CS350 Lecture 8
Software Testing

Doo-Hwan Bae

SoC, KAIST

bae@se.kaist.ac.kr

Overview of SW Testing

Terminology: Error, Fault, and Failure(1/4)

- Error
  - Difference between the actual output and the correct output.
  - Measurement of the difference between the actual and ideal
  - (IEEE) human action(mistake) that produces an incorrect result
- Fault = Bug
  - A condition(potential) that causes the software to fail.
  - Manifestation of an error in program
- Failure
  - The inability of a system to perform a required function.
  - Is produced only when there is a fault
  - Presence of a fault does not mean a failure.

Terminology: V(erification) & V(alidation) (2/3)

- Verification
  - The process of evaluating products of a development phase to find out whether they meet the specified requirements.
  - *Do we build the system right?*
- Validation
  - The process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
  - *Do we build the right system?*

Classification of V&V Activities

- Most of the V&V activities can be classified as dynamic or static
  - Dynamic
    - testing(so called dynamic testing)
  - Static
    - reviews
    - program proving
    - code reading

Static vs. Dynamic: Characteristics

- Static
  - Do not observe system behavior
  - Not looking for system failures
  - Faults are directly detected
- Dynamic(Testing)
  - System behavior is observed
  - Determine the existence of failures
  - Reveals the presence of faults, but not their absence
  - Actual faults will be identified by "debugging" activity
  - No failure does not means no fault

What is (Dynamic) Testing ?

- How: Exercising a system or component
  - with defined inputs
  - capturing monitored outputs
  - comparing outputs with specified or intended requirements
- For What: purposes
  - to identify discrepancies between actual results and correct or expected behavior
  - to provide high assurance of reliability, correctness, ……
- What is NOT software testing?
  - Debugging..

Fundamental Testing Questions

- What cases should we test?
    - What test data should we use ?
    - What aspects of the software should we test?
- Are we there yet? When can we stop testing?
    - Did we find a sufficient number of failures?
    - Did we cover the product satisfactorily?
- How well do we do?
    - Did we provide high assurance?
- Infinite possibilities with limited resources?
    - Working overnight? Agile? Call experts? Let the maintenance people do more? Take care of yourself by avoiding hidden bombs…

State of the Testing Practice: Effect of SW Defects

- Financial loss due to SW failure (2017): US$ 1.7 trillion

  (source: https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/)

- Persons affected by SW failure: 3.7Billion/7.4 Billion total

  (출처: https://www.tricentis.com/wp-content/uploads/2018/01/20180119_Software-Fails-Watch_Small_Web.pdf)

Testing Principles

- Q : Choose the correct principles

(Y,N) Testing is the process of executing a program with the intention of proving the correctness of the program.

(Y,N) It is possible to completely test any nontrivial module or any system.

(Y,N) Testing takes creativity, experience, and hard work.

(Y,N) Testing can prevent errors from occurring

(Y,N) Testing is best done by independent testers.

Fisherman's Dilemma

- You have three days for fishing and 2 lakes to choose from.
- Day 1 at lake X nets 8 fishes.
- Day 2 at lake Y nets 32 fishes.
- Q: Which lake do you return to for day 3 to catch more fishes?
- Fishing for errors
  - In general, the probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
- Problems of error-prone modules: A study showed that ..

Test Cases

- Two kinds of problems to be discovered by testing:
    - The program DOES NOT do something it is supposed to do.
    - The program DOES something it is NOT supposed to do.
- Test cases must be written for INVALID and UNEXPECTED, as well as valid and expected input conditions.
- What are parts of a test case?
    - Description of input conditions.
    - Description of expected results.
- Q: Why expected results should be included in a test case?

Test Every Input Conditions?

- Situation:
  - A module has two input integer parameters.
  - Word size: 32 bits.
  - Number of input conditions: ???
  - Test completely automated.
  - Time to execute one test case: 0.0000000001 second.
  - Total time required: 584.94 years!!
- Generally impossible, for even trivial examples, to test a program using every possible input conditions.
- Realistic goal of testing
  - Design a small number of test cases that can establish a reasonable level of confidence on the absence of undiscovered faults.

More Terminology(3/4)

- Black box testing (Functional testing)
  - Test cases derived from examining the requirements, specifications or user documents.
- White box testing (Structural testing)
  - Test cases derived from examining the actual code..

More Terminology(4/4)

- Driver:
  - When an element under test cannot be executed by itself, a DRIVER is needed to invoke the element.
  - The DRIVER supplies appropriate data and reports the results.
- Stub:
  - When an element under test calls other elements which are not present yet, STUBs are needed to take their places.
  - The STUBs can return planned, random or constant data.
- Scaffolding in testing:
  - Building drivers and stubs for testing

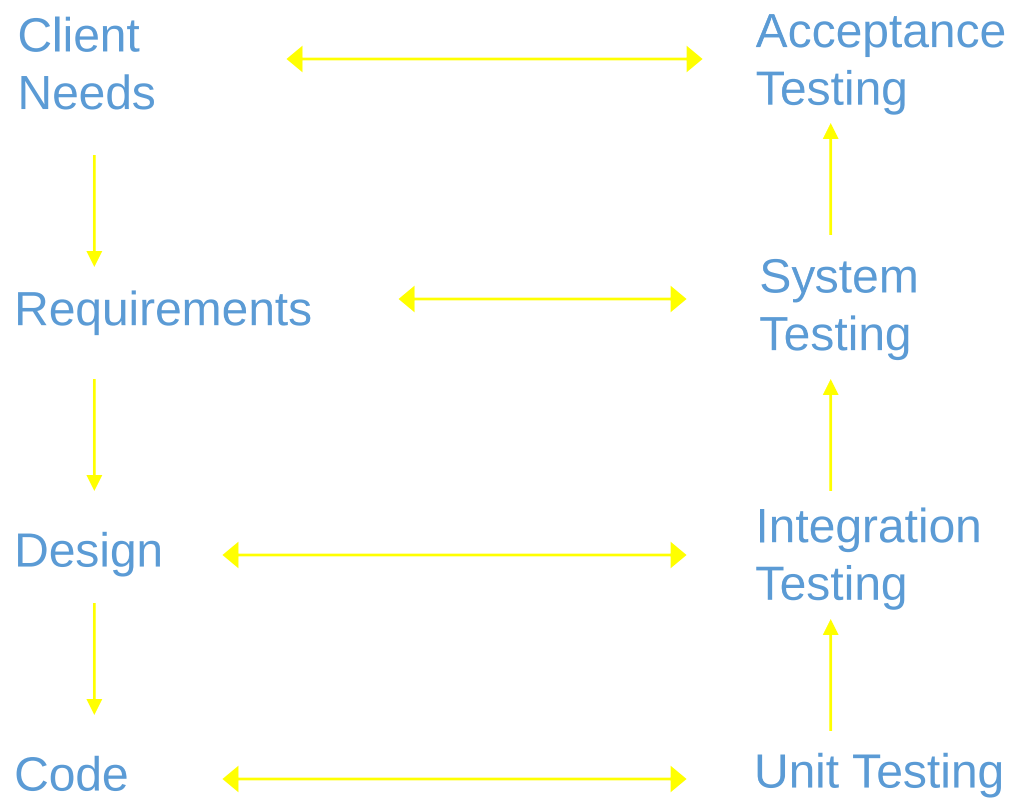
Categories of Testing Techniques: by Testing Levels (Phases)

Testing Phases (Levels)

- Unit Testing
  - testing of a unit comparing with requirements
- Integration Testing
  - systematic combination and testing of software components to insure consistency of component interface
- System Testing
  - testing an integrated hardware and software system (in target environment)
- Acceptance Testing
  - Carried out by users

Levels of Testing in SDLC

- Attempts to detect different type of faults



Integration Test

- Testing needed as the elements of a system are being combined.
- Types:
  - Nonincremental (Big-Bang) testing
  - Incremental testing
- Nonincremental testing
  - All the elements are combined at once and tested.
  - It is difficult because errors cannot be easily attributed to particular elements.
- Incremental testing
  - Elements are added and tested incrementally.
  - In What orders should the elements be integrated.
    - Top-down/Bottom-up/Threads/…

Top-Down Strategy(1/2)

- The root module is tested first.
- Stubs take the place of other modules called by the root module.
- A new module called by one of the already integrated modules is added and tested with others.
- The process continues in this manner until all modules have been integrated and tested.

Top-Down Strategy(2/2)

- Advantages:
  - Advantageous if major flaws occur toward the top of the program.
  - Early skeletal program allows partial demonstration.
- Disadvantages:
  - Before the I/O modules are added, the representation of test cases in stubs can be difficult.
  - Test cases are possibly in a different form for each level and must be continuously altered.

Bottom-Up Strategy

- Lower level modules are tested first using drivers.
- Higher level modules are added after all the modules they call have been added.
- The process continues in this manner until all modules have been integrated and tested.
- Advantages:
  - Advantageous if major flaws occur toward the bottom of the program.
  - Test conditions are easier to create.
  - Observation of test results is easier.
- Disadvantages:
  - The program as an entity does not exist until the last module is added.
  - As system subtrees are integrated at higher levels, several modules may be brought together for the first time in one step.

Thread Strategy

- A variation of top-down strategy.
- Input and output modules are added as early as possible.
- Choose a set of modules (the thread) that represents a normal, important case, or a critical path through the system.
- Advantages:
  - Other threads can be integrated in parallel.
  - Early skeleton versions are available early.
  - It also improves programmers morale.

Other Integration Strategies

- Critical Module First
- I/O First

- When is the I/O first strategy useful?

Categories of Testing Techniques: by Test Case Selection Criteria

Test Case Selection Criteria

- Test case selection criteria can be categorized as
  - Functional
  - Structural

Functional v.s. Structural

- Structural testing
  - test cases selected based on software structure/implementation
  - views the component as a white box
- Functional testing
  - test cases selected based on functional specification(require document, high-level design)
  - views the component as a black box
- Both functional and structural testing must be done.
  - These two are complementary each other

White Box Testing

- Using knowledge on internal structures(i.e. program paths) and maybe even ignoring specification.
- Uses internal structures of the program to derive the test data
- Will "exhaustive(or very through) white-box testing" solve testing problem ?

White Box Testing Steps

- Steps involved:
  - Examine the program logic.
  - Design test cases to satisfy a chosen coverage criteria.
  - Run the test cases.
  - Compare actual results and report errors.
  - Compare actual coverage to expected coverage.

Coverage Criteria in White-Box Testing

- Statement coverage
- Decision(branch) coverage
- Path coverage
- Condition coverage
- Decision/Condition Coverage

Statement Coverage

- Every statement must be executed at least once
- Weakest logic coverage criterion.

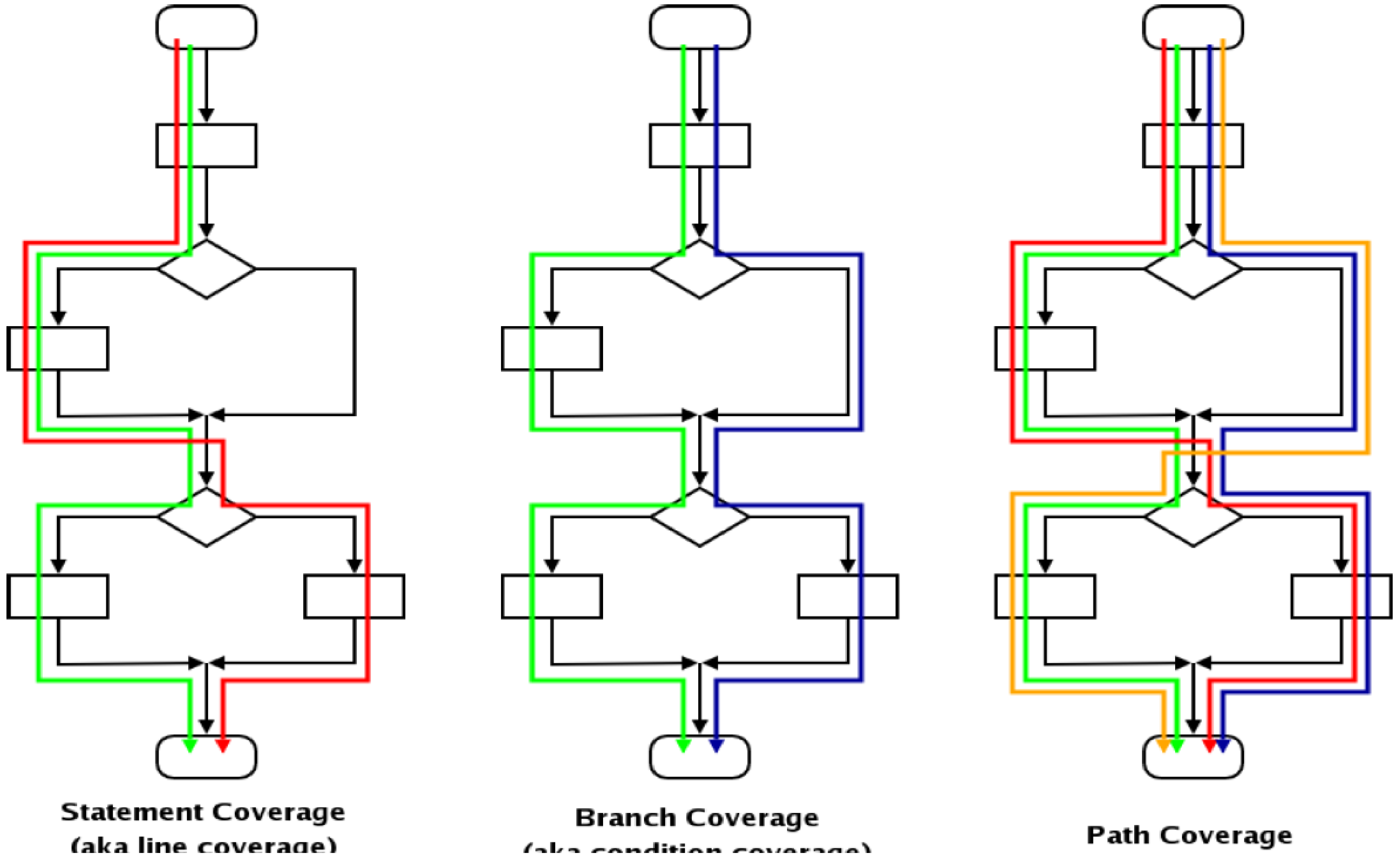
Decision Coverage (Branch Coverage)

- Each branch must be taken at least once.
- Branches:
  - If-then, if-then-else statement:
    - True and False branches.
  - Case statement:
    - each branch and others or default.
    - failure
  - While, repeat and for statements:
    - Must enter loop body at least once and must exit.

Condition Coverage

- Each condition in a decision must take on all possible outcomes at least once.
- Verify that the following test cases satisfy the condition coverage.
- Q: Is condition coverage the same as branch coverage?

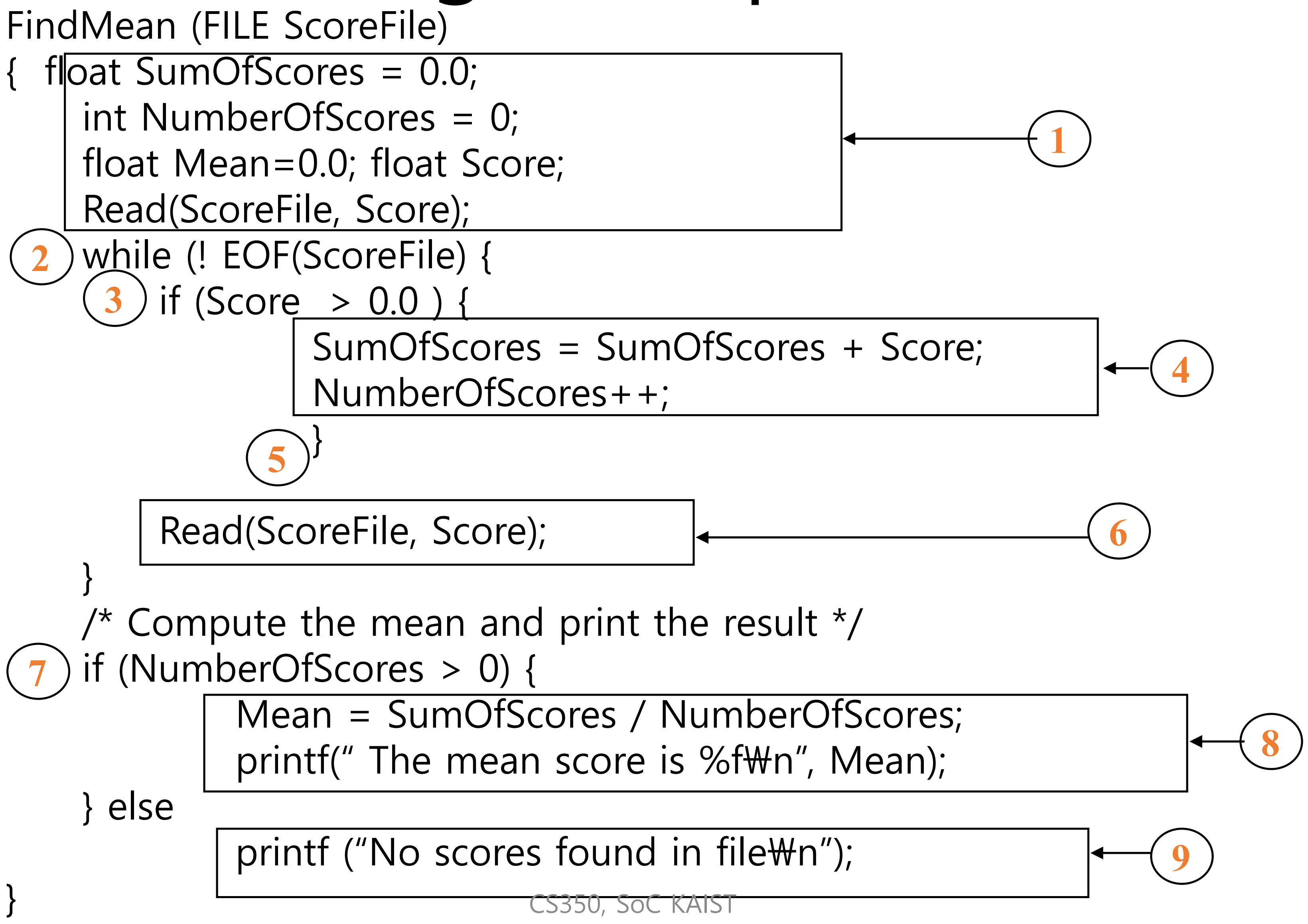
Major Coverages Comparison



White-box Testing Example: Code

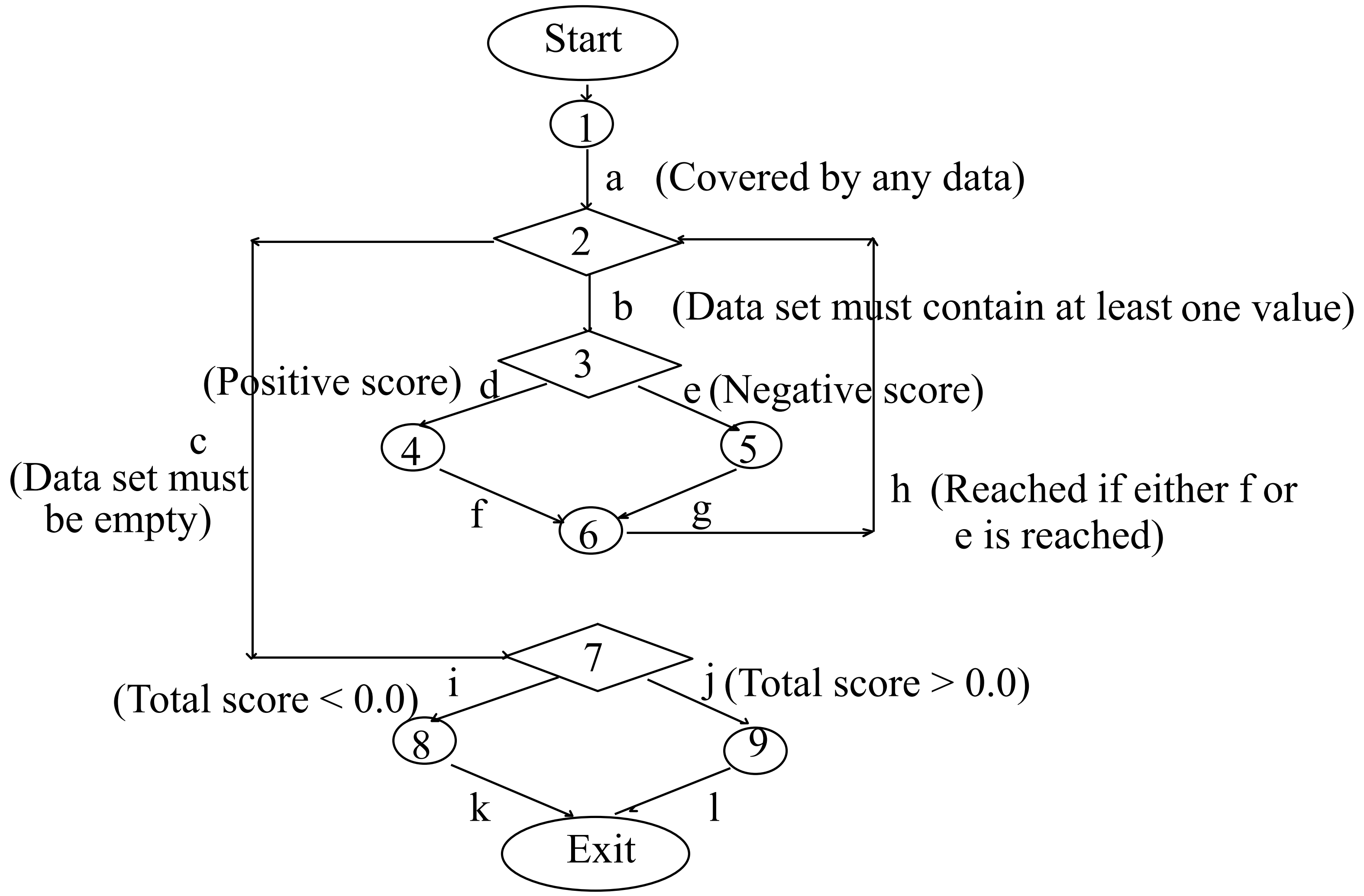
```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
                SumOfScores = SumOfScores + Score;
               NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
             Mean = SumOfScores/NumberOfScores;
             printf("The mean score is %f \n",  Mean);
 } else
             printf("No scores found in file\n");
}
```

White-box Testing Example: Control Blocks



```
FindMean (FILE ScoreFile)
{  float SumOfScores = 0.0;
   int NumberOfScores = 0;
   float Mean=0.0; float Score;
   Read(ScoreFile, Score);
   while (! EOF(ScoreFile) {
      if (Score  > 0.0 ) {
            SumOfScores = SumOfScores + Score;
            NumberOfScores++;
            }
      Read(ScoreFile, Score);
   }
   /* Compute the mean and print the result */
   if (NumberOfScores > 0) {
         Mean = SumOfScores / NumberOfScores;
         printf(" The mean score is %f₩n", Mean);
   } else
         printf ("No scores found in file₩n");
}
```

Finding the Test Cases: Control Flow Graph



Cyclomatic Complexity (McCabe)

- Originally proposed to measure the quality of source code
    - Recommended complexity: CC < 10
- Can be used to identify number of test cases for path coverage
- Give a CFG for a source code segment,
    - CC = E-N+2
        - E: number of edges
        - N: number of nodes
- Try with the example code above:
- Usage of CC for white-box testing
    - # of test cases for branch coverage,
    - # of test cases for path coverage
    - Q: relationship to CC

So far,

- We have looked at the white-box testing approaches
- Reviewed major coverages such as statement, decision(branch), path, etc

Functional Testing Heuristics

- Test data are developed from documents that specify the software's intended behavior
- Basic goal is to test each specified software feature(function)
- Heuristics
  - equivalence partitioning
  - boundary value analysis
  - special value coverage
  - output domain coverage
  - Error guessing

Equivalence (Class) Partitioning

- Divide the input space into equivalence class.
- Items in each equivalence class are treated the same in the sense that a test of any item in the class is equivalent to a test of any other item in the class
- Output space partitioning may be also considered.
- Both valid and invalid equivalence classes are considered.
- Pick at least one element from each equivalence class to test.

Guidelines for Partitioning (1/2)

- If an input condition specifies a range of values,
  - e.g. -- the item count can be from 1 to 999.
  - Identity one valid equivalent class and two invalid classes:
  - Test cases:
- If an input conditions specifies a set of values and there is a reason to believe that each is handled differently by the program, identify a valid equivalent class for each value and one invalid equivalence class.
  - e.g. --type of the variable must be integer or real.
  - Test cases:

Guidelines for Partitioning (2/2)

- If an input condition specifies a "must be" situation, identify one valid and one invalid equivalence class.
  - e.g. -- type of index must be positive integer.
  - Test cases:
- If there is any reason to believe that elements in an equivalence class are not handled in an identical manner by the program, split the equivalence class into smaller pieces.

Coverage Matrix

- Show each test case's coverage in valid equivalence classes and invalid equivalence classes.

	TC 1	TC2	TC3	“”				
V1	o							
v2		o						
V3	o	o						
…								
I1			o					
I2				o				

Heuristics for Identifying Test Cases

- Until all valid equivalence class have been covered by test cases, make a new test case covering as many of the uncovered valid equivalence classes as possible.
- Until all invalid equivalence classes have been covered by test cases, make a new test case that covers ONE and ONLY ONE of the uncovered invalid equivalence classes.

Boundary Value Analysis

- A simple method with a high payoff in finding errors.
- Focus on the elements near the boundary of the equivalence class.
- Consider boundaries in both input and output spaces.

Guidelines for Boundary Value Analysis

- If an input condition specifies a range of values,
  - e.g. [-1.0, +1.0]
  - Test the boundary:
  - Test the just outside of the boundary:
- If an input condition specifies a number of values,
  - e.g. 1--255
  - Test the maximum and minimum values:
  - Test the value one beneath the minimum and the value one beyond the maximum:
- For an ordered set, focus on the first and last element.
- Also check the output boundaries:
  - Test maximum(minimum) output.
  - Attempt to exceed maximum(minimum).
  - Test no output.

Output Range Coverage

- Based on module interface and function to be tested
- Requires expertise in the function to be tested
- Test data is choosen to cover the output space
  - typical and extreme values
  - valid and invalid values
  - For example, for an output with a specified range, inputs would be tested to produce output just inside, and jus outside the boundaries as well as interior value
  - Inputs that compute the outputs must be derived
  - Ensures that functions are tested for range of output conditions and that all possible exceptions and error messages have been produced

Special Value Coverage

- Based on the function to be tested
- Derive test data form algorithmic characteristics and well-known sensitive values
  - properties of function to be tested can aid in selecting that indicate the accuracy of the computed solution
  - for example, the periodicity of the sine function suggests use data which differs by multiples of 360 degree

Error Guessing

- Based on intuition and experience.
- Try to guess what the programmer may have overlooked.
- Make test cases for assumptions not made explicit in the spec.

- ** It is known that highly skilled and experienced testers are 40 times more efficient than novice engineers in testing.

An Example

- A payroll program is to have two inputs.
- The first input is a five digit id-number.
- The second input is a real number that is the number of hours worked
- The pay for the first 40 hours is $6.00 per hour.
- After 40 hours, the person is paid time-and-half for overtime
- Maximum hours per week is 56 hours
- The output is to be the name and gross pay for the person

Partition Input Space

- Will be done in class

Coverage Matrix

- Will be done in class

Test Cases

- Will be done in class

Additional Testing Approaches

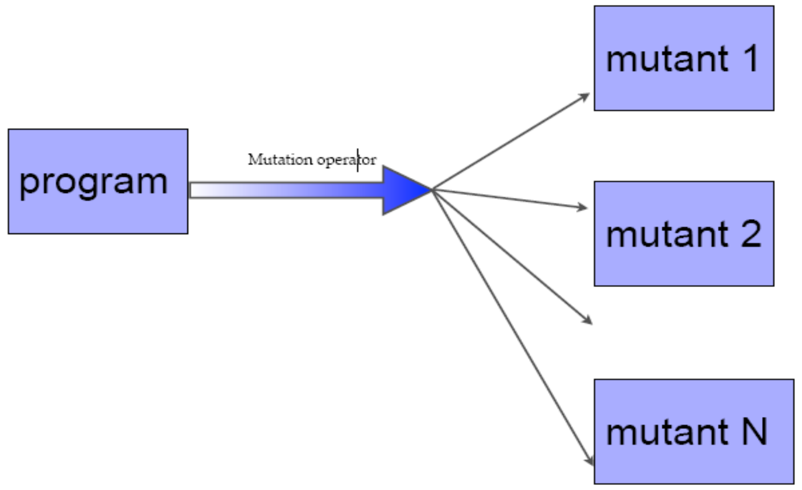
- In addition to testing approaches covered so far, such as white-, black-box testing,
- There are others such as …
  - Mutation Testing
  - Regression Testing

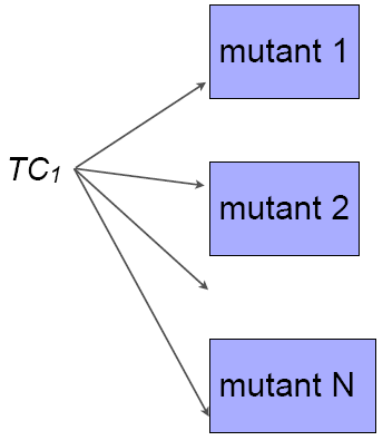
Mutation Testing

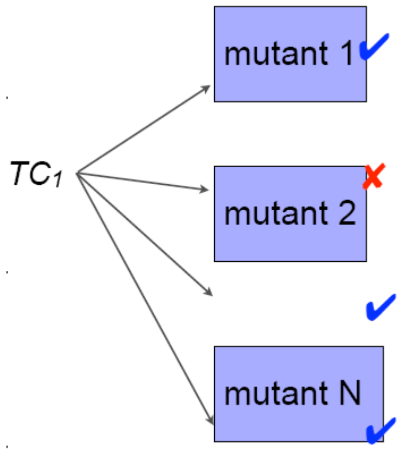
- Fault-based testing
- Goal of mutation testing is different from testing.
  - Testing is for checking source code for its quality
  - Mutation testing is for checking test (data) set(called test suite) for its quality

Mutation Testing

- *Mutant:* a copy of the original program with a small change (seeded fault)
- *Mutant killed:* if its behaviors/outputs differ from those of the original program
- *Mutant score:* number of killed mutants

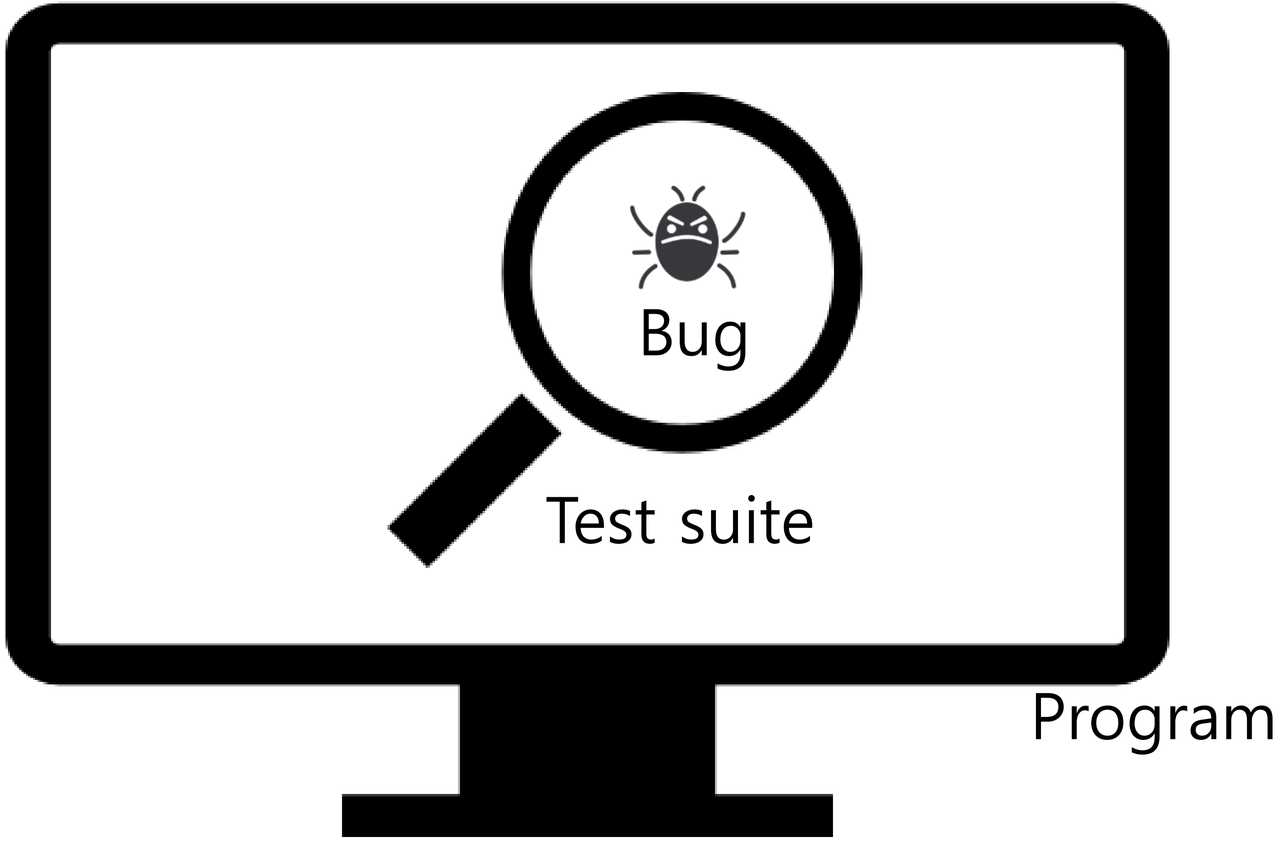






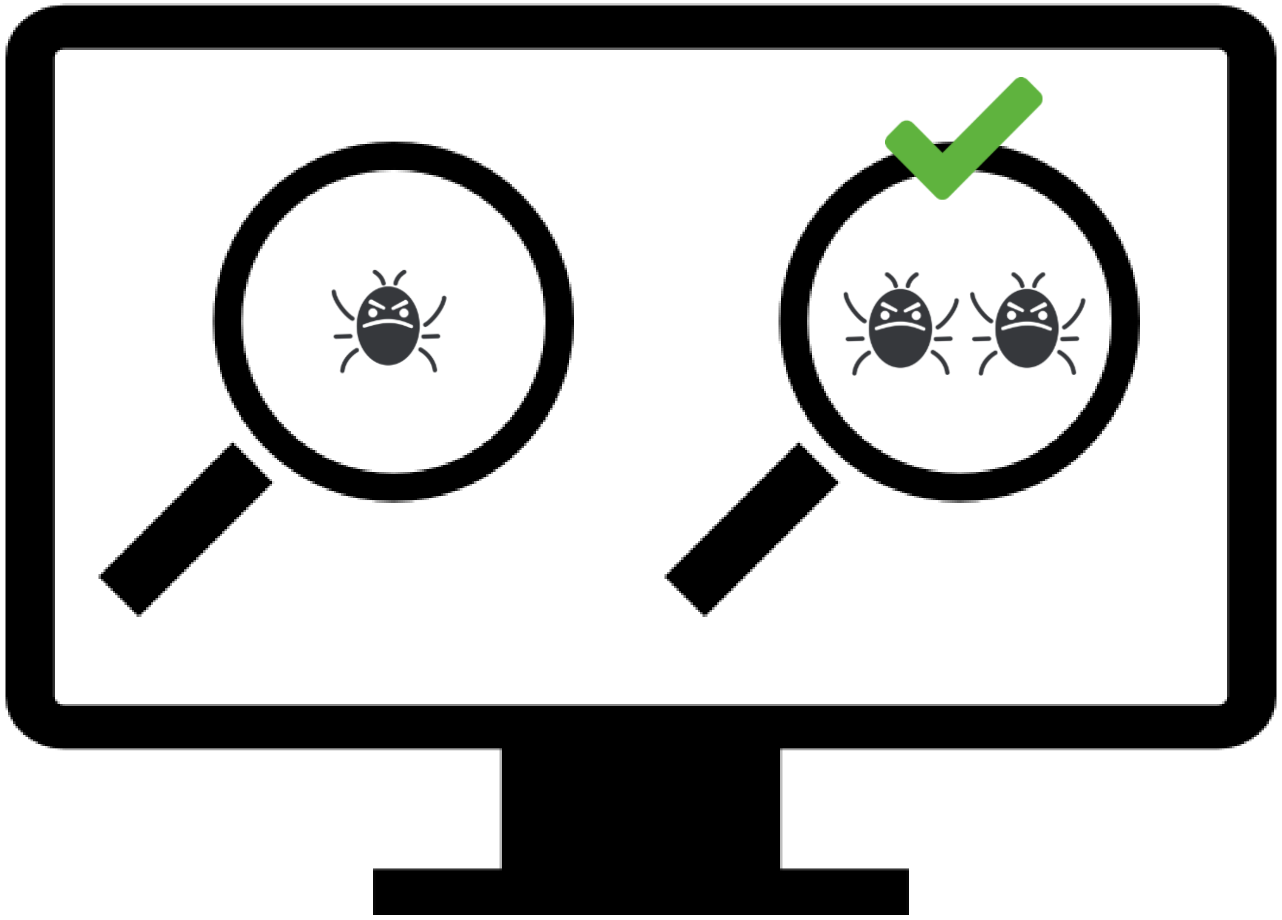
Testing: Find Bugs Using Test Suites
(Mutation Testing: prepared by Shin, Donghwan)

- We can find bugs by **executing** test suites.



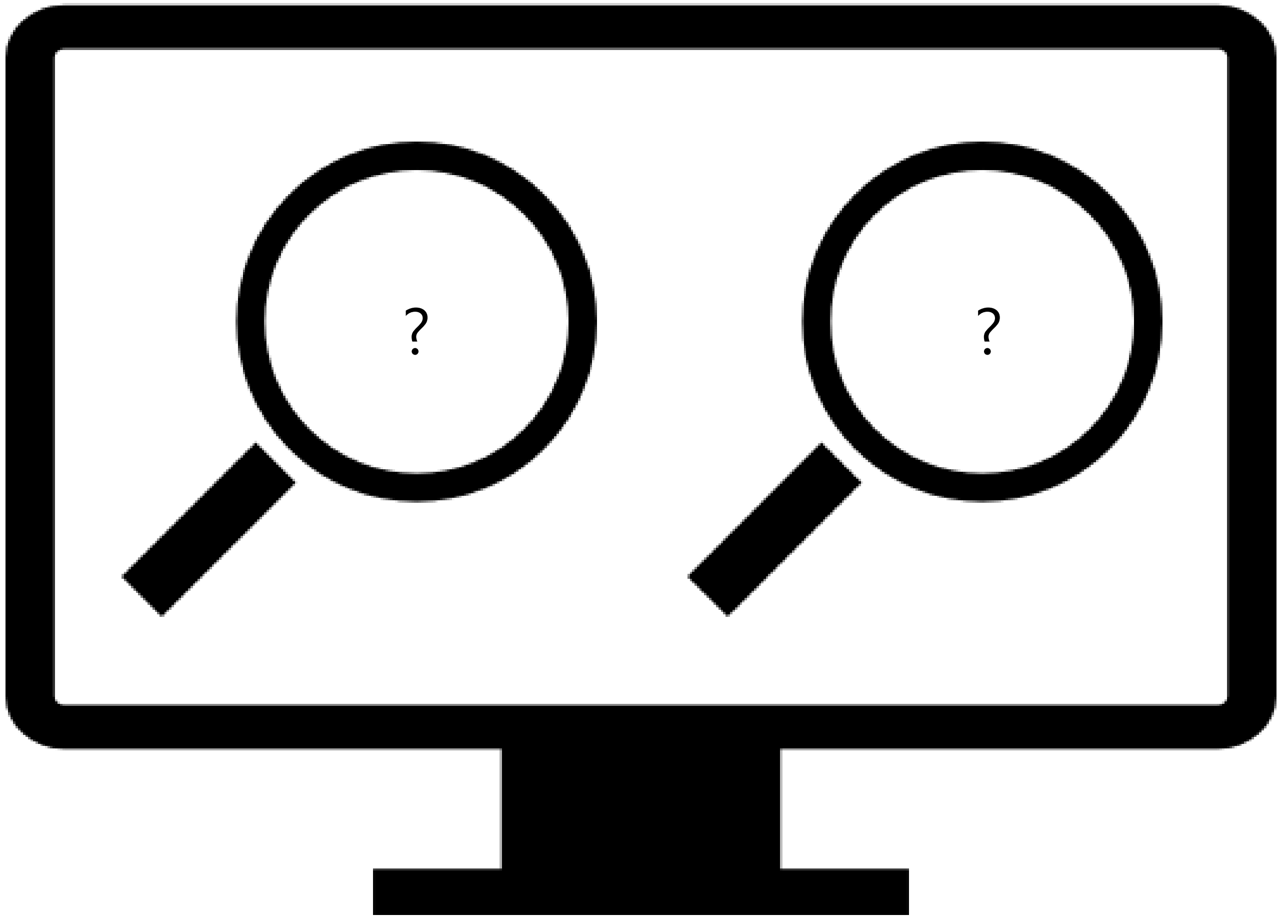
Whose Bug Detection Capability Is Higher?

- It is trivial when we already know the bugs.



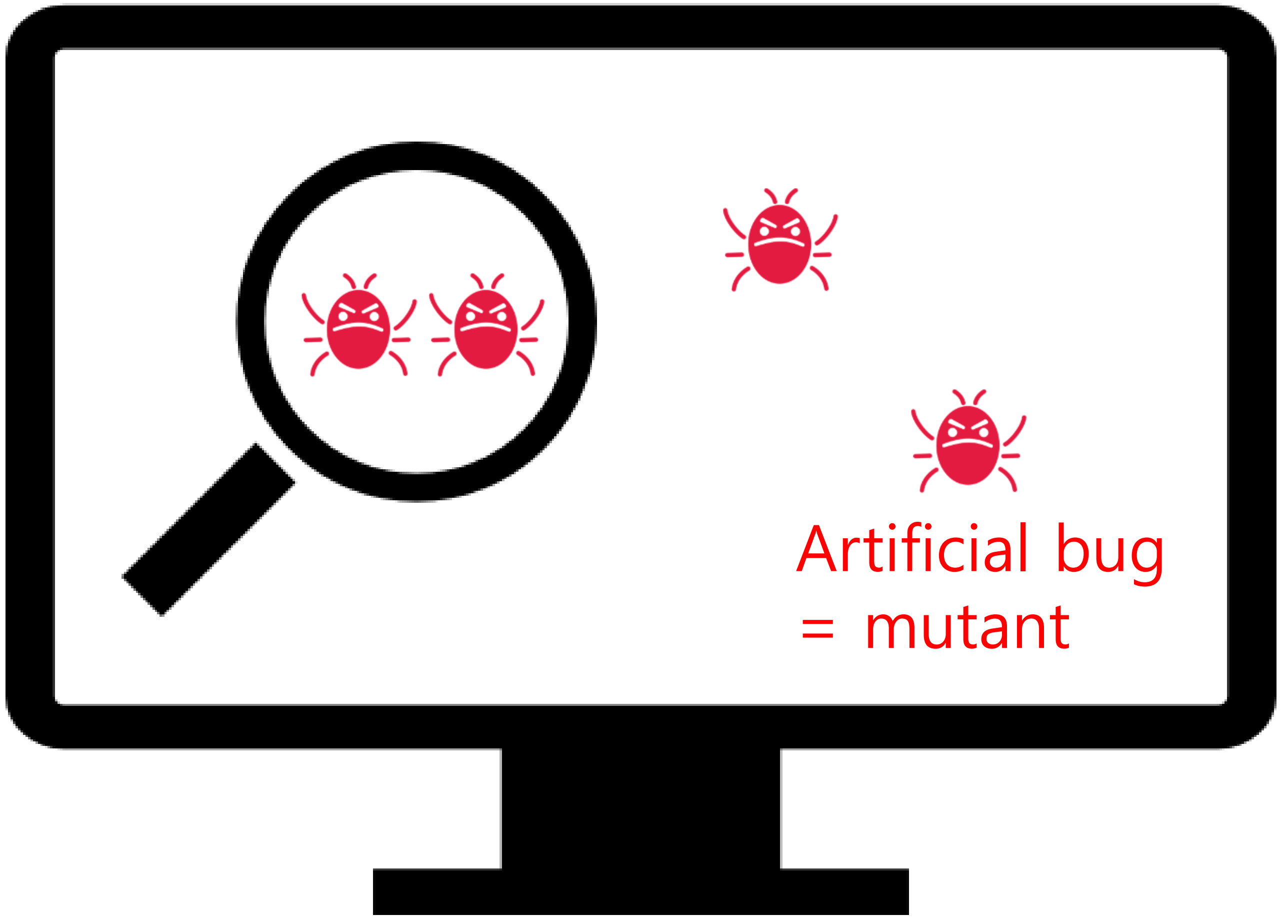
Can We Guess Before Finding Bugs?

- We cannot simply answer which one is better.



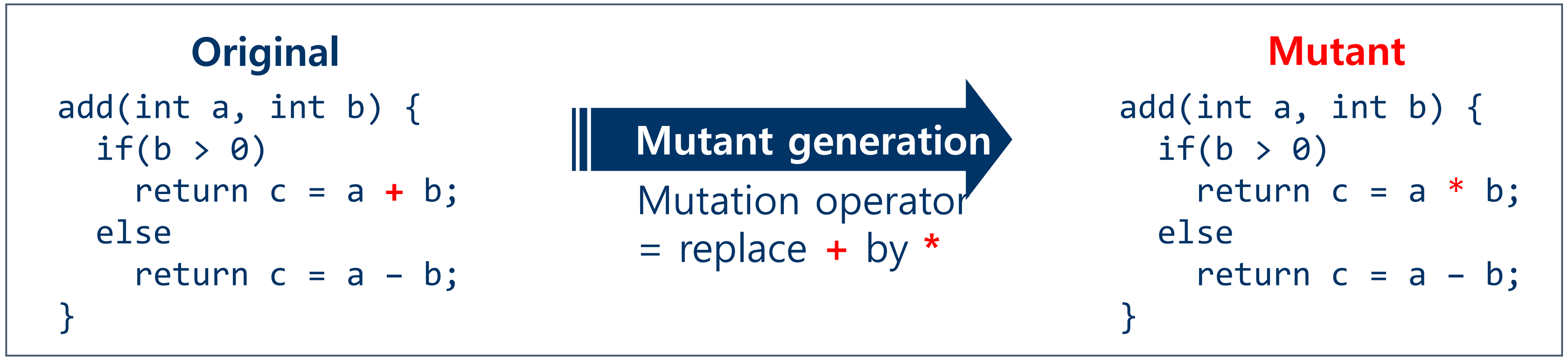
Insight: Inject “artificial” Bugs

- The more detect mutants, the more detect real bugs.



How To Systematically Inject Mutants

- From an original artifact, mutants are generated by applying pre-defined syntactic change rules called mutation operators.



Original

```
add(int a, int b) {
  if(b > 0)
    return c = a + b;
  else
    return c = a – b;
}
```

Mutant generation

Mutation operator = replace + by *

Mutant

```
add(int a, int b) {
  if(b > 0)
    return c = a * b;
  else
    return c = a – b;
}
```

- ✓ A test data [a=2, b=1] returns [3] for the original but [2] for the mutant.
- ✓ In this case, we say that the mutant is detected (or killed) by the test data.
- ✓ % of killed mutants for a test suite → its fault detection capability

Tools For Mutation Testing In Practice

- MuJava (https://cs.gmu.edu/~offutt/mujava/)
  - Well-known mutation testing tool for Java in research
- PIT (http://pitest.org/)
  - Real world mutation testing tool
  - Provide Eclipse / intelliJ plugins
- Major (http://mutation-testing.org/)
  - Recently developed high-performance tool
  - Embedded in Defects4J

Finally! A Few Words More…

- My old professor’s last lecture in his class many years ago.
- Be proud of yourself; who/where you are now.
- Become a global leader;
- Invest to yourself
- Read many books