Mutation Testing

Mutation Testing: Motivation

- Determine the power (i.e., sensitivity) of your test suite
- When weaknesses are found, identify ways in which your test suite can be improved
- “Who will test the tests?”

Terminology

- “Program mutation”: The act of changing the program intentionally to simulate small bugs.

- “Mutant”: A single version of a mutated program.

- “Killing a mutant”: At least one test case fails in a way that it does not on the un-mutated program.

- “Mutation operator”: One of a set of systematic recipes for mutating the program. Ideally, these are designed to simulate real bugs or to target particularly problematic types of bugs. Example: Replace a “<” in a `for`-loop predicate with a “<=”. (Another example would be to replace the upper limit, `i`, with `i-1`)

- “Mutation score”: (# of mutants killed) / (total # of mutants)

Typical Use

- The original program is tested to identify the set of test cases that pass (and possibly the set of failures with the particular way that they fail).

- Generally, thousands or millions of mutants are generated.

- Each mutant is run on the entire test suite.

- Each test case result (and possibly output) is compared with the original, un-mutated result for that test case.

- If any test case fails when it did not before, that mutant is “killed”.

Weak vs. Strong Mutation

- "PIE" model of failure: ***E***xecution, ***I***nfection, ***P***ropagation to output

- **Strong mutation** is the traditional way described thus far — all three conditions are needed.

- **Weak mutation** requires only execution and infection, but not necessarily propagation.

- "Equivalent mutant": Output of mutant is (guaranteed) always the same as the original program. In these cases, weak mutation is the strongest that can be expected.

Equivalent Mutants

Example:

```
int i = 2;
if ( i >= 1 ) {
    return "foo";
}
```

```
//...
int i = 2;
if ( i > 1 ) {
    return "foo";
}
```

Common cases are related to logging or debugging code.

Tools

- PIT
- muJava
- muClipse
- Jumble
- Javalanche
- JavaMut
- Jester
- …

Maven Quick Start

Improve this page

Installation

PIT is available from maven central since version 0.20.

Getting started

Add the plugin to build/plugins in your pom.xml

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
</plugin>
```

That's it, you're up and running.

By default pitest will mutate all code in your project. You can limit which code is mutated and which tests are run using `targetClasses` and `targetTests`. Be sure to read the globs section if you want to use exact class names.

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
    <configuration>
        <targetClasses>
            <param>com.your.package.root.want.to.mutate*</param>
        </targetClasses>
        <targetTests>
            <param>com.your.package.root*</param>
        </targetTests>
    </configuration>
</plugin>
```

If no `targetClasses` are provided in versions before 1.2.0, pitest assumes that your classes live in a package matching your projects group id. In 1.2.0 and later versions pitest will scan your project to determine which classes are present.