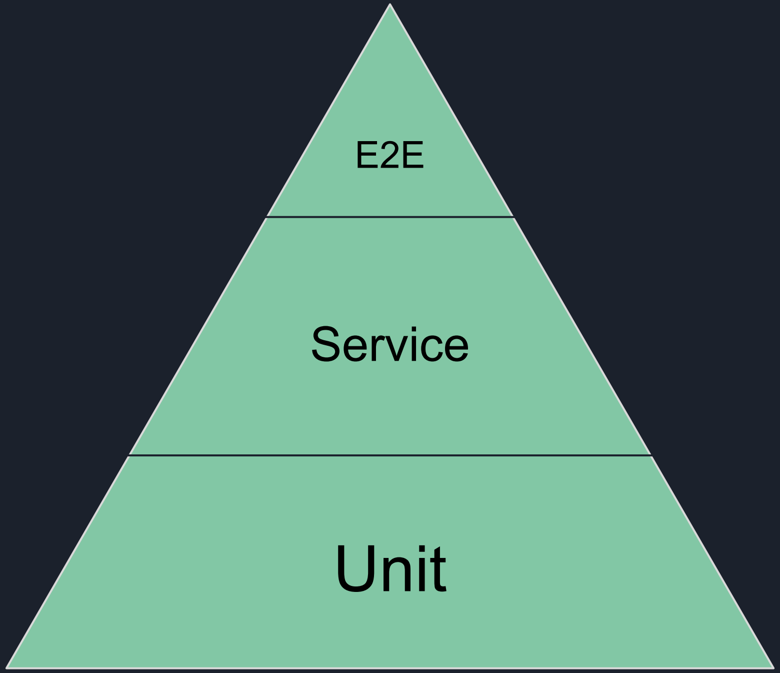
Mutation testing

Aleksandr Elmekeev

- Overview
- Terminology
- Problems
- Tools
- Summary

Overview

Test Pyramid



Test coverage

Criteria	JaCoCo	Istanbul
Function coverage	+	+
Statement coverage	+ (Instruction coverage)	+
Branch coverage	+	+
Modified condition/decision coverage	-	-
Linear Code Sequence and Jump (LCSAJ) coverage	-	-
Parameter value coverage	-	-

Quis custodiet ipsos custodes?

Who watches the watchmen?

Goals

- identify weakly tested pieces of code
- identify weak tests
- get rid of useless code / tests

Terminology

Mutation Operator (Mutator)

Type	Example: before	Example: after
Arithmetic	a + b	a - b
Array declaration	[1, 2, 3]	[]
Boolean	true	false
Conditional	for (var i = 0; i < 10; i++) { }	for (var i = 0; false; i++) { }
Equality	a < b	a <= b
Logical	a && b	a || b
Void	voidMethod();	// no voidMethod call

Mutant

By number of mutators:

- Simple (first order)
- Complex (high order)

By end state:

- Killed
- Timeout
- Error
- Survived / Escaped
- Equivalent

Mutant: RIP

- A test must **reach** the mutated statement.
- Test input data should **infect** the program state by causing different program states for the mutant and the original program.
- The incorrect program state must **propagate** to the program's output and be **revealed** by the test.

Algorithm

1. Run tests T against original program P.
2. Generate a set of mutants P’.
3. Run tests T against each mutant P’.

Mutation score = killed / total

Problems

High Computational Cost

- reduce number of mutants
  - **Mutant Sampling** — random subset of all mutants
  - **Selective Mutation** — certain types of mutators to generate mutants
  - **Mutant Clustering** — includes analysis of tests to identify subset
  - **Higher Order Mutation** — combines mutators (FOM × N = HOM) to make a single one with the same possibility to fail as a set of others
- optimize execution process
  - break the program by modules
  - Bytecode Translation technique
  - parallel runs
  - incremental analysis
  - etc

Problems Related To Human Effort

- equivalent mutant problem (e.g. doesn’t work well with Defensive Programming)
  - suggest (SEM)
  - detect (DEM)
  - avoid (AEMG)
- human oracle problem

Tools

Tools

- C# — Stryker.NET
- Java — Pitest, Descartes
- Javascript, Typescript — Stryker
- PHP — infection
- Python — mutmut
- Ruby — mutant
- Scala — Stryker4s
- LLVM (C, C++, Swift, Rust) — Mull

Java

PIT

- Test Frameworks:
  - JUnit (JUnit5 plugin)
  - TestNG
- Build Systems:
  - Ant
  - Maven (multi model support plugin)
  - Gradle plugin
- Other:
  - IntelliJ plugin
  - Sonarqube plugin
  - Extreme mutation testing (pitest-descartes mutation engine)

pitest.org

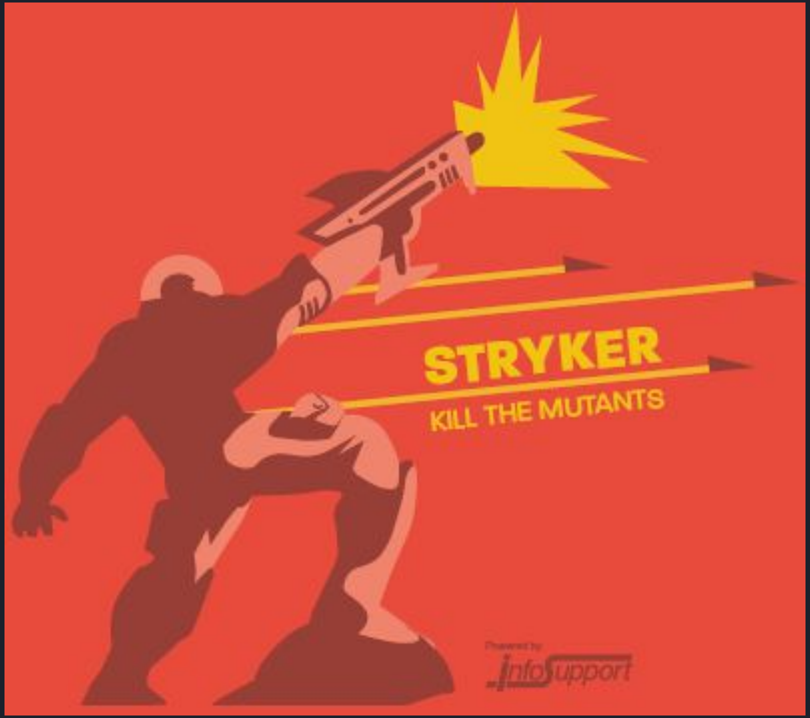
PIT: Configuration

- `targetClasses` — classes to be mutated
- `targetTests` — specifies list of tests to run
- `dependencyDistance` — allows to limit tests to run based on “distance” between it and mutated code
- `threads` — number of threads to use during mutation testing
- `mutators` — list of mutators to be applied
- `avoidCallsTo` — allows to avoid mutation of code that calls methods from particular classes / packages that we consider outside the scope of mutation testing
- `timeoutFactor` and `timeoutConst` — defines timed out mutants
- `outputFormats` — CSV / HTML / XML
- `historyInputLocation` and `historyOutputLocation` — incremental analysis config

Typescript / Javascript

Stryker

- Runners:
  - stryker-jest-runner
  - stryker-karma-runner
  - stryker-mocha-runner
  - stryker-wct-runner
- Reporters:
  - stryker-html-reporter
- Mutators:
  - stryker-javascript-mutator
  - stryker-typescript
  - stryker-vue-mutator



Stryker: Configuration

- `transpilers` — typescript / webpack / babel
- `mutate` — files to mutate
- `mutator` — mutator to use as long as mutator operators to exclude
- `maxConcurrentTestRunners` — maximum number of concurrent test runners to spawn
- `coverageAnalysis` — specifies coverage analysis strategy (off / all / perTest)
- `timeoutMS` & `timeoutFactor` — defines timed out mutants
- `reporters` — clear text, HTML

Summary

Usage

When to use:

1. critical parts of software;
2. project with continuous delivery;
3. if you want to validate quality of existing tests;
4. new tests to make sure the quality of them is good enough.

Be careful with:

1. file operations.

Useful Links

Read:

- Mutation testing on Wikipedia
- Mutation Testing Repository by Yue Jia and Mark Harman
- Analysis of Java Mutation Testing frameworks by PIT team and by scoban

Watch:

- Is Mutation Analysis Ready For Prime Time? by Jeff Offutt: part 1 and part 2
- Testing like it's 1971 by Henry Coles
- Mutation Analysis: What Code Coverage Doesn't Tell Us by Gleb Smirnov (slides)
- Mutate and Test Your Tests by Benoit Baudry
- Using Mutation Testing to improve your Javascript tests by Simon de Lang