Taller: PiTest

Generación Automática de Casos de Test - 2018

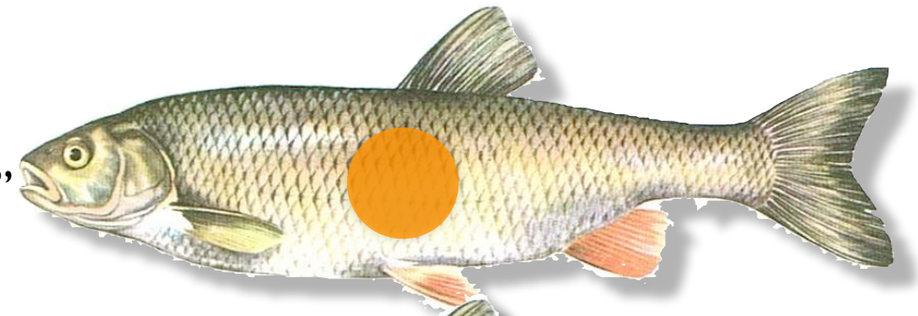
El Problema del Oráculo

- Ejecutar todo el programa suele no ser suficiente
- Necesitamos chequear el comportamiento funcional del programa
- Hace el programa realmente lo que queremos?

Oráculo de Delfos

Contando Mutantes

50 defectos "artificiales" detectados



300 defectos "naturales" detectados

Mutation Testing se centra en First Order Mutants (FOMs)

Hipótesis del Programador Competente

Efecto Acoplamiento

Mejoras

Do fewer

- Sampling de Mutantes
- Mutación Selectiva

Do smarter

- Paralelización
- Mutación Débil
- Uso de Cobertura
- Impacto

Do faster

- Mutar bytecode
- Meta-mutantes

Presentado en ISSTA 2016

PIT a Practical Mutation Testing Tool for Java (Demo)

Henry Coles
NCR, Edinburgh
henry@pitest.org

Thomas Laurent
Lero@UCD, Ireland & École Centrale de Nantes, France
thomas.laurent@eleves.ec-nantes.fr

Christopher Henard
University of Luxembourg
christopher.henard@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Anthony Ventresque
Lero@UCD, UCD, Ireland
anthony.ventresque@ucd.ie

ABSTRACT

Mutation testing introduces artificial defects to measure the adequacy of testing. In case candidate tests can distinguish the behaviour of mutants from that of the original program, they are considered of good quality – otherwise developers need to design new tests. While, this method has been shown to be effective, industry-scale code challenges its applicability due to the sheer number of mutants and test executions it requires. In this paper we present PIT, a practical mutation testing tool for Java, applicable on real-world codebases. PIT is fast since it operates on bytecode and optimises mutant executions. It is also robust and well integrated with development tools, as it can be invoked through a command line interface, Ant or Maven. PIT is also open source and hence, publicly available at http://pitest.org/

CCS Concepts

•**Software and its engineering** → **Software testing**

much of the source code is covered (i.e., merely executed) by the tests. Coverage metrics imply that the more coverage the merrier. This notion, while widely applied, has a major drawback; it only checks if a line/instruction is tested, not how well it is tested.

Mutation testing [4] is a technique that gives a better understanding of what the tests exercise on the program under analysis. Mutation introduces defects, in the form of small code modifications, which should result in an abnormal behaviour when exercised by tests. If the tests fail to expose the defects then the testers/developers can reasonably infer that the tests are not checking every possible behaviour and that the tests need to be improved.

This paper presents PIT, a mutation testing system for Java. PIT is considerably fast as it manipulates bytecode and runs only the tests that have a chances to kill the used mutants (i.e., the tests that execute the instruction where the mutant is located). PIT's major advantage is that it is robust, easy to use and well integrated with development

Real world mutation testing

PIT is a state of the art **mutation testing** system, providing **gold standard test coverage** for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started

User Group Issues Source Maven Central

Follow Tweet G+1

PiTest

- Utiliza meta-mutantes (no necesita recompilar N mutantes)
- Strong Mutation (test pasa o no pasa)
- Open-Source (https://github.com/hcoles/pitest)

PiTest

- Integrado con Gradle, ANT, Maven
- Puede seleccionar los operadores a aplicar
- Produce reportes HTML
  - Mutantes Vivos (ya sea que estén cubiertos o no por algún Test)
  - Mutantes Muertos (ie detectados por al menos un Test)

Operadores Activos por Default

- Conditionals Boundary Mutator
- Increments Mutator
- Invert Negatives Mutator
- Math Mutator
- Negate Conditionals Mutator
- Return Values Mutator
- Void Method Calls Mutator

Operadores no Activos por Default (activables)

- Constructor Calls Mutator
- Inline Constant Mutator
- Non Void Method Calls Mutator
- Remove Conditionals Mutator
- Experimental Member Variable Mutator
- Experimental Switch Mutator

pitest.org

CONDITIONALS_BOUNDARY

- < es reemplazado por <=
- <= es reemplazado por <
- > es reemplazado por >=
- >= es reemplazado por >

pitest.org

Taller #2: PiTest

- Vamos a usar el Test Suite que crearon en el Taller #1 para StackAr
- Vamos a usar PiTest como un plugin de Maven
- Vamos a mejorar el Test Suite para hacerlo más robusto wrt Mutation Testing

- Apache Maven es un *"software project management and comprehension tool"*
- Nos permite hacer deployment de proyectos (símil Makefiles)
- A fines del Taller, sólo es una línea de comando

- Descargar el .zip file en
  - www.dc.uba.ar/autotest/descargas/Taller02-pitest/stackar.zip
- Descomprimirlo y moverse a la carpeta “stackar”
- Ejecutar desde una terminal:
  - $ mvn clean install org.pitest:pitest-maven:mutationCoverage

- ¿Qué hizo Maven?
  - Descargo las dependencies (librerías) necesarias para el proyecto (PiTest,JUnit)
  - Compilo el código fuente
  - Ejecutó los JUNITs
  - Ejecutó PiTest y generó los reportes HTML

Reporte de Mutantes Muertos/Vivos de PiTest

- En la carpeta stackar/target encontrará:
  - Una carpeta “pit-reports” con el reporte de la actividad de PiTest
    - $ open target/pit-reports/*/index.html

Pit Test Coverage Report

Package Summary

org.autotest

Number of Classes	Line Coverage		Mutation Coverage	
1	0%	0/52	0%	0/37

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
StackAr.java	0%	0/52	0%	0/37

StackAr.java

```
1   package org.autotest;
2
3   import java.util.Arrays;
4
5   public class StackAr {
6
7           private final static int DEFAULT_CAPACITY = 10;
8
9           private final Object[] elems;
10
11          private int readIndex = -1;
12
13          public StackAr() {
14                  this(DEFAULT_CAPACITY);
15          }
16
17          public StackAr(int capacity) throws IllegalArgumentException {
18  2               if (capacity < 0) {
19                          throw new IllegalArgumentException();
20                  }
21                  this.elems = new Object[capacity];
22          }
23
24          public int size() {
25  2               return readIndex+1;
26          }
27
28          public boolean isEmpty() {
29  2               return size() == 0;
30          }
```

Enunciado (1)

1. Reemplazar TestStackAr.java con el test suite creado para el Taller #1

- Copiar su archivo TestStackAr.java a la carpeta stackar/src/test/java/org/autotest/

Enunciado (2)

2. Ejecutar PiTest usando el nuevo test suite:
   - Ejecutar el comando:
     - $ mvn clean install org.pitest:pitest-maven:mutationCoverage
- ¿Cuántas líneas cubiertas reporta PiTest?
- ¿Cuántos mutantes vivos reporta PiTest?
- ¿Cuál es el mutation score (mutantes vivos /mutantes totales) que reporta PiTest?

Enunciado (3)

3. Extender el test suite TestStackAr para obtener el mejor mutation score posible con PiTest
   - ¿Cuál es el mejor mutation score que pudo obtener?
   - ¿Cuántos mutantes equivalentes encontró?