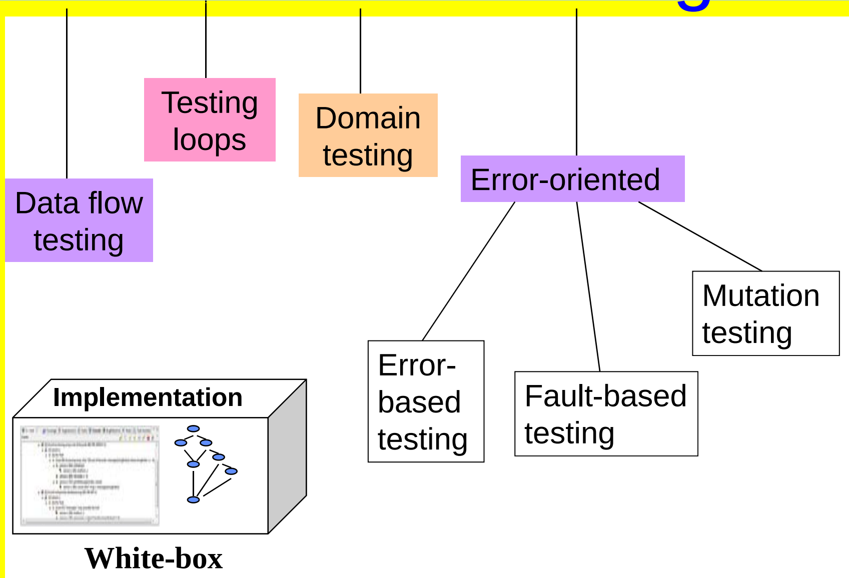
Course Structure

1. Software Quality Assurance
2. Testing Fundamentals
3. Code-based Techniques – Part II
4. Specification-based Techniques
5. Inspection Technique
6. Test Tools
7. Measuring Software Quality
8. TDD

Code-based Testing



Data Flow Testing

► Data flow testing

Testing loop

Domain testing

Error-oriented

- Data flow testing uses the relationship between points where variables are *defined* and points where they are *used* to design test.
- It requires the coverage of various **d**efinition-**u**se pairs (or du-pairs).

Why test this? test cases are inadequate if they do not exercise the various define and use combinations.

- An ***incorrect definition*** that is never used during a test will not be detected.
- If a given location ***incorrectly uses*** a particular definition, but that define-use pair is never tried during a test, the fault will not be detected.

Definition-clear Path

► Data flow testing

Testing loop

Domain testing

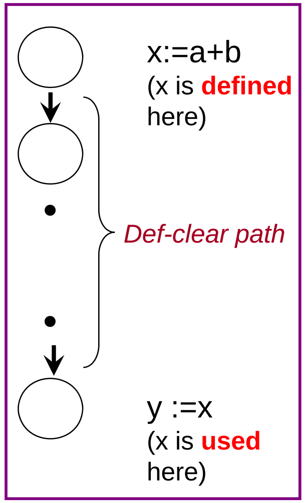
Error-oriented

Definition-clear path for variable x:

a subpath p such that x is not defined in any node in p, and x does not become undefined in any node in p

Example: a definition-clear path for variable `number of line` in *`Program Word`* is:

Line 8 to line 13.



Example: CountWord Program

```
#define YES 1
#define NO 0
main()
{
  int c, number_of_line, number_of_word, number_of_char, inword;
  inword = NO;
  number_of_line = 0;
  number_of_word = 0;
  number_of_char = 0;
  c = getchar();
   while (c != EOF) {
      number_of_char = number_of_char + 1;
      if (c == '\n')
          number_of_line = number_of_line + 1;
      if (c == '' || c == '\n' || c == '\t')
          inword = NO;
      else if (inword == NO) {
          inword = YES;
         number_of_word = number_of_word + 1;
      }
      c = getchar();
   }
   printf("%d \n", number_of_line);
   printf("%d \n", number_of_word);
   printf("%d \n", number_of_char);
 }
```

Data Flow Testing

► Data flow testing
Testing loop
Domain testing
Error-oriented

- ⌘ Create test cases to exercise <u>define and use pairs</u>.
- ⌘ Data flow testing is useful for
  - ➔ programs that are computationally intensive
  - ➔ control intensive programs, if control variables are computed

<u>Data flow criteria</u> measure the flow of data between **variable assignments** and **references to the variables**.

Data Flow Coverage Criteria

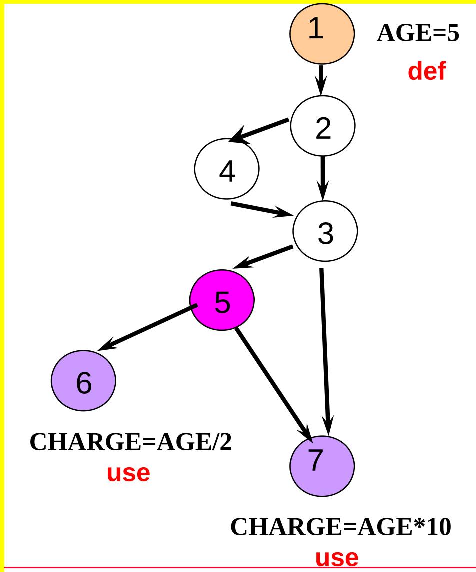
► Data flow testing
Testing loop
Domain testing
Error-oriented

- ⌘ *All-DU-path* requires that <u>all</u> paths between all defs and all uses it reaches be exercised
- ⌘ *All-Uses* requires <u>some</u> path between all defs and all uses it reaches be exercised (***use*** can be c-use or p-use)
- ⌘ *All-Defs* requires <u>some</u> path between all defs and some uses it reaches be exercised

Criteria	Number of Uses	Paths to Uses
All-DU-path	all	all
All-Uses	all	some
All-Defs	some	some

Example

► Data flow testing
Testing loop
Domain testing
Error-oriented



The definition of **AGE** at node 1 has 2 uses at nodes 6 and 7.

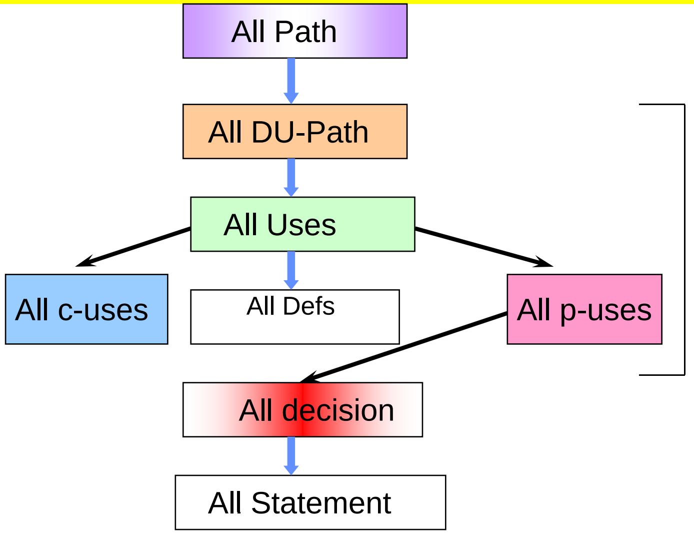
du-pairs for **AGE** are: (1, 7) and (1, 6)

Test cases required to achieve coverage:

All-DU-path:	6
All-Uses:	2
All-Defs:	1

Revised Subsumption Relation

▸ **Data flow testing**
Testing loop
Domain testing
Error-oriented



After adding the data flow coverage criteria

Code-based Testing vs. Code Coverage

Data flow testing
Testing loop
Domain testing
Error-oriented

⌘ **Code-based testing** is performed for the purpose of *creating* test cases based on the code.

⌘ **Code coverage measurements** are performed for the purpose of *evaluating* test cases and can use either code-based or specification-based test cases

⌘ As a **measure**, coverage is usually expressed as a percentage.

- Code coverage may be used as an **adequacy criterion** of testing (i.e. determine when to stop testing).
  e.g., once we achieve 100% statement coverage, we can stop testing

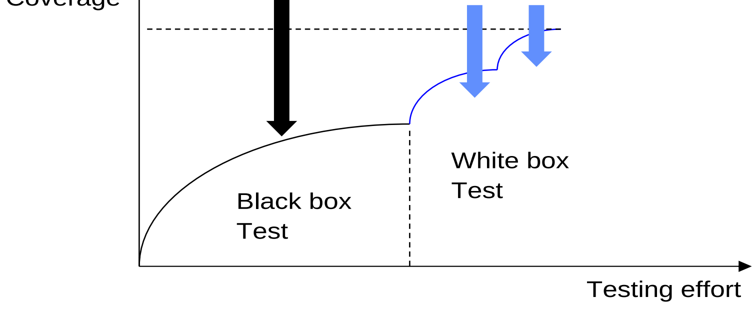
Typical testing strategy

Data flow testing
Testing loop
Domain testing
Error-oriented

How to construct tests and get everything covered?

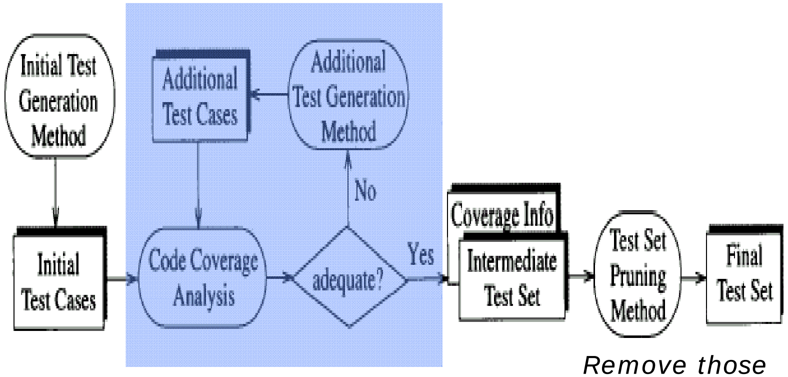
Guidelines:

1. Develop test cases based on black box methods
2. Measure coverage of the black box tests
3. If not enough, add more white-box test cases



A Testing Strategy using Coverage

Data flow testing
Testing loop
Domain testing
Error-oriented



Remove those tests which do not increase the coverage %.

Which Coverage Criterion to use?

→ A bit stronger than branch coverage is very effective, e.g. branch + condition coverage

→ **Industrial practice**:

Varies from statement coverage (100%) to multiple condition coverage (?%)

e.g., HuaWei uses statement coverage

Code Coverage in Practice

What coverage % to use?

- ➔ Some companies use 85% or 90%
- ➔ But, it is better to do more analysis:
  - What is the coverage of each component of the program?
  - Have we tested the key component enough?
  - Key components should have higher coverage!

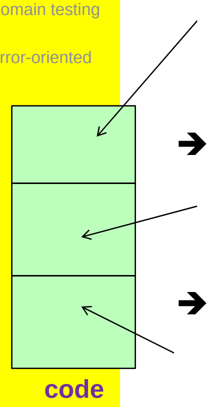
A better code coverage strategy (1)

Divide the code under test into 3 categories:

➔**High risk code** could cause severe damage (wipe out data, injure someone), or has many users, or seems likely to have many mistakes whose costs will add up.

➔**Medium risk code** is between high risk and low risk code. Defects would not be individually critical, but having too many of them would cause a schedule slip.

➔**Low risk code** is unlikely to have defects important enough to stop or delay a shipment, even when all the defects are summed together. They would be annoyance defects in non-essential features.



A better code coverage strategy (2)

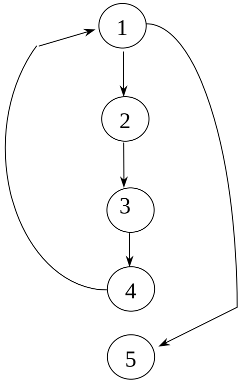
- ➔ Test the high risk code thoroughly.
- ➔ Then, use most of the remaining time testing the medium risk code.
- ➔ Don't intentionally test the low risk code. But, they may be exercised incidentally by tests that target higher-risk code.
- ➔ Check for coverage
  - ➢ Expect good coverage of high risk code
  - ➢ Expect lower coverage for medium risk code. Check to see whether we have overlooked something.
  - ➢ Check for any module that was never entered. Create tests for it.

Testing Loop

Data flow testing
▸Testing loop
Domain testing
Error-oriented

Typical Defects in Loop:

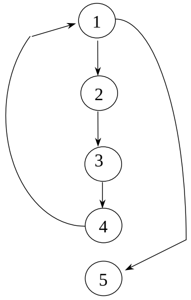
- ⌘ missing the first case
- ⌘ terminating too early
- ⌘ terminating too late
- ⌘ never ending (infinite loop)



Strategy for Testing **Simple** Loops

Data flow testing
▸Testing loop
Domain testing
Error-oriented

- ⌘ **Bypass**: any value that causes the loop to be exited immediately
- ⌘ **Once**: values that cause the loop to be executed once
- ⌘ **Twice**: loop to be executed twice
- ⌘ **Typical**: A typical number of iterations (normal case)
- ⌘ **Maximum**: maximum number of allowed iterations
- ⌘ **Max+1**: one more than the max. allowed
- ⌘ **Min**: minimum required
- ⌘ **Min-1**: one less than the minimum required



Example

Data flow testing
▸Testing loop
Domain testing
Error-oriented

If “minimum value” is 1, then

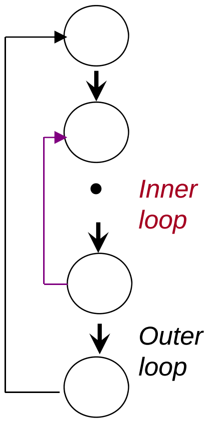
Min-1 = Bypass the loop
Min = Once
Min+1=Twice

- Some of the test cases listed in the previous slide overlap with each other.
- Usually don’t need many test cases as shown in the previous slide.

Strategy for Testing Nested Loops

Data flow testing
▸Testing loop
Domain testing
Error-oriented

- ⌘ Start with the innermost loop. Set all other loops to <u>minimum values</u>
- ⌘ Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration values.
- ⌘ Work outward, conducting tests for the next loop, but keeping all other outer loops at the minimum iteration count.
- ⌘ Continue until all loops have been tested.

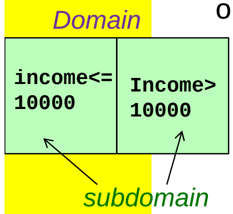


Domain Testing (1)

Data flow testing
Testing loop
Domain testing
Error-oriented

Focus on testing the boundaries of the input domains.

- ⌘ **Domain:** a subset of input space of the program.
- ⌘ The domain is made up of subdomains in which all elements are treated similarly.
- ⌘ The domain is determined from the program, not the specification!
- ⌘ Domains are sets of boundary inequalities defined over the input space.



Boundary inequality: an algebraic expression over the input variables that defines which points in the input space belong to the domain of interest.

Example: inequality **`income<=10000`**, the points in the domain of interest all have an **income** 10000 or less

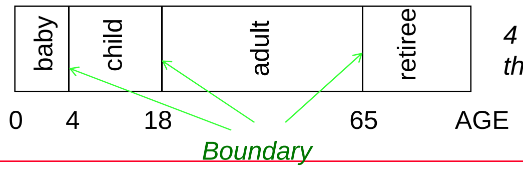
Domain Testing (2)

Data flow testing
Testing loop
Domain testing
Error-oriented

Suppose we have a program to calculate the discount based on the age of the user:

```
If (AGE < 4) then discount = 0
If (4 <= AGE < 18) then discount =10
If (18 <= AGE < 65) then discount = 20
If (AGE > 65) then discount = 30
```

- ⌘ From analysing the program, we can identify the following 4 subdomains:



4 subdomains for the input AGE

Domain Testing (3)

Data flow testing
Testing loop
Domain testing
Error-oriented

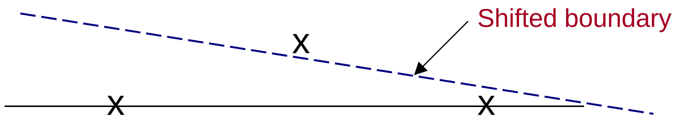
What test cases to use?

A **boundary test** tries to establish that the boundary is in the correct place.

Test cases are chosen on the boundary as close as possible on the open side.

2-on, 1-off means 2 test points on the boundary spread far apart and one test point just off the boundary in between the other 2 points.

If we choose the off points very far from the boundary, then the boundary can be shifted and we cannot detect this problem.



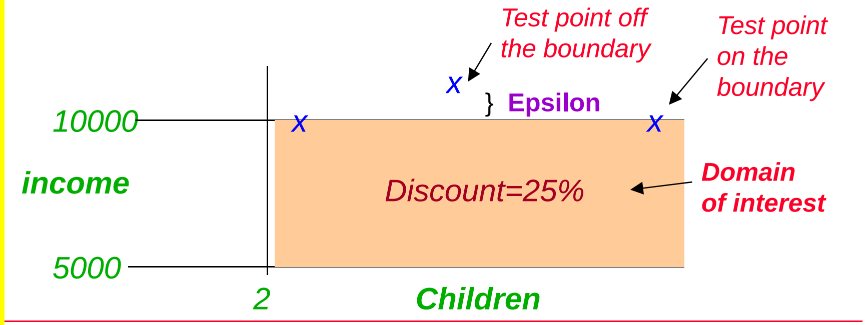
Example

Data flow testing
Testing loop
Domain testing
Error-oriented

Program

```
If (income>5000) then
   if (income<=10000) then
      if (children >2) then
         discount=25%
   else discount=50%
```

One of the domain boundaries is **`income=10000`**



Domain Testing

Domain testing is best apply to

- ☺ heavy numerical processing with lots of conditional logic, such as: financial calculations, payroll processing
- ☺ numerical inputs and heavy input data validation and categorization.

Limitations:

- ⌘ Difficult to handle loop. (Hard to work out the domains)
- ⌘ Blind to errors of less than the chosen Epsilon.
- ⌘ Can't detect coincidental correctness

Coincidental Correctness & Blindness

Coincidental correctness:

Actual and expected outcomes (or outputs) can match but there are defects in the software because outcomes matched by coincidence.

Example: if the correct expression should be **$y=x^2$**
But, we coded **$y=x*2$**
If we use a test case with x=2, we will not detect this fault!

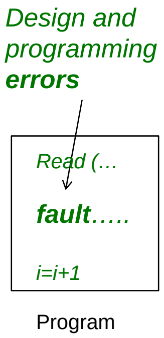
Blindness:

Every test method is blind to defects specific to that technique. Example: many techniques are blind to coincidental correctness.

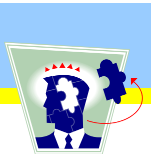
Each testing technique has its limitations!

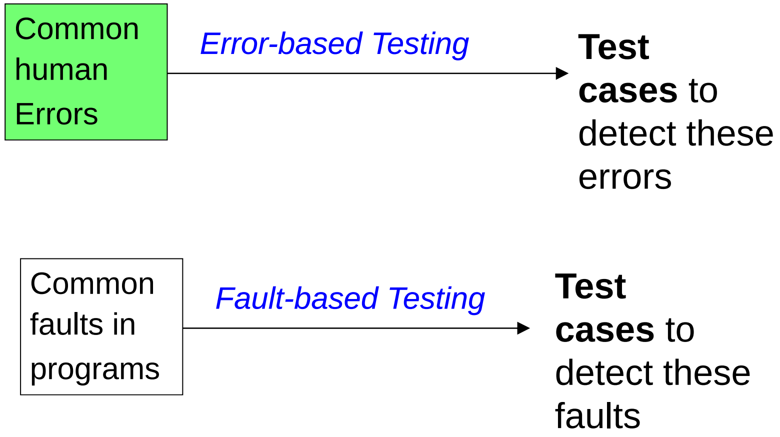
Error-oriented Testing

- Error-based Testing
- Fault-based Testing
- Mutation Testing



Pattern-based – based on common errors and/or faults (*error patterns, fault patterns*)




Common human Errors
Error-based Testing
Test cases to detect these errors
Common faults in programs
Fault-based Testing
Test cases to detect these faults

Error-based Testing

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ Error-based
Fault-based
Mutation testing

- ⌘ Begin with the ***design*** and ***programming process***
- ⌘ Identify potential errors in that process
- ⌘ Determine how those errors are reflected as faults
- ⌘ Seek to demonstrate the absence of those faults.

Aim: show absence of errors

Example:

Code **(a+b)** rather than **(a*b)**

Test case: a=2, b=2 will not detect the fault!

But, Test case: a=1, b=2 will identify the fault

Error-based testing

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ Error-based
Fault-based
Mutation testing

We use this testing technique because human errors are common.

Example:

IF ((this AND that) OR (that AND other)
AND NOT (this AND other)
AND NOT (other OR NOT another))

Can you figure out what this is doing?

Programmer will likely make a mistake here.

Careful with Programming Language

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ Error-based
Fault-based
Mutation testing

Some programming language features are more error-prone than others, because:

- ⌘ Programmers are prone to making errors when using the feature
- ⌘ Compilers are prone to incorrect implementation of the feature
- ⌘ Programs using the feature may be difficult to analyze, test or prove.

Dangerous Programming Language Features

- ⌘ Pointer. It is easy for programmers to become confused about where a pointer is pointing at. Programs which use pointers can be difficult to understand or analyze.
- ⌘ Memory allocation. Programmer often allocate memory and subsequently forget to deallocate it. Compilers and OS frequently fail to fully recover deallocated memory. Errors are dependent on execution time, and the system may fail after a period of correct operation.
- ⌘ Unstructured programming (including the use of goto)
- ⌘ Multiple entry points and exit points to loops, blocks, procedures and functions.
- ⌘ Variant data, where the type of data in a variable changes, or the structure of a record changes, is difficult to analyze and cause confusion.
- ⌘ Implicit declaration and implicit initialization. It is hard to detect errors in these.
- ⌘ Procedural parameters, or passing one procedure or function as a parameter to another procedure or function, is difficult to analyze and test.
- ⌘ Recursion is difficult to analyze and test. It leads to unpredictable real time behavior

Requirement **Errors** (from NASA)

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ **Error-based**
Fault-based
Mutation testing

- Incompleteness (decomposition, requirement description)
- Omitted/missing (requirement, external constants, initial system state)
- Incorrect (external constants, input or output, initial system state, assignment of resources)
- Ambiguous
- Infeasible
- Inconsistent (external conflicts, internal conflicts)
- Over-specification
- Non-verifiable
- Misplaced
- Redundant or duplicate

Example of Logic & Control **Errors**

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ **Error-based**
Fault-based
Mutation testing

- unreachable code
- improper nesting of loops and branches
- incomplete predicates
- improper sequencing of processes
- infinite loops
- instruction modification
- failure to save or restore registers
- unauthorized recursion
- missing labels or code
- unreferenced labels

Example of Computational **Errors**

- missing validity tests
- incorrect access of array components
- mismatched parameter lists
- initialization faults
- undefined variables
- undeclared variables
- misuse of variables (locally and globally)
- data fields unconstrained by natural or defined data boundaries

Other Errors

- calls to subprograms that do not exist
- improper program linkages
- input-output faults
- failure to implement the design

Error Guessing (1)

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ **Error-based**
Fault-based
Mutation testing

⌘ A sub-class of error-based testing

Basic Idea: Given the description (specification) of a program, the tester guesses, both by intuition and experience, certain probable types of error and then select test cases to expose them.

⌘ Use tests that are likely to expose errors.

⌘ Assumption: defects that have been there in previous projects are likely going to be in the new project.

Approach:

- Make a list of possible errors or error-prone situations (or error models)

Example: empty or null lists/strings, zero instances/occurrences, blanks or null characters in strings, negative numbers

- Design test-cases to cover all error models.

Example

Data flow testing
Testing loop
Domain testing
Error-oriented
▸ **Error-based**
Fault-based
Mutation testing

Given the following specification:

Given a text consisting of words separated by BL (blank space) and CR (carriage return) characters, reformat it so that

(1) line breaks are made only where the given text has BL or CR;

(2) each line is filled as far as possible; and

(3) no line contains more than MAXCHAR characters.

Example input:

How are you? CR I am fine.CR

Example (continued)

An experienced tester will check if the program work correctly for the following cases:

- an input text of length zero
- a text containing a very long word (length>MAXCHAR)
- a text containing nothing but BL's and CR's
- a text with an empty line
- words separated by 2 or more consecutive BL's or CR's (e.g. How are you? CR CR CR I am…)
- a line with BL as the first or last character
- a text containing digits or special characters
- a text containing nonprintable characters

Fault-based Testing

Try to show the absence of certain **faults** in the code.

Begin with the code and ask what are the potential faults in it, regardless of what error in the design and programming process caused them.

Different from error-based testing

⌘ For each plausible fault, create an input that will make the given, incorrect expression evaluate wrong.

This testing misses 2 types of defects:

⌘ Incorrect specification

⌘ Interactions among subsystems, as it is local in scope and driven more by the product than the customer

Examples: Off by One Fault

Off by one is a common fault, such as by starting at 0 when we should have started at 1 or vice-versa, or by writing < N instead of <= N or vice-versa.

Example 1: A loop needs to execute 10 times, and we program:

```
For (x=0; x<=10: x++)
```

This will execute 11 times!

Example 2: Try to set x to array_max on the last pass through the loop:

```
for (x=array_min; x<array_max; x++)
```

But, it is wrong!

Example Faults

```
IF (a and not b or c) THEN calculate maximum
```

Possible faults:

(1) '`and`' should be '`or`'

(2) '`not`' should not be used

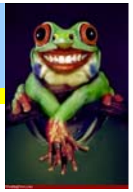
(3) There should be parenthesis around '`not b or c`'

What values to use for a, b, and c to show these faults?

For fault (1), what input will cause `(a and not b or c)` to give a different value from `(a or not b or c)`

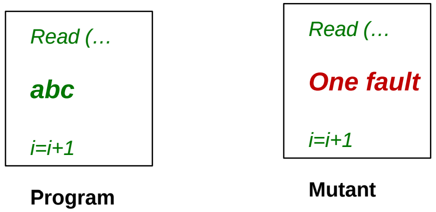
Mutation Testing

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**



What is a Mutant?

It is a program with one <u>injected fault</u>



- **Each mutant consists of a single fault.**
- Each time a test case causes a mutant to produce a different output from the original program, that mutant is considered '**killed**'.

Mutation Testing

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**

- **Goal** of testing: find a test set that causes the mutant to generate an output different from that of <u>the program under test</u>, thereby distinguishing the mutant from the program (i.e., kill the mutant.)
- When all mutants have been killed, the test set is <u>**effective**</u>, as it can distinguish faulty versions from the correct program.

➔ Mutation testing is complicated and time-consuming to perform without an automated tool.

➔ Some mutation tools are available:
Insure++ v4.0 from Parasoft
Open source mutation tool: Jester, http://jester.sourceforge.net/

Mutation Testing

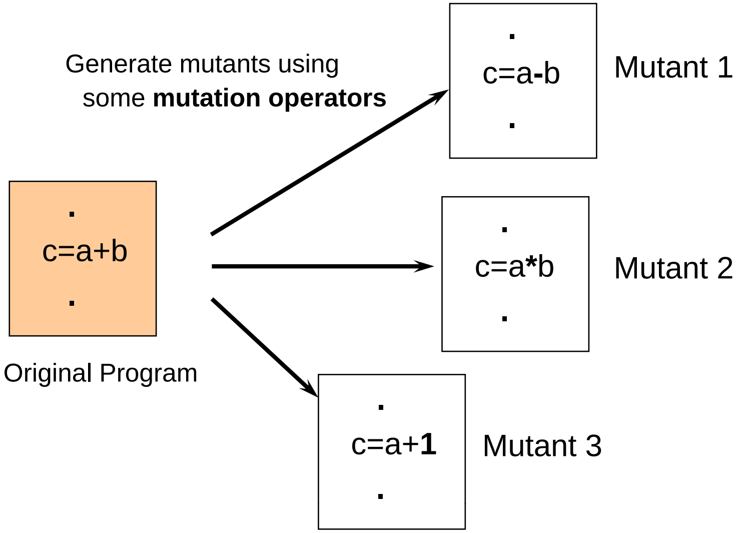
Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**

- A mutant is produced by a ***mutation operator*** - simple <u>syntactic change</u>

  Examples:
  - Statement deletion.
  - Replace each boolean subexpression with *true* and *false*.
  - Replace each arithmetic operation with another one, e.g. + with *, - and /.
  - Replace each boolean relation with another one, e.g. > with >=, == and <=.
  - Replace each variable with another variable declared in the same scope (variable types should be the same).
- Applying the mutation operator at each possible point in the program can generate many mutants (several thousand!).

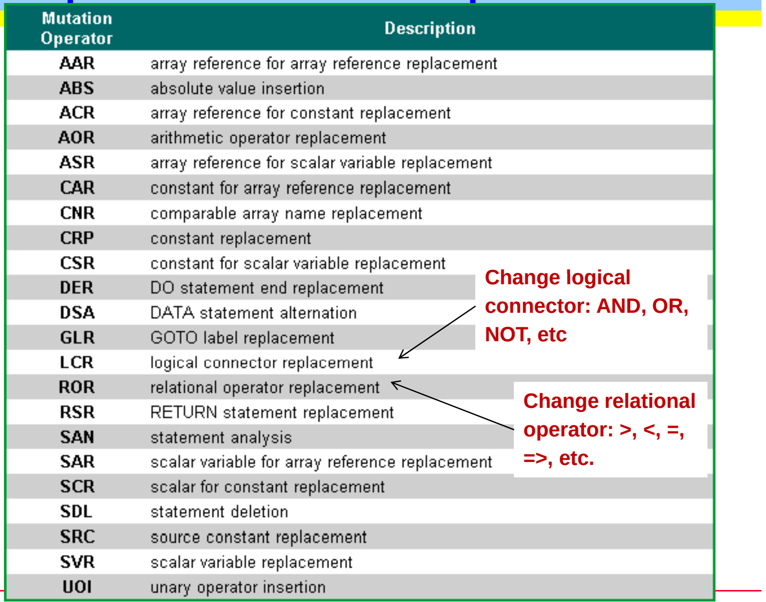
Generating Mutants

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**



Example Mutation Operators

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**



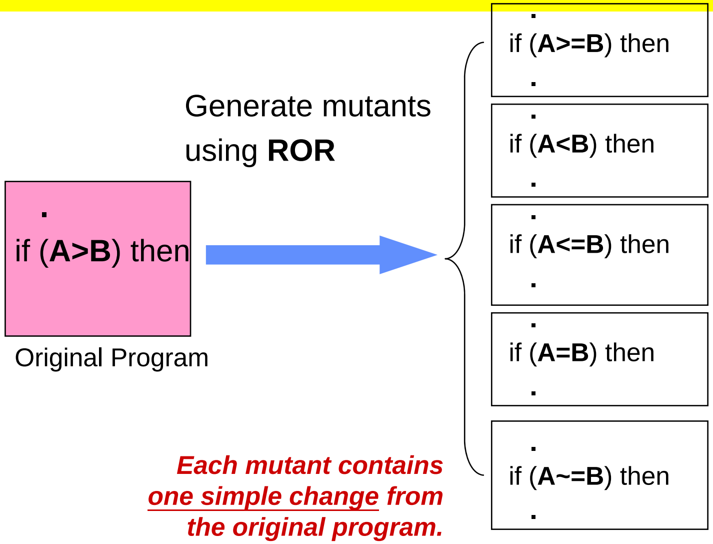
Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alternation
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Change logical connector: AND, OR, NOT, etc

Change relational operator: >, <, =, =>, etc.

Example: Generating Mutants by ROR

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**



Generate mutants using **ROR**

Each mutant contains <u>one simple change</u> from the original program.

Adequacy of Test (1)

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**

- **adequacy criteria:** Conditions that are required to be satisfied during testing.
- Example: testing may be considered <u>inadequate</u> if the test data
  - ➢ do not include boundary cases specified by the requirements,
  - ➢ do not cause execution of every branch, or
  - ➢ do not cause the program to exercise certain error-prone situations.
- Even correct program can be poorly tested. Adequacy criteria may be viewed as quantifying the <u>quality of testing.</u>

Adequacy of Test (2)

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**

- Useful adequacy criteria seek to improve the likelihood of finding faults in the program, regardless of the quality of the program being tested.

For mutation testing, **test case adequacy** is judged by demonstrating that injected faults are caught.

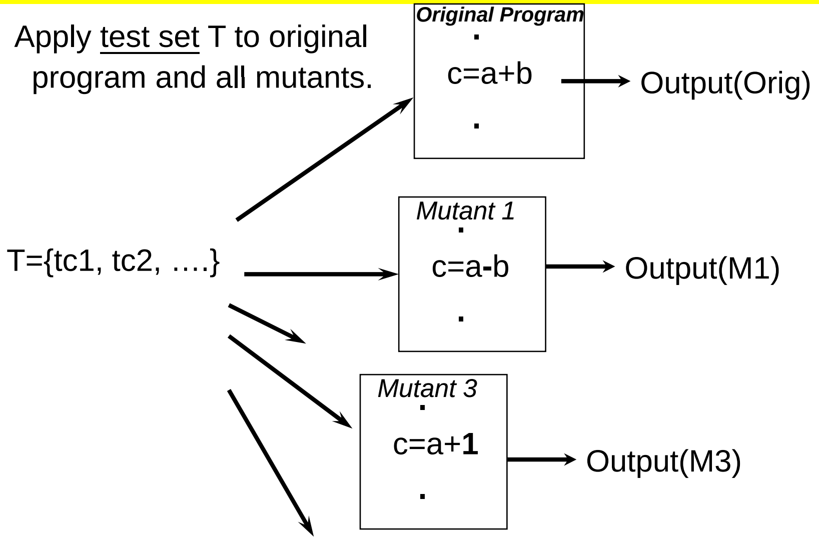
- The mutation testing process mainly focusing on evaluating **how 'good' is a test set**.

⌘ The test data is judged *<u>adequate</u>* only if each mutant in this set is either functionally equivalent to the original program or computes different output than the original program.

Run the test cases:

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▶ Mutation testing

Apply test set T to original program and all mutants.



Two possible cases:

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▶ Mutation testing

Case 1: Output(M1) not equal Output(Orig)

Mutant 1 is killed

Case 2: Output(M1) = Output(Orig)

What does this mean?

- Mutant 1 is equivalent to Original program. Need to confirm this by doing some code analysis.
- Test cases are not good. Try to develop more tests.

Mutation Adequacy Score, S

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▶ Mutation testing

$$S = \frac{D}{(M-E)}$$

➔ D is the number of dead mutants (killed)

➔ M the total mutants

➔ E the equivalent mutants

The closer S to 1, the better.

Example:

There are 25 mutants, 10 are found to be dead and 5 are equivalent mutants, then S = 10/(25-5)=0.5

Mutation Testing Hypotheses

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▶ Mutation testing

The mutation testing method assumes 2 hypothesis are true!!

1. Competent programmer hypothesis: a competent programmer will write code that is close to being correct. The correct program can be produced by some straightforward syntactic changes to the code.

Example: if the correct code is

`A=B*C`

The programmer will not make a big mistake and write

`A=C-D*K+H`

The mistakes are like: **`A=B-C`** or **`A=B*1`** (small mistakes)

Mutation Testing Hypotheses

Data flow testing
Testing loop
Domain testing
Error-oriented
Error-based
Fault-based
▸ **Mutation testing**

The mutation testing method assumes 2 hypothesis are true!!

2. Coupling effect hypothesis: test cases that reveals simple faults will uncover complex faults as well. Thus, only single mutants need be eliminated, and combinatoric effects of multiple mutants need not be considered.

Complex fault such as **A=B*D–C** are related to simple fault, such as **A=B+1**

We do not need to test for complex faults, because when we look for simple faults, we will also detect complex faults at the same time.

Review Exercise

Data flow testing
Testing loop
Domain testing
Error-oriented

For the **liability procedure**,

1. How many mutants if we apply ROR?
2. How many mutants if we apply LCR?

```
procedure liability(age, sex, married, premium);
begin
  premium := 500;
  if ((age<25) and (sex=male) and (not married))
then
      premium := premium + 1500;
  else (if (married or (sex = female)) then
              premium := premium-200;
        if ((age > 45) and (age < 65)) then
              premium := premium-100;)
 end;
```

Review Questions

Data flow testing
Testing loop
Domain testing
Error-oriented

1 Most defects occur in which phase?
2 Why early testing (V&V) is importance?
3 Why random testing is ineffective?
4 What is the best way to improve software quality?
5 What should be the key objective of testing?
6 What is the key objective of white-box testing?
7 How many test cases are needed to cover all branches?

8. For **Program CountWord**, estimate how many test cases are required to fully test it.

Time to Reflect

Related to white-box testing,

- What have you learned?
- What else you want to learn?
- How can you use white-box testing in your project?
- What topics we have not discussed?