

# CS 5154: Software Testing

## Syntax-Based Coverage and Mutation Testing

Instructor: Owolabi Legunsen

Fall 2021

# Recall the four software models in this course

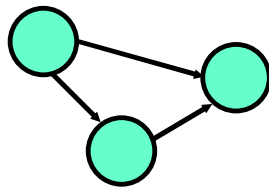


Input  
Domains

```
A: {0, 1, >1}  
B: {600, 700, 800}  
C: {cs, ece, is, sds}
```



Graphs



Logic  
Expressions

```
(!x | !y) & a & b
```



Syntax

```
if (x > y)  
    z = x - y;  
else  
    z = 2 * x;
```

## Conversation with a student after ISP HW

**Student:** I feel like I followed everything that you taught us, but I still don't know if I am doing it right...

**Owolabi:** How do you mean?

**Student:** Well, I don't know if the tests that I wrote are good enough and I don't know when to stop!

**Owolabi:** 😞

## Question for you...

- You have followed all the MDTD criteria that we taught you
- You have written test that satisfy “strong” coverage criteria
- But, how good are those tests for finding faults?

# CS5154 is organized into six themes

1. How to automate the execution of tests?
2. How to design and write high-quality tests?
3. How to measure the quality of tests?
4. How to automate the generation of tests?
5. How to reduce the costs of running existing tests?
6. How to deal with bugs that tests reveal?

What do we do in mutation testing?

Make small syntactic changes to source code and see if a test suite is strong enough to detect them

# Benefits of mutation testing

- Mutation testing provides a way to evaluate the quality of test suites
- Mutation testing also helps discover tests that should be added
- Mutation testing can also help to discover faults in programs

# Why learn about mutation testing?

- The “P” in the RIPR model
  - Check whether errors are propagating to the state that test oracles check
- There is a lot of tool support for mutation testing
- Mutation testing is gaining adoption in industry and in open source



# Companies are using mutation testing

2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice

## State of Mutation Testing at Google

Goran Petrović  
Google Inc.  
goranpetrovic@google.com

Marko Ivanković  
Google Inc.  
markoi@google.com

2018

## An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions

Goran Petrović Marko Ivanković  
Google Switzerland GmbH  
Zurich, Switzerland  
{goranpetrovic, markoi}@google.com

Bob Kurtz Paul Ammann  
George Mason University  
Fairfax, VA, USA  
{rkurtz2, pammann}@gmu.edu

René Just  
University of Massachusetts  
Amherst, MA, USA  
rjust@cs.umass.edu

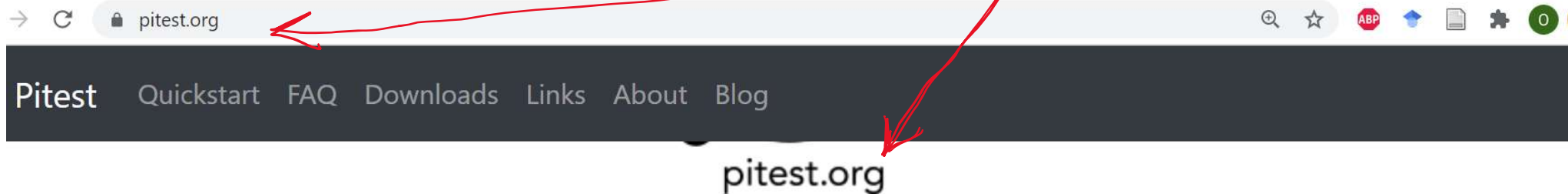
2019

- Articles should be [available for FREE @ Cornell Library](#)
- **Do NOT** pay to read these articles

# How we will learn mutation testing

- Today: see mutation testing in action
- Next: discuss mutation testing in more detail

## Demo on mutation testing



# 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

*Start here*

# Getting Started

[Pitest](#) [Quickstart](#) [FAQ](#) [Downloads](#) [Links](#) [About](#) [Blog](#)

## Quickstart

 [Improve this page](#)

Out of the box PIT can be launched from the command line, ant or maven. Third party components provide integration with Gradle, Eclipse, IntelliJ and others (see the [links](#) section for details).

The impatient can jump straight to the section for their chosen build tool - it may however be helpful to read the basic concepts section first.

### Getting started

[Maven quick start](#)

[Command line quick start](#)

We showed how to set up  
for maven

# Basic Concepts

Read the whole page

🔄 [pitest.org/quickstart/basic\\_concepts/](https://pitest.org/quickstart/basic_concepts/) ☆ ABP 📄 ⚙️ 0

Pitest Quickstart FAQ Downloads Links About Blog

## Basic Concepts

✎ Improve this page

### Mutation Operators

PIT applies a configurable set of **mutation operators** (or **mutators**) to the byte code generated by compiling your code.

For example the **CONDITIONALS\_BOUNDARY\_MUTATOR** would modify the byte code generated by the statement

```
if ( i >= 0 ) {  
    return "foo";  
} else {  
    return "bar";  
}
```

To be equivalent to

```
if ( i > 0 ) {  
    return "foo";  
} else {  
    return "bar";  
}
```

# Mutation Operators (or mutators)



## Available mutators and groups

Why groups?

The following table list available mutators and whether or not they are part of a group :

Mutators	"OLD_DEFAULTS" group	"DEFAULTS" group	"STRONGER" group	"ALL" group
<a href="#">Conditionals Boundary</a>	yes	yes	yes	yes
<a href="#">Increments</a>	yes	yes	yes	yes
<a href="#">Invert Negatives</a>	yes	yes	yes	yes

Be familiar with all default mutators

# What we covered in the demo

- GitLab pages setup for Jacoco and PIT
- Parsing the PIT Maven output
- Parsing the PIT reports
- Killed Mutants and Surviving Mutants
- Equivalent Mutants
- Benefits of mutation testing: strengthening test suites and finding faults
- Mutation Operators
- Saving costs by using groups with fewer mutators
- Higher-order mutants

# I want to read more about these concepts

- See “notes” links for mutation testing dates on the course webpage
- Those “notes” contain the relevant PIT web pages that we discussed in the demo
- Focus on the relevant parts, but you can learn a lot by reading them all
  - 40 pages, but lots of space in there



# Killing Mutants

**Given a mutant  $m \in M$  for a program  $P$  and a test  $t$ ,  $t$  is said to kill  $m$  if and only if the output of  $t$  on  $P$  is different from the output of  $t$  on  $m$ .**

- Testers can keep adding tests until all mutants are killed
- Or, the process of killing mutants can help developers to find faults

## Some types of mutants

- *Dead mutant* : A test case has killed it
- *Stillborn mutant* : Syntactically illegal
- *Trivial mutant* : Almost every test can kill it
- *Equivalent mutant* : No test can kill it (same behavior as original)
- *Higher-order mutant* : differs from original in more than one location

# Mutation operators

- Rules for making small syntactic changes to the original program
- Mutation testing: can the tests can detect those changes?
- Well-designed mutation operators yield very powerful tests
- Operators are designed for different prog. languages and goals

## One syntax-based coverage criteria

Mutation Coverage (MC) : For each  $m \in M$ , TR contains exactly one requirement, to kill  $m$ .

# Mutation testing and the RIPR model

- *Reachability* : Tests cause faulty (i.e., mutated) statements to be reached
- *Infection* : Tests cause faulty statement to result in an incorrect state
- *Propagation* : The incorrect state propagates to incorrect output
- *Revealability* : The oracles must observe part of the incorrect output

RIPR model yields **two variants** of mutation coverage

## Variant 1: Strongly Killing Mutants

Given a mutant  $m \in M$  for a program  $P$  and a test  $t$ ,  $t$  is said to *strongly kill*  $m$  if and only if the output of  $t$  on  $P$  is different from the output of  $t$  on  $m$

## Variant 2: Weakly Killing Mutants

Given a mutant  $m \in M$  that modifies a location  $l$  in a program  $P$ , and a test  $t$ ,  $t$  is said to *weakly kill*  $m$  if and only if the state of the execution of  $P$  on  $t$  is different from the state of the execution of  $m$  on  $t$  immediately after  $l$

Weakly killing satisfies **reachability** and **infection**, but not **propagation**



# Strong vs. Weak Mutant Killing

```
1 boolean isEven (int X){  
2   if (X < 0)  
3     X = 0 - X;  
Δ3   X = 0;  
4   if (double) (X/2) == ((double)  
5     return (true);  
6   else  
7     return  
8 }
```

Reachability :  $X < 0$

Infection :  $X \neq 0$

( $X = -6$ ) will kill mutant  $\Delta 3$  under weak mutation

Propagation :  $((\text{double}) ((0-X)/2) == ((\text{double}) 0-X) / 2.0)$

$\neq ((\text{double}) (0/2) == ((\text{double}) 0) / 2.0)$

That is,  $X$  is not even ...

Thus ( $X = -6$ ) does not kill the mutant under strong mutation

## Exercise: More on Weak mutant killing

```
int Min (int A, int B) {  
    int minVal;  
    minVal = A;  
    Δ1 minVal = B;  
    if (B < A){  
        minVal = B;  
    }  
    return (minVal);  
} // end Min
```

1. Find a test input that **weakly kills** the mutant, but not strongly
2. Generalize : What predicate **must be true** to weakly kill the mutant, but not strongly?
3. Write down the **conditions** needed to (i) **reach** the mutated statement, (ii) **infect** the program state, and (iii) **propagate** to the output

## More on Weak Mutation (contd)

1. Find a test that **weakly kills** the mutant, but not strongly

$A = 5, B = 3$

2. Generalize : What predicate **must be true** to weakly kill the mutant, but not strongly?

$B < A$  // minVal is set to B for both

3. RIP **conditions**

Reachability : *true* // we always reach

Infection :  $A \neq B$  // minVal has a different value

Propagation :  $(B < A) = false$  // Take a different branch

# Why does mutation testing work?

## Fundamental Premise of Mutation Testing

If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

- This is not an absolute! (note the “usually”)
- The mutants guide the tester to an effective set of tests
- Of course, this depends on the mutation operators ...

# Some notes on mutation operators

- At the **method level**, mutation operators for different programming languages are similar
- Mutation operators do one of two things :
  - Mimic typical programmer **mistakes** ( incorrect variable name )
  - Encourage common test **heuristics** ( cause expressions to be 0 )

# Exercise: Mutation Testing and Subsumption

- Mutation **subsumes** other (ISP, graph-based, and logic-based) criteria by including specific mutation operators
- See pages 251 to 255 in the textbook

# What we saw so far

- Mutation is widely considered the **strongest** test criterion
  - And most **expensive** !
  - By far the most test requirements (each mutant)
  - Usually, yields the most tests

# Next

1. How to automate the execution of tests?
2. How to design and write high-quality tests?
3. How to measure the quality of tests?
4. How to automate the generation of tests?
5. How to reduce the costs of running existing tests?
6. How to deal with bugs that tests reveal?