

Universidad de Sevilla  
Escuela Técnica Superior de Ingeniería Informática

## **Mutation testing**



Grado en Ingeniería Informática – Ingeniería del Software  
Diseño y Pruebas II  
Curso 2019 – 2020

### **Miembros del equipo**

Jorge Andrea Molina  
Juan Carlos Cortés Muñoz  
María Elena Molino Peña  
Alejandro Muñoz Aranda  
Mario Ruano Fernández  
Fernando Ruíz Robles

<https://github.com/dp2-g3-7/petclinic.git>

## Índice

Pruebas de mutación .....	2
¿En qué consisten las pruebas de mutación? .....	2
Tipos de mutaciones .....	4
Implementación de pruebas de mutación en el proyecto .....	5
Ejecución de las pruebas de mutación.....	7
Resultados obtenidos.....	8
Conclusión .....	10

## Pruebas de mutación

### ¿En qué consisten las pruebas de mutación?

Las pruebas de mutación no están enfocadas a probar la calidad del software desarrollado. Dicho de otra manera, no persiguen medir la calidad de la implementación de las diferentes funcionalidades que se integran en el proyecto, sino en testear la calidad de nuestra pruebas unitarias: ¿cómo de buenos son nuestros test unitarios? ¿realizamos los asertos correctos? ¿son frágiles ante los cambios en el código?

Evidentemente, como la gran mayoría de pruebas de código, las pruebas de mutación también ayudan a detectar fallos y errores de manera indirecta, lo que implicará una refactorización del código fuente en etapas tempranas del ciclo de desarrollo y supondrá un importante ahorro de costes a medio y largo plazo.

Poder tener una respuesta segura acerca de la fiabilidad de las pruebas unitarias de un proyecto nos permite contar con la certeza de cómo de robusta es nuestra suite de tests.

Para poder alcanzar esta meta, las pruebas de mutación se presentan como el resultado de la creación de variantes del código fuente original. Se realizan modificaciones en nuestro código, las cuales se llaman mutaciones, y se comprueba si el test unitario que prueba dicha parte del código continúa pasando satisfactoriamente o, sin embargo, se rompe y falla.

Es curioso que el objetivo primordial de las pruebas de mutación sea romper los tests que anteriormente se han implementado y probado minuciosamente, tratando de que pasen satisfactoriamente. Cuanto mayor sea el impacto de las mutaciones, mucho mejor en términos de fiabilidad de la suite de tests.

Signo de que los tests son robustos, y de que realmente están probando que la implementación cuenta con una buena cobertura de pruebas de calidad, es que los tests se rompan con estas modificaciones. Cuando esto ocurre se dice que el mutante ha muerto. En caso contrario, el mutante queda vivo.

En concreto, la calidad de la suite de tests se mide mediante el "mutation score" que viene dado por la expresión  $MS(P,T) = K / (M-E)$  donde P es el programa que se está probando, T es la suite de tests, K el número de mutantes muertos, M el número total de mutantes y E es el número de mutantes funcionalmente equivalentes.

Según esto, hay un paso previo necesario que se deduce como un requisito fundamental para que los datos que se obtengan de las pruebas de mutación sean datos de peso. Si nuestros tests unitarios son escasos y el porcentaje de cobertura del código a probar es bajo, es muy probable que los datos reportados por estas pruebas sean poco fiables, por buenos que pudieran ser.

Por lo tanto, tras conseguir implementar una suite suficiente que cubra el código a poner a prueba, se podrán llevar a cabo las pruebas de mutación.

## Tipos de mutaciones

A continuación, se presentan las mutaciones más habituales que se producen en el código, las cuales quedan identificadas por un operador, y una breve descripción de en qué consiste.

Opera- dor	Descripción
UOI	Inserción de operador unario. Por ejemplo, cambiar un signo + por un signo -
SDL	Eliminación de una sentencia. Por ejemplo, eliminar una variable
RSR	Sustitución de la instrucción de return
ROR	Sustitución de un operador relacional. Por ejemplo cambiar la condición de un if
CRP	Sustitución del valor de una constante
AOR	Sustitución de un operador aritmético
ACR	Sustitución de una referencia variable a un array por una constante
ABS	Sustitución de una variable por un valor absoluto

## Implementación de pruebas de mutación en el proyecto

Con las ideas expuestas anteriormente, se puede llegar a la conclusión de que la implementación de este tipo de pruebas no pasa por una labor manual, ya que los cambios requeridos (el número de mutaciones) deben ser muy grandes para poder obtener un resultado de evaluación fiable.

Por esta razón se hace necesario el uso de alguna herramienta que automatice los tests de mutación y que sea capaz de generar un gran volumen de mutantes del código fuente y lanzar, para cada uno de ellos, los tests unitarios.

Para ello, en el proyecto se ha utilizado [PIT](#) como herramienta para crear y ejecutar las pruebas de mutación.

Siguiendo las instrucciones de la [documentación de PIT](#) para integrar este plugin en el proyecto, se debe especificar la configuración de las pruebas en el archivo pom.xml.

```
<plugins>
  <plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.5.2</version>
    <dependencies>
      <dependency>
        <groupId>org.pitest</groupId>
        <artifactId>pitest-junit5-plugin</artifactId>
        <version>0.12</version>
      </dependency>
    </dependencies>
    <configuration>
      <targetClasses>
        <param>org.springframework.samples.petclinic.service.StayService</param>
        <param>org.springframework.samples.petclinic.web.StayController</param>
        <param>org.springframework.samples.petclinic.service.AppointmentService</param>
        <param>org.springframework.samples.petclinic.web.AppointmentController</param>
      </targetClasses>
      <targetTests>
        <param>org.springframework.samples.petclinic.service.StayServiceTests</param>
        <param>org.springframework.samples.petclinic.web.StayControllerTests</param>
        <param>org.springframework.samples.petclinic.service.AppointmentServiceTests</param>
        <param>org.springframework.samples.petclinic.web.AppointmentControllerTests</param>
      </targetTests>
    </configuration>
  </plugin>
</plugins>
```

**Servicios testeados.** Se despliegan los tests de mutación correspondientes a las clases StayService.java, StayController.java, AppointmentService.java y AppointmentController.java. Estas serán las cuatro clases del proyecto que van a experimentar mutaciones. Se les aplicarán los tests unitarios correspondientes a las clases StayServiceTests.java, StayControllerTests.java, AppointmentServiceTests.java y AppointmentControllerTests.java respectivamente.

**Integración con JUnit 5.** Para que la integración del plugin de PIT sea correcto y funcione bien con la versión JUnit 5 que se utiliza en el proyecto, es necesario contar con la [dependencia](#) pitest-junit5-plugin que también se especifica en el archivo pom.xml, la cual da soporte a PIT con las tests unitarios del proyecto.

## Ejecución de las pruebas de mutación

Una vez que se ha configurado el archivo pom.xml se pueden ejecutar las pruebas de mutación mediante el siguiente comando Maven:

```
> mvn org.pitest:pitest-maven:mutationCoverage
```

Esto comenzará con la generación de mutantes y la ejecución de los correspondientes tests unitarios en un segundo plano.

Tras finalizar la ejecución, la traza de la consola nos arroja un reporte breve de las pruebas realizadas:

```
=====
===
- Statistics
=====
===
>> Generated 156 mutations Killed 121 (78%)
>> Ran 267 tests (1.71 tests per mutation)
[INFO] -----
--
[INFO] BUILD SUCCESS
[INFO] -----
--
[INFO] Total time: 01:00 min
[INFO] Finished at: 2020-05-21T19:00:28+02:00
[INFO] -----
--
```

Se han generado 156 mutaciones, de las cuales 121 han muerto (78%) y se han ejecutado un total de 267 tests. Es un porcentaje bastante alto, cercano al 80%, por lo que se puede concluir que la calidad de los tests unitarios de estos servicios es bastante buena.



## Resultados obtenidos

Para este caso, se han realizado los tests de mutación para los servicios de Stay y Appointment con la siguiente cobertura de tests unitarios:

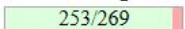
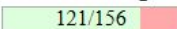
- StayService.java: 100%
- StayController.java: 96.3%
- AppointmentService.java: 82.4%
- AppointmentController.java: 92.6%

Como se puede observar, la cobertura es muy alta, en todos los casos superior al 80%, por lo que los resultados que se obtengan de las pruebas de mutación serán consistentes.

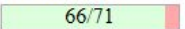
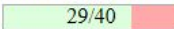
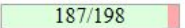
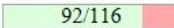
PIT genera también unos reportes técnicos detallados de cada una de las pruebas realizadas y del porcentaje de mutantes muertos y mutantes vivos, una vez que finalizado con la ejecución. Estos informes se generan en HTML en el directorio target/pit-reports.

## Pit Test Coverage Report

### Project Summary

Number of Classes	Line Coverage	Mutation Coverage
4	94%  253/269	78%  121/156

### Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
<a href="#">org.springframework.samples.petclinic.service</a>	2	93%  66/71	73%  29/40
<a href="#">org.springframework.samples.petclinic.web</a>	2	94%  187/198	79%  92/116

Esta es la vista principal del informe correspondiente al caso de ejemplo que se está siguiendo. Desde aquí se puede navegar a los diferentes paquetes y a las clases que se han probado. En cada una de ellas se tiene un detalle de la cobertura de los test de mutación y del resultado obtenido en cada uno de los test.

Con KILLED se indica que se ha matado al mutando, es decir, el test unitario ha fallado y se contabiliza como éxito.

## Mutations

```
1. removed call to org.springframework.samples.petclinic/model/Appointment::setOwner → KILLED
1. removed call to org.springframework.samples.petclinic/model/Appointment::setPet → SURVIVED
1. replaced return value with null for org.springframework.samples.petclinic/web/AppointmentController
1. replaced return value with Collections.emptyList for org.springframework.samples.petclinic/web/App
1. negated conditional → KILLED
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. negated conditional → KILLED
2. negated conditional → KILLED
1. negated conditional → KILLED
2. negated conditional → KILLED
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. negated conditional → KILLED
1. negated conditional → KILLED
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. removed call to org.springframework.samples.petclinic/service/AppointmentService::saveAppointment
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
1. negated conditional → KILLED
2. negated conditional → KILLED
1. negated conditional → KILLED
1. replaced return value with "" for org.springframework.samples.petclinic/web/AppointmentController
```

Esto también es muy útil para ver dónde hay posibles fallos y qué puede mejorarse, qué tests unitarios pueden faltar por implementar o qué otros son redundantes o poco efectivos.

## Conclusión

Todos éramos desconocedores de la existencia de las pruebas de mutación antes de la realización de este proyecto. Consideramos que son unas pruebas muy útiles para valorar la calidad del trabajo de testeo unitario, más cuando se está comenzando a adquirir experiencia en el campo del desarrollo y de las pruebas software, y es una herramienta bastante correctiva.

Además, ir evaluando la suite de tests conforme se va desarrollando es bastante beneficioso para reducir costes en el proyecto, mejorando en la eficiencia del trabajo del equipo.