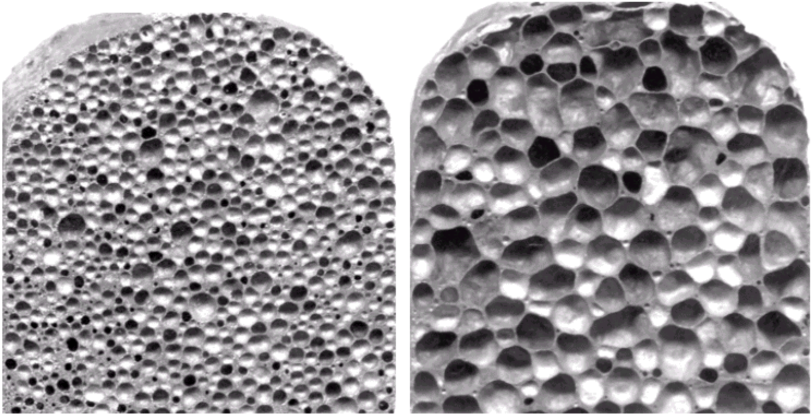
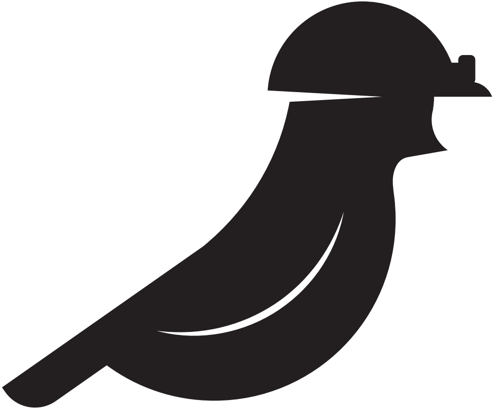
Mutation testing

a practitioners perspective

Hello



2000



pitest.org

Grown from a codebase doing something else starting in 2009

One of a handful of tools to be used in real teams

One of a handful of tools to be used in real teams

Apparently also now popular in academia

Developed without reference to academic papers

Main external input was an existing open source tool

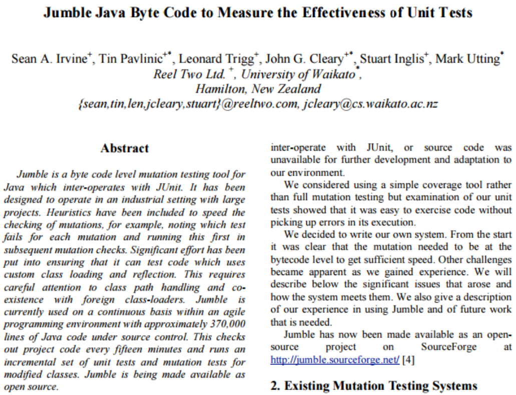
Jumble

My big “innovation”

My big “innovation”

Using coverage data to target tests against mutations

Actually first proposed by Irvine et al



Jumble Java Byte Code to Measure the Effectiveness of Unit Tests

Sean A. Irvine[+], Tin Pavlinic[+*], Leonard Trigg[+], John G. Cleary[+*], Stuart Inglis[+], Mark Utting[*]
Reel Two Ltd. [+], *University of Waikato*[*],
Hamilton, New Zealand
{sean,tin,len,jcleary,stuart}@reeltwo.com, jcleary@cs.waikato.ac.nz

Abstract

Jumble is a byte code level mutation testing tool for Java which inter-operates with JUnit. It has been designed to operate in an industrial setting with large projects. Heuristics have been included to speed the checking of mutations, for example, noting which test fails for each mutation and running this first in subsequent mutation checks. Significant effort has been put into ensuring that it can test code which uses custom class loading and reflection. This requires careful attention to class path handling and co-existence with foreign class-loaders. Jumble is currently used on a continuous basis within an agile programming environment with approximately 370,000 lines of Java code under source control. This checks out project code every fifteen minutes and runs an incremental set of unit tests and mutation tests for modified classes. Jumble is being made available as open source.

inter-operate with JUnit, or source code was unavailable for further development and adaptation to our environment.

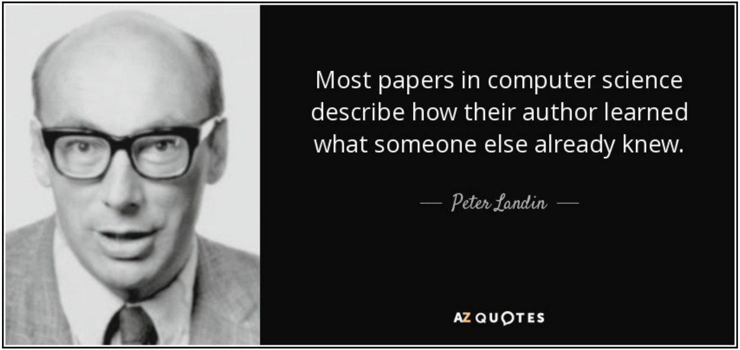
We considered using a simple coverage tool rather than full mutation testing but examination of our unit tests showed that it was easy to exercise code without picking up errors in its execution.

We decided to write our own system. From the start it was clear that the mutation needed to be at the bytecode level to get sufficient speed. Other challenges became apparent as we gained experience. We will describe below the significant issues that arose and how the system meets them. We also give a description of our experience in using Jumble and of future work that is needed.

Jumble has now been made available as an open-source project on SourceForge at http://jumble.sourceforge.net/ [4]

2. Existing Mutation Testing Systems

And first implemented in Javalanche about 2 years before pitest



Most papers in computer science describe how their author learned what someone else already knew.

— Peter Landin —

Programming and writing computer science papers have more in common than it first appears

Henry Coles

What's in this talk?

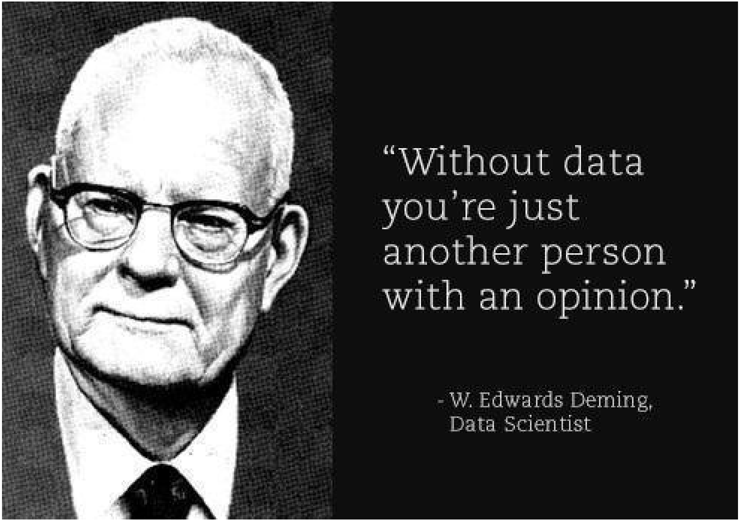
2. Look at some little discussed **implementation tradeoffs**

1. Look at what mutation testing is actually useful for

1. **Look at what mutation testing is actually useful for**

If you are an industry programmer

(and while doing this look at why people are using pitest instead of javalanche)

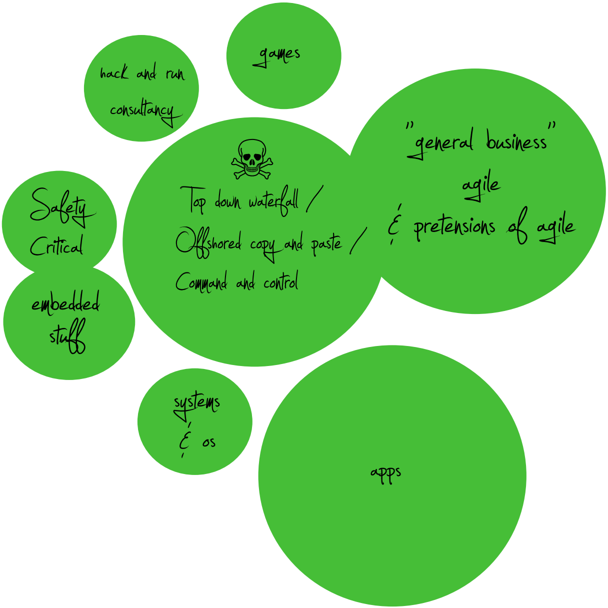


"Without data you're just another person with an opinion."

- W. Edwards Deming,
Data Scientist

What do programmers want?

What do I mean by programmers?



hack and run consultancy

games

☠

Top down waterfall /
Offshored copy and paste /
Command and control

"general business"
agile
& pretensions of agile

Safety Critical

embedded stuff

systems & os

apps

Programmers want Feedback

Useful feedback

Useful feedback

Actionable

Useful feedback

Actionable

Repeatable

Useful feedback

Actionable

Repeatable

Easy to collect

Useful feedback

Actionable

Repeatable

Easy to collect

Timely

Actionable

Actionable

There is a clear thing I can do

Actionable

There is a clear thing I can do

Actions for higher order mutants?

Repeatable

Repeatable

I can re-run and see if I fixed it

Repeatable

I can re-run and see if I fixed it

Random sampling?

Easy to collect

Easy to collect

We are lazy with short attention spans

Easy to collect

We are lazy with short attention spans

We resist inconvenience and extra work

Easy to collect

We are lazy with short attention spans

We resist inconvenience and extra work

So no speed bumps

Easy to collect

We are lazy with short attention spans

We resist inconvenience and extra work

So no speed bumps

(no matter how small)

- “**Manually edit the . . .**”

- “Manually edit the . . .”
- “Just download the modified version of Java”

- “Manually edit the . . .”
- “Just download the modified version of Java”
- “Launch the swing GUI and . . .”

- "Manually edit the . . ."
- "Just download the modified version of Java"
- "Launch the swing GUI and . . ."
- "It works but doesn't support . . ."

- "Manually edit the . . ."
- "Just download the modified version of Java"
- "Launch the swing GUI and . . ."
- "It works but doesn't support . . ."
- "Enable some of AOIS, JID, JSI, EOC, COR, ROR COI . . ."

Needs to run from existing build tool

Needs to run from existing build tool

This has not meant Ant since 2007

Needs to run from existing build tool

This has not meant Ant since 2007

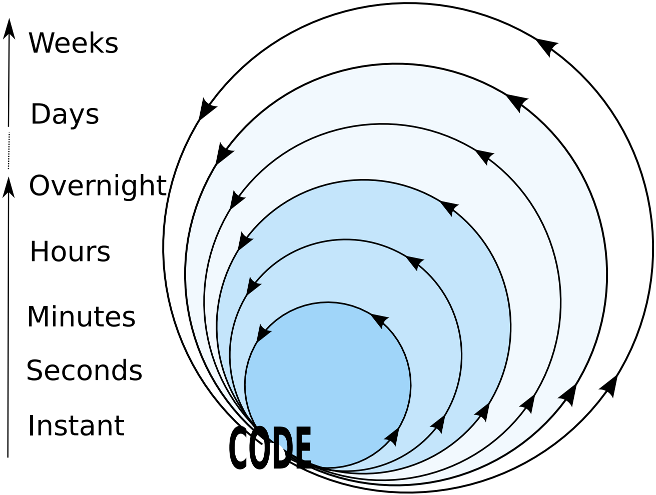
IDE integration is nice, but build tool is essential

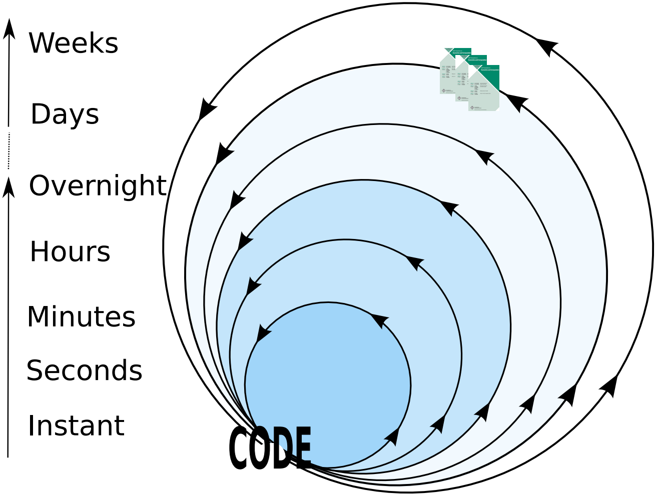
Timely

When do we want this feedback?

When do we want this feedback?

As *early* as possible


Weeks
Days
Overnight
Hours
Minutes
Seconds
Instant
CODE



Weeks

Days

Overnight

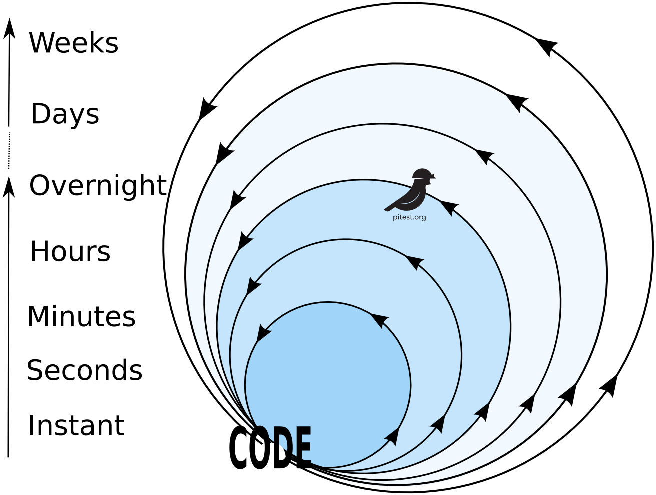
Hours

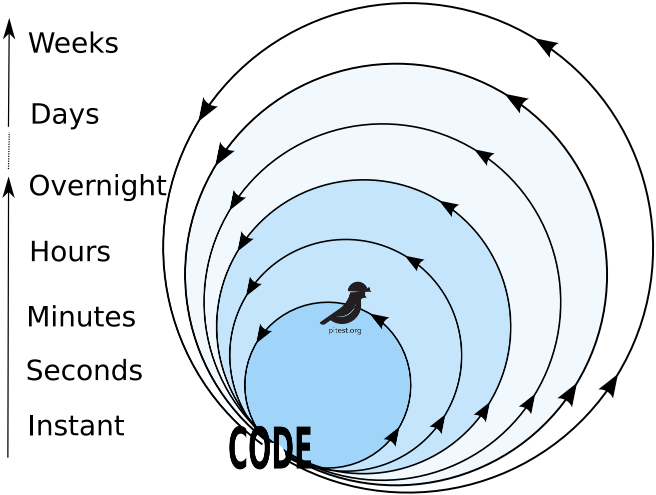
Minutes

Seconds

Instant

CODE


Weeks
Days
Overnight
Hours
Minutes
Seconds
Instant
CODE
pitest.org


Weeks
Days
Overnight
Hours
Minutes
Seconds
Instant
CODE
pitest.org

Mutation testing is most useful for developers as they write the code

Assessing for equivalence takes less time

More likely to take action

The 3 minute build

How much mutation feedback can you collect in 3 minutes?

How much mutation feedback can you collect in 3 minutes?

Quite a lot

Can analyse many real projects in 3 minutes

When you can't just mutate the slice that's changed

What actions does it prompt?

Highlights missing test cases

Highlights missing test cases

Not a surprise

Highlights missing test cases

Not a surprise

Add a test

Highlights weak/buggy tests

Highlights **weak/buggy** tests

Not a surprise

Highlights weak/buggy tests

Not a surprise

Fix a test

Highlights code needing “closer inspection”

?

Equivalent mutants can also provide useful feedback

Actions for equivalent mutants

Is the code neccessary?

Is the code **neccessary**?

Delete the code

Is this code for performance optimisation?

Is this code for performance optimisation?

Is it worth it?

Does the same class of equivalent mutant appear multiple times?

Does the same class of equivalent mutant appear multiple times?

Is there duplication that can be removed?

Can I re-express the code and not have the mutant?

Can I re-express the code and not have the mutant?

Does the code look 'cleaner' now?

Can I re-express the code and not have the mutant?

Does the code look 'cleaner' now?

(this one is much more subjective than others)

Trivial made up example

```
public static int doStuff(int a, int b) {
  int c = 0;
  if ( a == 2 && b == 2) {
    c = a * b;
  }
  return c;
}
```

Trivial made up example

```java
public static int doStuff(int a, int b) {
  int c = 0;
  if ( a == 2 && b == 2) {
    c = a + b; // <--- mutated
  }
  return c;
}
```

Trivial made up example

```
public static int doStuff(int a, int b) {
  int c = 0;
  if ( a == 2 && b == 2) {
    c = 4; // better?
  }
  return c;
}
```

Real example from google truth

Real example from google truth

A small assertion library

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; // Unequal-lengthed arrays are not equal.
    }
    List<Integer> unequalIndices = new ArrayList<Integer>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
     // failWithRawMessage(
     //   "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; // Unequal-lengthed arrays are not equal.
    }
    List<Integer> unequalIndices = new ArrayList<Integer>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (areEqual(actual,expected,tolerance)) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
private boolean areEqual(double[] actual, double[] expected, double tolerance) {
  if (actual == expected) return true;

  if (expected.length != actual.length) return false;

  return compareArrayContents(actual, expected, tolerance);
}
private boolean compareArrayContents(double[] actual, double[] expected,
    double tolerance) {
  List<Integer> unequalIndices = new ArrayList<Integer>();
  for (int i = 0; i < expected.length; i++) {
    if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
      unequalIndices.add(i);
    }
  }
  return unequalIndices.isEmpty();
}
```

```
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (areEqual(actual,expected,tolerance)) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
private boolean areEqual(double[] actual, double[] expected, double tolerance) {
  if (false) return true; // <--------- mutated

  if (expected.length != actual.length) return false;

  return compareArrayContents(actual, expected, tolerance);
}
private boolean compareArrayContents(double[] actual, double[] expected,
    double tolerance) {
  List<Integer> unequalIndices = new ArrayList<Integer>();
  for (int i = 0; i < expected.length; i++) {
    if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
      unequalIndices.add(i);
    }
  }
  return unequalIndices.isEmpty();
}
```

Summary

Summary

A tool for industry must be **low friction**

- Runs from our build tools (maven and gradle)

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps
- Gives useful feedback in 3 minutes or less

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps
- Gives useful feedback in 3 minutes or less
- (which will probably mean only analysing the slice we're working on)

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps
- Gives useful feedback in 3 minutes or less
- (which will probably mean only analysing the slice we're working on)
- Doesn't make us change anything we already do

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps
- Gives useful feedback in 3 minutes or less
- (which will probably mean only analysing the slice we're working on)
- Doesn't make us change anything we already do
- Has visible support channels

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps
- Gives useful feedback in 3 minutes or less
- (which will probably mean only analysing the slice we're working on)
- Doesn't make us change anything we already do
- Has visible support channels

- Runs from our build tools (maven and gradle)
- Needs no/minimal setup
- Needs no manual steps
- Gives useful feedback in 3 minutes or less
- (which will probably mean only analysing the slice we're working on)
- Doesn't make us change anything we already do
- Has visible support channels

Javalanche did get some of this right

2. Implementation tradeoffs

Mutant isolation

Mutant isolation

Mutants can poison their environment

Mutant isolation

Mutants can poison their environment

(which for Java means the jvm)

Side effects caused by mutants

Side effects caused by mutants

- Bad state in a static variable

Side effects caused by mutants

- Bad state in a static variable
- Exhausted memory

Side effects caused by mutants

- Bad state in a static variable
- Exhausted memory
- Unexpected classes loaded

Side effects caused by mutants

- Bad state in a static variable
- Exhausted memory
- Unexpected classes loaded
- Others?

Correctness vs performance

Correctness vs performance

We need to isolate mutants from each other, but that has a cost

Strategies to isolate mutants

Strategies to isolate mutants

Often inadvertently selected by tool authors due to method of insertion

1. Don’t

1. Don't

Results may not be correct

1. Don’t

Results may not be correct

Default strategy for mutant schemata & instrumentation api

2. Launch a JVM for each mutant

2. Launch a JVM for each mutant

Most robust approach

2. Launch a JVM for each mutant

Most robust approach

Takes about 1 second to start a jvm

2. Launch a JVM for each mutant

Most robust approach

Takes about 1 second to start a jvm

Can take much longer to load classes

2. Launch a JVM for each mutant

Most robust approach

Takes about 1 second to start a jvm

Can take much longer to load classes

Default strategy if you manipulate source and compile to disk

3. Use classloaders

3. Use classloaders

Still need to load classes multiple times

3. Use classloaders

Still need to load classes multiple times

Breaks things

3. Use classloaders

Still need to load classes multiple times

Breaks things

(pitest used to do this)

Current pitest approach

Current pitest approach

Group mutants and launch jvm per group

Current pitest approach

Group mutants and launch jvm per group

By default 1 group per class

Current pitest approach

Group mutants and launch jvm per group

By default 1 group per class

But can set mutationUnitSize=1

Current pitest approach

Group mutants and launch jvm per group

By default 1 group per class

But can set mutationUnitSize=1

(also tries to detect low memory)

Alternate approach

Alternate approach

(only considers poisoning via static state)

Run a static analysis of which mutants might corrupt/be corrupted by a static variable

Run a static analysis of which mutants might corrupt/be corrupted by a static variable

Group them separately from each other

Faster (2x speedup)

Faster (2x speedup)

More correct?

Faster (2x speedup)

More correct?

Harder to understand

The mutant approach

Markus Schirp

Before inserting each mutant insert a no-op mutant

Before inserting each mutant insert a no-op mutant

Run the tests

Before inserting each mutant insert a no-op mutant

Run the tests

If a test fails create a fresh environment

Can be fine tuned between correctness and speed

Can be fine tuned between correctness and speed

(only insert no-op x% of the time)

Can be fine tuned between correctness and speed

(only insert no-op x% of the time)

(only run x% of the tests against the no-op mutant)

Can be combined with grouping based strategies

Test targeting

Test targeting

How tests are targeted has a large impact on analysis time

Most common approaches are

Most common approaches are

- No targeting (run whole suite)

Most common approaches are

- No targeting (run whole suite)
- Naming convention

Most common approaches are

- No targeting (run whole suite)
- Naming convention
- Coverage targeting

Naming convention

Naming convention

Assume the class Foo is tested by FooTest

Naming convention

Assume the class Foo is tested by FooTest

Unfortunately not all tests are written this way

Naming convention

Assume the class Foo is tested by FooTest

Unfortunately not all tests are written this way

This was one of the main issues with using Jumble in practice

Coverage targeting

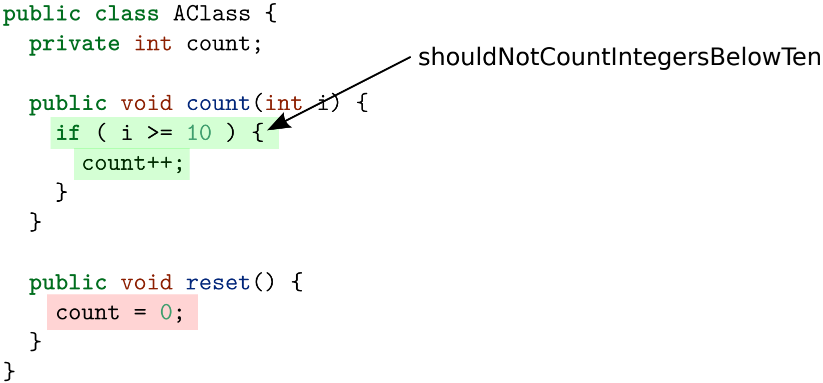
Coverage targeting

As used by Javalanche and Pitest

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

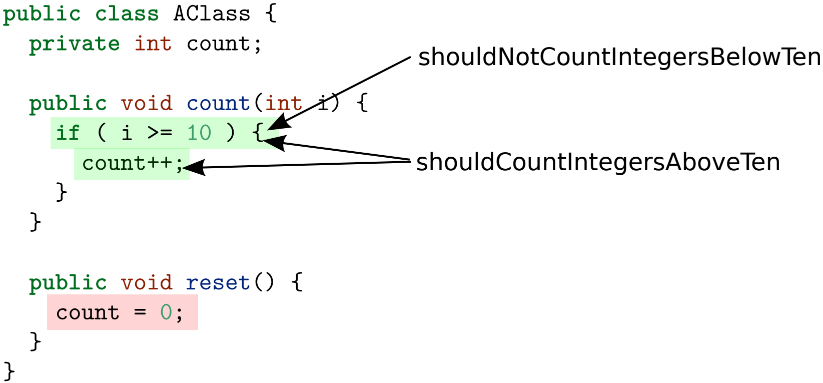


```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

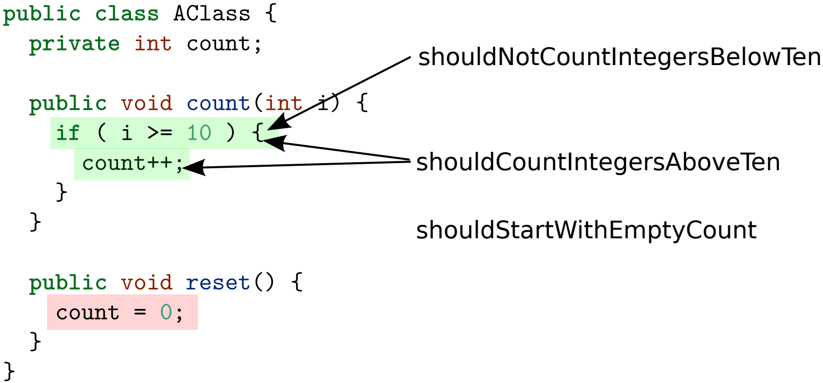
shouldNotCountIntegersBelowTen



```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```



```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

shouldNotCountIntegersBelowTen

shouldCountIntegersAboveTen

shouldStartWithEmptyCount

```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i > 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

```
 1  public class AClass {
 2    private int count;
 3
 4    public void count(int i) {
 5      if ( i > 10 ) {
 6        count++;
 7      }
 8    }
 9
10    public void reset() {
11      count = 0;
12    }
13  }
```

- We will only run 2 tests for the mutation on line 5

```
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i > 10 ) {
6        count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- We will only run 2 tests for the mutation on line 5
- The mutation will survive as we're missing an effective test case

```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
     //count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

```
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6       //count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- We will run only 1 test for the mutation on line 6

```java
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6       //count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- We will run only 1 test for the mutation on line 6
- The mutation will be **killed**

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 1;
  }
}
```

```
 1  public class AClass {
 2    private int count;
 3
 4    public void count(int i) {
 5      if ( i >= 10 ) {
 6        count++;
 7      }
 8    }
 9
10    public void reset() {
11      count = 1;
12    }
13  }
```

- **We will run no tests for the mutation on line 11**

```
 1  public class AClass {
 2    private int count;
 3
 4    public void count(int i) {
 5      if ( i >= 10 ) {
 6        count++;
 7      }
 8    }
 9
10    public void reset() {
11      count = 1;
12    }
13  }
```

- We will run **no tests** for the mutation on line 11
- The mutation will be instantly marked as **survived**

Much faster than other approaches

Much faster than other approaches

Targeting can be as precise as per statement

Much faster than other approaches

Targeting can be as precise as per statement

Makes no assumptions about how tests are written

We are however trading off some correctness

We are however trading off some correctness

```
public class LateNinetiesCode {
  private static LateNinetiesCode instance
     = new LateNinetiesCode();

  private LateNinetiesCode() {
    lots of mutable code in here
  }
}
```

We are however trading off some correctness

```
public class LateNinetiesCode {
  private static LateNinetiesCode instance
     = new LateNinetiesCode();

  private LateNinetiesCode() {
    lots of mutable code in here
  }
}
```

Code that sets up static state will be executed by the first test to run

We are however trading off some correctness

```
public class LateNinetiesCode {
  private static LateNinetiesCode instance
     = new LateNinetiesCode();

  private LateNinetiesCode() {
    lots of mutable code in here
  }
}
```

Code that sets up static state will be executed by the first test to run

And only that test

We could fix this by

We could fix this by

- Starting a JVM or classloader for each test

We could fix this by

- Starting a JVM or classloader for each test
- Performing a static analysis to identify static construction code

We could fix this by

- Starting a JVM or classloader for each test
- Performing a static analysis to identify static construction code

We could fix this by

- Starting a JVM or classloader for each test
- Performing a static analysis to identify static construction code

Pitest does option 2, but it is limited

Questions?

Questions?

Or a 5 minutes bonus topic?

Bonus topic

Bonus topic

A possibly stupid different use for mutation

Mutation testing measures test strength

Mutation testing measures test strength

Does a test fail when something changes?

Half of the art of test automation is
making the test code
sensitive to things you care about and
insensitive to things you don't care about

Dale Emery

Test suites are meant to enable refactoring

Test suites are meant to enable refactoring

Many real ones prevent refactoring by being tied to implementation detail

Reverse mutation testing?

Reverse mutation testing?

Do the tests still pass when the implementation changes but behaviour remain the same?

Reverse mutation testing?

Do the tests still pass when the implementation changes but behaviour remain the same?

i.e are the tests tied to implementation detail?

Reverse mutation testing?

Do the tests still pass when the implementation changes but behaviour remain the same?

i.e are the tests tied to implementation detail?

Is this what the Parasoft Insure++ tool did?

Problems

Problems

A technique slower than mutation testing

Problems

A technique slower than mutation testing

(can't stop when a test fails)

Operators for reverse mutation testing

The obvious ones wouldn't add much benefit over static analysis

The obvious ones wouldn't add much benefit over static analysis

Rename a private method

The obvious ones wouldn't add much benefit over static analysis

- Rename a private method
- Rename a field

The obvious ones wouldn't add much benefit over static analysis

Rename a private method

Rename a field

Switch a collection for a compatible one

Are there better ones?

Questions?