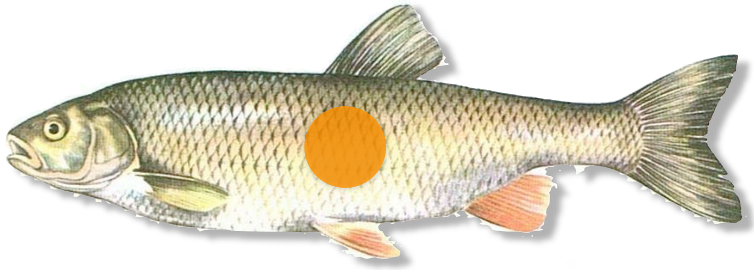
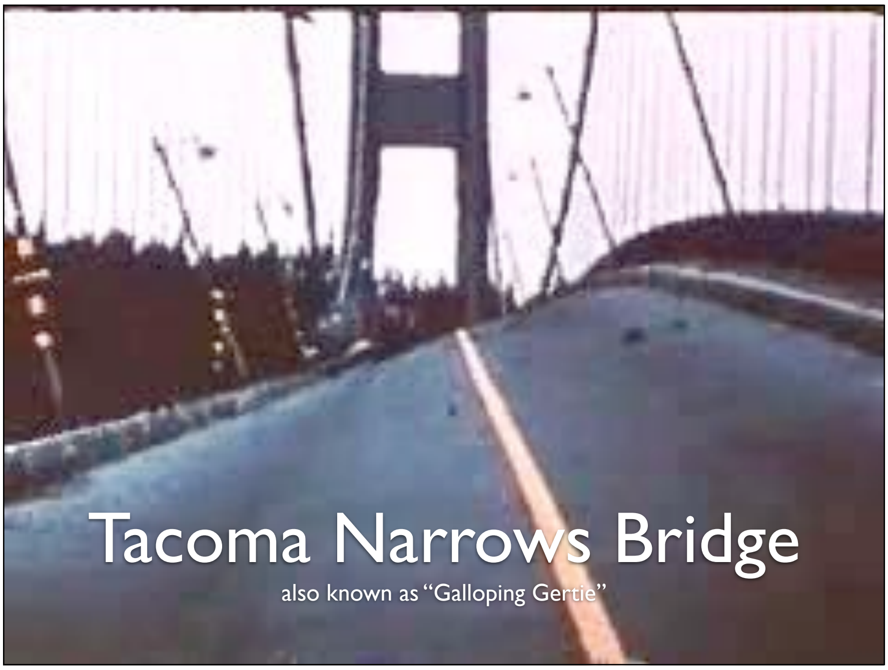
Mutation Testing

Andreas Zeller



1



2

The Tacoma Narrows Bridge is a pair of mile-long (1600 meter) suspension bridges with main spans of 2800 feet (850 m), they carry Washington State Route 16 across the Tacoma Narrows of Puget Sound between Tacoma and the Kitsap Peninsula, USA. The first bridge, nicknamed Galloping Gertie, was opened to traffic on July 1, 1940, and became famous four months later for a dramatic wind-induced structural collapse that was caught on color motion picture film.

Learning from Mistakes

- Key idea: Learning from earlier mistakes to prevent them from happening again
- Key technique: *Simulate earlier mistakes* and see whether the resulting defects are found
- Known as *fault-based testing* or *mutation testing*

3

A blue lobster (one in two million), an example of a genuine mutant. Blue American lobster (Homarus americanus). Taken at the New England Aquarium (Boston, MA, December 2006. Copyright © 2006 Steven G. Johnson and donated to Wikipedia under GFDL and CC-by-SA.

4

Seeding Defects

- We *seed defects* into the program
  generating a *mutant* – a *mutation* of the original program
- We run the test suite to see whether it detects the defects ("kills the mutants")
- A mutant not killed indicates a weakness of the test suite
  The mutant may also be 100% equivalent to the original program

5

6

7

8

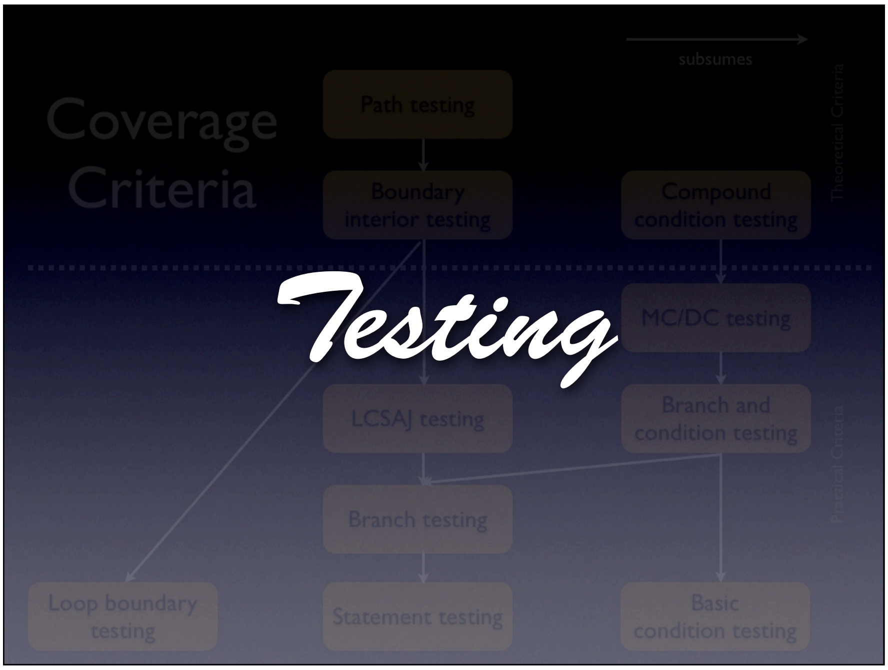
Hans-Peter is moving into this building – actually, he built it, too. He's worried that everything might be okay. But he's not that worried.

9

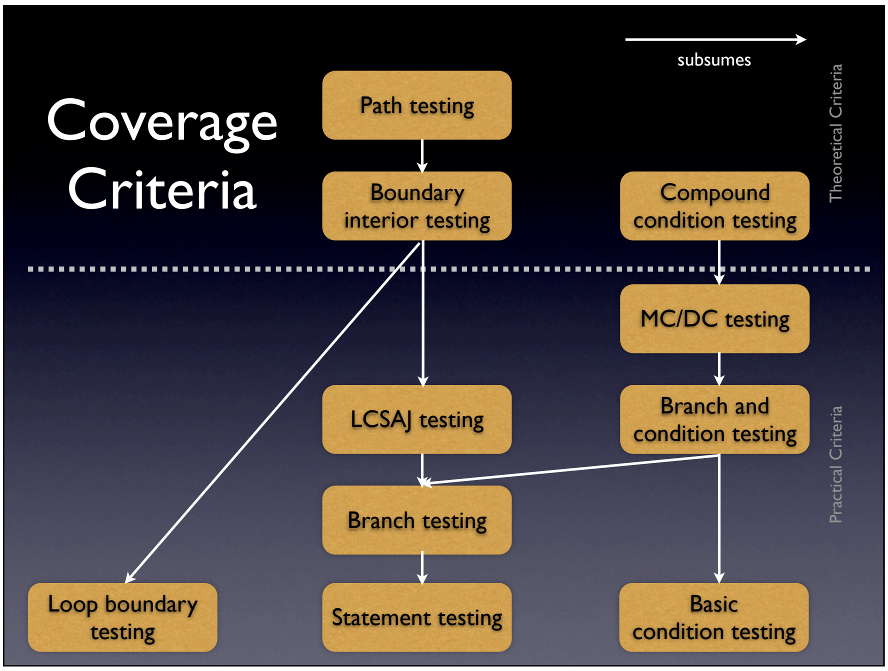
If you're building not a building, but a piece of software, you have many more reasons to be worried.

10

It’s not like this is the ultimate horror…



11



12

And this is the summary of structural testing techniques.

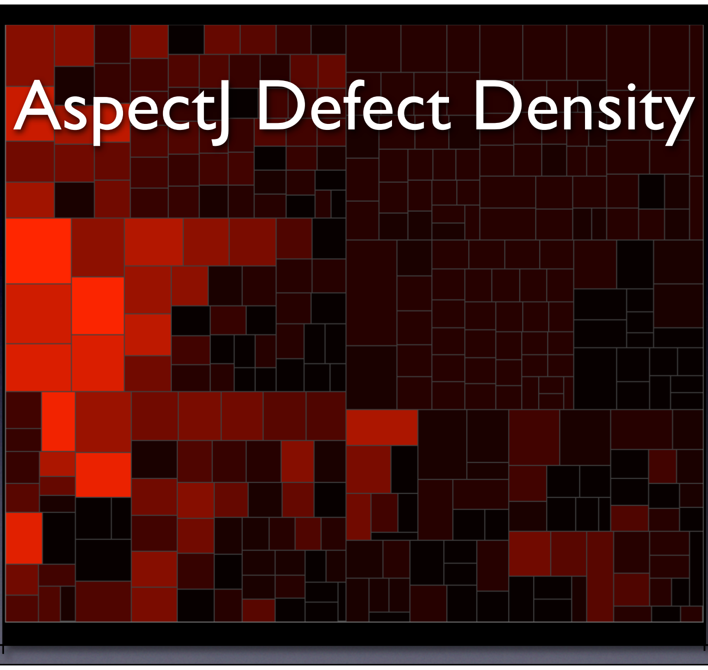
Weyuker's Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.

Established by a number of studies done by E. Weyuker at AT&T. "Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case".

13

AspectJ Defect Density



14

A Bad Test

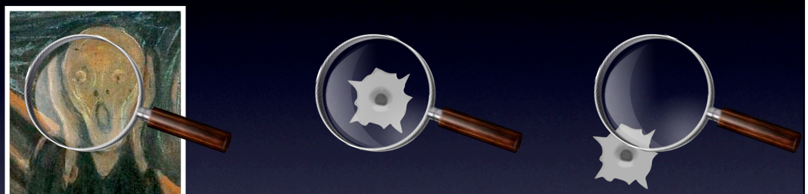
```
class TrueStoryTest {
    public int test_all(Object other)
    {
        executeForSomeTime();
        assertTrue(true);
    }
}
```

100% coverage – and never fails

15

Mutation Testing

DeMillo, Lipton, Sayward 1978



Program

16

A Mutation

```
class Checker {
    public int compareTo(Object other)
    {
        return 1;
    }
}
```

not found by AspectJ test suite

17

Mutation Operators

id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by `abs(e)`	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Pezzé and Young, Software Testing and Analysis

from Pezze + Young, “Software Testing and Analysis”, Chapter 16

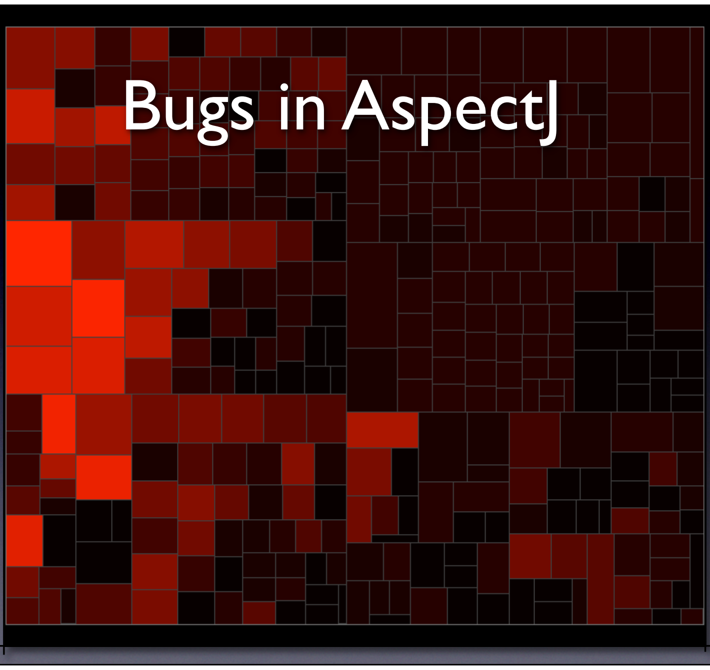
If one ever needed a proof that testing is a destructive process – here it is

18

Does it work?

- Generated mutants are similar to real faults
  Andrews, Briand, Labiche, ICSE 2005
- Mutation testing is more powerful than statement or branch coverage
  Walsh, PhD thesis, State University of NY at Binghampton, 1985
- Mutation testing is superior to data flow coverage criteria
  Frankl, Weiss, Hu, Journal of Systems and Software, 1997

Bugs in AspectJ

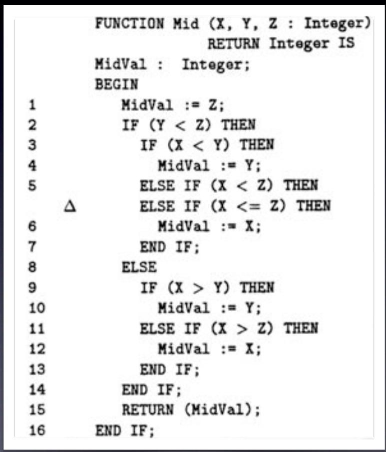


Issues

Efficiency

Inspection

Mid



```
FUNCTION Mid (X, Y, Z : Integer)
                RETURN Integer IS
MidVal :  Integer;
BEGIN
    MidVal := Z;
    IF (Y < Z) THEN
      IF (X < Y) THEN
        MidVal := Y;
      ELSE IF (X < Z) THEN
Δ     ELSE IF (X <= Z) THEN
        MidVal := X;
      END IF;
    ELSE
      IF (X > Y) THEN
        MidVal := Y;
      ELSE IF (X > Z) THEN
        MidVal := X;
      END IF;
    END IF;
    RETURN (MidVal);
END IF;
```

Offut and Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths", 1997

Efficiency

- Test suite must be re-run for every single mutation
- Expensive

Efficiency

How do we make mutation testing efficient?

Efficiency

- Manipulate byte code directly
  rather than recompiling every single mutant
- Focus on few mutation operators
  - replace numerical constant *C* by *C*±1, or 0
  - negate branch condition
  - replace arithmetic operator (+ by –, * by /, etc.)
- Use mutant schemata
  individual mutants are guarded by run-time conditions
- Use coverage data
  only run those tests that actually execute mutated code

25

A Mutation

```
class Checker {
    public int compareTo(Object other)
    {
        return 1;
    }
}
```

not found by AspectJ test suite
because it is not executed

– and since we know it's not executed, we don't even apply this mutation.

26

Efficiency

- 6.5 CPU hours for Jaxen
  XPath engine with 12,500 LOC
- Mutation testing is feasible in practice

27

Inspection

- A mutation may leave program semantics unchanged
- These *equivalent mutants* must be determined manually
- This task is tedious.

28

An Equivalent Mutant

```
public int compareTo(Object other) {
  if (!(other instanceof BcelAdvice))
    return 0;
  BcelAdvice o = (BcelAdvice)other;

  if (kind.getPrecedence() != o.kind.getPrecedence()) {
    if (kind.getPrecedence() > o.kind.getPrecedence())
      return +2;
    else
      return -1;
  }
  // More comparisons...
}
```

no impact on AspectJ

29

To check this, we need to look at 50+ places!

Inspection is Costly

- In a Jaxen sample, 40% of non-detected mutants were equivalent
  - Assessing a single mutation took us 30 minutes
  - 1,933 mutations were not detected
- This ratio *grows* as the test suite improves
  and approaches 100% with a perfect test suite
- Such false positives are just *worthless*
  Using coverage, false positives at least imply dead

1,000 hours, or 10 weeks for a Microsoft programmer.

30

Frankl's Observation

We also observed that […]
mutation testing was *costly*.
Even for these small subject programs,
the human effort needed to check a large
number of mutants for equivalence
was *almost prohibitive*.

P. G. Frankl, S. N. Weiss, and C. Hu.
All-uses versus mutation testing:
An experimental comparison of effectiveness.
Journal of Systems and Software, 38:235–253, 1997.

31

Inspection

How do we determine equivalent mutants?

32

Aiming for Impact

33

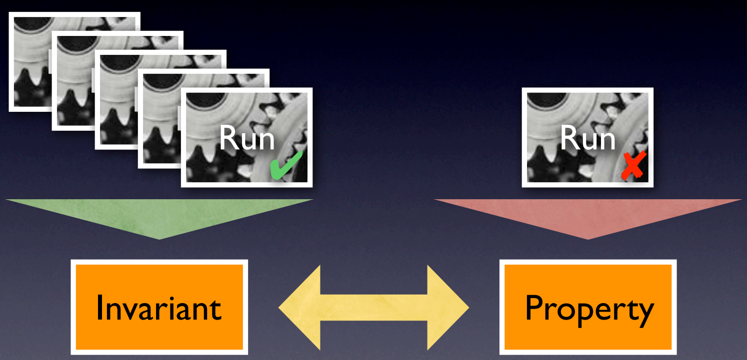
Measuring Impact

- How do we characterize "impact" on program execution?
- Idea: Look for changes in *pre- and postconditions*
- Use *dynamic invariants* to learn these

34

Dynamic Invariants

pioneered by Mike Ernst's Daikon



At f(), x is odd

At f(), x = 2

35

Example

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

Precondition

```
n == size(b[])
b != null
n <= 13
n >= 7
```

Postcondition

```
b[] = orig(b[])
return == sum(b)
```

- Run with 100 randomly generated arrays of length 7–13

36

Obtaining Invariants

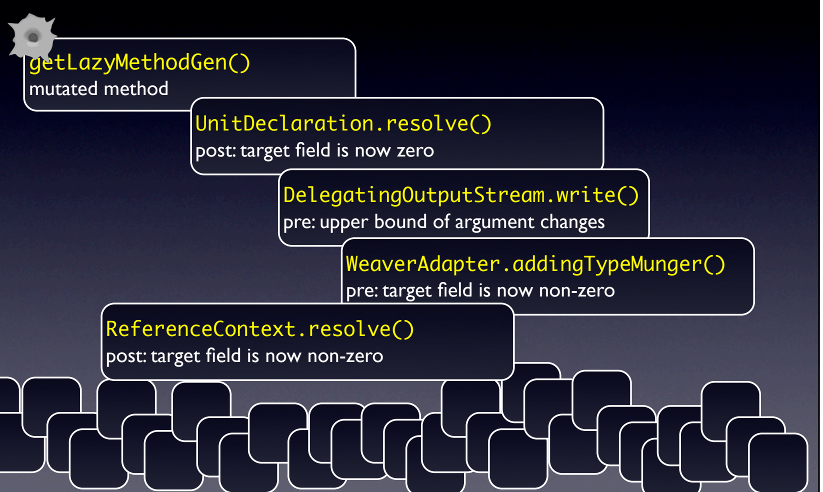
37

Impact on Invariants

```
public LazyMethodGen getLazyMethodGen(String name,
    String signature, boolean allowMissing) {
  for (Iterator i = methodGens.iterator(); i.hasNext();) {
    LazyMethodGen gen = (LazyMethodGen) i.next();
    if (gen.getName().equals(name) &&
      ! gen.getSignature().equals(signature))
      return gen;
  }
  if (!allowMissing)
    throw new BCException("Class " + this.getName() +
        " does not have a method " + name +
        " with signature " + signature);
  return null;
}
```

38

Impact on Invariants



39

Impact on Invariants

```
public LazyMethodGen getLazyMethodGen(String name,
    String signature, boolean allowMissing) {
  for (Iterator i = methodGens.iterator(); i.hasNext();) {
    LazyMethodGen gen = (LazyMethodGen) i.next();
    if (gen.getName().equals(name) &&
        gen.getSignature().equals(signature))
      return gen;
  }
  if (!allowMissing)
    throw new BCException("Class " + this.getName() +
        " does not have a method " + name +
        " with signature " + signature);
  return null;
}
```

impacts 39 invariants in 18 methods
but undetected by AspectJ unit tests

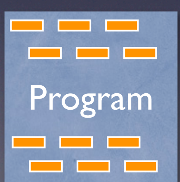
40

- Mutation Testing Framework for Java
  12 man-months of implementation effort
- Efficient Mutation Testing
  Manipulate byte code directly • Focus on few mutation operators • Use mutant schemata • Use coverage data
- Ranks Mutations by Impact
  Checks impact on dynamic invariants • Uses efficient invariant learner and checker

41

Mutation Testing

with Javalanche

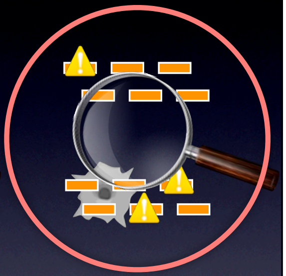


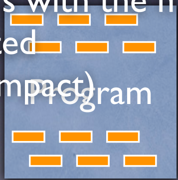
42

Mutation Testing

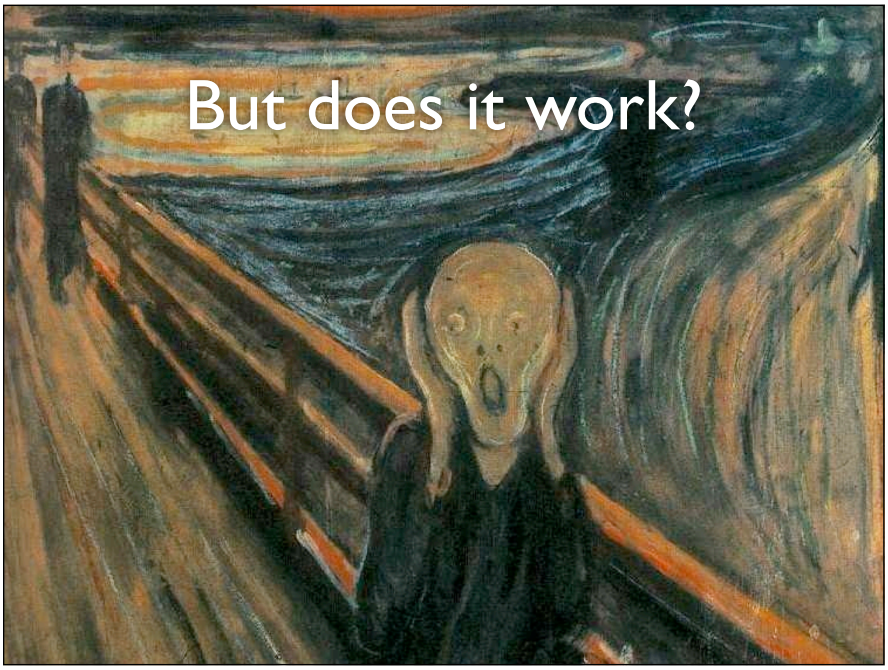
with Javalanche

1. Learn invariants from test suite
2. Insert invariant checkers into code
3. Detect impact of mutations
4. Select mutations with the most invariants violated
   (= the highest impact)





43



44

Evaluation

1. Are mutations with impact *less likely to be equivalent?*
2. Are mutations with impact *more likely to be detected?*
3. Are mutants with the *highest* impact *most likely* to be detected?

45

Evaluation Subjects

Name	Lines of Code	#Tests
AspectJ Core AOP extension to Java	94.902	321
Barbecue Bar Code Reader	4.837	137
Commons Helper Utilities	18.782	1.590
Jaxen XPath Engine	12.449	680
Joda-Time Date and Time Library	25.861	3.447
JTopas Parser tools	2.031	128
XStream XML Object Serialization	14.480	838

Mutations

Name	#Mutations	%detected
AspectJ Core	47.146	53
Barbecue	17.178	67
Commons	15.125	83
Jaxen	6.712	61
Joda-Time	13.859	79
JTopas	1.533	72
XStream	5.186	92

% detected means covered mutations

Performance

- Mutation testing is feasible in practice
  14 CPU hours for AspectJ, 6 CPU hours for XStream
- Learning invariants is very expensive
  22 CPU hours for AspectJ – one-time effort
- Creating checkers is somewhat expensive
  10 CPU hours for AspectJ – one-time effort

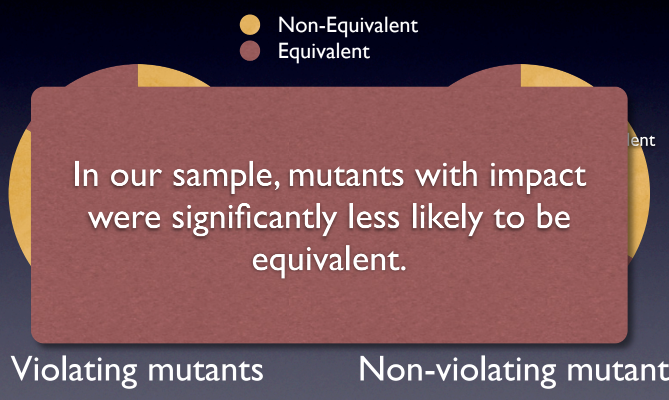
49

Are mutations with impact *less likely to be equivalent?*

- Randomly selected non-detected *Jaxen* mutants – 12 violating, 12 non-violating
- Manual inspection: *Are mutations equivalent?*
- Mutation was proven non-equivalent iff we could create a detecting test case
- Assessment took 30 minutes per mutation

50

Are mutations with impact *less likely to be equivalent?*



Difference is statistically significant according to Fisher test
Mutations and tests made public to counter researcher bias

51

Are mutations with impact *more likely to be detected?*

1. Mutations detected by the test suite are *non-equivalent.*
2. The more of my mutations are detected, the fewer equivalent mutations I have generated.

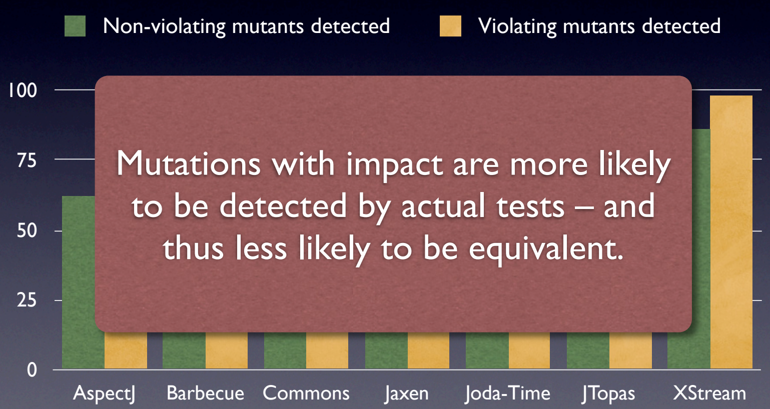
52

Are mutations with impact *more likely to be detected?*

$$\frac{\begin{array}{c}\text{detected by test} \Rightarrow \text{non-equivalent} \\ \text{has impact} \overset{?}{\Rightarrow} \text{detected by test}\end{array}}{\text{has impact} \overset{!}{\Rightarrow} \text{non-equivalent}}$$

53

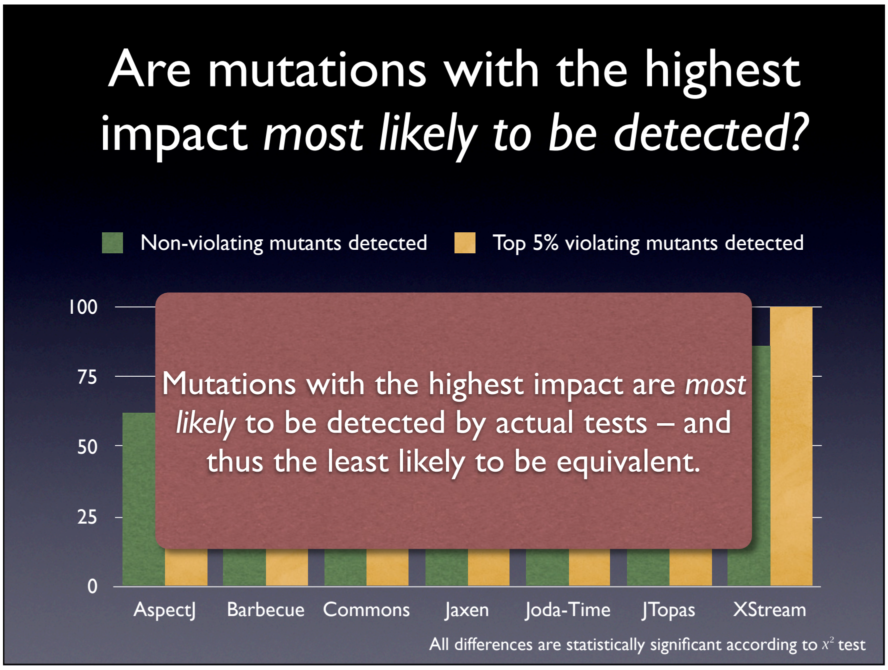
Are mutations with impact *more likely to be detected?*



Mutations with impact are more likely to be detected by actual tests – and thus less likely to be equivalent.

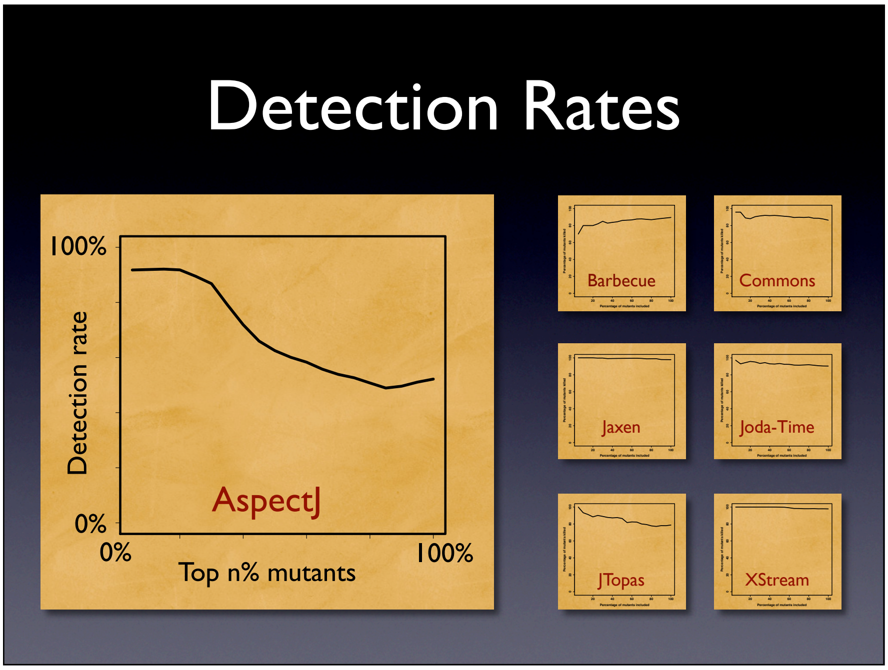
All differences are statistically significant according to χ^2 test

54



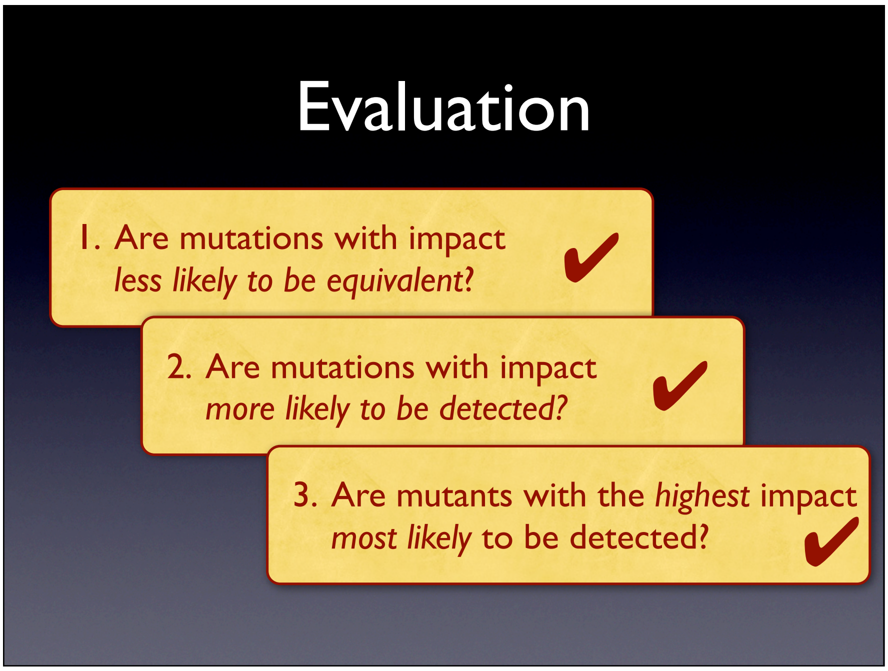
Are mutations with the highest impact *most likely to be detected?*
Non-violating mutants detected
Top 5% violating mutants detected
100
75
50
25
0
AspectJ
Barbecue
Commons
Jaxen
Joda-Time
JTopas
XStream
Mutations with the highest impact are *most likely* to be detected by actual tests – and thus the least likely to be equivalent.
All differences are statistically significant according to x^2 test

55



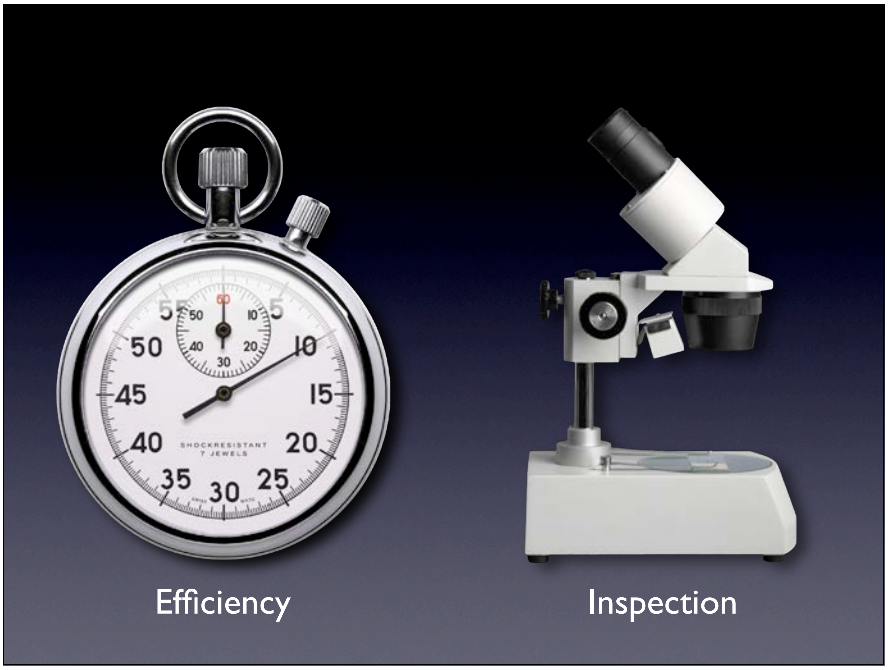
Detection Rates
100%
Detection rate
0%
0%
Top n% mutants
100%
AspectJ
Barbecue
Commons
Jaxen
Joda-Time
JTopas
XStream

56

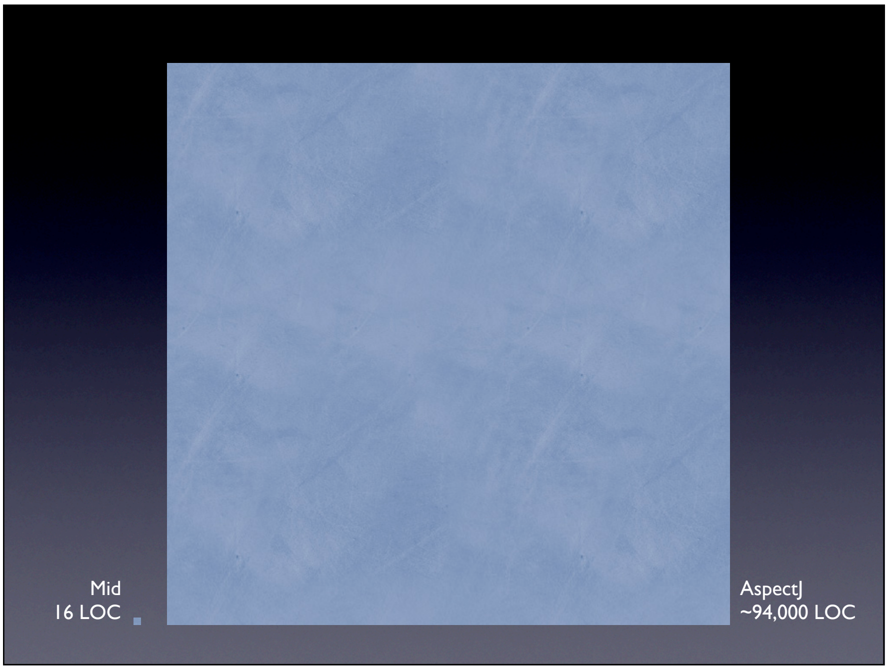


Evaluation
1. Are mutations with impact *less likely to be equivalent?* ✔
2. Are mutations with impact *more likely to be detected?* ✔
3. Are mutants with the *highest* impact *most likely* to be detected? ✔

57



58



Factor 6,666 – plus full automation due to lack of inspection

59

Future Work

- How effective is mutation testing?
  on a large scale – compared to traditional coverage
- Alternative impact measures
  Coverage • Program spectra • Method sequences
- Adaptive mutation testing
  Evolve mutations to have the fittest survive
- Automatic fixes
  Choose fixes (mutations) with minimal impact

60

Estimating #Defects

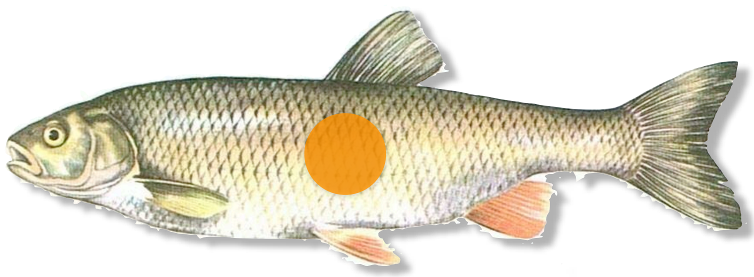
- How many defects remain in our software?
- With mutation testing, we can make an *estimate* of remaining defects

61

Let's consider a lake. How many fish are in that lake?

62

Fish Tag



- We catch 1,000 fish and tag them

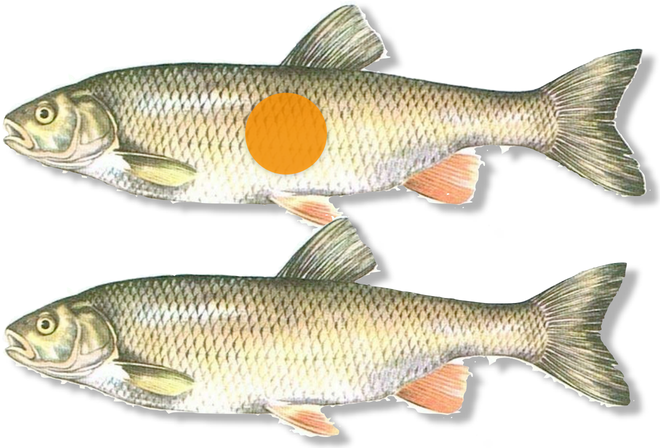
Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

63

Counting Tags

50

300



Let's assume over the next week, we ask fishermen to count the number of tags. We find 300 untagged and 50 tagged fish.

64

Estimate

$$\frac{1{,}000}{\text{untagged fish population}} = \frac{50}{300}$$

…and we can thus estimate that there are about 6,000 remaining untagged fish in the lake.

65

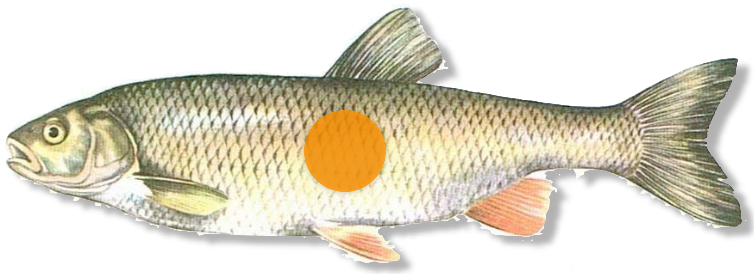
That's how we can tell how many fish there are.

66

Now let's assume our lake is not a lake, but our program.

67

A Mutant



- We seed 1,000 mutations into the program

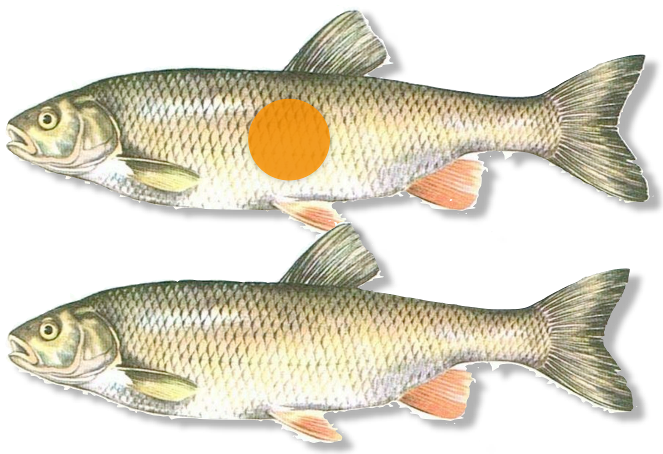
Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

68

Counting Mutants

50

300



Our test suite finds 50 mutants, and 300 natural faults.

69

Estimate

$$\frac{1{,}000}{\text{remaining defects}} = \frac{50}{300}$$

…and we can again estimate that there are about 6,000 remaining defects in our program. (A test suite finding only 50 out of 1,000 mutations is a real bad sign.)

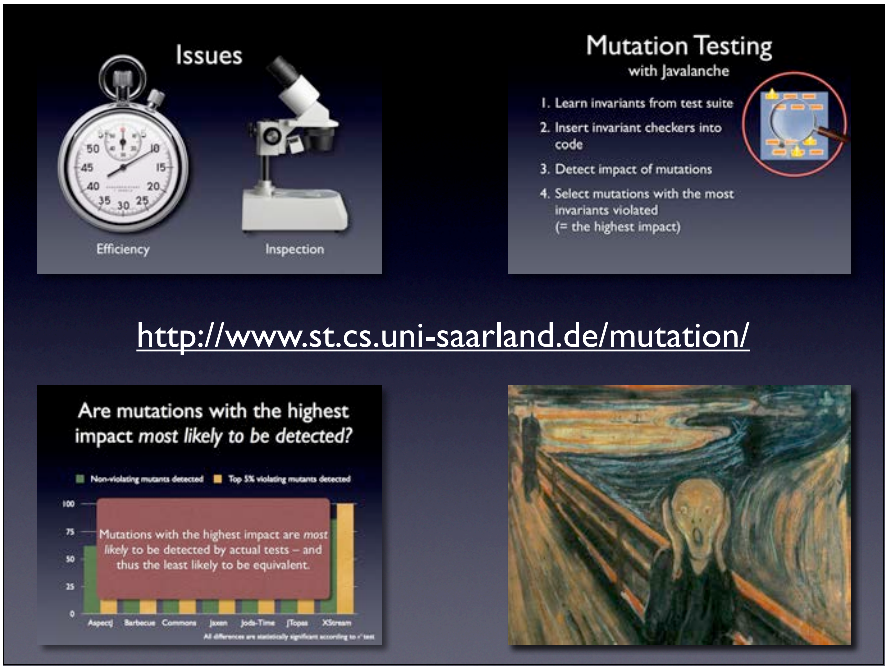
70

71

Assumptions

- Mutations are representatives for earlier mistakes
  so-called *competent programmer hypothesis*
- Failures come to be because of a combination of minor mistakes
  but there may be *logical errors* that cross-cut the program
- These hypotheses are not proven

72


Issues
Efficiency
Inspection
Mutation Testing
with Javalanche
1. Learn invariants from test suite
2. Insert invariant checkers into code
3. Detect impact of mutations
4. Select mutations with the most invariants violated (= the highest impact)
http://www.st.cs.uni-saarland.de/mutation/
Are mutations with the highest impact most likely to be detected?
Non-violating mutants detected
Top 5% violating mutants detected
Mutations with the highest impact are most likely to be detected by actual tests – and thus the least likely to be equivalent.
Aspectj
Barbecue
Commons
Jaxen
Joda-Time
JTopas
XStream

73