Software Testing

Testing Levels

- **Acceptance testing**
  - Test whether software is acceptable to the user
    (Acceptable = software meets user/business needs)
- **System testing**
  - Test the overall functionality of a system
- **Integration testing**
  - Test how modules/libraries interact with one another
- **Module testing (We as programmers care about this one)**
  - Module = Collection of related units that are assembled in a file, package or class
  - Test modules *in isolation* including how the components interacts
  - (Potentially) of a smaller scope than integration testing as it may test only a small collection of modules
  - Responsibility of the programmer
- **Unit test (We care about this one too)**
  - Test units (methods individually)
  - Responsibility of the programmer

Black-Box/White-Box Testing

- **Black-Box Testing**
  - Tests are derived from external descriptions of the software
- **White-Box Testing**
  - Tests are derived from source code internals of the software
  - More expensive to apply

Why is Software Testing the Big Hard?

- Exhausting testing is infeasible (you go try and test every integer parameter you've ever wrote)
- And writing random/statistical testing is not great

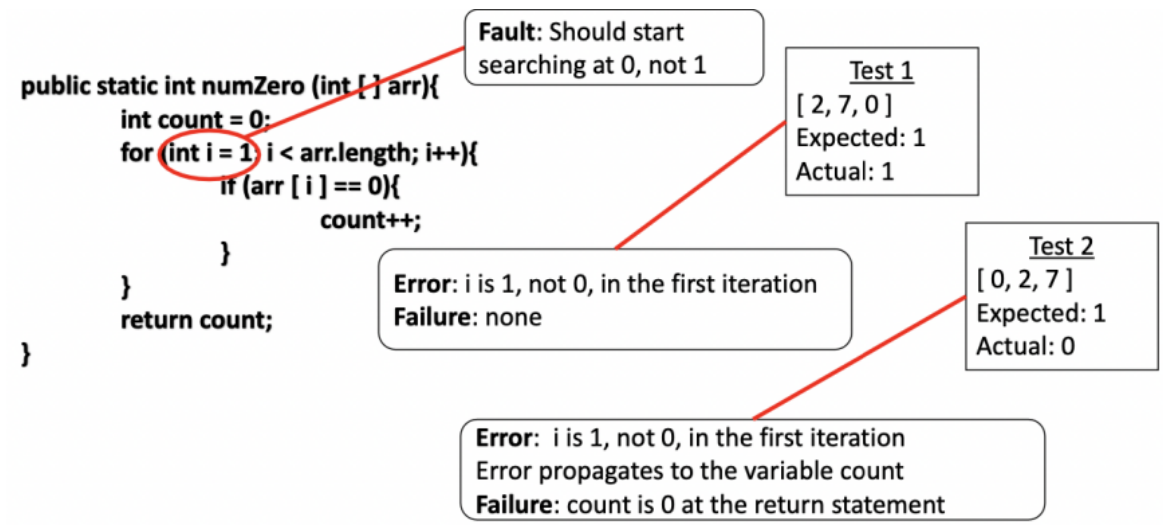
This is hard, why do we do it anyway?

- Software is kinda everywhere
- And software failures have some wild consequences

Fault/Error/Failure

- **Software failure**
  - A difference from the expected result
  - Expected result = Requirements or description of expected results from code
  - Related to the propagation of the program (see RIPR)
  - This is the problem you observe
- **Software fault**
  - The cause of a failure
  - In the code, this is a static defect
- **Software error**
  - The mistake which caused the fault to occur
  - Related to the infection of the program (see RIPR)
  - The manifestation of some fault

Example:



- **Why do we give three different labels to a "bug"?**
  - Saying *failure* means you know something is wrong but don't know the cause
  - Saying *fault* means you know the cause but don't know why it happened
  - Saying *error* means you know why it occurred (e.g. the coder was distracted by watching AMC stocks drop down)

RIPR

R - Reachability

- The part of the code that contains a bug is reached and executed

I - Infection

- The “State” of the program is wrong after the buggy code is reached
- E.g. on the second iteration of a loop, a variable is 2 instead of 1

P - Propagation

- The final “State” of the program is incorrect
- E.g. The program returns the incorrect value

R - Revealability

- In the testing, the error is revealed to the user
- E.g. when the program returns the incorrect value, we have a statement like assertTrue that catches the error and makes the test fail

```
public int indexOf(int x, int [] A){
        int i=0;
        while(i<A.length){
                if(A[i] == x)
                        return i;
                else
                        i = i + x; //fault, should be i = i + 1;
        }
        return -1;
}
```

Using the RIPR model, indicate for each of the following test cases the conditions that are satisfied.

Test case	Reachability	Infection	Propagation	Revealability
int [] A = {3,8,2}; assertTrue(indexOf(8,A) == 1);	✓	✓	✓	✓
int [] A = {3,8,2}; assertTrue(indexOf(3,A) == 0);				
int [] A = {3,8,2}; assertTrue(indexOf(1,A) == -1);	✓			
int [] A = {3,8,2}; assertTrue(indexOf(2,A) == 2);	✓	✓		
int [] A = {3,2,2}; assertTrue(indexOf(2,A) != -1);	✓	✓	✓	

Test case 1:
R - i = i+x is reached and executed
I - the variable i becomes 8 instead of 1, so the state of the program is incorrect
P - The program returns to wrong value, -1 instead of 1
R - The assertTrue fails since -1 != 1 and the test case fails

Test case 2:
R - i = i+x is **not** reached since the program returns immediately on the first iteration
Since it not reached the rest of RIPR doesn't apply

Test case 3:
R - i = i+x is reached
I - the state is **not** correct since x = 1, thus i = i + x ⇔ i = i + 1

Test case 4:
R - i = i + x is reached
I - the state is incorrect after the first iteration where x = 2 instead of 1
P - The final state of the program is **not** correct, since 2 is the correct index, thus the infection did not propagate throughout the program

Test case 5:
R - i = i + x is reached
I - the state is incorrect after the first iteration where x = 2 instead of 1
P - the infection is propagated to the return where 2 is returned instead of 1
R- the error is **not** revealed since the assertion only checks that -1 is not returned and does not catch the error and the test case does not fail when it should

Coverage criteria

A *coverage criterion* is a set of requirements for tests.

e.g., graph coverage, logic (predicate/clause) coverage

They systematically segment the possible input space, so that our tests are suitably comprehensive and minimally redundant. (Meaning of “suitably comprehensive” depends on what we’re testing.)

A criterion C_1 *subsumes* C_2 when every test set satisfying C_1 also satisfies C_2.

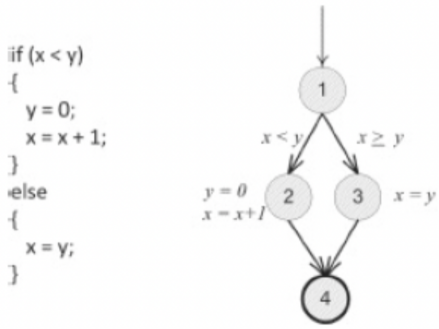
Graph coverage

Graph coverage criteria model programs as graphs. Examples include:

- *Node coverage* (pass every node)
- *Edge coverage* (pass every edge)
- *Path coverage* (pass every path)

Graphs can be constructed from methods/calls (*call graphs*), basic blocks / branches (*control flow graphs* (*CFGs*)), or Java bytecode.

- Example (Control Flow Graph)



In CFG coverage, statements are nodes and branches are edges, so node coverage is called *statement coverage* and edge coverage is called *branch coverage*.

Logic coverage

Logic coverage criteria impose requirements on the results of *predicates* and/or *clauses*.

Predicates are boolean expressions that are not constituents of any other boolean expressions.

Clauses are boolean expressions that do not include the operators && or ||.

Examples include:

- *Predicate coverage* (make every predicate evaluate to true and false)
- *Clause coverage* (make every clause evaluate to true and false)
- *Active clause coverage*
- *Inactive clause coverage*

For example, consider the predicate p: ((a>b) || c) && (x<y))

Predicate coverage requires a test case that makes p false, and one that makes p true.

Clause coverage requires test cases that make each clause of p evaluate to true and false.

Active clause coverage

Ensures that clauses affect the predicate.
For each predicate p of the program:

For each clause c of p:

We call c the *major clause* and all non-c clauses of p *minor clauses*.

Active clause coverage requires:

- A test case where c is true and c determines p
- A test case where c is false and c determines p

c *determines* p when changing c changes p.

For example, consider the predicate p: ((a>b) || c) && (x<y))
Active clause coverage requires:

- A test case where $a < b$ determines p and is false
- A case where $a < b$ determines p and it true
- A case where c determines p and is false
- A case where c determines p and is true
- A case where $x < y$ determines p and is false
- A case where $x < y$ determines p and is true

Inactive clause coverage

Ensures that clauses do *not* affect the predicate.
For each predicate p of the program:

For each clause c of p:

Inactive clause coverage requires:

- A test case where c is true and p is true
- A test case where c is true and p is false
- Same thing but such that c is false

Inactive clause coverage is not always possible to satisfy. e.g., if p consists of one clause, then p will always be determined by that clause.