Testing Fundamentals

CS 438
Ali Aburas PhD

Program 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.
  - You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- **Testing can reveal the presence of errors NOT their absence.**
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

Program testing goals

- To 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.
  - For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.

Verification and Validation

- **Verification**: "Are we building the product right".
  - The process of proving that the software conforms to its specified functional and non-functional requirements.
    - The software should conform to its specification.
- **Validation**: "Are we building the right product".
  - The process of proving that the software meets the customer's true requirements, needs, and expectations.
    - The software should do what the user really requires.

These two sound similar, but they're a little different.

Verification and Validation

Which is more important?

- Both are important.
  - A well-verified system might not meet the user's needs.
  - A system can't meet the user's needs unless it is well-constructed.

When do you perform V&V?

- Constantly, throughout development.
  - Verification requires specifications, but can begin then and be executed throughout development.
  - Validation can start at any time by seeking feedback.

Inspections and testing

- **Software inspections** Concerned with analysis of the static system representation to discover problems (**static verification**)
  - May be supplement by tool-based document and code analysis.
- **Software testing** Concerned with exercising and observing product behaviour (**dynamic verification**)
  - The system is executed with test data and its operational behaviour is observed.

Development testing

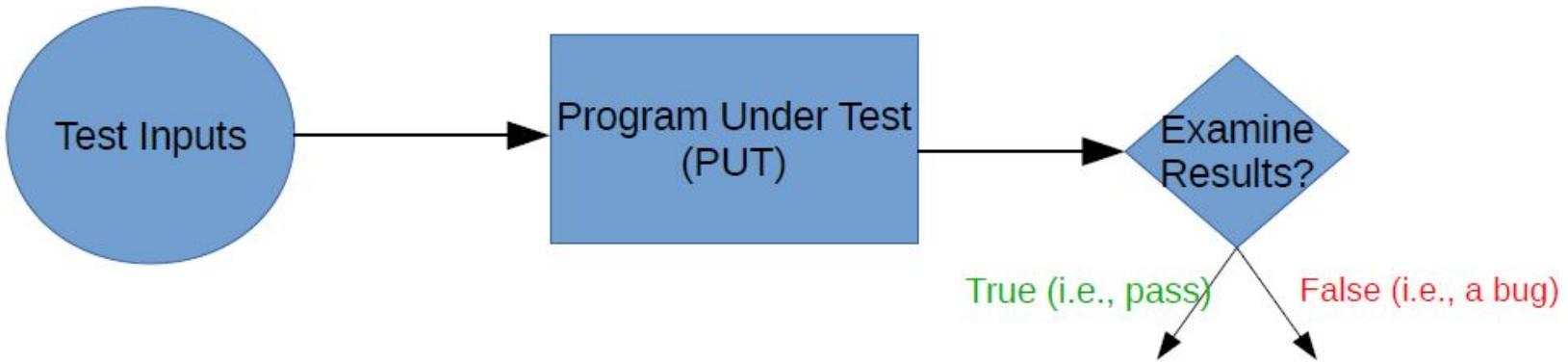
Software Testing Levels

- Development testing includes all testing activities that are carried out by the team developing the system.
- There are generally four levels of tests:
  - **Unit testing:** Does each unit work as specified? **Key focus in CS438: unit testing**
    - where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - **Integration testing (Component testing):** Do the units work when put together?
    - where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces
  - **System testing:** Does the system work as a whole?
    - where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.
  - **Acceptance Testing:** which is the validation of the software against the customer requirements.
    - Give product to a set of users to check whether it meets their needs.
    - Also called alpha/beta testing.
  - **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

How is testing done?

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

1. Choose input data
2. Define the expected outcome (oracles)
3. Run program under test against the input and record the results
4. Examine results against the expected outcome



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.

A Real-world Example

You have written a function to add two numbers:

- int Max(int a, int b) {

```
if a>=b
    return a;
else
    return b;
}
```

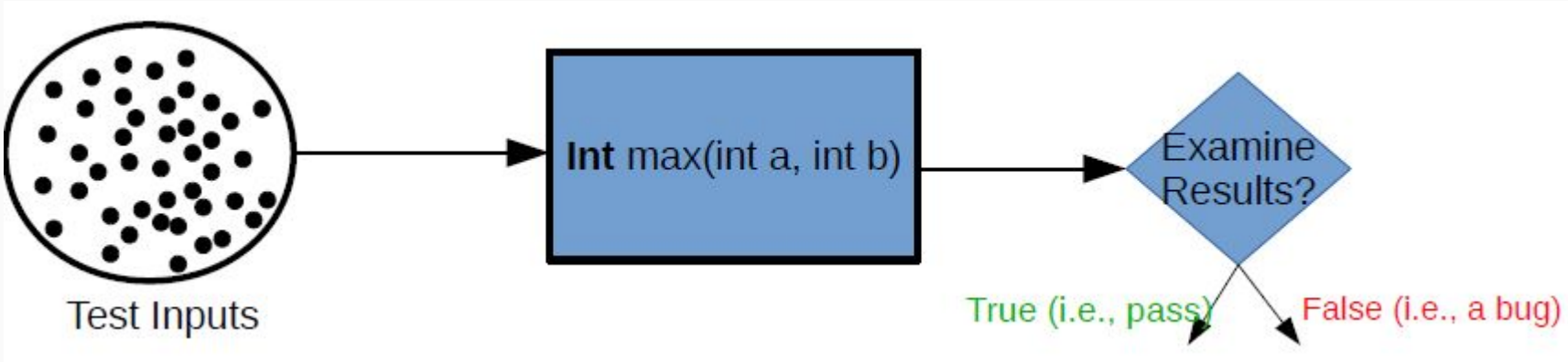
A unit test code would look something like this:

- void TestMax1() { Assert.IsEqual(Max(5, 10), 10) }
- void TestMax2() { Assert.IsEqual(Max(500, 1000), 1000) }
- void TestMax3() { Assert.IsEqual(Max(0, 1000), 1000) }
- void TestMax4() { Assert.IsEqual(Max(-100, 100), 100) }
- void TestMax5() { Assert.IsEqual(Max(-100, -1100), -100) }

Why is So Hard about Testing?

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

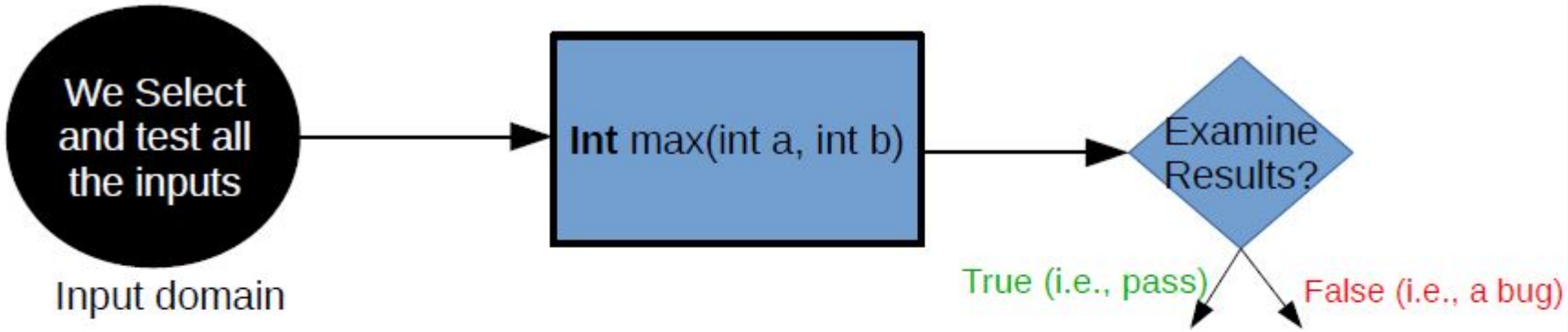
```
int max(int a, int b)
// effects: a > b => returns a
// a < b => returns b
// a = b => returns a
```



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
```

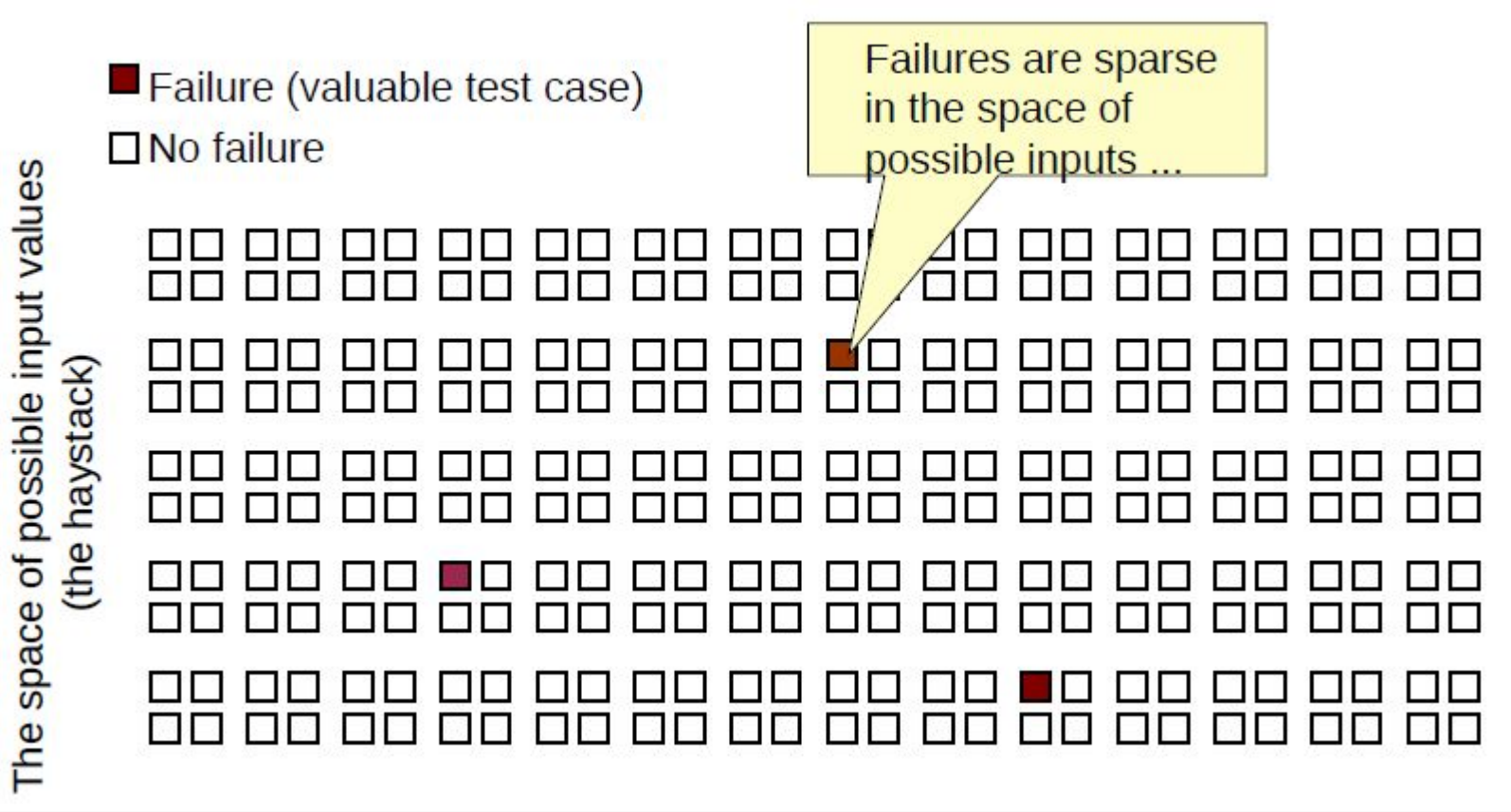
- So we can randomly choose 3 test cases

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?



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)

Choice #3: Black Box (Functional) Testing

- Choosing tests without knowledge of the program's internal structure and design.
- 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)

Choice #4: White-Box (Structural) 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 ≥ b (e.g., (5,3) => 5)

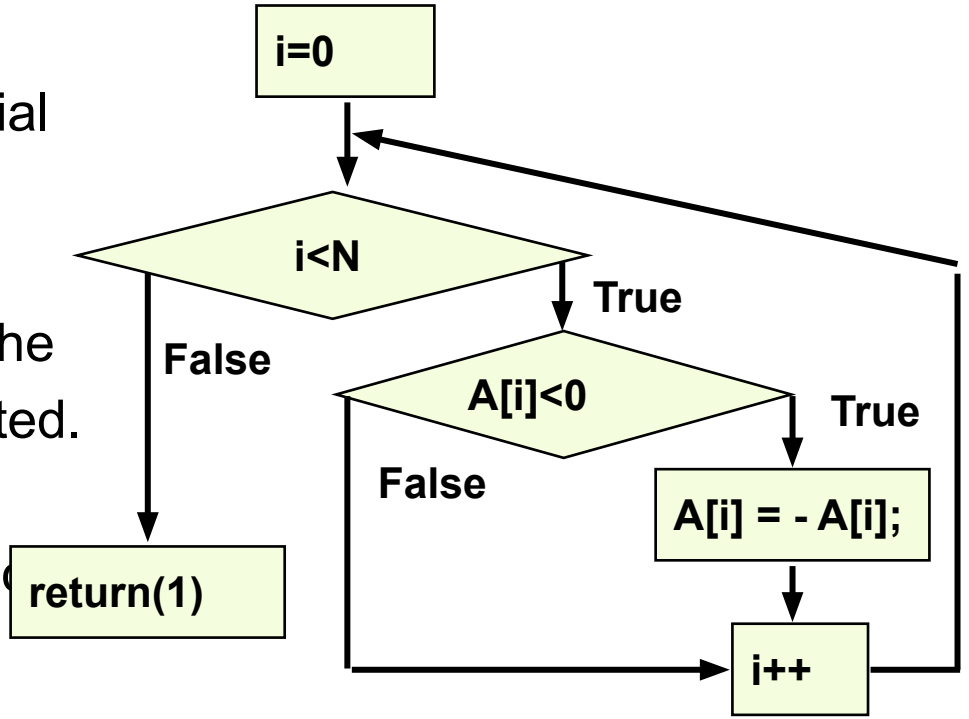
- It covers every statement/line
- It misses the bug!

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

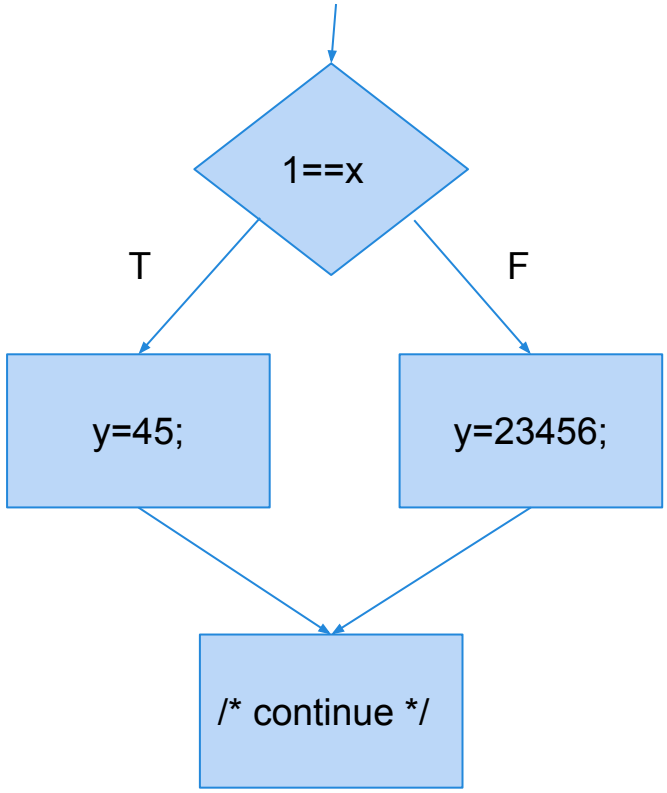
Control-Flow Graphs

- A directed graph representing the flow of control through the program.
- Nodes represent sequential blocks of program commands.
- Edges connect nodes in the sequence they are executed. Multiple edges indicate conditional statements (lo if statements, switches).



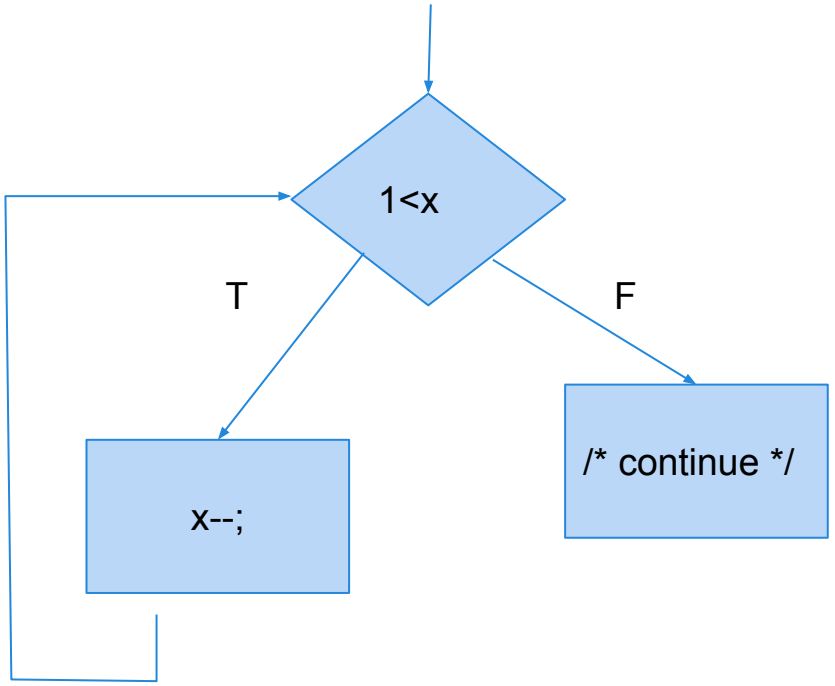
If-then-else

```
1 if (1==x) {
2       y=45;
3 }
4 else {
5       y=23456;
6 }
7 /* continue */
```



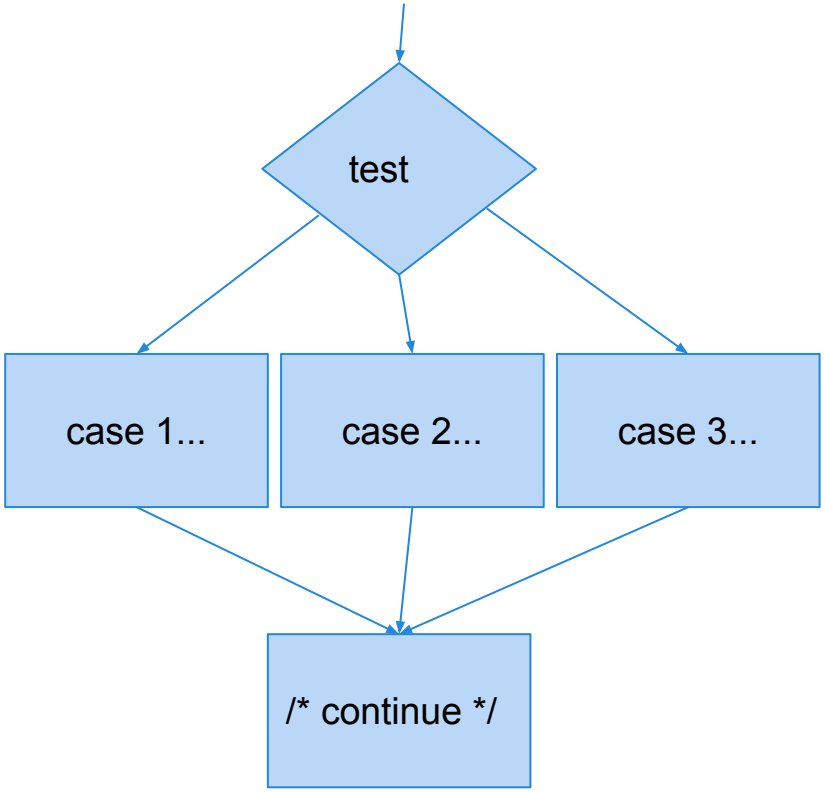
Loop

```
1 while (1<x) {
2       x--;
3 }
4 /* continue */
```



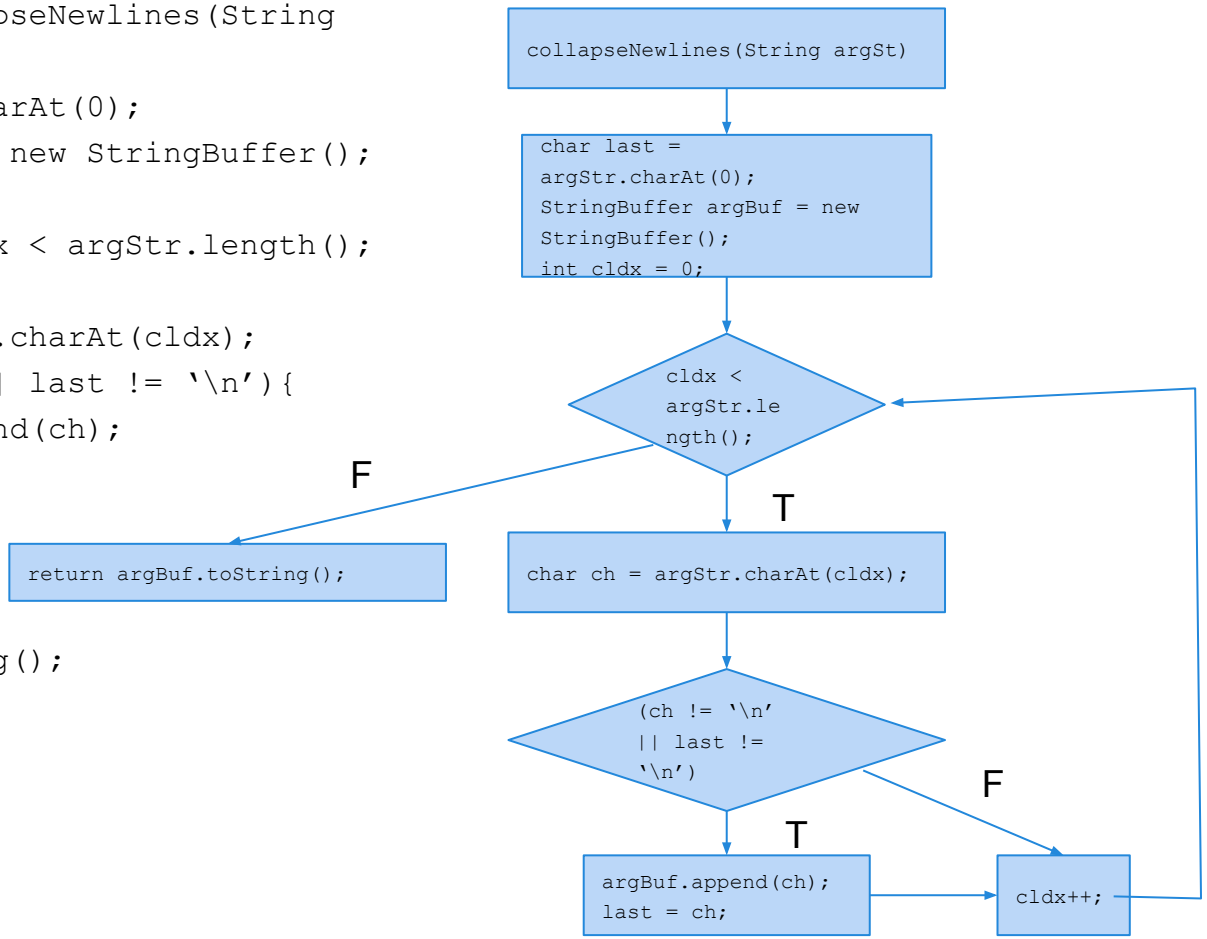
Case

```
1 switch (test) {
2       case 1 : ...
3       case 2 : ...
4     case 3 : ...
5 }
6 /* continue */
```



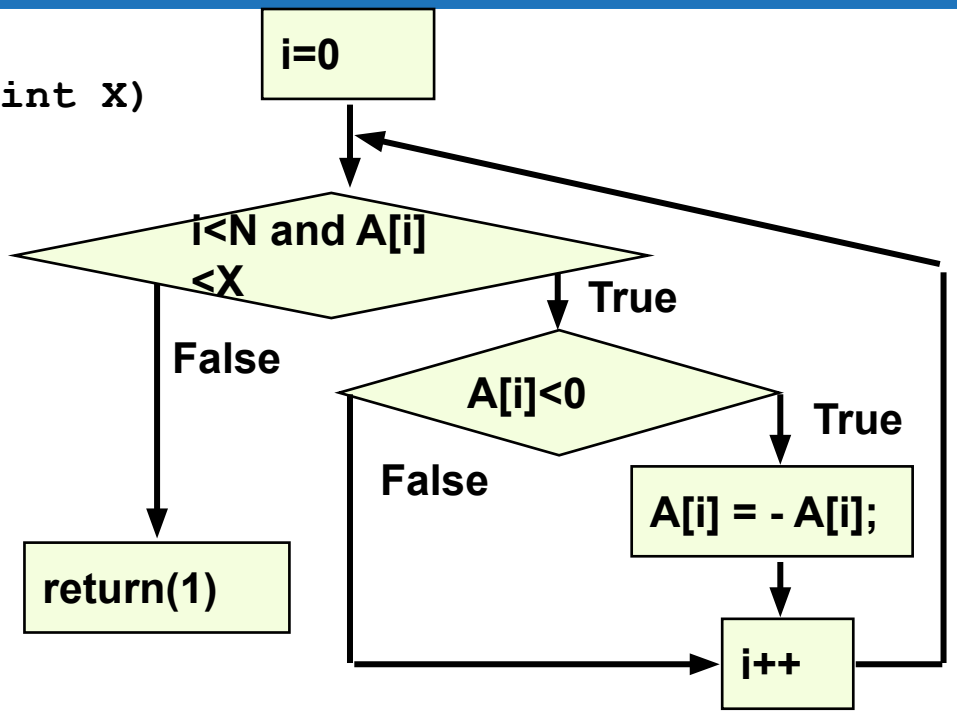
Control Flow Graph Example

```
public static String collapseNewlines(String
argSt){
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for(int cldx = 0; cldx < argStr.length();
      cldx++){
            char ch = argStr.charAt(cldx);
            if (ch != '\n' || last != '\n'){
                  argBuf.append(ch);
                  last = ch;
            {
      }

      return argBuf.toString();
}
```



Statement Coverage

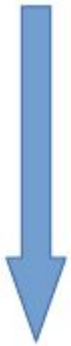
```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```



How many tests do we need to provide coverage?
What kind of faults could we miss?
Where would we want to use statement coverage?

Structural Coverage Criteria

- Criteria based on exercising of:
  - Statements (nodes of CFG)
  - Branches (edges of CFG)
  - Conditions
  - Paths
  - … and many more



Increasing number of test cases.

- Measurements used as (in)adequacy criteria
  - If significant parts of the program are not tested, testing is surely inadequate.

About Coverage

- 100% coverage is not always a reasonable target. why?
  - There might be a dead code (dead code)
    - **Empty Control Structures**: Loops or conditionals that don't contain any executable code or their content has been commented out or removed.
    - **Obsolete Code**: Code that was once useful but is no longer relevant due to changes in the software's requirements or functionality.
    - **Commented-Out Code**: Sometimes developers "comment out" code instead of deleting it, intending to use it later.
    - **Default Case Code in Switch Statements**: In some cases, the default case in a switch statement might never be executed if all possible values are covered in other cases.
- Two purposes: to know what we **have & haven't tested**, and to know when we can "safely" **stop testing**.
- Coverage measures what is **executed**, not what is **checked**.

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

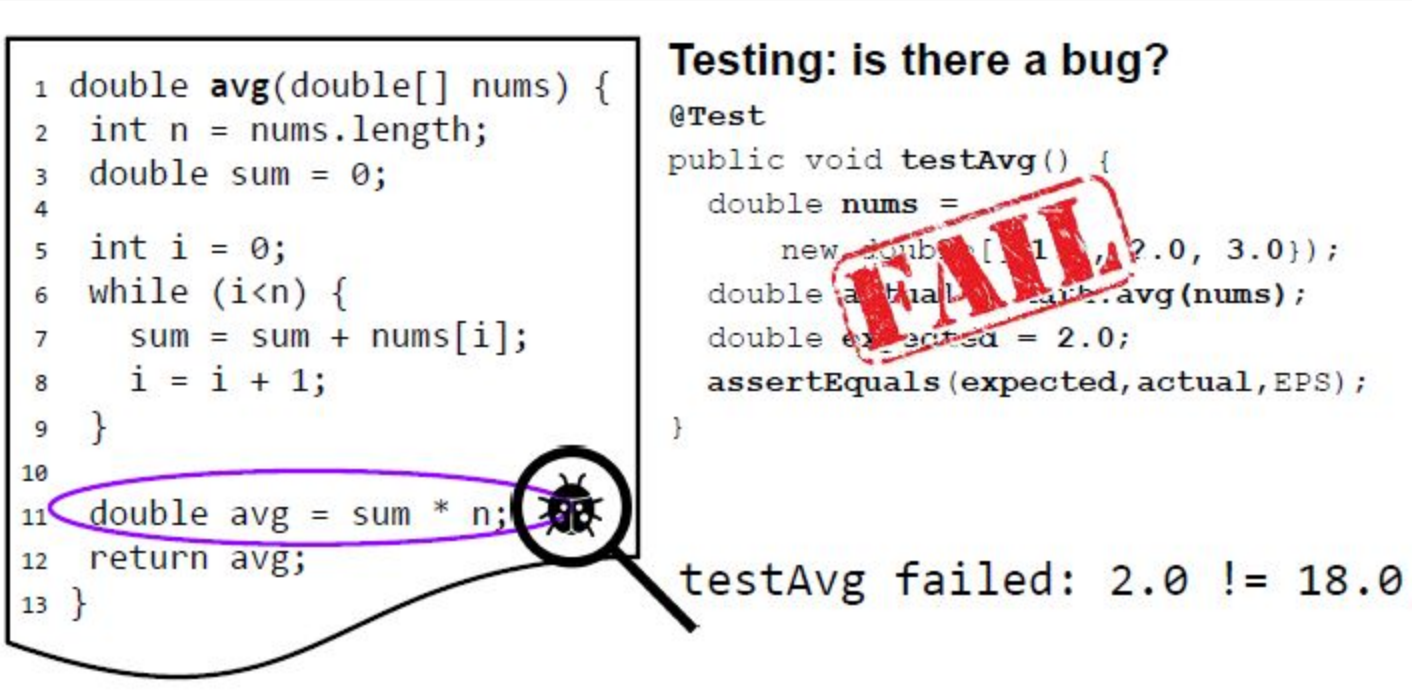
- Java JUnit,

```
1  double avg(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i<n) {
7      sum = sum + nums[i];
8      i = i + 1;
9    }
10
11   double avg = sum * n;
12   return avg;
13 }
```

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



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

Debugging

- Debugging is the process of fixing errors and problems that have been discovered by testing.
- Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error.
- This process is often supported by interactive debugging tools that provide extra information about program execution.