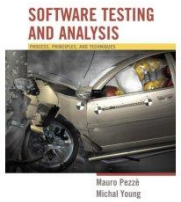
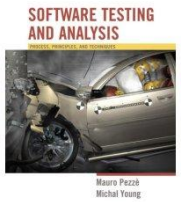
# Unit 7
# Fault-Based Testing

Mr. C. R. Belavi
Assistant Professor
Dept. Of. CSE, HSIT, NIDASOSHI



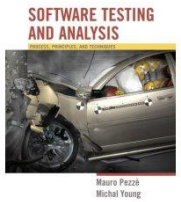
# contents

- Overview
- Assumption in fault Based Testing
- Mutation Analysis
- Fault Based Adequacy Criteria
- Variation on Mutation Analysis



# Overview

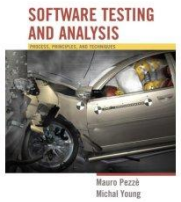
- Study the failure how to prevent similar failure in the future.
- Ex1: Failure of the Tacoma Narrow Bridge.
- Ex2: Airline crash.



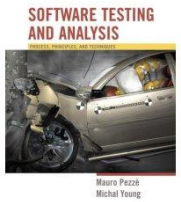
# Assumption in fault Based Testing

- The basic concept of fault based testing is to select test cases that would distinguish the program under test from alternative program that contains hypothetical faults.

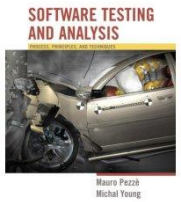
- Mutation testing is based on two assumptions: *the competent programmer hypothesis* and *the coupling effect*.



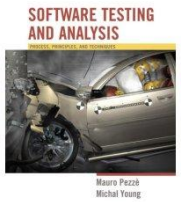
- The **competent programmer hypothesis** assumes that competent programmers tend to write nearly "correct" programs .
- That is programs written by experienced programmers may not be correct, but they will differ from the corrected version by some relatively simple faults such as **off-by-one fault.**



- The coupling effect stated that a set of test data that can uncover all simple faults in a program is also capable of detecting more complex faults.



- Competent programmer hypothesis:
  - Programs are nearly correct
    - Real faults are small variations from the correct program
    - => Mutants are reasonable models of real buggy programs
- Coupling effect hypothesis:
  - Tests that find simple faults also find more complex faults
    - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too



# Mutation Analysis

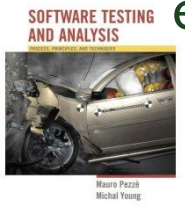
- Mutation Testing is a powerful error-based testing technique for unit testing. It provides high test coverage and detects many simple syntactic faults.

- A *mutant* is a copy of a program with a *mutation*

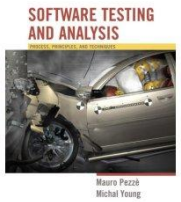
- A *mutation* is a syntactic change (a seeded bug)
  - Example: change (i < 0) to (i <= 0)

- Run test suite on all the mutant programs
- A mutant is *killed* if it fails on at least one test case

- If many mutants are killed, infer that the test suite is also effective at finding real bugs

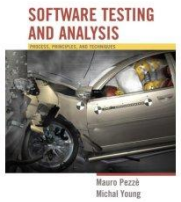


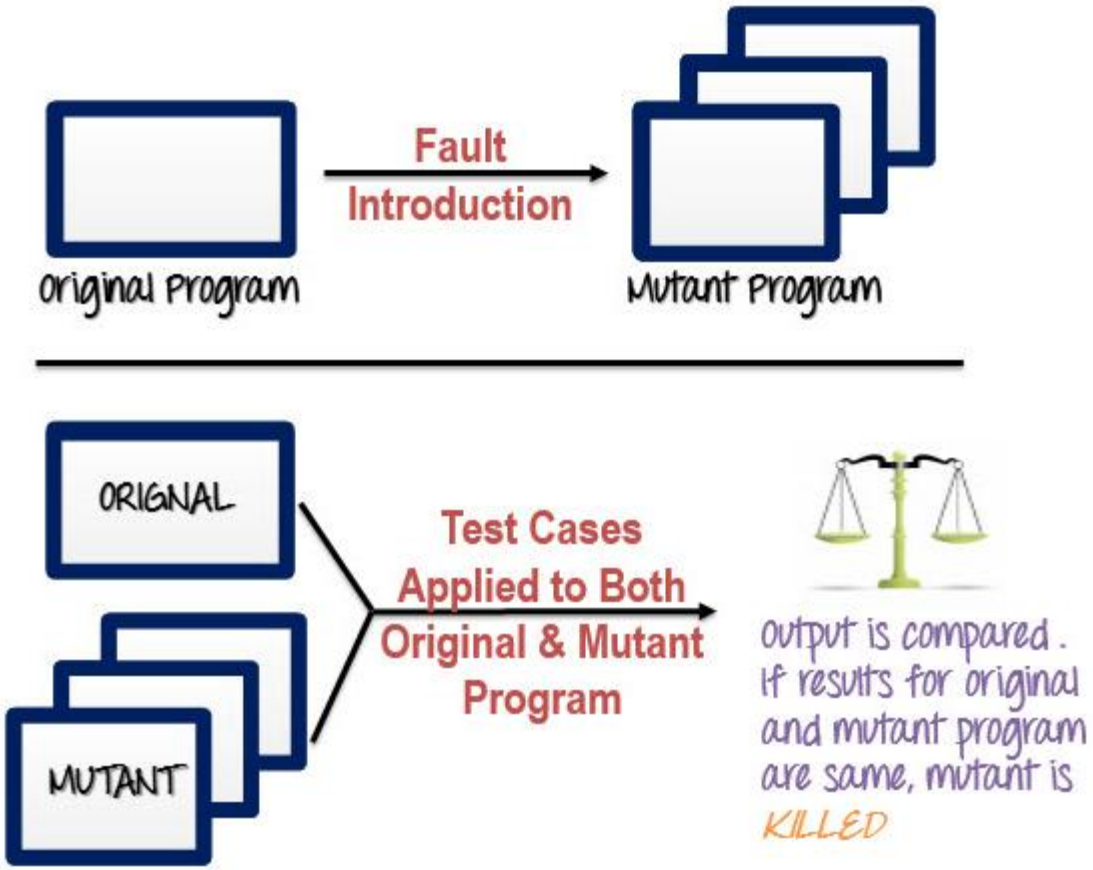
- The goal of Mutation Testing is to assess the quality of the test cases which should be robust enough to fail mutant code.
- To maintain the effectiveness of test sets.
- Mutation was originally proposed in 1971 but lost fervor due to high costs involved. Now, again it has picked steam and is widely used for languages such as Java and XML.

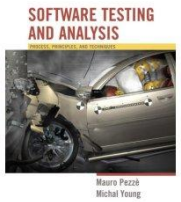


# Different types of Mutants

- Stillborn mutants: Syntactically incorrect, killed by compiler, e.g., x = a ++ b
- Trivial mutants: Killed by almost any test case
- Equivalent mutant: Always acts in the same behavior as the original program, e.g., x = a + b and x = a – (-b)
- None of the above are interesting from a mutation testing perspective
- Those mutants are interesting which behave differently than the original program, and we do not have test cases to identify them (to cover those specific changes)

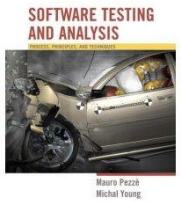



Fault
Introduction
original Program
Mutant Program
ORIGNAL
MUTANT
Test Cases
Applied to Both
Original & Mutant
Program
output is compared.
If results for original
and mutant program
are same, mutant is
KILLED


SOFTWARE TESTING
AND ANALYSIS
Mauro Pezzè
Michal Young

# Following are the steps to execute mutation testing:

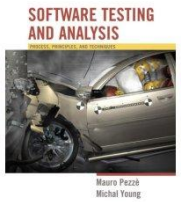
**Step 1**: Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases.
**Step 2**: Test cases are applied to the original program and also to the mutant program. A test case should be adequate, and it is tweaked to detect faults in a program.
**Step 3**: Compare the results of original and mutant program.
**Step 4**: If the original program and mutant programs generate the same output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program.
**Step 5**: If the original program and mutant program generate different output, Mutant is kept alive. In such cases , more effective test cases need to be created that  kill all mutants.



# How to Create Mutant Programs?

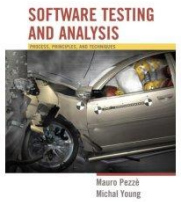
- A mutation is nothing but a single syntactic change that is made to the program statement. Each mutant program should differ from the original program by one mutation.

| Original Program | Mutant Program |
|---|---|
| If (x>y)<br>Print “Hello”<br>Else<br>Print “Hi” | If(**x)**<br>**Print “Hello”**<br>**Else**<br>**Print “Hi”** |



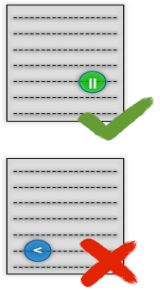
# Automation of Mutation Testing:

- Mutation testing is extremely time consuming and complicated to execute manually. To speed up the process, it is advisable to go for automation tools. Automation tools reduce cost of testing as well.
- List of tools available -
- Ninja Turtles- .net mutation testing tool
- Mutagenesis- PHP mutation testing framework
- Heckle- Ruby Mutation Testing Tool
- Jester- Mutation Testing Tool for Java



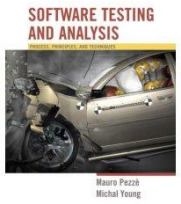
# Mutation Score:

- The mutation score is defined as the percentage of killed mutants with the total number of mutants.
- Mutation Score = (Killed Mutants / Total number of Mutants) * 100
- The effectiveness of the test data set is measured by the percentage of mutants killed.



Mutation Score:

$$\frac{\text{Killed Mutants}}{\text{Total Mutants}}$$

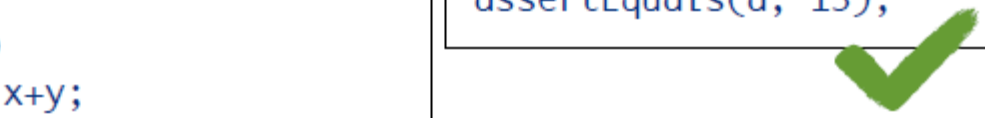


```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

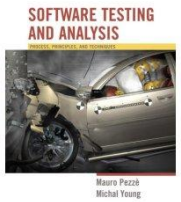


```
int do_something(int x, int y)
{
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

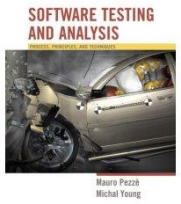
Test



# Example of a Program Mutation

```
1 int max(int x, int y)
2 {
3 int mx = x;
4 if (x > y)
5     mx = x;
6 else
7     mx = y;
8 return mx;
9 }
```

```
1 int max(int x, int y)
2 {
3 int mx = x;
4 if (x < y)
5     mx = x;
6 else
7     mx = y;
8 return mx;
9 }
```



# Example of Testing By Mutation

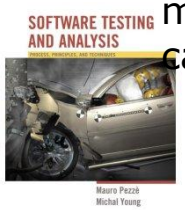
```
function MAX(M<N:INTEGER)
   return INTEGER is
begin
    if M>N then
       return M;
   else
       return N;
   end if:
end MAX;
```

First test data set--M=1, N=2

- the original function returns 2
- mutants: replace”>“ operator in if statements by (>=,<,<=or=)
- executing each mutant:

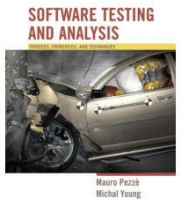
| Mutants | Outputs | Comparison |
|---|---|---|
| if M>=N then | 2 | dead |
| if M<N then | 1 | live |
| if M<=N then | 1 | live |
| if M=N then | 2 | dead |
| if M< >N then | 1 | live |

- adding test data M=2, N=1 will eliminate the latter live mutant, but the former live mutant remains live because it is equivalent to the original function. No test data can eliminate it.



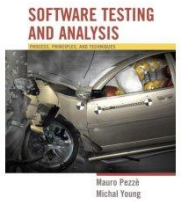
# Advantages of Mutation Testing:

- It is a powerful approach to attain high coverage of the source program.
- This testing is capable comprehensively testing the mutant program.
- Mutation testing brings a good level of error detection to the software developer.
- This method uncovers ambiguities in the source code, and has the capacity to  detect all the faults in the program.
- Customers are benefited from this testing by getting most reliable and stable system.



# Disadvantages of Mutant testing:

- Mutation testing is extremely costly and time consuming since there are many mutant programs that need to be generated.
- Since its time consuming, it's fair to say that this testing cannot be done without an automation tool.
- Each mutation will have the same number of test cases than that of the original program. So, a large number of mutant programs may need to be tested against the original test suite.
- As this method involves source code changes, it is not at all applicable for black box testing.

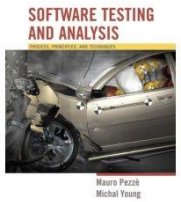


# Estimating #Defects

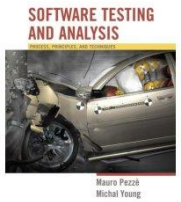
- How many defects remain in our software?

- With mutation testing, we can make an

Estimate of remaining defects

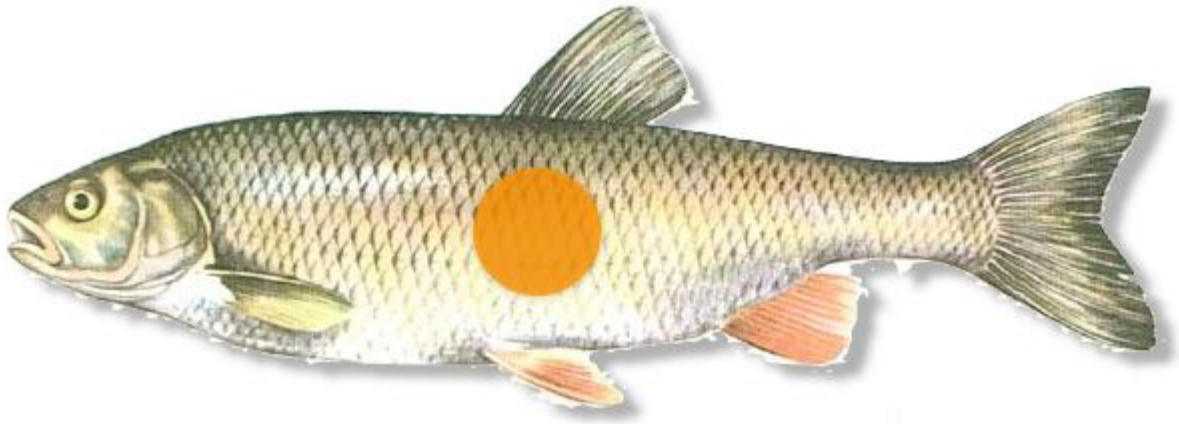


# Let’ s consider a lake. How many fish are in that lake?

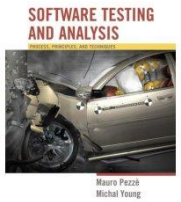


Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

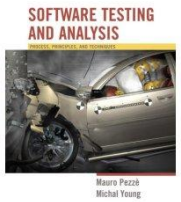
## Fish Tag



- We catch 1,000 fish and tag them



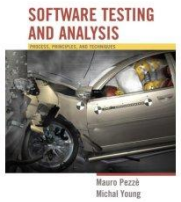
Let's assume over the next week, we ask fishermen to count the number of tags. We find 300 untagged and 50 tagged fish.

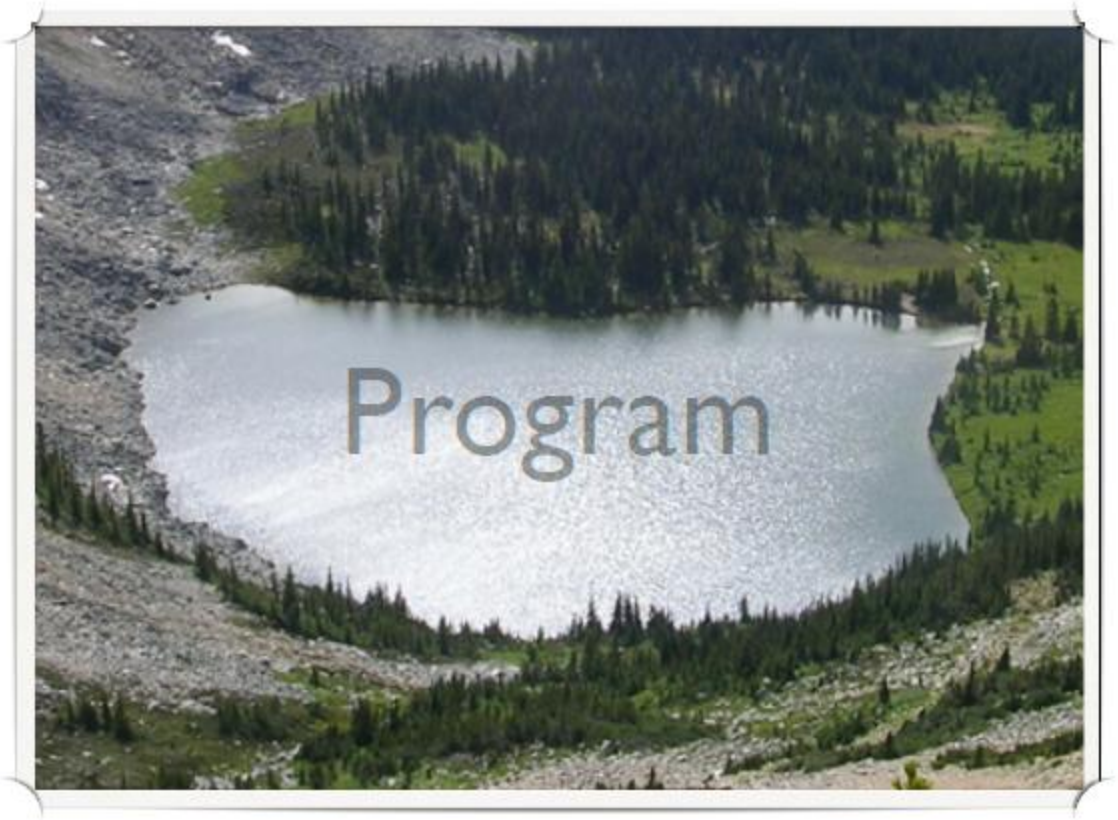
we can thus estimate that there are about 6,000 remaining untagged fish in the lake.

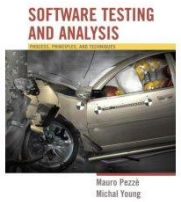
Estimate

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



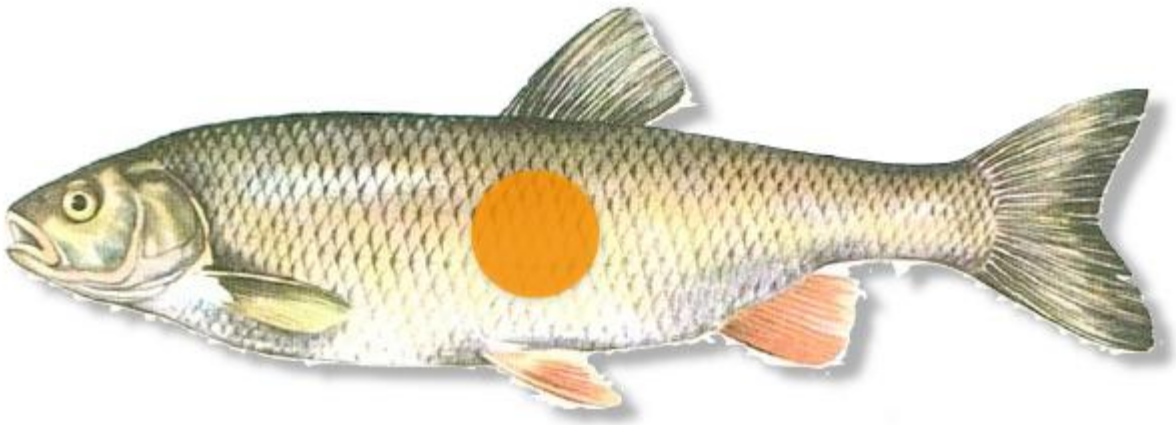
# Now let's assume our lake is not a lake, but our program.



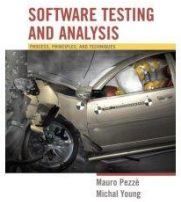



Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

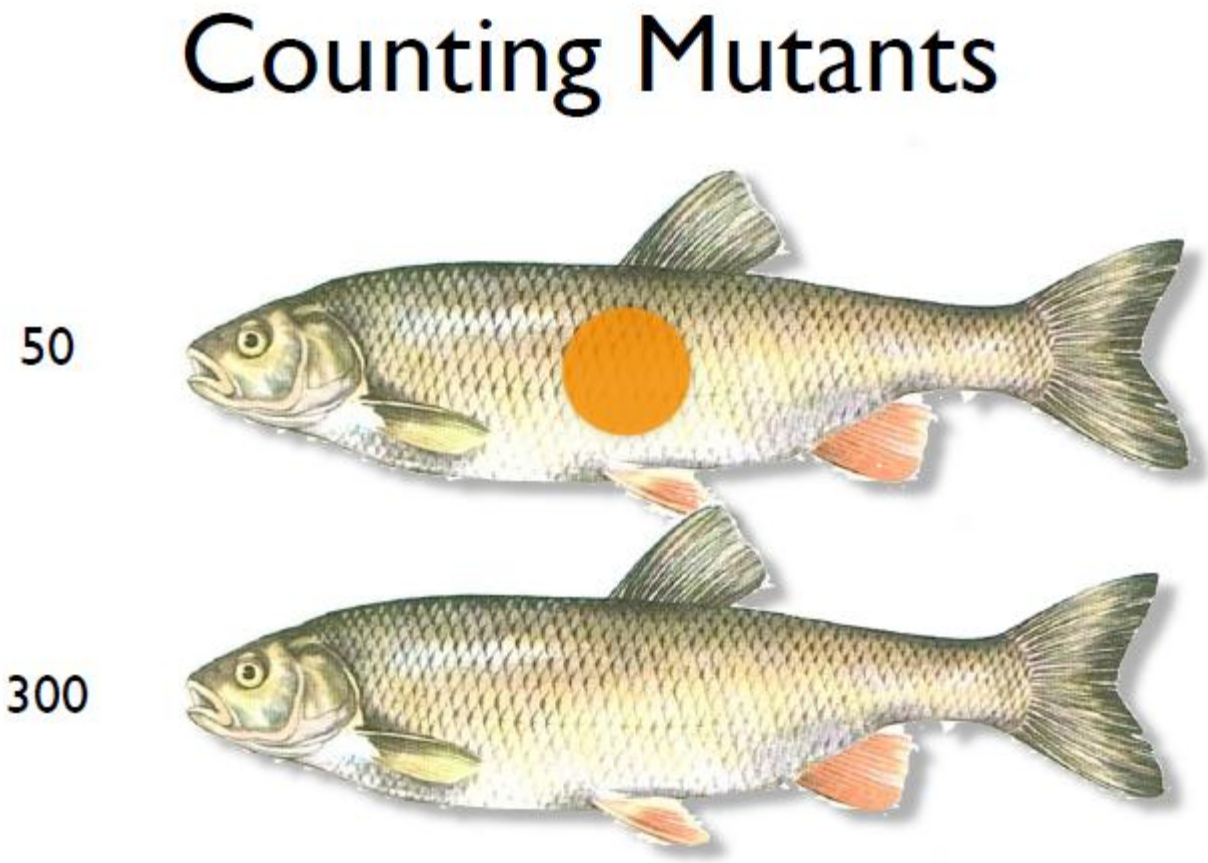
## A Mutant

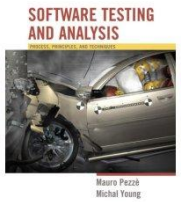


- We seed 1,000 mutations into the program



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

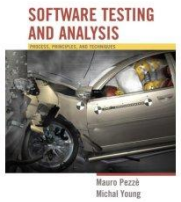


we can again estimate that there are about 6,000 remaining defects in our program. (A test suite finding only 50 out of 1,000 mutations is a real bad sign.)

# Estimate

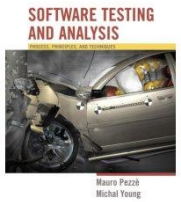
$$\frac{\text{(Seeded fault) } 1{,}000}{\text{remaining defects}} = \frac{50 \text{ (seeded fault detected)}}{300 \text{ (natural fault detected)}}$$



# Let's count marbles ... a lot of marbles

Photo credit: (c) KaCey97007 on Flickr, Creative Commons license

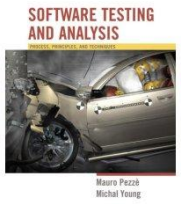
- Suppose we have a big bowl of marbles. How can we estimate how many?
  - I don't want to count every marble individually
  - I have a bag of 100 other marbles of the same size, but a different color
  - What if I mix them?



# Estimating marbles

- I mix 100 black marbles into the bowl
  - Stir well ...
- I draw out 100 marbles at random
- 20 of them are black

- How many marbles were in the bowl to begin with?



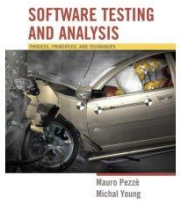
# Fault Based Adequacy Criteria

- Given a program and a test suite T. Mutation analysis consists of the following steps

1. Select mutation operator

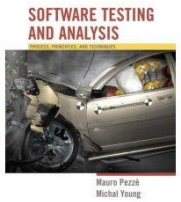
2. Generate mutants

3. Distinguish mutants



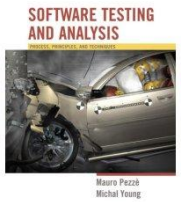
# Mutation operator

| Type | Description |
|---|---|
| aar | array for array replacement |
| abs | absolute value insertion |
| acr | array constant replacement |
| aor | arithmetic operator replacement |
| asr | array for variable replacement |
| car | constant for array replacement |
| cnr | comparable array replacement |
| csr | constant for scalar replacement |
| der | DO statement end replacement |
| lcr | logical connector replacement |
| ror | relational operator replacement |
| sar | scalar for array replacement |
| scr | scalar for constant replacement |
| svr | scalar variable replacement |
| crp | constant replacement |
| dsa | data statement alterations |
| glr | goto label replacement |
| rsr | return statement replacement |
| san | statement analysis |
| sdl | statement deletion |
| src | source constant replacement |
| uoi | unary operation insertion |



# Mutation Operators

- Syntactic change from legal program to legal program
  - So: Specific to each programming language. C++ mutations don't work for Java, Java mutations don't work for Python
- Examples:
  - crp: constant for constant replacement
    - for instance: from (x < 5) to (x < 12)
    - select from constants found somewhere in program text
  - ror: relational operator replacement
    - for instance: from (x <= 5) to (x < 5)
  - vie: variable initialization elimination
    - change int x =5; to int x;



```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

# ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

# AOR - Arithmetic Operator Replacement

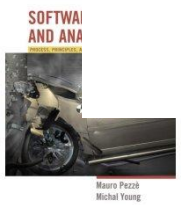
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

+, -, *, /, %, **, x, y

# ROR - Relational Operator Replacement

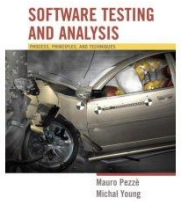
```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

<, >, <=, >=, =, !=, false, true



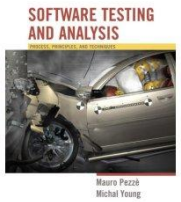
# Live Mutants

- Scenario:
  - We create 100 mutants from our program
  - We run our test suite on all 100 mutants, plus the original program
  - The original program passes all tests
  - 94 mutant programs are killed (fail at least one test)
  - 6 mutants remain *alive*
- What can we learn from the living mutants?



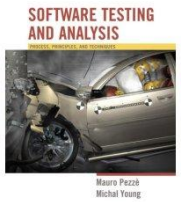
# How mutants survive

- A mutant may be equivalent to the original program
  - Maybe changing (x < 0) to (x <= 0) didn't change the output at all! The seeded "fault" is not really a "fault".
    - Determining whether a mutant is equivalent may be easy or hard; in the worst case it is undecideable
- Or the test suite could be inadequate
  - If the mutant could have been killed, but was not, it indicates a weakness in the test suite
  - But adding a test case for just this mutant is a bad idea. We care about the real bugs, not the fakes!



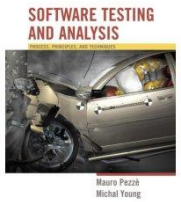
# Variation in mutation Testing

- Since the number of mutants that can be generated is large (the number is usually on the order of $N^2$, where N is the number of variable references in the program), methods have been suggested to reduce the computational expenses of this testing technique.
- Methods proposed over the years to combat the expensive computation problem include *weak mutation*, *Strong mutation*, *Statistical mutation*.



# Variations on Mutation

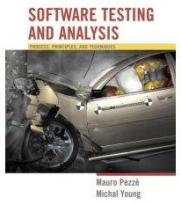
- Weak mutation
- Statistical mutation



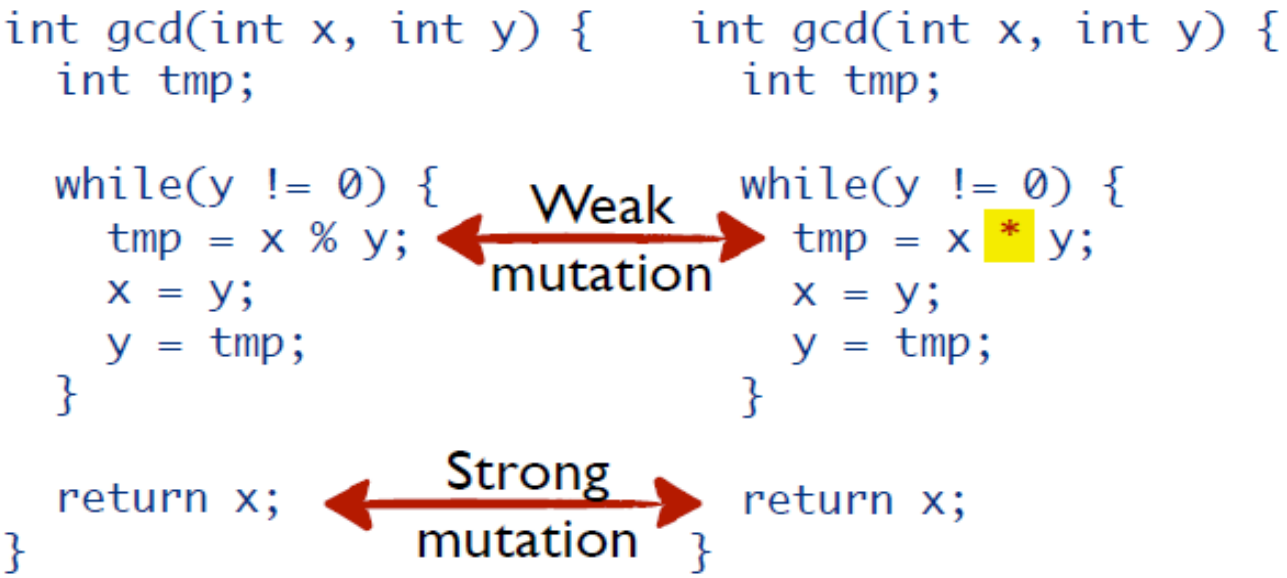
# Weak mutation

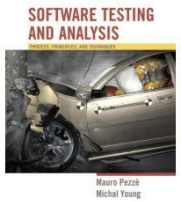
- **Weak mutation** only requires the test data to cause a mutated component to take on a different value in at least one execution, instead of outputting a different value from the expected result.

- Problem: There are lots of mutants. Running each test case to completion on every mutant is expensive
  - Number of mutants grows with the square of program size
- Approach:
  - Execute meta-mutant (with many seeded faults) together with original program
  - Mark a seeded fault as "killed" as soon as a difference in intermediate state is found
    - Without waiting for program completion
    - Restart with new mutant selection after each "kill"



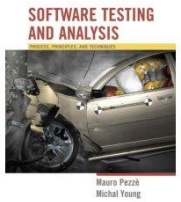
# Strong vs. Weak Mutation



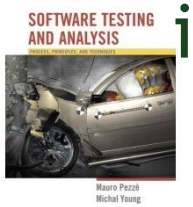
# Statistical Mutation

- Problem: There are lots of mutants. Running each test case on every mutant is expensive
  - It's just too expensive to create $N^2$ mutants for a program of N lines (even if we don't run each test case separately to completion)
- Approach: Just create a random sample of mutants
  - May be just as good for *assessing* a test suite
    - Provided we don't design test cases to kill particular mutants (which would be like selectively picking out black marbles anyway)



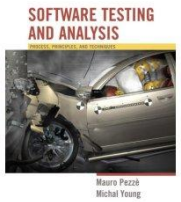
# In real life ...

- Fault-based testing is a widely used in semiconductor manufacturing
  - With good *fault models* of typical manufacturing faults, e.g., “stuck-at-one” for a transistor
  - But fault-based testing for *design errors* is more challenging (as in software)
- Mutation testing is not widely used in industry
  - But plays a role in software testing research, to compare effectiveness of testing techniques
- Some use of fault models to design test cases is important and widely practiced



# Summary

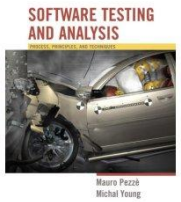
- If bugs were marbles ...
  - We could get some nice black marbles to judge the quality of test suites
- Since bugs aren't marbles ...
  - Mutation testing rests on some troubling assumptions about seeded faults, which may not be statistically representative of real faults
- Nonetheless ...
  - A model of typical or important faults is invaluable information for designing and assessing test suites



# Contents

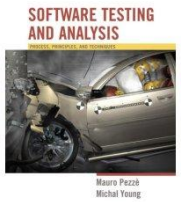
Test Execution:

- Overview from test case specifications to test cases.
- Scaffolding.
- Generic versus specific scaffolding.
- Test oracles.
- Self-checks as oracles.
- Capture and replay.



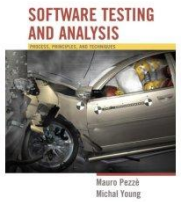
# Automating Test Execution

- Designing test cases and test suites is creative
  - Like any design activity: A demanding intellectual activity, requiring human judgment
- Executing test cases should be automatic
  - Design once, execute many times
- Test automation separates the creative human process from the mechanical process of test execution



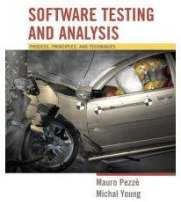
# Generation: From Test Case Specifications to Test Cases

- Test design often yields test case specifications, rather than concrete data
  - Ex: “a large positive number”, not 420023
  - Ex: “a sorted sequence, length > 2”, not “Alpha, Beta, Chi, Omega”
- Other details for execution may be omitted
- Generation creates concrete, executable test cases from test case specifications



# Example Tool Chain for Test Case Generation & Execution

- We could combine ...
  - A combinatorial test case generation (like genpairs.py) to create test data
    - Optional: Constraint-based data generator to “concretize” individual values, e.g., from “positive integer” to 42
  - DDSteps to convert from spreadsheet data to JUnit test cases
  - JUnit to execute concrete test cases
- Many other tool chains are possible ...
  - depending on application domain

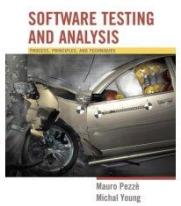


# Scaffolding

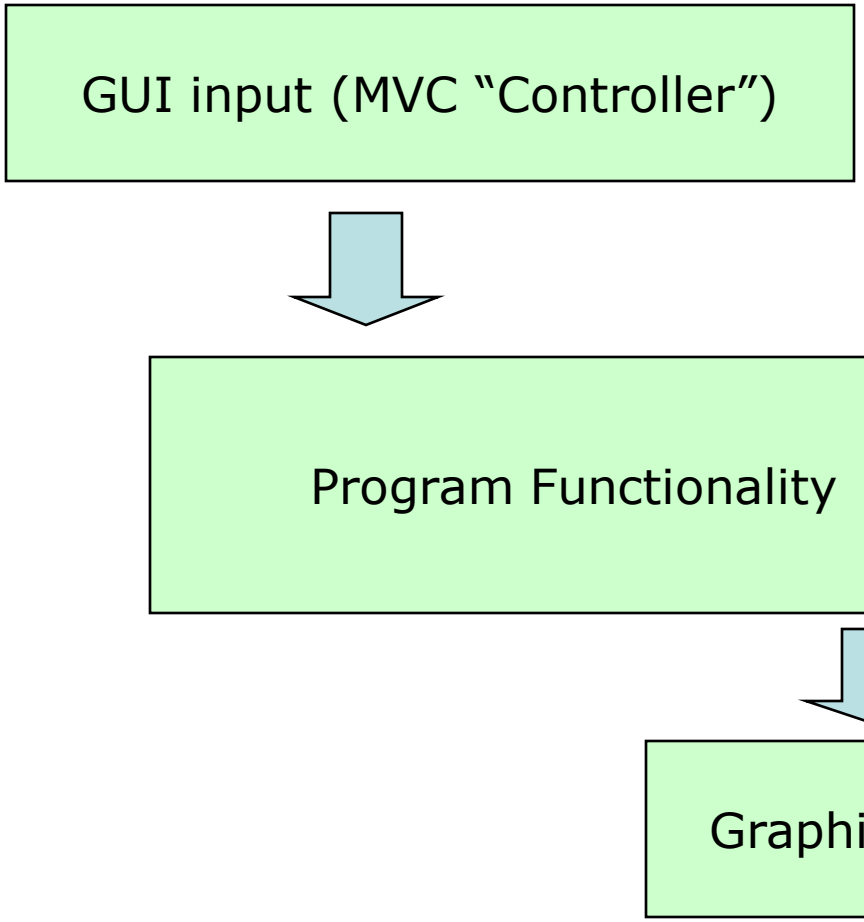
- Code produced to support development activities (especially testing)
  - Not part of the “product” as seen by the end user
  - May be temporary (like scaffolding in construction of buildings
- Includes
  - Test harnesses, drivers, and stubs

# Scaffolding ...

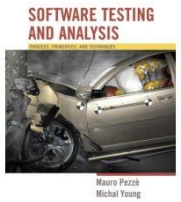
- Test driver
  - A “main” program for running a test
    - May be produced before a “real” main program
    - Provides more control than the “real” main program
      - To driver program under test through test cases
- Test stubs
  - Substitute for called functions/methods/objects
- Test harness
  - Substitutes for other parts of the deployed environment
    - Ex: Software simulation of a hardware device



# Controllability & Observability

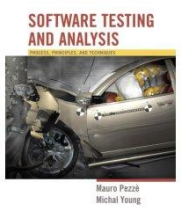



Example: We want automated tests, but interactive input provides limited control and graphical output provides limited observability

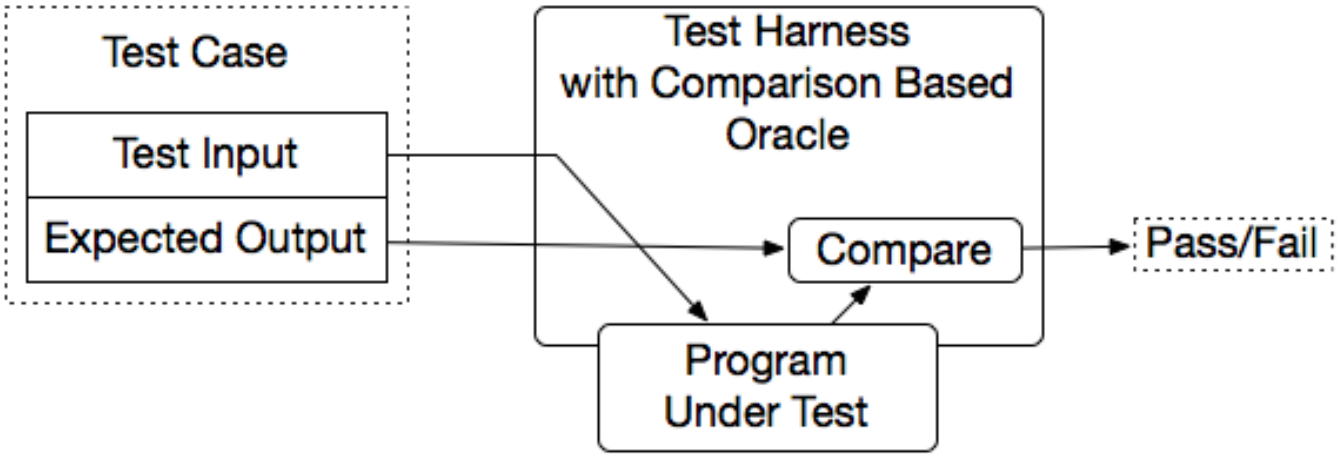


# Generic or Specific?

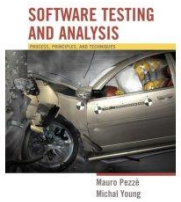
- How general should scaffolding be?
  - We could build a driver and stubs for each test case
  - ... or at least factor out some common code of the driver and test management (e.g., JUnit)
  - ... or further factor out some common support code, to drive a large number of test cases from data (as in DDSteps)
  - ... or further, generate the data automatically from a more abstract model (e.g., network traffic model)
- A question of costs and re-use
  - Just as for other kinds of software



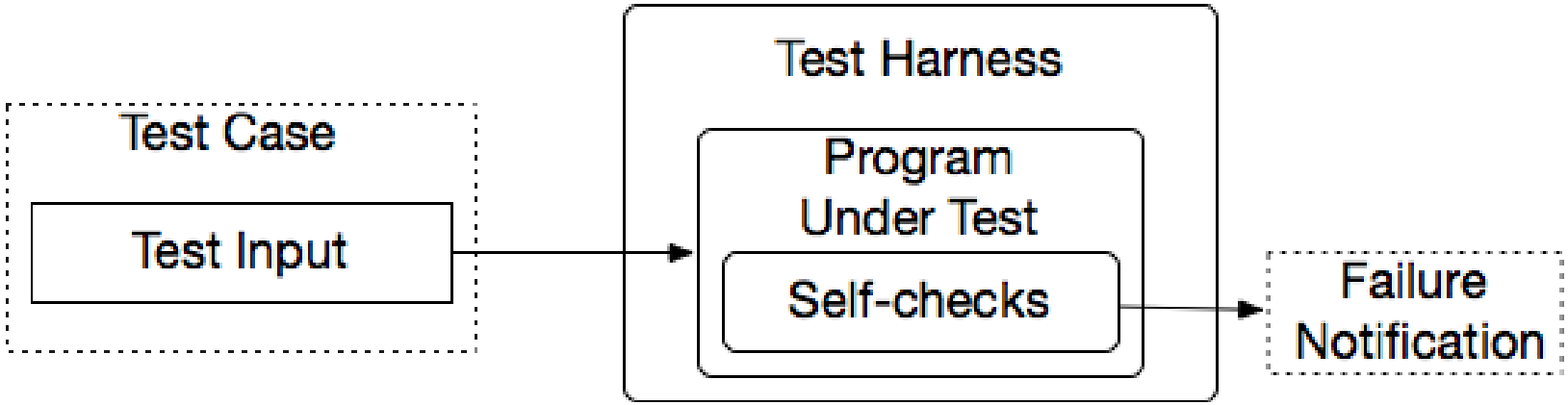
# Comparison-based oracle



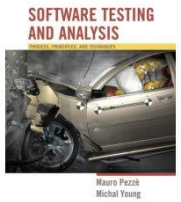

- With a comparison-based oracle, we need predicted output for each input
  - Oracle compares actual to predicted output, and reports failure if they differ
- Fine for a small number of hand-generated test cases
  - E.g., for hand-written JUnit test cases



# Self-Checking Code as Oracle

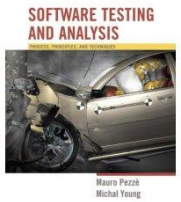



- An oracle can also be written as *self-checks*
  - Often possible to judge correctness without predicting results
- Advantages and limits: Usable with large, automatically generated test suites, but often only a *partial* check
  - e.g., structural invariants of data structures
  - recognize *many* or *most* failures, but not all



# Capture and Replay

- Sometimes there is no alternative to human input and observation
  - Even if we separate testing program functionality from GUI, some testing of the GUI is required
- We can at least cut *repetition* of human testing
- *Capture* a manually run test case, *replay* it automatically
  - with a comparison-based test oracle: behavior same as previously accepted behavior
    - reusable only until a program change invalidates it
    - lifetime depends on abstraction level of input and output



# Summary

- Goal: Separate creative task of test design from mechanical task of test execution
  - Enable generation and execution of large test suites
  - Re-execute test suites frequently (e.g., nightly or after each program change)
- Scaffolding: Code to support development and testing
  - Test drivers, stubs, harness, including oracles
  - Ranging from individual, hand-written test case drivers to automatic generation and testing of large test suites
  - Capture/replay where human interaction is required