Why Testing?

- Testing finds errors (Does not guarantee absence of bugs!)
- An early agreement upon the test plan at requirement stage is important

Test Driven Development

Specs

Test plan
A Test Case

- A test case is described through the choice of inputs for the unit to be tested.

- For example, following are three different test cases to test a function `int f (int x, int y)` that compares two numbers and reports the maximum:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>- <code>X</code> and <code>Y</code> same</td>
<td><code>(10,10)</code></td>
</tr>
<tr>
<td>- <code>X</code> &gt; <code>Y</code></td>
<td><code>(10, 8)</code></td>
</tr>
<tr>
<td>- <code>Y</code> &gt; <code>X</code></td>
<td><code>(2, 25)</code></td>
</tr>
</tbody>
</table>
Important Questions

- What should be tested
- What criteria should be used to design tests?
- How to analyze the output?
Is the output of a test correct?

- Manual Validation
  - Time consuming

- Test Oracles to automatically test and validate against the expected

- The testing strategies, the test specifications and results are documented
What criteria to use?

- Should the testing be done based on externally observable behavior

OR

- should the code be seen to design test cases?
Black box Vs. White Box Testing

- **External Vs. Internal View**
  - Black box testing is based on what is required from external point of view
  - White box testing is based on an insider’s view of the given artifact
    - Tester has a knowledge of code
    - Test cases are generated to test the coding structures
External Vs. Internal View

- Black box testing is based on what is required from external point of view

- White box testing is based on an insider’s view of the given artifact
Black Box Testing

- Also called Functional Testing
- Test the artifact from external point of view
- Specs are used to generate test data

- E.g. a data sorting function is tested on different sets of data
- Data can be randomly generated based on input types
White Box Testing

- Also called Structural Testing
- Test the artifact from internal (implementation) point of view
- Cannot detect absence of features
- Coverage measures are used, e.g:
  - Statement Coverage
    - Each statement is Covered in testing
  - Branch Coverage
    - Each branch is covered (e.g. in if-then-else)
  - Path oriented testing
    - Select data such that chosen paths in the program are covered
Black Box Testing (External)

- Functional/Feature Testing
- Boundary Value Testing
- Tests for absence of features

Use of specs to generate test data
White Box Testing (Internal)

- Structural Testing
  - Internal structure used to generate test data

- Test for coverage of statements, conditional branches, paths

- Does not detect absence of features of software
Static Analysis for Testing Code

- No actual execution is done
- Check for not well formed control paths
  - Unstructured programs
  - Unreachable code
- Variable anomalies
  - Unused variables
  - Misused variables references
Statement Coverage

- Select a test suit such that each statement in the program is executed at least once

- Motivation: An error may get masked if tests do not execute parts of the program

- What is an elementary statement?
  Use of syntactic definition of language:
  - Assignment statement
  - procedural calls
  - i/o statements in conventional block structured languages
Statement coverage

- A test case may cover many statements

- One may try to minimize the number of test cases such that all the statements are still covered
An Example for Statement Coverage

```c
int fib (int n) {
   /* defined on n=0+ */
   if (n<2)
      return n;  \( \leftarrow \) elementary statement
   else if (n>=2)
      return (fib(n-1)+fib(n-2));  \( \leftarrow \)
         elementary statement
}
```

Test suit: ???
An Example for Statement Coverage

```c
int fib (int n) {
    /* defined on n=0+ */

    if (n<2)
        return n;  \< an elementary statement \\
    else if (n>=2)
        return (fib(n-1)+fib(n-2)); \< an elementary statement

}
```

Test suit: \n<\n<n=3>,<n=1>
Observations

- Missing features are not detected
  - E.g. if number supplied is negative (n=-1), the function does not report an error
  - This test case is not needed for statement coverage criteria

- Does not cover implicit statements
  - (example on next slide)
Implicit statements

```cpp
bool flip (bool var) {
    bool local;
    if (isTrue(var))
        local = false;  \leftarrow \text{an elementary statement}
    return local;  \leftarrow \text{an elementary statement}
}
```

Test suit: <???> sufficient to cover all statements?
Implicit statements

bool flip (bool var) {
    bool local;
    if (isTrue(var))
        local = false; // an elementary statement
    return local; // an elementary statement
}

Test suit: <var=True> sufficient to cover all statements
Error that flip does not work with <var=false> is not detected
Implicit statements

```c
bool flip (bool var) {
    bool local = var;
    if (isTrue(var))
        local = false; \[elementary statement\]
    else {
    } \[implicit else statement\]
    return local; \[elementary statement\]
}
```

Test suit: `<var=True>` is **not sufficient** to cover all statements
Error that flip does not work with `<var=false>` will get detected
with suit: `<var=True>,<var=False>`
Basic Path Testing

◆ Select test suit such that basic paths are covered
  – This guarantees that every statement gets covered

◆ Representations for:
  – Elementary statements: assignment, i/o, call
  – Conditional statements: If then else
  – Conditional Loops: While-do, Repeat-until
  – Sequential composition: Two sequential statements
A Sequential Composition

S1: interest = balance * (x/100);
S2: balance = balance + interest;
A Branching Statement

S1: If (employee.performance=HIGH)
S2: incentive = x;
S3: else incentive = x/2;
S4: Print (employee.id, incentive)
A While Statement

S1: while (!end_of_file (file))
S2: read a value from file, and print it;
S3: file.close();
A Repeat Statement

A: tmp1 = x; tmp2 = x * x; i=0;
B: repeat
S1: x = x + tmp; i=i+1;
S2: until (x == tmp2);
S3: print (x, i);
A Switch Case Statement

S1: switch (choice) {
S2: case Tea: drink=prepareTea(); break;
S3: case Coffee: drink=prepareCoffee(); break;
S4: case Juice: drink=prepareJuice(); break;
}
S5: serve (drink);
A Compound Condition

If (a OR b)
x:    do_some_thing;
else
y:    do_something_else;

S1:a   S2:b
Cyclomatic Complexity

- No. of independent paths in the basis set
- Upper bound on no. of test that must be conducted
  - to ensure all statements get covered at least once
- = No. of regions (count outer region also)
- = No. of predicate nodes + 1
- = No. of edges – no. of vertices + 2
Cyclomatic Complexity Example

- =No. of regions (count outer region also): 3
- =No. of predicate nodes + 1: 2+1= 3
- =No. of edges – no. of vertices + 2 = 7-6+2=3
- i.e. basis set has 3 paths
The 3 independent paths in basis set
- a is found to be true
- a is not found to be true, but b is found to be true
- a is not found to be true, b is also not true

Test suite: <a=true, b=false>, <a=false, b=true>, <a=false, b=false> to test
→ If (a or b) then x else y;
Condition Testing

- **Simple conditions**
  - Boolean variable
  - Relational operator ( <, >, <=, ==, >= )

- **Compound conditions**
  - Composition of 2 or more conditions with
    Boolean operators (&&, ||, !)

- **Error could occur due to**
  - wrong variable values
  - Wrong choice of operators
  - Wrong expression inside conditions
  - Parenthesis problems
Condition Testing Strategies

- **Branch testing:**
  - Test for true and false for C
  - Every simple condition in C is executed at least once

- **Domain Testing:**
  - Boolean expression with n variables
    - Test for all possible $2^n$ values
  - Relational operators
    - For $a \, R \, b$:
      - test for $a$ less than $b$
      - $a$ greater than $b$
      - $a$ equal to $b$. 
Data Flow Testing Strategies

- Selection of paths according to data definition and usage

- DEF(S) = \{D| \text{statement S contains definition of D}\}
- USE(S) = \{D| \text{statement S contains use of D}\}

- DU chain of variable X: [X,S,S’] such that
  - X is in DEF(S);
  - X is in USE (S’);
  - Definition of X in statement S is live at statement S’
    - The definition is not overridden by another definition)
- A strategy: cover all DU chains at least once
Mutation Testing

- Change the program code a bit
- Test this mutant
- If the test does not generate a detectable error, the test case is not enough
  - Test once more and continue thus with mutation testing
Exercise: Try different strategies on the below program, make a few errors and try the tests again

```
AwardGrades (List L) {
1. int current=0;
2. while (current<L.size) {
3.     if (L[current].marks < 35)
        L[current].grade='F';
    else
4.        if (L[current].marks <50)
5.            L [current].grade='D';
    else
6.        if (L [current].marks <70)
7.            L [current].grade='C';
    else
8.            if (L [current].marks <90)
9.                L [current].grade='B';
    else
10.               L[current].grade='A';
11.               current=current+1;
}
}``