Lesson-13:Software testing

CS 438

Ali Aburas PhD

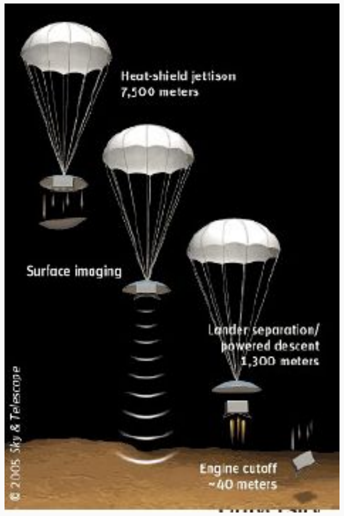
Today's Goals

- Software testing
- Unit Testing

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTERNOON!

Why Does Testing Matter?

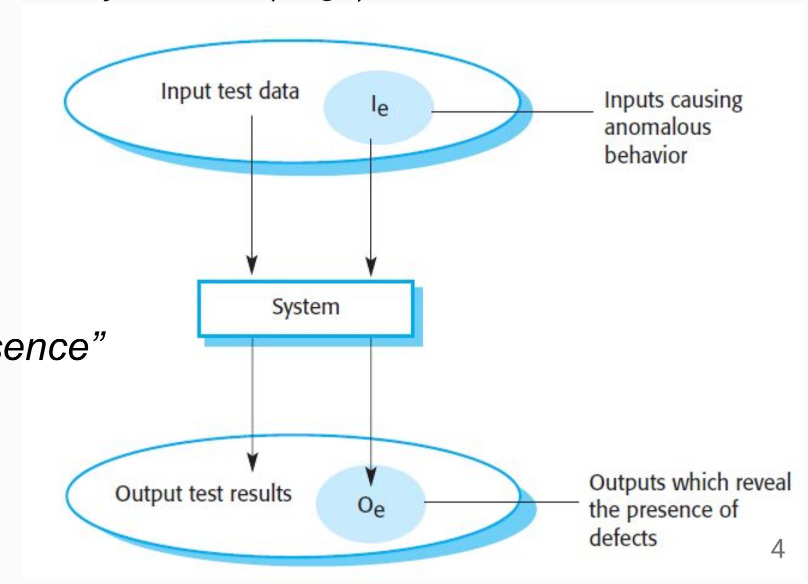
- It is important to understand the consequences that some failures might have.
- **Ariane 5(link):**
  - The rocket self-destructed 37 seconds after launch
  - Reason: conversion from 64-bit floating to 16-bit signed value has caused an exception.
  - Total cost: over US$362 million
- **Mars Polar Lander(link)**
  - Sensor signal falsely indicated that the craft touched down (130 feet above the surface)
  - Reason: The software that controlled the orbiter thrusters use imperial units (e.g., foot, inch), rather than metrics units (e.g., meter, cm) as specified by NASA
  - Total cost: US$165 million



Software Testing

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
    - When you test software, you execute a program using artificial data.
- When you test software, you are trying to do two things:
    1. Demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document.
    2. Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are caused by defects (bugs) in the software.

- Testing **cannot** demonstrate that the **software is free of defects** or that it will **behave as specified in every circumstance**.


Input test data
I_e
Inputs causing anomalous behavior
System
Output test results
O_e
Outputs which reveal the presence of defects

“Testing can only show the presence of errors, not their absence”

Edsger Dijkstra

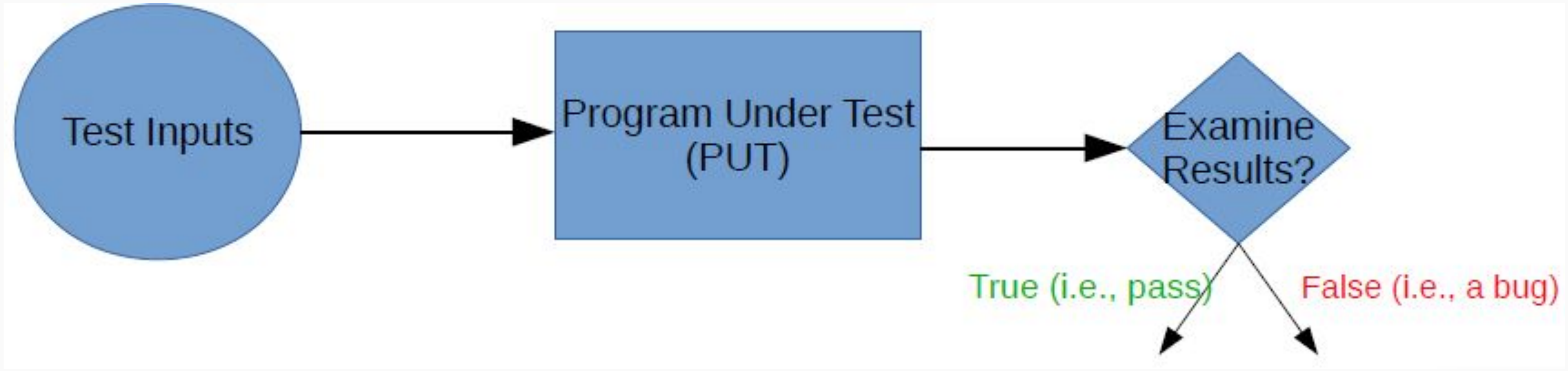
Bugs? What are Those?

- Bug is an overloaded term - does it refer to the bad behavior observed, the source code problem that led to that behavior, or both?
- **Failure**
    - An execution that yields an incorrect result.
- **Fault**
    - The problem that is the source of that failure.
    - For instance, a typo in a line of the source code.
- When we observe a failure, we try to find the fault that caused it.
- The main purpose of testing is to find faults:

How is testing done?

Basic steps of a test (i.e., unit test)

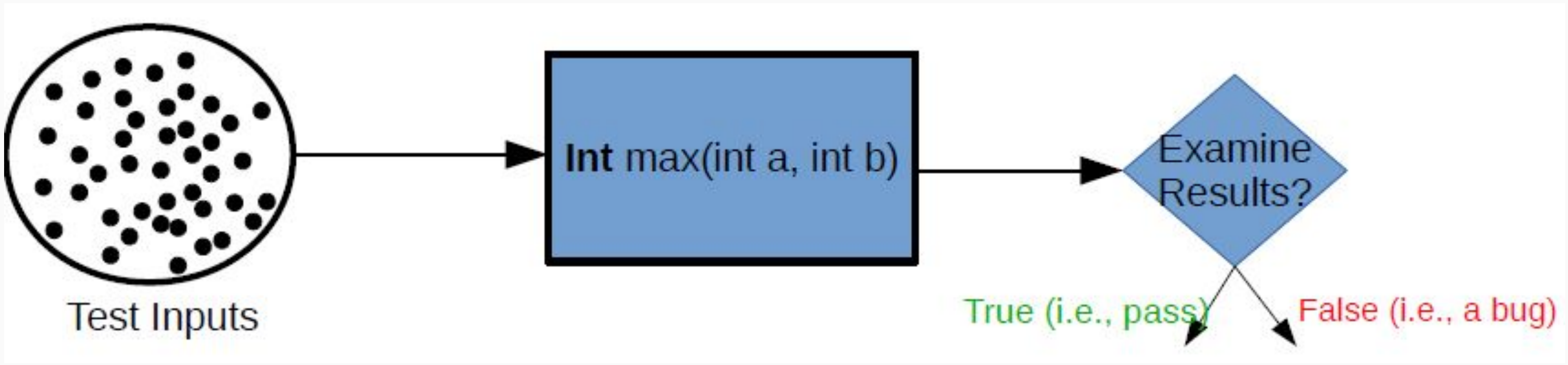
1) Choose input data (**Test Input**)
2) Define the expected outcome (**Test Oracle**)
3) Run program under test against the input and record the results
4) Examine results against the expected outcome



Why is So Hard about Testing?

Example: Let's consider a program under test (PUT) that takes two integer and returns the maximum value.

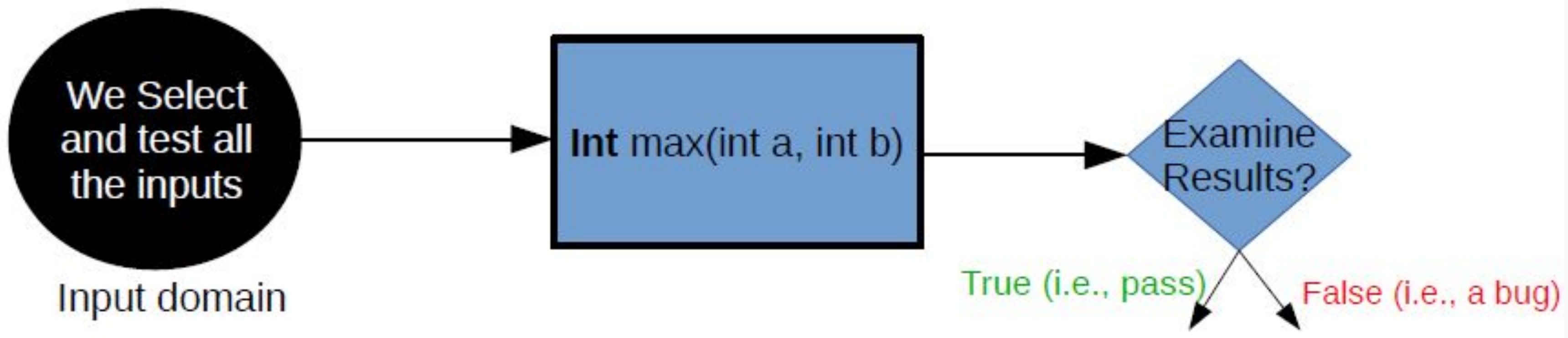
```
int max(int a, int b)
// effects: a > b => returns a
// a < b => returns b
// a = b => returns a
```



The question is: How can we select a set of inputs from input domain (i.e., test inputs) for our PUT that after we run them, we have **enough confidence** that the PUT is implemented correctly?

Choice #1: Exhaustive Testing

First approach: we select all the inputs (i.e., do exhaustive testing)

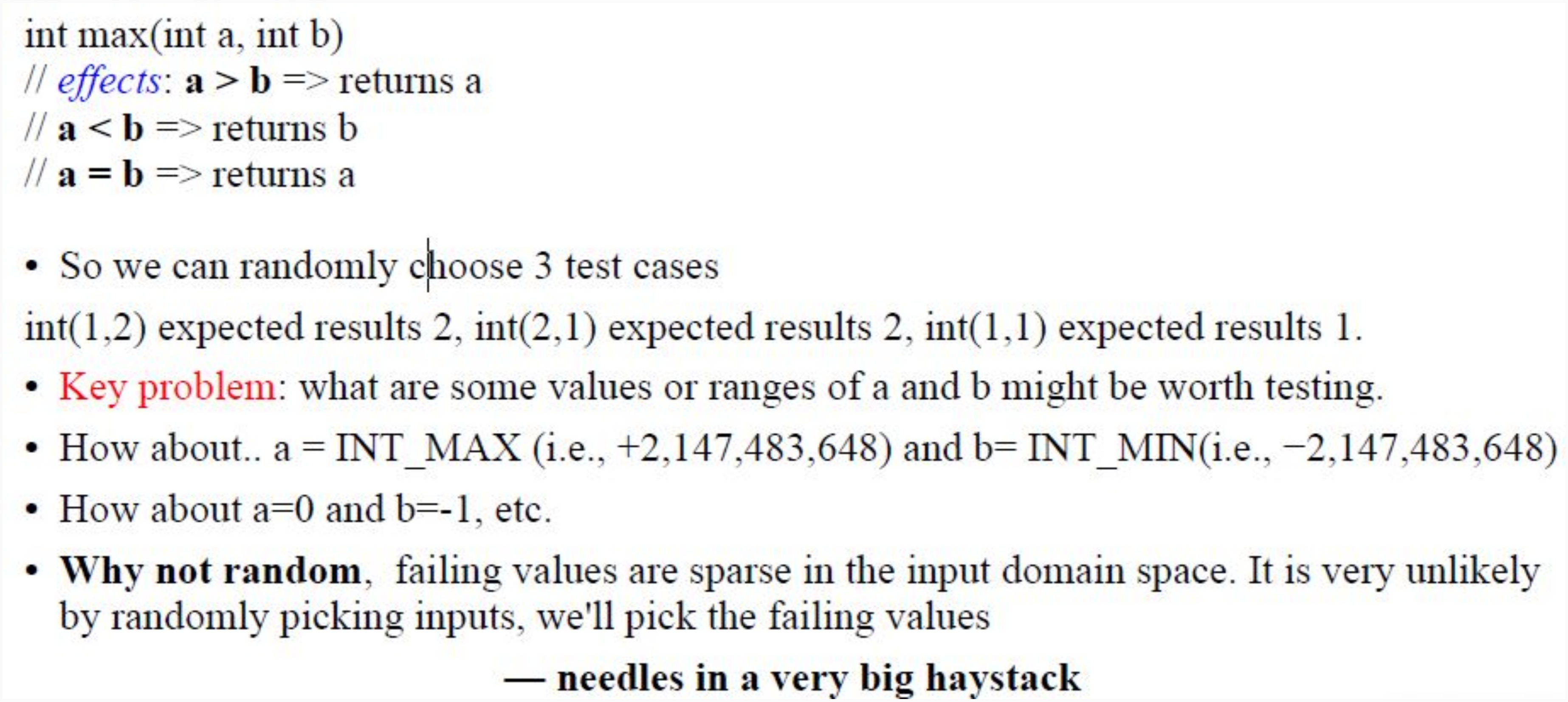


- If **a** and b are 32bit integers, we'll have a total number of combinations, which is $2^{32} \times 2^{32} = 2^{64} \simeq 10^{19}$
- So we need to run 10^{19} test cases to cover the whole input domain for the **PUT** int max(int a, int b)
- Exhaustive testing would require hundreds of years to cover all possible inputs.

Sounds totally impractical – and this is a trivial small problem

Choice #2: Random Testing

- **Second approach**: choose our test inputs randomly (i.e., do random testing)



```
int max(int a, int b)
// effects: a > b => returns a
// a < b => returns b
// a = b => returns a
```

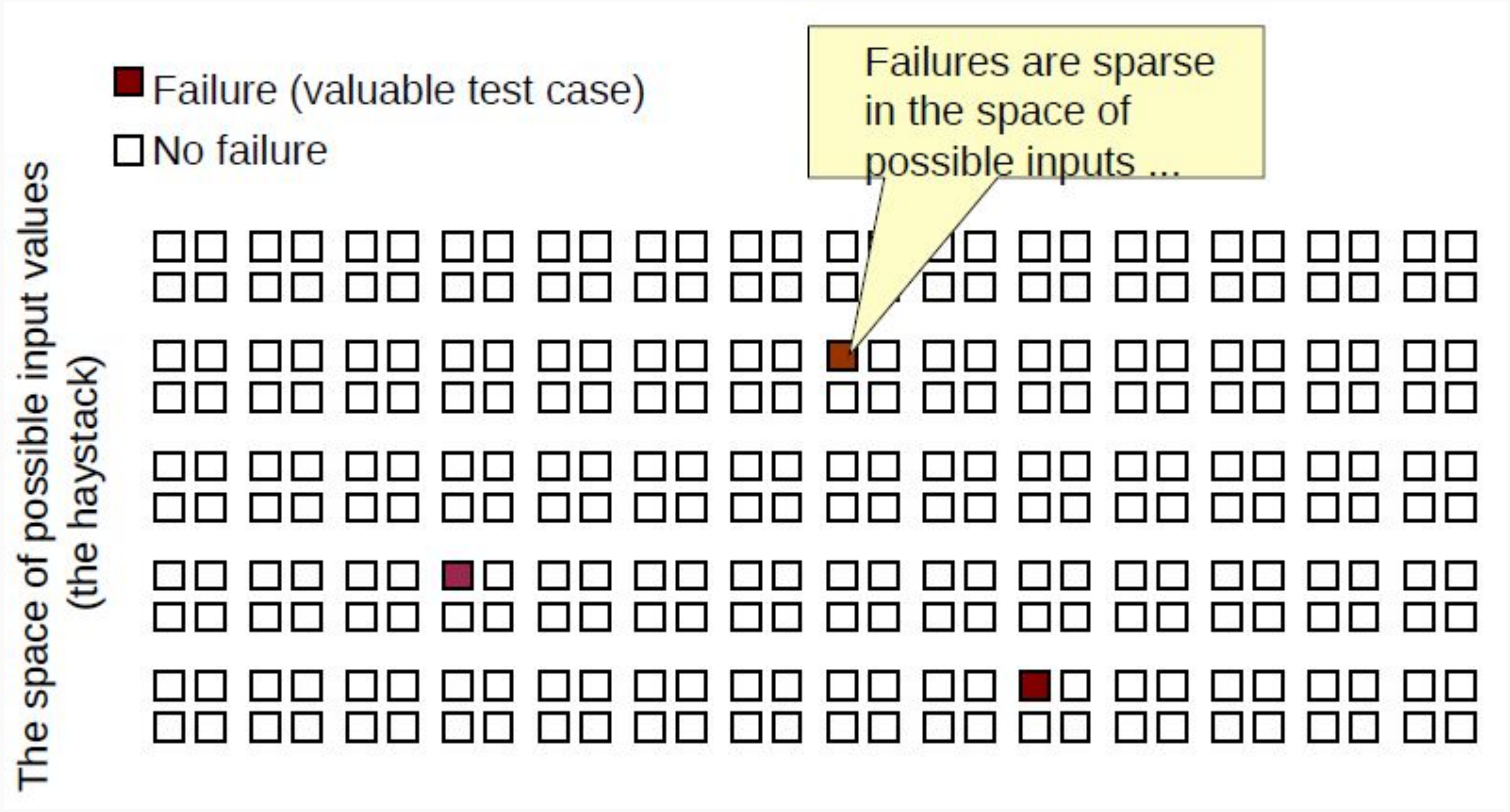
- So we can randomly choose 3 test cases

int(1,2) expected results 2, int(2,1) expected results 2, int(1,1) expected results 1.

- Key problem: what are some values or ranges of a and b might be worth testing.
- How about.. a = INT_MAX (i.e., +2,147,483,648) and b= INT_MIN(i.e., −2,147,483,648)
- How about a=0 and b=-1, etc.
- **Why not random**, failing values are sparse in the input domain space. It is very unlikely by randomly picking inputs, we'll pick the failing values

— needles in a very big haystack

Why not RANDOM?



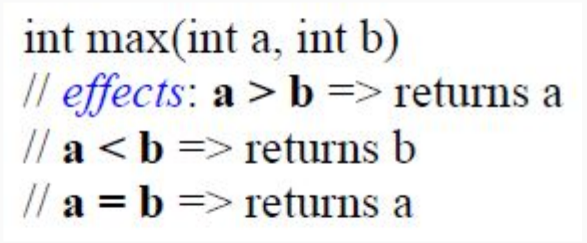
■ Failure (valuable test case)
□ No failure
Failures are sparse in the space of possible inputs ...
The space of possible input values (the haystack)

Black and White Box Testing

- **Black Box (Functional) Testing**
  - Designed without knowledge of the program’s internal structure and design.
  - Based on functional and non-functional requirement specifications.
- **White Box (Structural) Testing**
  - Examines the internal design of the program.
  - Requires detailed knowledge of its structure.
  - Tests typically based on coverage of the source code (all statements /conditions /branches have been executed)

Black Box Testing

- Choosing tests (test input) based on partitioning the input domain.
    - Identify sets (partitions) with same behavior
    - Try one input from each set
- “Just try it and see if it works for int max(int a, int b) //which is a simple program to find maximum of two numbers”

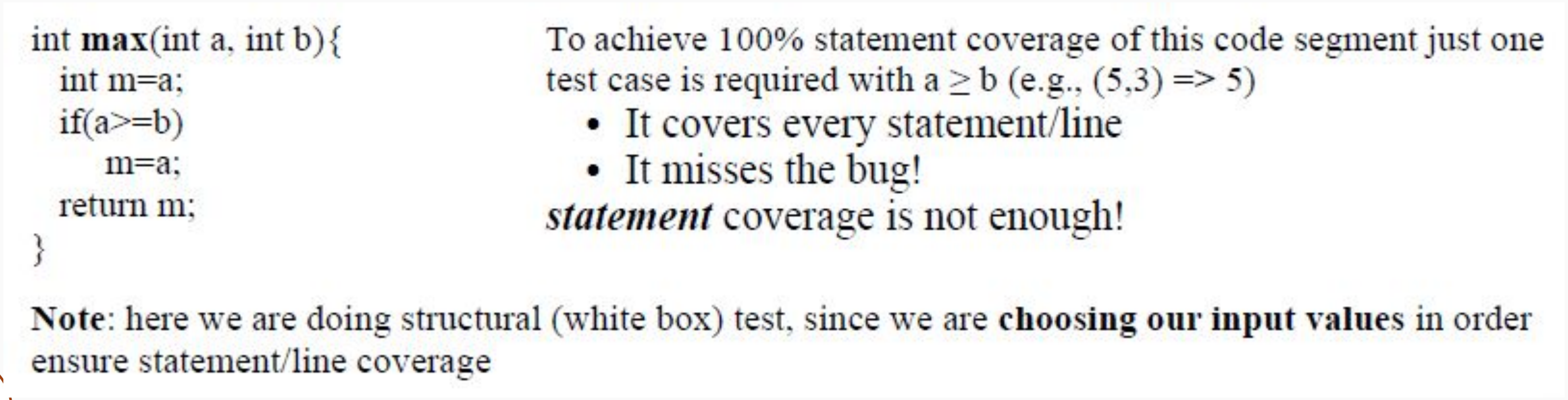


```
int max(int a, int b)
// effects: a > b => returns a
// a < b => returns b
// a = b => returns a
```

- **Some possible partitions**:
    - a > 0 b>0
    - a <0 b <0
    - a = 0 b =0
- Boundary Testing: create tests at the edges or extreme ends of partitions input values
    - a=INT_MIN b=INT_MIN (boundary values)
    - a=INT_MAX b=INT_MAX (boundary values)

Structural (White-Box) Testing

- White Box Testing requires two basic steps
  1. Understand the source code
  2. Create test cases and execute
- The goals
  1. Ensure test cases (test suites) cover (executes) all the program
  2. Measure quality of test cases (test suites) with % coverage
- Varieties of coverage
  - For example, Statement coverage, Branch Coverage, Path Coverage
- What is full Coverage?



```
int max(int a, int b){
  int m=a;
  if(a>=b)
     m=a;
  return m;
}
```

To achieve 100% statement coverage of this code segment just one test case is required with a $\geq$ b (e.g., (5,3) => 5)

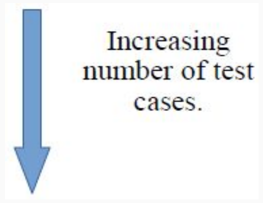
- It covers every statement/line
- It misses the bug!

statement coverage is not enough!

Note: here we are doing structural (white box) test, since we are **choosing our input values** in order ensure statement/line coverage

Structural (White-Box) Testing

- Covering all the program under test (PUT)
  - Statement coverage
  - Branch coverage
  - Path Coverage
  - Condition Coverage (Predicate Coverage)



- 100% coverage is not always a reasonable target
  - There might be a dead code (indefeasible code)
- Two purposes:
  - to know what we have & haven't tested, and
  - to know when we can "safely" stop testing.

Software Testing Levels

- There are generally four levels of tests:
  - **Unit testing**
    - Does each unit work as specified?
  - **Integration testing**
    - Do the units work when put together?
  - **System testing**
    - Does the system work as a whole?
  - **Acceptance Testing**
    - which is the validation of the software against the customer requirements.
- **Regression Testing**
  - We perform regression testing every time that we change our system
  - We need to make sure that the changes behave as it is intended and the unchanged code is not negatively affected by these changes/modifications

Software Testing Levels

- There are generally four levels of tests:
    - **Unit testing**
        - Does each unit work as specified?
    - **Integration testing**
        - Do the units work when put together?
    - **System testing**
        - Does the system work as a whole?
    - **Acceptance Testing**
        - which is the validation of the software against the customer requirements.
- **Regression Testing**
    - We perform regression testing every time that we change our system
    - We need to make sure that the changes behave as it is intended and the unchanged code is not negatively affected by these changes/modifications

Key focus in 438: unit testing

UNIT TESTING

- A unit is the smallest testable part of the software system (e.g., a method in a Java class).
- Goal: Verify that each software unit performs as specified.
- Focus:
  - Individual units (not the interactions between units).
  - Usually input/output relationships.

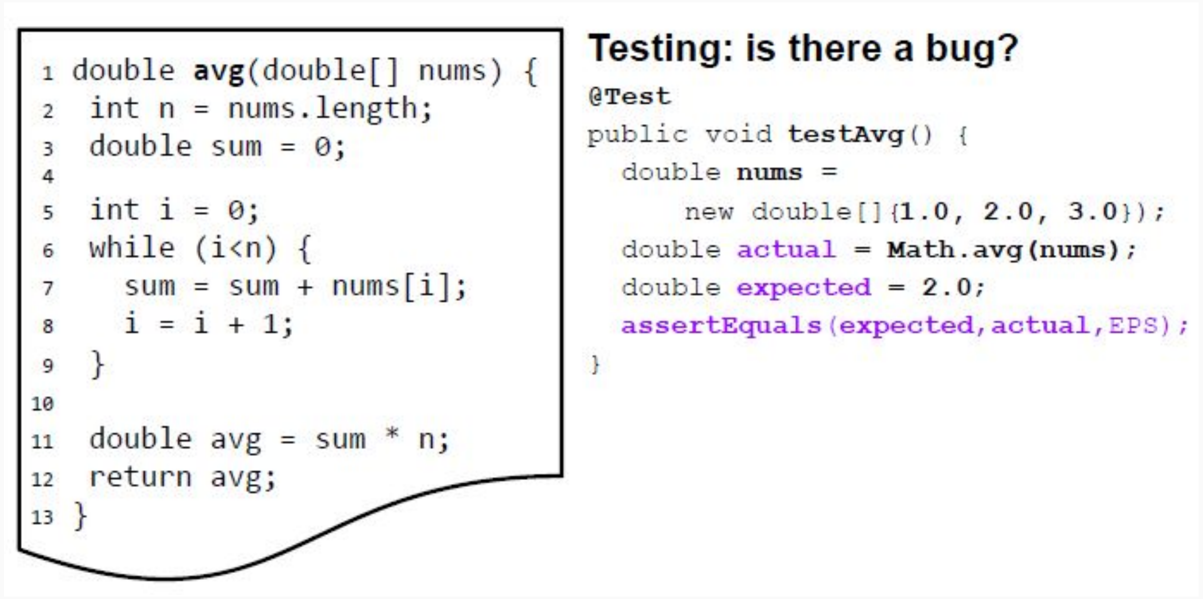
Writing and executing a Unit Test

- How do you run test cases on the program?
  - You could run the code and check results by hand (manual testing).
  - **Please don’t do this.**
  - Manual testing is slow, expensive and error-prone!
- **Test Automation** is the development of software to separate repetitive tasks from the creative aspects of testing.
  - Automation allows control over how and when tests are executed.

- Test-automation infrastructure (Tools):
  - Java JUnit,
  - Mocha,
  - Python Pytest,
  - .Net xUnit,
  - C/C++ gunit.

Writing a Unit Test

○ Java JUnit,

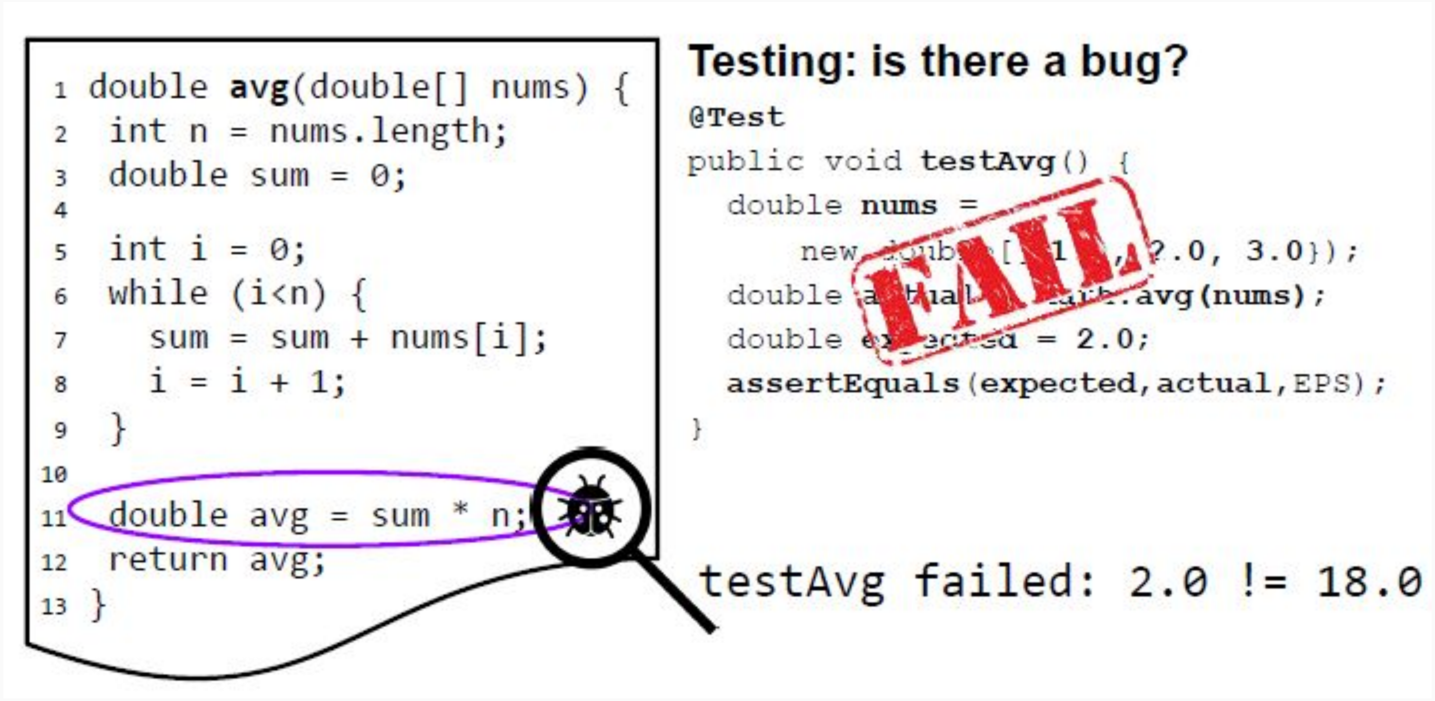


```
double avg(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i<n) {
   sum = sum + nums[i];
   i = i + 1;
 }

 double avg = sum * n;
 return avg;
}
```

Testing: is there a bug?

```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = Math.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```

Writing a Unit Test



```
double avg(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i<n) {
   sum = sum + nums[i];
   i = i + 1;
 }

 double avg = sum * n;
 return avg;
}
```

Testing: is there a bug?

```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = Arith.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```

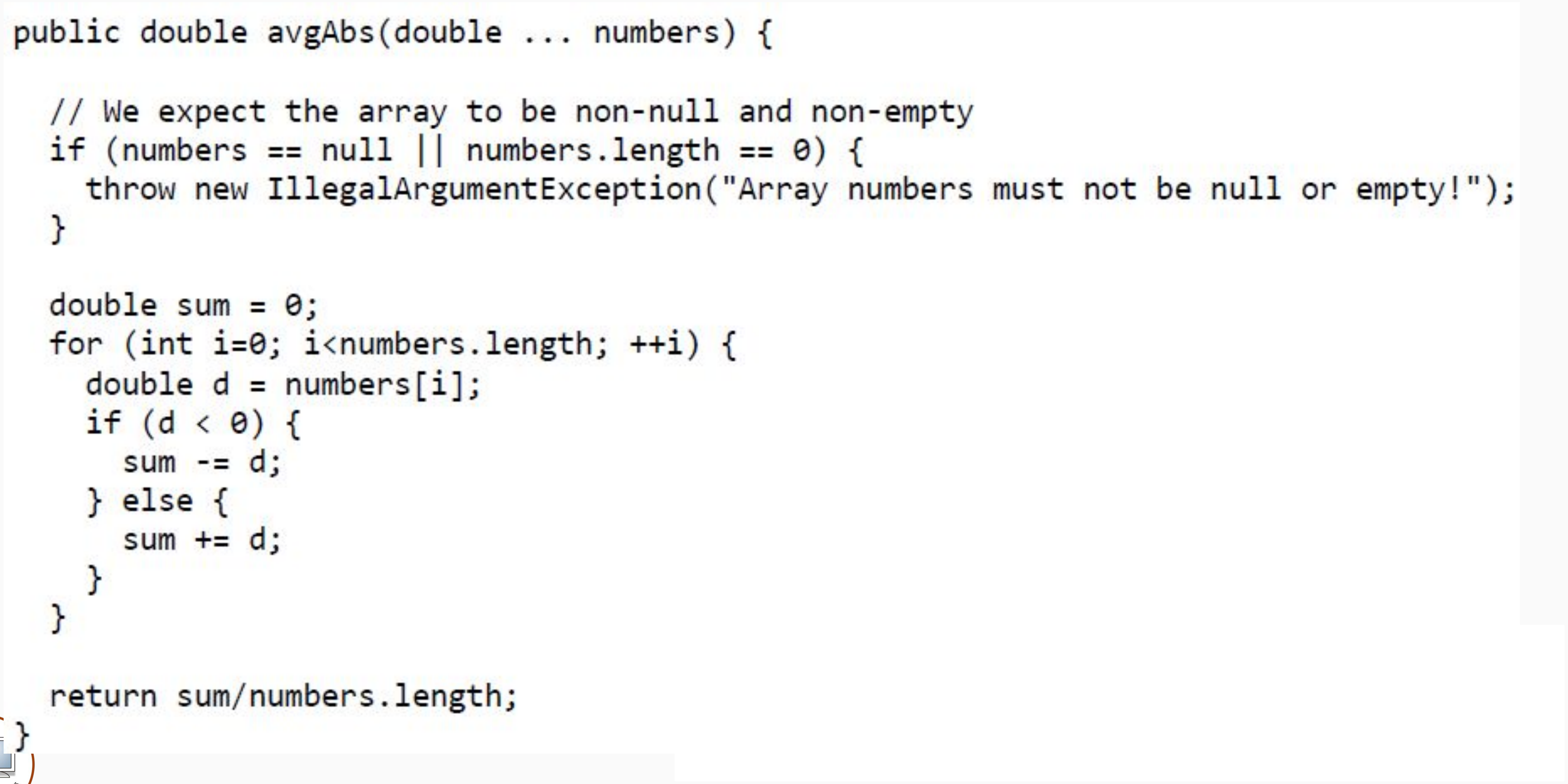
```
testAvg failed: 2.0 != 18.0
```

- **Debugging:**
  - **where is the bug?**
  - **how to fix the bug?**

How to spend less time debugging

Method: Average of the absolute values of an array of doubles

What tests should we write for this method?

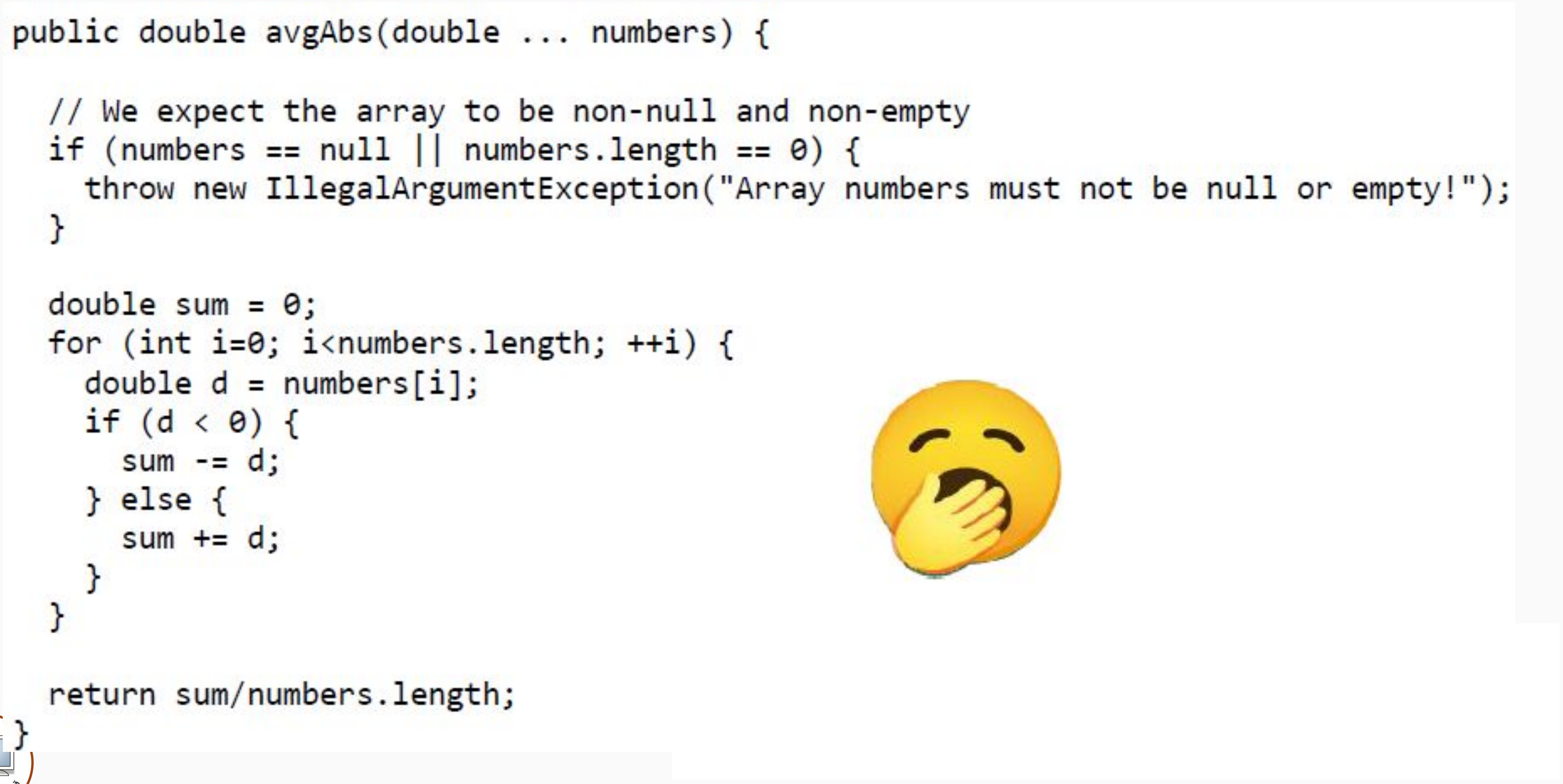


```
public double avgAbs(double ... numbers) {

  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException("Array numbers must not be null or empty!");
  }

  double sum = 0;
  for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
      sum -= d;
    } else {
      sum += d;
    }
  }

  return sum/numbers.length;
}
```

How to spend less time debugging

Method: Average of the absolute values of an array of doubles

What tests should we write for this method?

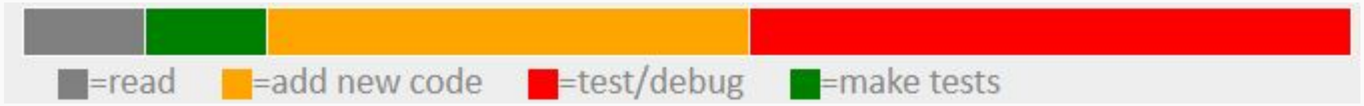


```
public double avgAbs(double ... numbers) {

  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException("Array numbers must not be null or empty!");
  }

  double sum = 0;
  for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
      sum -= d;
    } else {
      sum += d;
    }
  }

  return sum/numbers.length;
}
```

How to spend less time debugging

First approach:

1. You read the description, think it through, and then code up the entire solution.
2. Then, you write all the tests at once.
3. When you run the tests, there are dozens of compiler warnings and errors.
4. After solving those, you find that none of the code works.
5. You debug one issue after another until finally, everything works



How to spend less time debugging

Second approach: TDD (Test Driven-Development)

Assignment-5 (TDD)

Example: Unit Testing