Software Engineering

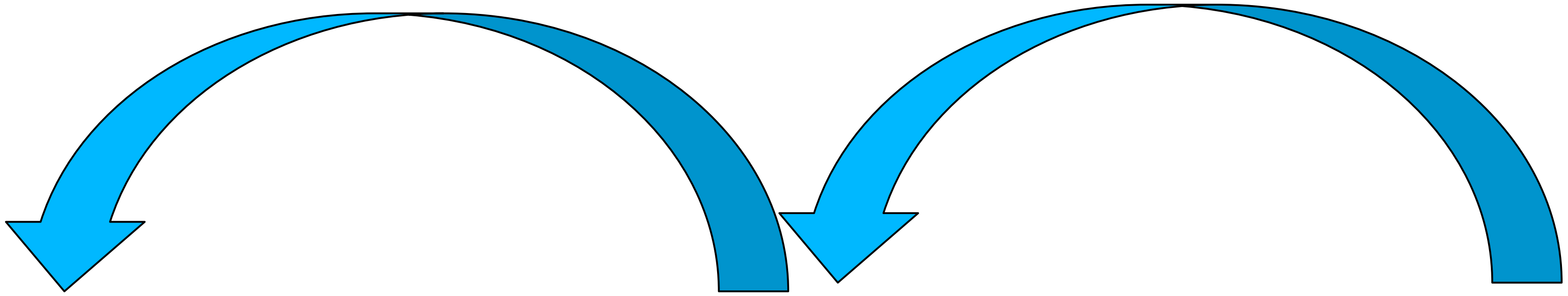
Lecture-40

Mutation Testing

Topics Covered

- Mutation Testing
- Goals
- Testing Method
- Mutation Process
- Traditional Syntactical Mutation Operators

What is Mutation Testing?



Verify Quality of…

Verify Quality of…

Source Code

Tests

Mutation Testing

Introduction

Who Watches The Watchmen?

In this case: What Tests The Tests?

Mutation Testing is a method of inserting faults into programs to test whether the tests pick them up, thereby validating or invalidating the tests

History of Mutation

Can trace birth of Mutation Testing back to a student paper written in 1971, by Lipton

More interest in the late 70s (DeMillo *et al.*)

Died down due to problems of cost

Being researched again recently due to availability of much higher computing power

Is most recently being used on non-imperative languages such as Java and XML

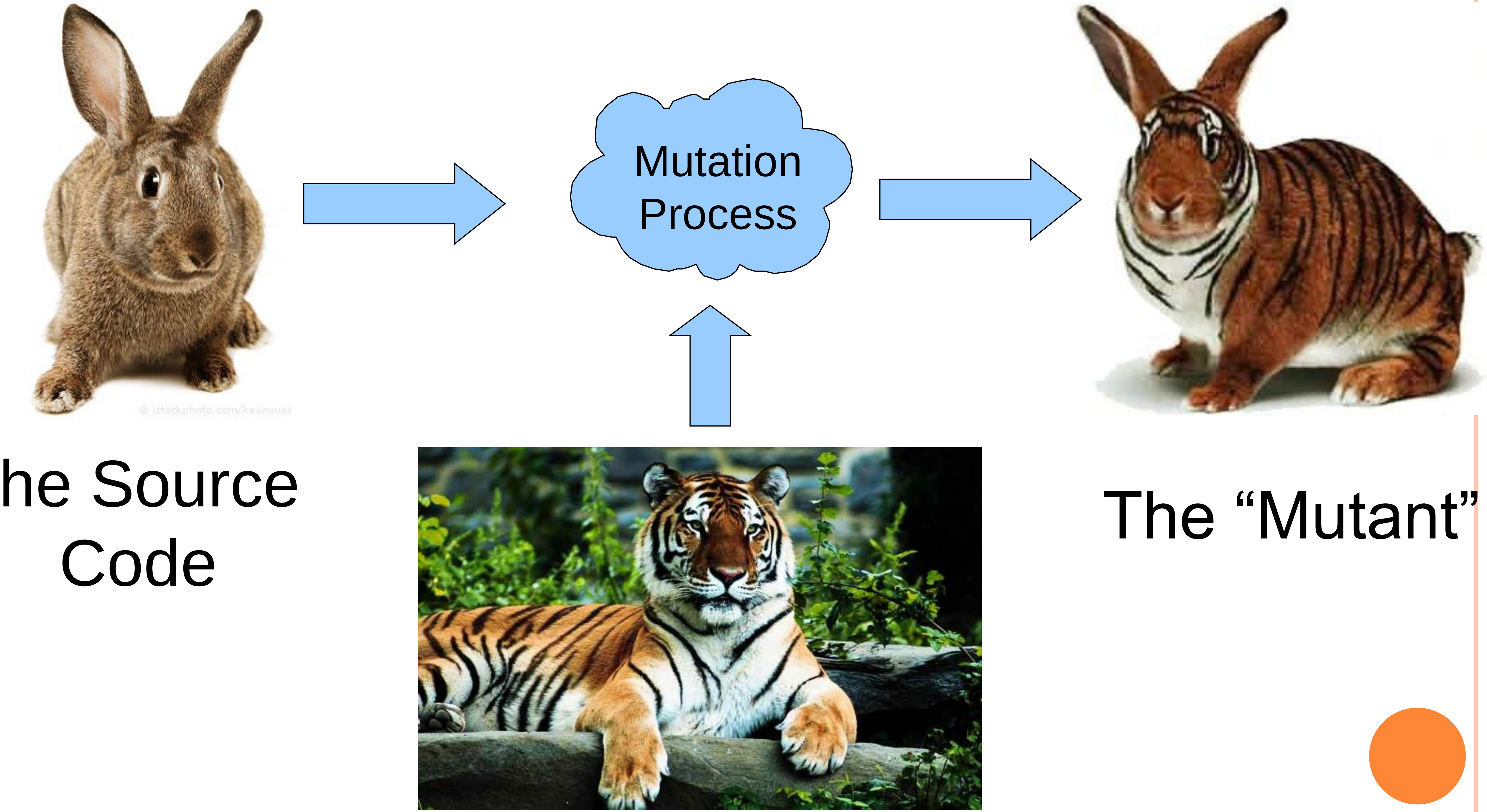
GOALS

To assess the quality of the tests by performing them on mutated code

To use these assessments to help construct more adequate tests

To thereby produce a suite of valid tests which can be used on real programs

How does it work?
1st Step: Create the Mutant



Mutation Process

The Source Code

The “Mutant”

The Mutation “Operator”

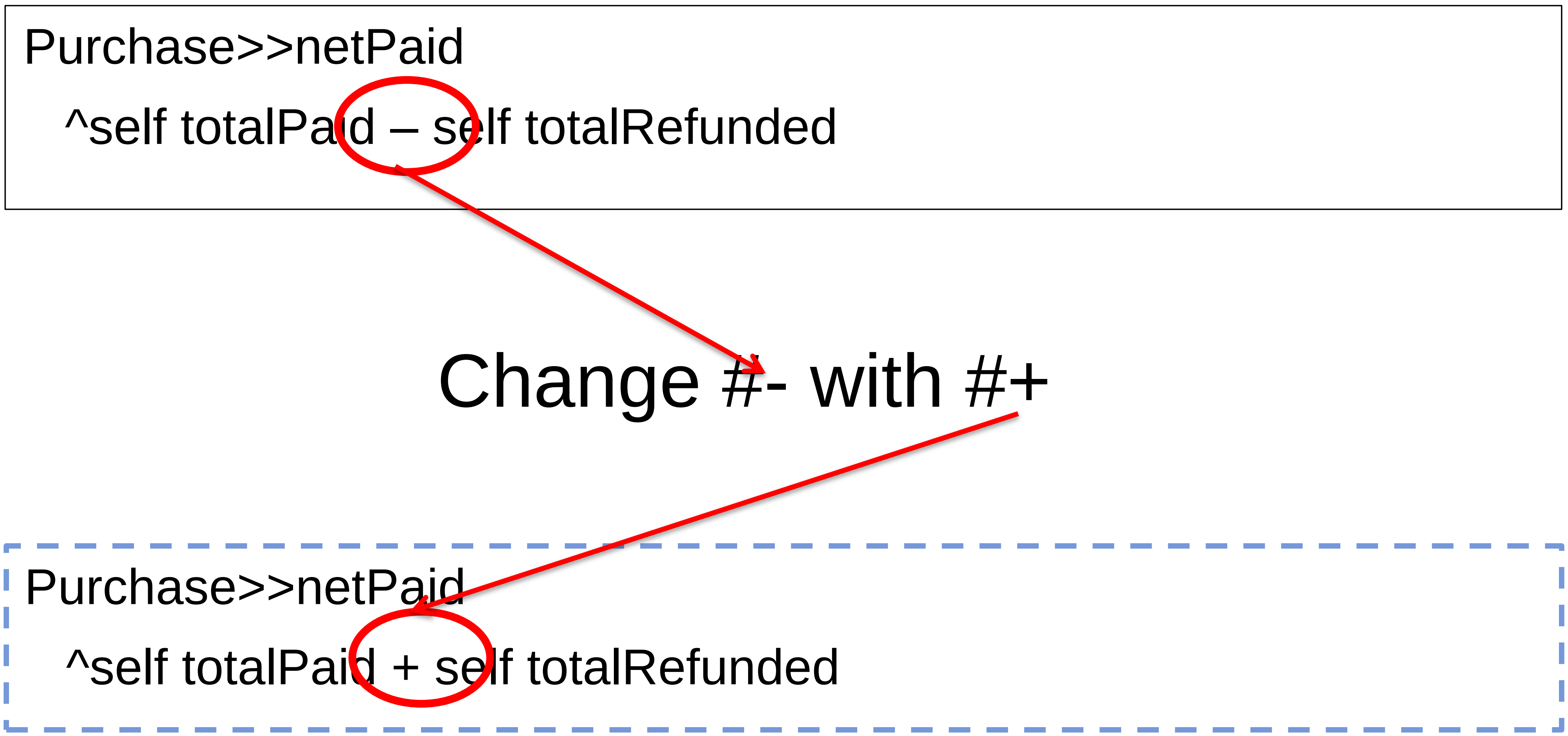
Examples

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
    and: [ number = anotherDebitCard number ]
```

Operator: Change #and: by #or:

```
CreditCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
    or: [ number = anotherDebitCard number ]
```

Examples



```
Purchase>>netPaid
   ^self totalPaid – self totalRefunded
```

Change #- with #+

```
Purchase>>netPaid
   ^self totalPaid + self totalRefunded
```

Why?
How does it help?

How does it work?
2nd Step: Try to Kill the Mutant

The "Mutant"

A Killer
tries to kill the Mutant!

The Test Suite

All tests run → The Mutant Survives!!!

A test fails or errors → The Mutant Dies

MEANING…

The Mutant Survives → The case generated by the mutant is not tested

The Mutant Dies → The case generated by the mutant is tested

Testing Method

Mutant processes are created to try to mimic typical syntactic errors made by programmers

Many differing mutants are run against the specified tests to assess the quality of the tests

The tests are attributed with a score as to whether they can distinguish between the original and the mutants

Traditional Syntactical Mutation Operators

Deletion of a statement

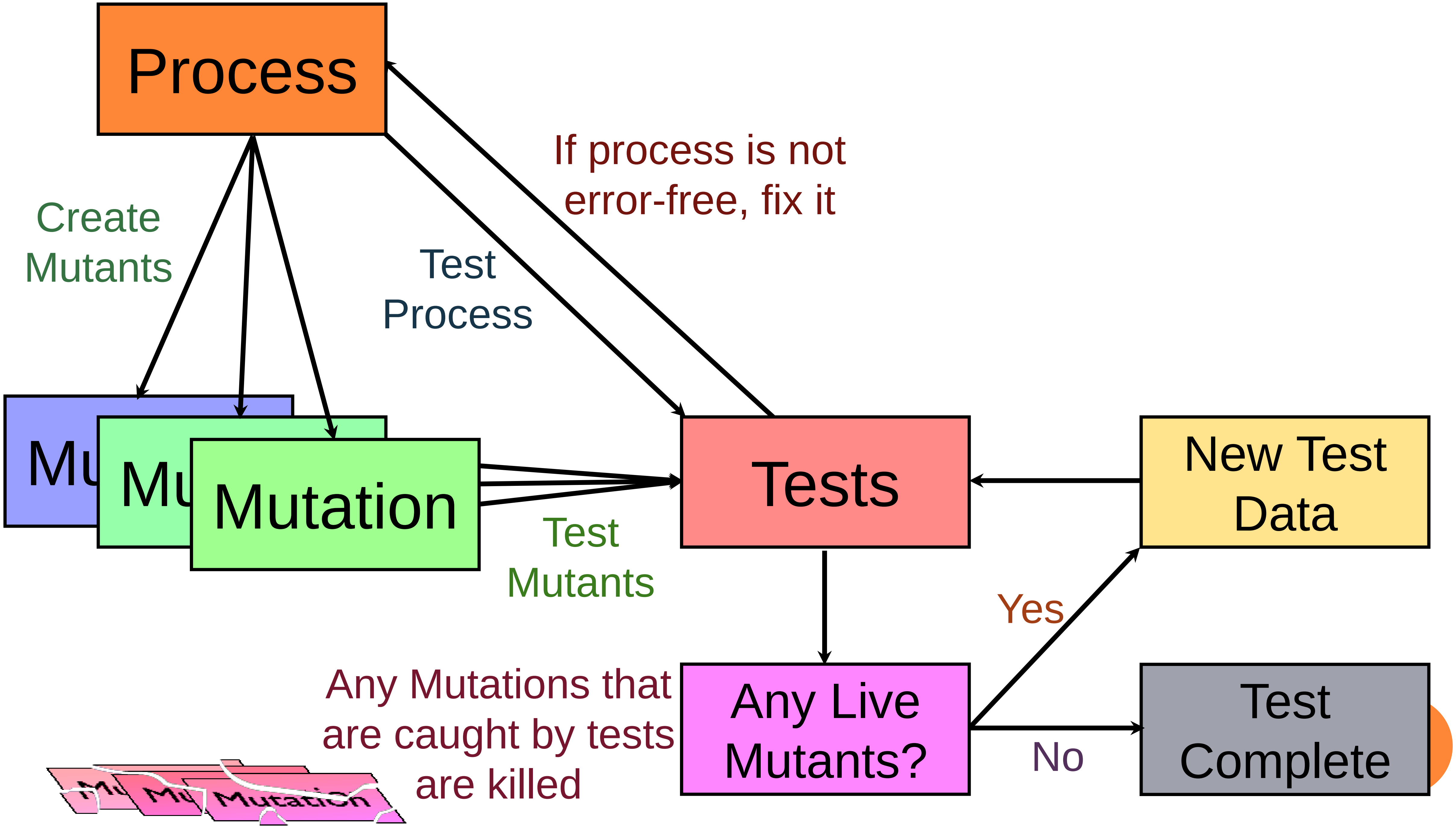
Boolean:

- Replacement of a statement with another
  - eg. == and >=, < and <=
- Replacement of boolean expressions with *true* or *false*
  - eg. a || b with *true*

Replacement of arithmetic
  eg. * and +, / and -

Replacement of a variable (ensuring same scope/type)

The Mutation Process



How does it work? - Summary

- Changes the original source code with special “operators” to generate “Mutants”
- Run the test suite related to the changed code
  - If a test errors or fails → Kills the mutant
  - If all tests run → The Mutant survives
- Surviving Mutants show not tested cases

The Important Thing!

Why is not widely used?

Is not new … - History

- Begins in 1971, R. Lipton, “Fault Diagnosis of Computer Programs”

- Generally accepted in 1978, R. Lipton et al, “Hints on test data selection: Help for the practicing programmer”

Why is not widely used?

- Technical Problem: It is a Brute Force technique!

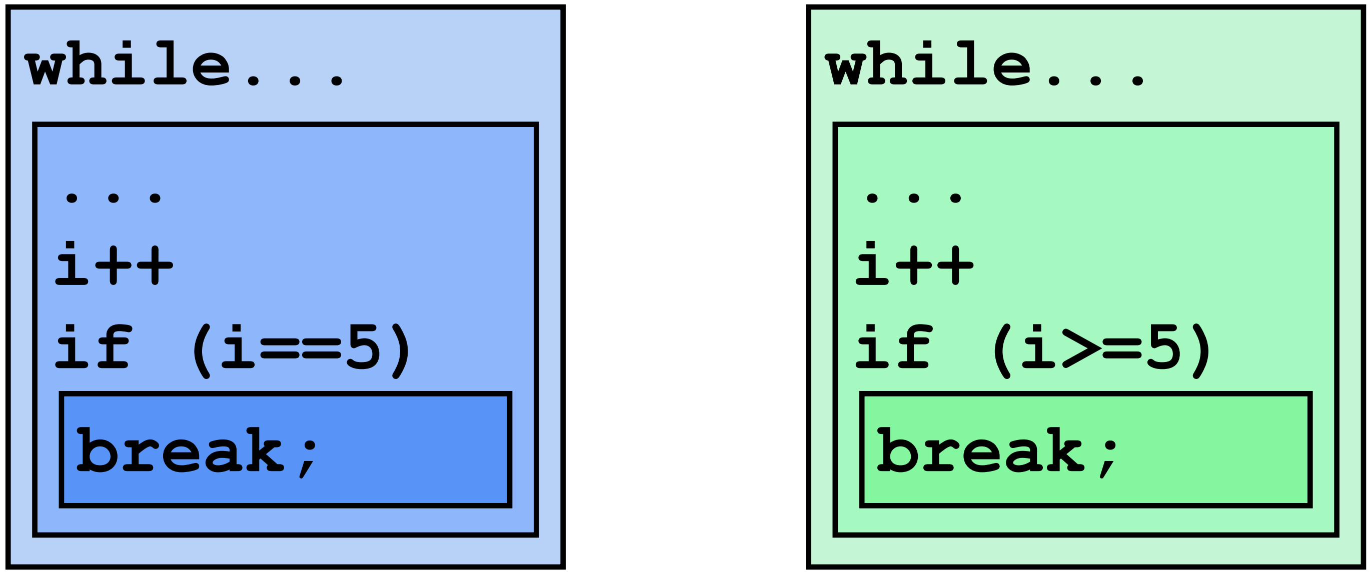
Technical Problems

- Brute force technique
  - N x M
    - N = number of tests
    - M = number of mutants

- Number of Tests: 666
- Number of Mutants: 1005
- Time to create a mutant/compile/link/run: 10 secs. each aprox.?
- Total time:
  - 6693300 seconds
  - 1859 hours, 15 minutes

Mutant Equivalence

- There may be surviving mutants that cannot be killed, these are called Equivalent Mutants
- Although syntactically different, these mutants are indistinguishable through testing.
- They therefore have to be checked ‘by hand’



```
while...
  ...
  i++
  if (i==5)
    break;
```

```
while...
  ...
  i++
  if (i>=5)
    break;
```

Mutant Equivalence

- Checking through all the Equivalent Mutants can make Mutation Testing cost-prohibitive

“Even for these small programs the human effort needed to check a large number of mutants for equivalence was almost prohibitive” - Frankl *et al.*, 1997

- R.M. Hierons *et al.*, 1999 proposed Program Slicing could be used in imperative languages to help towards the problem of Equivalent Mutants
- Offutt and Pan, 1996 introduced an approach based on constraint solving that increased the equivalence detection rate up to 48%

Problems

- There are a few factors that stop Mutation Testing from being more than an academic research topic, and being a practical method of testing:
  - The undecidability of Equivalent Mutants, and the cost of checking ‘by hand’
  - The relatively high computational cost of running all the mutations against a test set
  - The need for a Human Oracle to verify the contents of output is made more expensive by increases in test cases; this is especially the case using Mutation Testing
- However, methods for limiting the costs involved are continuing to be developed, increasing the chances of industry adoption

References

1. R. A. DeMillo, R. J. Lipton and F. G. Sayward (1978), “Hints on test data selection: Help for the practical programmer,” IEEE Computer no. 11, pp. 34-41.

2. P. G. Frankl, S. N. Weiss and C. Hu (1997), “All-uses vs mutation testing: An experimental comparison of effectiveness,” Journal of Systems Software no. 38, pp. 235-253.

3. R. M. Hierons, M. Harman and S. Danicic (1999), “Using Program Slicing to Assist in the Detection of Equivalent Mutants,” Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 233-262.

4. A. J. Offutt and J. Pan (1996), “Detecting equivalent mutants and the feasible path problem,” Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press, pp. 224-236.

5. R. T. Alexander, J. M. Bieman, S. Ghosh and J. Bixia (2002), “Mutation of Java objects,” 13th International Symposium on Software Reliability Engineering, Fort Collins, CO, USA, 2002, pp. 341-351.

6. J. S. Bradbury, J. R. Cordy, and J. Dingel (2006), “Mutation operators for concurrent Java (J2SE 5.0),” Proc. of the 2nd Workshop on Mutation Analysis (Mutation 2006), pp. 83–92.

7. K. M. Sacha (2006), “Software engineering techniques: design for quality,” Springer, pp 274-275

8. M. Harman and Y. Jia (2009), http://www.dcs.kcl.ac.uk/pg/jiayue/repository/index.php, “Mutation Testing Repository,” accessed on 26/11/2009.