

**SE 2AA4, CS 2ME3 (Introduction to Software
Development)**

Winter 2017

32 White Box Testing (Ch. 6) DRAFT

Dr. Spencer Smith

Faculty of Engineering, McMaster University

March 19, 2017



White Box Testing

- Administrative details
- Nonfunctional testing
- Functional testing
- Testing phases
- Theoretical foundations of testing
- Complete coverage principle
- Statement coverage
- Edge coverage
- Condition coverage
- Path coverage

Administrative Details

Fault Testing

- Common analogy involves planting fish in a lake to estimate the fish population
- T = total number of fish in the lake (to be estimated)
- N = fish stocked (marked) in the lake
- M = total number of fish caught in lake
- M' = number of marked fish caught
- $T = (M - M') * N / M'$
- Artificially seed faults, discover both seeded and new faults, estimate the total number of faults

Fault Testing Continued

- Method assumes that the real and seeded faults have the same distribution
- Hard to seed faults
 - ▶ By hand (not a great idea)
 - ▶ Independent testing by two groups and obtain the faults from one group for use by the other
- Want most of the discovered faults to be seeded faults
- If many faults are found, this is a bad thing
- The probability of errors is proportional to the number of errors already found

Nonfunctional System Testing

- Stress testing: Determines if the system can function when subject to large volumes
- Execution: Determines if the system achieves the desired level of proficiency in production status (performance)
- Recovery: Determines if the system has the ability to restart operations after integrity has been lost
- Operations: Determines if the operating procedures and staff can properly execute the system (documentation)
- Compliance (to process): Determines if the system has been developed in accordance with information technology standards, procedures and guidelines
- Security: Determines if the system can protect confidential information

Functional System Testing

- Requirements: Determines if the system can perform its function correctly and that the correctness can be sustained over a continuous period of time
- Error Handling: Determines the ability of the system to properly process incorrect transactions
- Manual Support: Determines that the manual support procedures are documented and complete, where manual support involves procedures, interfaces between people and the system, and training procedures
- Inter-systems: Determines that interconnections between systems function correctly

Functional System Testing Continued

- Control: Determines if the processing is performed in accordance with the intents of management
 - ▶ Includes data validation, file integrity, audit trail, backup and recovery, documentation and other aspects related to integrity
 - ▶ Controls are designed to reduce risks
- Parallel: Determines the results of the new application are consistent with the processing of the previous application or version of the application

Testing Phases

1. Unit testing
2. Integration testing
3. System testing
4. Acceptance testing

Theoretical Foundations of Testing

- P (program), D (input domain), R (output domain)
- $P : D \rightarrow R$ (may be partial)
- Correctness defined by $OR \subset D \times R$
 - ▶ $P(d)$ correct if $\langle d, P(d) \rangle \in OR$
 - ▶ P correct if all $P(d)$ are correct
- Desire a test set T that is a finite subset of D that will uncover all errors
- Determining an ideal T leads to several undecidable problems
- No algorithm exists to state if a test set will uncover all possible errors
- No algorithm exists to derive a test set that would prove program correctness

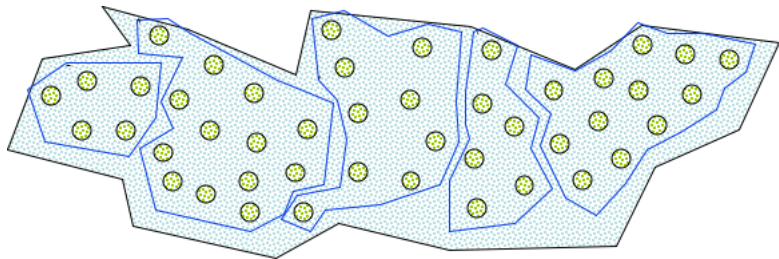
Empirical Testing

- Need to introduce empirical testing principles and heuristics as a compromise between the impossible and the inadequate
- Find a strategy to select **significant** test cases
- Significant means the test cases have a high potential of uncovering the presence of errors

Complete-Coverage Principle

- Try to group elements of D into subdomains D_1, D_2, \dots, D_n where any element of each D_i is likely to have similar behaviour
- $D = D_1 \cup D_2 \cup \dots \cup D_n$
- Select one test as a representative of the subdomain
- If $D_j \cap D_k = \emptyset$ for all $j \neq k$, (partition), any element can be chosen from each subdomain
- Otherwise choose representatives to minimize number of tests, yet fulfilling the principle

Complete-Coverage Principle



White-box Coverage Testing

- (In)adequacy criteria - if significant parts of the program structure are not tested, testing is inadequate
- Control flow coverage criteria
 - ▶ Statement coverage
 - ▶ Edge coverage
 - ▶ Condition coverage
 - ▶ Path coverage

Statement-Coverage Criterion

- Select a test set T such that every elementary statement in P is executed at least once by some d in T
- An input datum executes many statements - try to minimize the number of test cases still preserving the desired coverage

Example

```
read (x); read (y);  
if x > 0 then  
    write ("1");  
else  
    write ("2");  
end if;  
if y > 0 then  
    write ("3");  
else  
    write ("4");  
end if;
```

**$\{ \langle x = 2, y = -3 \rangle, \langle x = -13, y = 51 \rangle, \langle x = 97, y = 17 \rangle, \langle x = -1, y = -1 \rangle \}$
covers all statements**

**$\{ \langle x = -13, y = 51 \rangle, \langle x = 2, y = -3 \rangle \}$
is minimal**

Weakness of the Criterion

```
if x < 0 then  
    x := -x;  
end if;  
z := x;
```

$\{x < -3\}$ covers all
statements

it does not exercise the
case when x is positive
and the then branch is
not entered

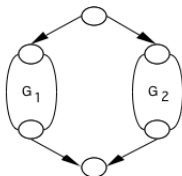
Edge-Coverage Criterion

- Select a test set T such that every edge (branch) of the control flow is exercised at least once by some d in T
- This requires formalizing the concept of the control graph and how to construct it
 - ▶ Edges represent statements
 - ▶ Nodes at the ends of an edge represent entry into the statement and exit

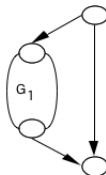
Control Graph Construction Rules



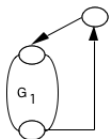
I/O, assignment,
or procedure call



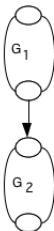
if-then-else



if-then



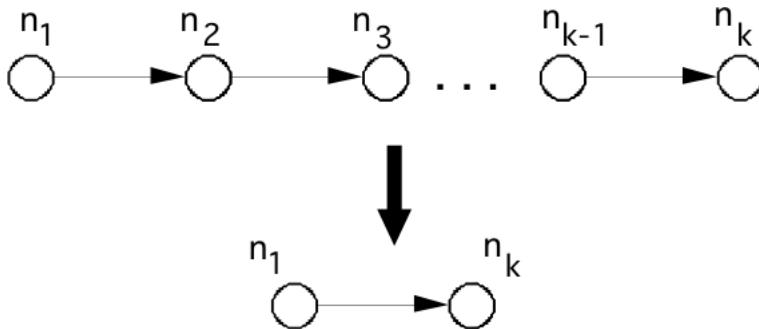
while loop



two sequential
statements

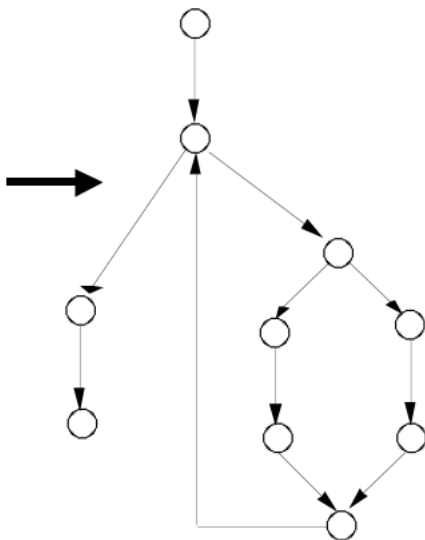
Simplification

A sequence of edges can be collapsed into just one edge



Example: Euclid's Algorithm

```
begin
  read (x); read (y);
  while  $x \neq y$  loop
    if  $x > y$  then
       $x := x - y$ ;
    else
       $y := y - x$ ;
    end if;
  end loop;
  gcd := x;
end;
```



Weakness

```
found := false; counter := 1;
while (not found) and counter < number_of_items loop
    if table(counter) = desired_element then
        found := true;
    end if;
    counter := counter + 1;
end loop;
if found then
    write ("the desired element is in the table");
else
    write ("the desired element is not in the table");
end if;
```

test cases: (1) empty table, (2) table with 3 items, second of which is the item to look for

Weakness

```
found := false; counter := 1;
while (not found) and counter < number_of_items loop
    if table(counter) = desired_element then
        found := true;
    end if;
    counter := counter + 1;
end loop;
if found then
    write ("the desired element is in the table");
else
    write ("the desired element is not in the table");
end if;
```

test cases: (1) empty table, (2) table with 3 items, second of which is the item to look for

Do not discover the error ($<$ instead of \leq)