**Actas XVIII JENUI 2012, Ciudad Real, 10-13 de julio 2012**
**I.S.B.N. 10: 84-615-7157-6 | I.S.B.N. 13:978-84-615-7157-4**
**Páginas 1-8**

# Enseñanza de la mutación en pruebas de software

Macario Polo Usaola y Pedro Reales Mateo
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4; 13071-Ciudad Real
{macario.polo, pedro.reales}@uclm.es

## Resumen

La mutación es una técnica de prueba de software desarrollada por investigadores y que, habitualmente, se ha utilizado casi de forma exclusiva con propósitos de investigación. Unos treinta y cinco años después de haber sido propuesta, la técnica está hoy suficientemente madura como para ser transferida a la industria y, también, para comenzar a ser introducida en la enseñanza reglada. Desde hace unos años venimos impartiendo mutación en la asignatura "Pruebas y seguridad de sistemas de información", del Máster Oficial en Tecnologías Informáticas Avanzadas de nuestra universidad, que cuenta con mención de calidad de la ANECA.

Lo cierto es que, mientras no introdujimos una metáfora, en la que se comparan los mutantes con faltas de ortografía, y los *test suites* con revisores ortotipográficos, la mutación era difícil de entender por parte de los alumnos o, tal vez, éramos los profesores los que no la explicábamos suficientemente bien.

En este trabajo describimos la metáfora que tan buenos resultados nos ha dado, así como algunos de los otros contenidos que impartimos, relacionados principalmente con la aplicación de la técnica y diversas estrategias para reducción de costes. Utilizamos una herramienta que hemos desarrollado y que se encuentra disponible libremente para su uso en universidades.

## Summary

Mutation is a software testing technique developed by researchers and usually only applied in research. Around 35 years after its proposal, the technique is today ready for both its transference to industry as for its inclusion in the syllabus of Computer Science. We teach mutation in "Testing and security of information systems", inside our quality-certified PhD program.

The first years we taught mutation, students took too much time in understanding the basic idea of mutation. In some moment, we introduced a metaphor (which compares the artificial faults inserted in programs with text typos, and test suites with typographic correctors) to describe mutation, obtaining excellent results in our explanations (maybe they had not been good enough until that moment) and in their comprehension.

This work describes the metaphor that so good results has given us, as well as the remaining contents taught. These are mainly related to the application of the technique and to several strategies for cost reduction. We use a tool we have developed and that is freely available for universities.

## Palabras clave:

Mutación *testing*, pruebas, metáforas.

## Introducción y motivación

La mutación es una técnica de pruebas propuesta en 1978 por DeMillo, Lipton y Sayward [1]. Durante muchos años, se ha utilizado fundamentalmente como técnica de validación de conjuntos de pruebas (*test suites)*.

El objetivo principal de las pruebas es el hallazgo de errores en el programa bajo prueba, de tal manera que si un *test suite* no encuentra errores no es, muy probablemente, porque el sistema no los tenga, sino porque los casos de prueba estén mal diseñados.

A partir de esta idea, la calidad de un *test suite* en mutación se mide en función del número de fallos artificiales sembrados en el sistema bajo prueba (el *SUT*, por sus siglas en inglés: *System Under Test)*. Para ello, se aplican una serie de operadores de mutación que generan copias defectuosas del SUT: en cada copia (llamada

*mutante*) se inserta al menos un fallo (realmente un cambio sintáctico o semántico, ya que hay mutantes en los que el cambio no altera el comportamiento del programa original, siendo entonces una optimización o *desoptimización* del código).

El operador de mutación AOR (*Arithmetic Operator Replacement)*, por ejemplo, sustituye un operador aritmético en una expresión por otro; el UOI (*Unary Operator Insertion)* inserta un operador unario en una variable: en los tres primeros mutantes de la Tabla 1, AOR ha sustituido el operador "+" del programa original por "-", "*" y "/". En el cuarto, el "+" original permanece, pero se ha insertado un operador de postincremento después de la variable *b*.

| Versión | Código |
|---|---|
| P (original) | int sum(int a, int b) {<br>return a + b;<br>} |
| Mutante 1 | int sum(int a, int b) {<br>return a **-** b;<br>} |
| Mutante 2 | int sum(int a, int b) {<br>return a ***** b;<br>} |
| Mutante 3 | int sum(int a, int b) {<br>return a / b;<br>} |
| Mutante 4 | int sum(int a, int b) {<br>return a + b++;<br>} |

Tabla 1. Una sencilla función y cuatro mutantes

Un buen *test suite* debería encontrar los cuatro errores insertados artificialmente.

| | Datos de prueba | | | |
|---|---|---|---|---|
| | (1, 1) | (0, 0) | (-1, 0) | (-1, -1) |
| P | 2 | 0 | -1 | -2 |
| Mutante 1 | 0 | 0 | -1 | 0 |
| Mutante 2 | 1 | 0 | 0 | 1 |
| Mutante 3 | 1 | Error | Error | 1 |
| Mutante 4 | 2 | 0 | -1 | -2 |

Tabla 2. Cuatro casos de prueba para el ejemplo de la Tabla 1

En la Tabla 2 se muestran los datos de entrada de cuatro casos de prueba construidos para encontrar los cuatro errores introducidos en los mutantes de la Tabla 1:

- El caso de prueba (1, 1) produce un 2 como salida en el programa original, y 0, 1, 1 y 2 en los mutantes. Esto significa que el caso encuentra los errores introducidos en los mutantes 1, 2 y 3, pero no en el 4º.
- El caso (0, 0) produce un 2 como salida en el original, y solamente una salida distinta en el mutante 3 (da un error al dividir 0 entre 0), por lo que no encuentra los errores introducidos en los mutantes 1, 2 y 4.
- El caso (-1, 0) encuentra los dos errores de los mutantes 2 y 3. Ante estos datos de entrada, los mutantes 1 y 4 devuelven las mismas salidas que el original.
- Finalmente, el caso (-1, -1) posee la misma efectividad ("es igual de bueno") que el primero, ya que encuentra los errores de los mutantes 1, 2 y 3.

Cuando un caso de prueba encuentra el error sembrado en un mutante se dice que el mutante está "muerto", y "vivo" cuando no hay ningún caso de prueba que encuentra el error. En nuestro ejemplo, los mutantes 1, 2 y 3 están muertos (todos ellos varias veces, además, si se nos permite la expresión), y el 4 está vivo.

Un mutante puede estar vivo o bien porque no haya ningún caso de prueba en el *test suite* que consiga matarlo, o bien porque el mutante sea "funcionalmente equivalente": es decir, que siempre mostrará, para cualquier caso de prueba, exactamente el mismo comportamiento que el programa original. El mutante 4 es uno de éstos (al menos en el lenguaje Java), ya que el postincremento se produce *después* de que la función devuelva su resultado, lo que impide observar la diferencia de estado entre el programa original y el mutante.

La calidad del test suite se mide mediante el *mutation score*, que es el porcentaje (o el tanto por uno) de mutantes no equivalentes que consigue matar el test suite (Ecuación 1).

$$MS(P,T) = \frac{K}{M-E} \quad (1)$$

P= Programa bajo prueba
T= *Test suite*
K= Número de mutantes muertos (*killed)*
M= Número total de mutantes
E= Número de mutantes funcionalmente equivalentes

La explicación y el ejemplo dados en los párrafos precedentes son muy similares a los que se encuentran en la literatura científica y, también, son los que hemos utilizado durante varios años en las clases de la asignatura del máster *Pruebas y seguridad de sistemas de información.*

Sin embargo, tanto nuestra percepción durante el desarrollo de las clases, como la de los trabajos prácticos que les encargábamos, como la obtenida de las respuestas dadas en texto libre por los alumnos a las encuestas de satisfacción que nosotros mismos elaborábamos evidenciaban que, ciertamente, costaba trabajo transmitir la idea de la mutación al alumnado.

Por ello, tras un periodo de reflexión, decidimos presentar los conceptos de la mutación con una metáfora, que luego hemos utilizado con éxito en otros cursos y seminarios.

En la Sección 2 presentamos el texto principal que utilizamos como metáfora, que nos sirve de base para presentar los conceptos de mutación que se explican en la Sección 3, y un texto secundario en la 4. Con un ejemplo ya más técnico, se presenta la idea de la subsunción de criterios de cobertura en la Sección 5. En la Sección 6 damos una breve descripción de la herramienta de mutación Bacterio que hemos desarrollado en el grupo de investigación y que también utilizamos en la docencia. Finalmente, presentamos nuestras conclusiones en la Sección 7.

## *La casada infiel*, de Federico García Lorca

Para ilustrar la idea que subyace en la técnica de pruebas con mutación, presentamos al alumno un fragmento del poema *La casada infiel*, de Federico García Lorca, en el que hemos introducido algunos fallos artificiales (Figura 1).

A los alumnos les pedimos que piensen que se están enfrentando a una prueba de contratación de una empresa editorial que desea contratar a un corrector ortotipográfico para la revisión de sus originales: el editor contratará a aquel aspirante que encuentre todos los errores existentes en el texto que se le presenta.

Además explicamos que, si al corrector seleccionado se le presenta un texto nuevo y no encuentra ningún fallo, lo que está sucediendo probablemente es que el texto esté libre de ellos.

Y que yo me la llevé al rio
creyendo que era mozuela,
pero tenia marido.
Fue la noche de Satniago
y casi por conpromiso.
Se apagaron los faroles
y se encendieron los grillos.
En las ultimas esquinas
toqué sus pechos dormidos,
y se me avrieron de pronto
como ramos de jacintos.
El almidon de su enagua
me sonaba en el oido,
como una pieza de seda
rasgada por diez cuchiyos.
Sin luz de plata en sus copas
los árboles han cresido,
y un horizonte de perros
ladra muy lejos del río.

Figura 1. Fragmento *mutante* de *La casada infiel*

Tras unos minutos, mostramos a los alumnos el texto original, resaltando en negrita los errores existentes en el texto mutante (Figura 2).

Y que yo me la llevé al **río**
creyendo que era mozuela,
pero **tenía** marido.
Fue la noche de **Santiago**
y casi por **compromiso**.
Se apagaron los faroles
y se encendieron los grillos.
En las **últimas** esquinas
toqué sus pechos dormidos,
y se me **abrieron** de pronto
como ramos de jacintos.
El **almidón** de su enagua
me sonaba en el **oído**,
como una pieza de seda
rasgada por diez **cuchillos**.
Sin luz de plata en sus copas
los árboles han **crecido**,
y un **horizonte** de perros
ladra muy lejos del **río**.

Figura 2. Fragmento original del poema, con los fallos resaltados

## Conceptos de mutación que incluye la metáfora

A partir de una sola transparencia (la Figura 3 muestra la versión inglesa) en la que mostramos el texto malo y desvelamos el texto bueno, podemos presentar y explicar varios conceptos.

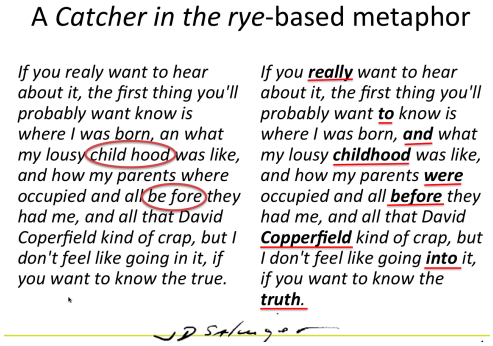


Figura 3. Transparencia utilizada en la versión en inglés

### Mutantes y mutantes de orden *n*

Por lo general, los mutantes de código incluyen solamente un fallo. Sin embargo, y con el fin de abaratar los costes de las pruebas, una técnica relativamente reciente es la mutación de orden *n* [2], por la cual en cada mutante se incluyen *n* errores artificiales. El hallazgo de uno solo de los *n* errores supone la *muerte* del mutante.

En la Figura 2 se observa que el texto presentado incluye 9 errores, lo que se corresponde con un mutante de "orden 9". De este modo, se presentan tanto el concepto de *mutante* como el de *mutante de orden n*.

### Operadores de mutación

Los errores artificiales se introducen mediante operadores de mutación (poníamos como ejemplo el AOR y el UOI, si bien hay muchos más: la Tabla 3 muestra algunos de los *tradicionales;* hay otros para orientación a objetos).

Cada operador de mutación introduce un tipo de error diferente. En el lado izquierdo de la transparencia mostrada en la Figura 3 se resaltan con un círculo las palabras *child hood* y *be fore*, cuyos correspondientes términos correctos son *childhood* y *before.* El error, en este caso, consiste en la inserción de un espacio en mitad de la palabra; otras palabras erróneas son *realy (really)* y *Coperfield (Copperfield),* que proceden de la supresión de una letra repetida. Estos dos tipos de cambios corresponden a dos distintos operadores de mutación que introducen en el texto espacios en blanco para separar palabras y suprimen letras repetidas.

Con estos ejemplos, se presenta el concepto de operador de mutación, que introduce un tipo de error bien determinado en el sistema bajo prueba.

| **Operadores tradicionales** | |
|---|---|
| ABS (absolute value) | Sustitución de una variable por el valor absoluto de dicha variable |
| ACR (array reference for constant replacement) | Sustitución de una referencia variable a un array por una constante |
| AOR (arithmetic operator replacement) | Sustitución de un operador aritmético |
| CRP (constant replacement) | Sustitución del valor de una constante |
| ROR (relational operator replacement) | Sustitución de un operador relacional |
| RSR (return statement replacement) | Sustitución de la instrucción Return |
| SDL (statement deletion) | Eliminación de una sentencia |
| UOI (unary operator insertion) | Inserción de operador unario (p.ej.: en lugar de x, poner – x) |

Tabla 3. Algunos operadores de mutación "tradicionales"

### *Test suite* y calidad del *test suite*

El *test suite* es el conjunto de casos de prueba y tiene por objetivo encontrar errores en el sistema bajo prueba. Desde este punto de vista, un *test suite A* es mejor que otro *B* si *A* encuentra en el SUT más errores que *B.*

En nuestra metáfora, cada uno de los alumnos se corresponde con un *test suite*. Para elegir al corrector ortotipográfico, la empresa editorial contratará a aquel revisor que encuentre más fallos en el texto mutado que se le esté presentando.

## Las tres enamoradas

Una vez planteados y bien entendidos los conceptos planteados en los puntos anteriores, pasamos a explicar a los alumnos métodos para mejorar la calidad del *test suite*, lo que lleva a describir el proceso de pruebas basado en mutación.

Un ingenioso texto que puede encontrarse en Internet y cuyo autor o autora, lamentablemente, no hemos sido capaces de encontrar, cuenta la historia de un joven que anda flirteando con tres hermanas. Éstas, un día, le piden que se decida por una, a lo que él les responde enviándoles los siguientes versos, que no ha tenido tiempo para puntuar, pidiéndoles que lo hagan ellas:

*Juana Teresa y Leonor*
*puestas de acuerdo las tres*
*me piden que diga cuál es*
*la que prefiere mi amor*

*Si obedecer es rigor*
*digo pues que amo a Teresa*
*no a Leonor cuya agudeza*
*compite consigo ufana*
*no aspira mi amor a Juana*
*que no es poca su belleza*

**Teresa** puntuó la segunda estrofa de esta manera:

*Si obedecer es rigor,*
*digo, pues, que amo a* ***Teresa****.*
*No a Leonor, cuya agudeza*
*compite consigo ufana.*
*No aspira mi amor a Juana,*
*que no es poca su belleza.*

**Leonor**, sin embargo, colocó los siguientes puntos y comas:

*Si obedecer es rigor,*
*¿digo, pues, que amo a Teresa?*
*No, a* ***Leonor****, cuya agudeza*
*compite consigo ufana.*
*No aspira mi amor a Juana,*
*que no es poca su belleza.*

**Juana**, la tercera hermana, pensándose la elegida, la puntuó así:

*Si obedecer es rigor,*
*¿digo, pues, que amo a Teresa?*
*No. ¿A Leonor, cuya agudeza*
*compite consigo ufana?*
*No. Aspira mi amor a* ***Juana****,*
*que no es poca su belleza.*

El **joven,** que finalmente era un carota (y cuyo texto se correspondería con el programa original), les dijo que las tres habían puntuado mal, ya que no quería compromiso con ninguna de ellas. En este punto, pedimos a los alumnos que puntúen el texto tal y como lo hizo el joven (de modo que obtengan así la versión correcta), que finalmente les revelamos tras dejarles que dediquen unos minutos a pensarlo:

***Juana Teresa*** *y* ***Leonor****,*
*puestas de acuerdo las tres,*
*me piden que diga cuál es*
*la que prefiere mi amor.*

*Si obedecer es rigor,*
*¿digo pues que amo a* ***Teresa****?*
*No. ¿A* ***Leonor****, cuya agudeza*
*compite consigo ufana?*
*No. ¿Aspira mi amor a* ***Juana****!*
*¡Que no! Es poca su belleza.*

Este ejemplo de las tres enamoradas sirve para ilustrar varios conceptos:

- No todos los errores introducidos por el joven (ausencia de signos de puntuación) han sido encontrados por los *test suites* de Juana, Teresa y Leonor, por lo que es necesario mejorarlos.
- Dado que los *test suites* han encontrado sólo unos pocos errores, deben mejorarse o bien añadiendo nuevos casos de prueba, o bien mejorando los oráculos de los casos de prueba ya existentes: es decir, dotando a los casos de prueba de un mejor mecanismo para encontrar las diferencias de significado.

El ejemplo de la versión inglesa es más sencillo. En este caso, la anécdota cuenta la historia de un oso panda que entra a un bar, pide un bocadillo, se lo come, dispara un tiro con una pistola y se va. El camarero lo persigue hasta la salida y le pregunta que por qué reacciona de esa manera. El oso responde que, según la enciclopedia, un oso panda es un *"Large black-and-white bear-like mammal, native to China. Eats, shoots and leaves"* ("Come, dispara y sale"), y le explica que él se quiere comportar como tal. El camarero busca la definición en la enciclopedia y advierte entonces que le sobra una coma, pues lo correcto sería: *"Eats shoots and leaves"* ("Come brotes y hojas").

A partir de aquí, explicamos el proceso de pruebas mediante mutación propuesto por Offutt [3], que consta de los siguientes pasos:

1. Partimos de *P* (programa bajo prueba) y *T* (el *test suite).*

2. Generamos el conjunto de mutantes: *M={m1, m2, ... mn}*.
3. Ejecutamos *T* contra todos los mutantes vivos que haya en *M*.
4. Eliminamos los mutantes equivalentes.
5. Calculamos el *mutation score.*
6. Sacamos de *M* el conjunto de mutantes muertos.
7. Eliminamos los casos de prueba inefectivos (los que no matan ningún mutante) y:
   7.1 Si se alcanzó en el punto 5 el *mutation score* deseado, entonces se ejecuta *T* contra *P:*
      7.1.1 Si *T* (que es completo, porque alcanza el *mutation score*) no encuentra errores en *P*, termina el proceso.
      7.1.2 Si *T* sí encuentra errores, entonces se corrige *P* y, dado que *P* ha cambiado, se vuelve al paso 2, repitiendo el proceso.
   7.2 Si no se llega al *mutation score* deseado, entonces hay que revisar los casos de prueba existentes (tal vez no tengan un oráculo suficientemente bien descrito como para detectar los cambios de estado) y quizás añadir nuevos casos de prueba a *T* y volver al paso 3.

## La mutación frente a otros criterios de cobertura. Subsunción de criterios. Enriquecimiento del *test suite*

Antes de tratar la mutación, en clase se trabaja sobre otros criterios de cobertura, como sentencias, decisiones, condiciones, condición-decisión, cobertura completa de condiciones y condición-decisión modificada (MC/DC).

La cobertura completa de condiciones es el método más exhaustivo para probar una sentencia condicional, pues se prueba la tabla de verdad completa de la decisión, dando a cada condición que la compone los valores *true* y *false*. El número de casos de prueba para cubrir este criterio, sin embargo, es elevadísimo ($2^n$, siendo *n* el número de condiciones), por lo que en la práctica se utiliza MC/DC, que requiere muchos menos casos de prueba y ofrece una capacidad de detección de errores prácticamente igual.

Para comparar la capacidad de hallazgo de errores de MC/DC frente a la de la mutación, presentamos a los alumnos la clase Java de la Figura 4, cuya operación *getMedian()* devuelve la mediana de los tres campos de la clase.

```
public class Median {
  private double median;
  private double a, b, c;

  public Median(double a,double b,double c){
        this.a=a; this.b=b; this.c=c;
  }

  public void calculateMedian() {
        if (a<b && b<c || a>b && b>c)
            median=b;
        else if (b<a && a<c || b>a && a>c)
            median=a;
        else median=c;
  }

  public double getMedian() {
        return this.median;
  }
}
```

Figura 4. Código de la clase *Median*

En el código anterior, el diseño de los casos de pruebas se centra, sobre todo, en pasar al constructor de la clase valores adecuados para que las dos decisiones de *calculateMedian* se cubran con cobertura MC/DC.

Las tablas 4 y 5 muestran, respectivamente, las condiciones de cada decisión, la propia decisión, los valores de prueba que deben pasarse como argumentos para lograr cobertura MC/DC (obsérvese que son los mismos casos, si bien se han construido en un orden distinto) y el resultado esperado. Con MC/DC, deben alcanzarse las ramas *true* y *false* de las dos decisiones gracias a la contribución individual de cada condición de la decisión. Tomando como ejemplo la tabla 4:

- Los valores *(1, 2, 3)* hacen ciertas las condiciones *a<b* y *b<c* y falsas las otras dos, por lo que la decisión vale *true* gracias a la contribución de *a<b* y de *b<c* (al estar éstas separadas por *&&*, no es posible ir a la rama *true* gracias sólo a una de las dos condiciones, por lo que ambas deben hacerse ciertas).
- Los valores *(2, 3, 1)* y *(2, 1, 3)* hacen falsa la decisión gracias, respectivamente, a las condiciones *b<c* y *a<b.*
- Las tres últimas ternas de valores representan las dos anteriores circunstancias, pero para la segunda parte de la decisión (sus dos últimas condiciones).

| Condiciones | | | | Decisión | Valores de prueba | | | Resultado esperado |
|---|---|---|---|---|---|---|---|---|
| a<b | b<c | a>b | b>c | (a<b&&b<c \|\| a>b&&b>c) | a | b | c | |
| T | T | F | F | T | 1 | 2 | 3 | b |
| T | F | F | F | F | 2 | 3 | 1 | a |
| F | T | F | F | F | 2 | 1 | 3 | a |
| F | F | T | T | T | 3 | 2 | 1 | b |
| F | F | T | F | F | 3 | 1 | 2 | c |
| F | F | F | T | F | 1 | 3 | 2 | c |

Tabla 4. Casos de prueba para alcanzar cobertura MC/DC en la primera decisión

| Condiciones | | | | Decisión | Valores de prueba | | | Resultado esperado |
|---|---|---|---|---|---|---|---|---|
| b<a | a<c | b>a | a>c | b<a && a<c \|\| b>a && a>c | a | b | c | |
| T | T | F | F | T | 2 | 1 | 3 | a |
| T | F | F | F | F | 3 | 2 | 1 | b |
| F | T | F | F | F | 1 | 2 | 3 | b |
| F | F | T | T | T | 2 | 3 | 1 | a |
| F | F | T | F | F | 1 | 3 | 2 | c |
| F | F | F | T | F | 3 | 1 | 2 | c |

Tabla 5. Casos de prueba para lograr cobertura MC/DC en la segunda decisión

Al escribir en formato JUnit los casos de prueba descritos en las dos tablas anteriores, los resultados obtenidos de su ejecución son todos satisfactorios, en el sentido de que coinciden en todos los casos con los esperados.

Ya que el *test suite* no encuentra errores en el programa bajo prueba y, además, lo recorre "completamente" (en el sentido de que verifica el criterio de cobertura elegido), el *test suite* es completo y el sistema está listo para pasarlo al entorno de producción.

Sin embargo, al generar mutantes con la herramienta Bacterio (que describimos en la sección siguiente), se obtiene un *mutation score* del 76,27%. En la Figura 5 se muestra el código de dos de estos mutantes, en los que el operador < se ha sustituido, en la primera y segunda condiciones, por <=.

```
(1) if (a<=b && b<c || a>b && b>c)
(2) if (a<b && b<=c || a>b && b>c)
```

Figura 5. Código del constructor de *Median* en un mutante

Teniendo en cuenta que lo habitual es que entre un 18 y un 20% de los mutantes sean equivalentes, al *tester* le queda aún un 4 o 5 por ciento de margen para llegar al 80-82% deseable. Por ello, debe inspeccionar el código de los mutantes vivos y, si no son funcionalmente equivalentes, escribir casos de prueba que consigan matarlos: para matar al mutante (1), los valores de los dos primeros parámetros del constructor deben ser iguales; para matar al (2), los parámetros segundo y tercero: se añaden para ellos dos casos de prueba compuestos por los valores de entrada *(1, 1, 2)* y *(1, 2, 2),* cuyos respectivos resultados esperados son *1* y *2.* Sin embargo, al ejecutarlos, JUnit muestra para estos dos últimos casos un veredicto de fallo (Figura 6).

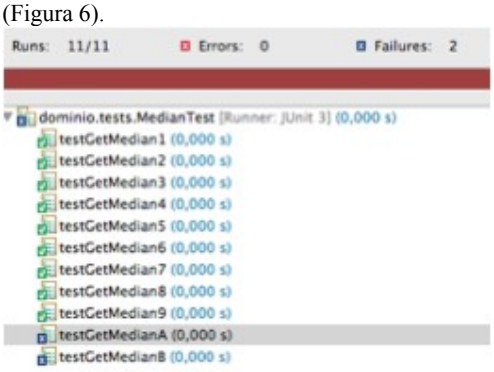


Figura 6. Los dos nuevos casos de prueba encuentran errores en el sistema

Obviamente, los errores se encuentran, precisamente, en que el método *calculateMedian* no contempla el paso de argumentos iguales.

En este caso, la mutación ha encontrado errores que, con otra técnica bien conocida y ampliamente utilizada, no habían sido encontrados. Explicamos, en este caso, que la mutación *subsume* los criterios de sentencias, decisiones, condiciones e, incluso, MC/DC.

## Bacterio: una herramienta para la automatización de pruebas con mutación

En el grupo de investigación hemos desarrollado la herramienta Bacterio, que permite realizar pruebas mediante mutación de sistemas Java, y que incluye la mayoría de las técnicas de reducción de costes propuestas en la literatura:

1. Mutación de orden *n*, con diferentes algoritmos de combinación de mutantes.
2. Mutación débil (*weak mutation),* fuerte (*strong),* débil flexible (*FWM: Flexible Weak*

*Mutation)* y basada en JUnit (la llamada *Functional Qualification)*.
3. Ejecución en paralelo de los casos de prueba, con cinco algoritmos de distribución de la carga de trabajo.
4. Mutación selectiva.
5. Selección aleatoria de mutantes.
6. Indicación de los mutantes visitados pero no muertos, que es un indicativo de la probabilidad de que el mutante sea equivalente.
7. Posibilidad de hacer pruebas de mutación a los niveles de sistema y de integración.
8. Diferentes formas de comparación de los estados de los mutantes con respecto al programa original.
9. Diferentes algoritmos para ejecutar los casos de prueba frente a los mutantes (es decir, diferentes formas de rellenar la matriz de muertos).
10. Posibilidad de hacer pruebas exploratorias de aplicaciones de escritorio, gracias a su funcionalidad de *Capture and replay*.

Teniendo claros los conceptos de la mutación, Bacterio es una herramienta fácil de usar, y permite a los alumnos poner en práctica la totalidad de los contenidos impartidos en la unidad dedicada a la mutación. Para la evaluación de esta parte de la asignatura, les solicitamos, entre otros, que propongan y desarrollen un trabajo en el que deben comparar al menos dos técnicas de prueba mediante su aplicación en un sistema real:

- Para la generación de los casos de prueba les sugerimos el uso de *CTWeb*, una herramienta web que también hemos desarrollado en el grupo y que está disponible en alarcosj.esi.uclm.es/CTWeb. *CTWeb* puede generar casos de prueba mediante algoritmos combinatorios o a partir de máquinas de estado.
- Una vez que los alumnos han generado los casos de prueba, los deben ejecutar con Bacterio aplicando más de una técnica.

De este modo profundizan y enfatizan, por un lado, en los contenidos de la asignatura; y, por otro, y ya que el trabajo lo han propuesto ellos (obviamente con el VºBº del profesor, que a veces lo completa y lo retoca) y son ellos los que deben proponer y discutir la bondad de las técnicas seleccionadas y aplicadas, se fomenta también su incipiente formación investigadora.

Bacterio está disponible libremente para su utilización en investigación y docencia universitaria en:
MutationAndCombinatorialTesting.blogspot.com

## Conclusiones

La utilización de metáforas es una técnica docente muy utilizada en multitud de disciplinas, incluyendo la propia Informática. Este trabajo se ha centrado, sobre todo, en la presentación de metáforas literarias para facilitar la comprensión del concepto de mutación a los estudiantes.

Si bien no disponemos de resultados cuantitativos respecto de las mejoras en tiempo o en dificultad, sí que tenemos el conocimiento fehaciente (tanto por nuestras impresiones subjetivas, como por los cambios de opinión que hemos encontrado en las respuestas dadas por los alumnos a las encuestas de satisfacción) de que el esfuerzo empleado en la búsqueda y presentación de las metáforas compensa con creces a la presentación puramente técnica de la mutación.

Además, la herramienta Bacterio ha mostrado ser una gran ayuda para poner en práctica todos los conceptos de mutación presentados en clase. En años anteriores, cuando no habíamos desarrollado todavía Bacterio, utilizábamos la herramienta MuJava, cuya configuración y uso son mucho más complicados.

## Referencias

DeMillo R, Lipton RJ and Sayward FG (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4), p. 34-41.

Polo M, Piattini M and García-Rodríguez I. (2008). Decreasing the cost of mutation testing with 2-order mutants. *Software Testing, Verification and Reliability*, 19(2), 111-131.

Offutt AJ. (1995). A practical system for mutation testing: help for the common programmer. *12th International Conference on Testing Computer Software*, pp. 99-109. 1995.