Note 5. Testing

Yan Shi

Lecture Notes for SE 3730 / CS 5730
Outline

- Formal and Informal Reviews
- Levels of Testing
  - Unit,
    - Structural Coverage Analysis
    - Input Coverage Testing:
      - Equivalence class testing
      - Boundary value analysis testing
    - CRUD testing
    - All pairs
      - integration,
      - system,
      - acceptance
- Regression Testing
Static and Dynamic Testing

- Static testing: the software is not actually executed.
  - Generally not detailed testing
  - Reviews, inspections, walkthrough

- Dynamic testing: test the dynamic behavior of the software
  - Usually need to run the software.
Black, White and Grey Box Testing

- **Black box testing**: assume no knowledge about the code, structure or implementation.

- **White box testing**: fully based on knowledge of the code, structure or implementation.

- **Grey box testing**: test with only partial knowledge of implementation.
  - E.g., algorithm review.
Reviews

- Static analysis and dynamic analysis
- Black-box testing and white-box testing

Reviews are **static white-box (?)** testing.
  - Formal design reviews: DR / FDR
  - Peer reviews: inspections and walkthrough
Formal Design Review

- The only reviews that are necessary for approval of the design product.
- The development team cannot continue to the next stage without this approval.
- Maybe conducted at any development milestone:
  - Requirement, system design, unit/detailed design, test plan, code, support manual, product release, installation plan, etc.
FDR Procedure

- **Preparation:**
  - find review members (5-10),
  - review in advance: could use the help of checklist.
- A short presentation of the document.
- Comments by the review team.
- Verification and validation based on comments.
- **Decision:**
  - Full approval: immediate continuation to the next phase.
  - Partial approval: immediate continuation for some part, major action items for the remainder.
  - Denial of approval: repeat FDR after action items.
- Generate FDR report.
- Follow up.
Inspections and Walkthrough

- **Inspection**: a step-by-step peer group review of a work product, with each step checked against predetermined criteria.

- **Roles:**
  - Moderator: the entire inspection procedure
  - Recorder: document defects found
  - Reviewer: everyone in the inspection team
  - Reader: read aloud small logical units
  - Producer: author of the work product.

- **Walkthrough**: the author leads the team through a manual execution of the product using predefined scenarios.
Requirement Reviews

- **Participants**: customers, system analysts, designers, developers, QAs.
- **Review leader**: project manager
- **Objectives**:
  - Check for accuracy and completeness of requirements and use cases.
  - Help designers, developers and QA understand the specifications and remove ambiguity.
- Customers can start acceptance test plan once this review is complete.
System Design Reviews

- **Participants**: system analysts, designers, developers and QAs.
- **Review leader**: system architect?
- **Objectives**:  
  - Check the design against the requirements
- System test plan is generally reviewed in conjunction with the system design and its requirements.
Detailed Design Reviews

- **Participants**: designers and developers
- **Review leader**: subsystem team leader?
- **Objectives**:
  - Make sure that detailed design of modules, classes and functions is consistent with system design.
- This usually includes a unit test plan and mapping of functions to requirements.
Test Plan Reviews

- **Participants:** QAs, development team
- **Review leader:** QA manager
- **Objectives:**
  - Everyone has the same interpretation on how the product will be tested.
  - Quality check on the test plan.
  - Final check on the requirements.
Code Inspections

- Participants: programmers
- When: unit testing has had at least one pass.
- Line by line, function by function code review against coding standards for
  - Efficiency
  - Coverage of requirements
  - Robustness against exceptions
  - Etc.
- Moderator’s job:
  - Organize and keep them on a positive note
  - Do not degenerate it into arguments on what is the best way to do something.
Unit Testing

- Test program units in **isolation**.
  - Errors found can be attributed to a specific unit ➔ easy to fix.
  - Verify that each distinct execution of a unit produces expected result.
- Often is part of the detailed design documents.

- Static unit testing: code inspections/walkthroughs
- Dynamic unit testing
- Test driven development
Dynamic Unit Test Environment

Test Driver

Unit under test

Results

Call and pass
Input parameters

Output parameters

Call

Acknowledge

Call

Acknowledge

Stub

Stub
Test Drivers and Stubs

- The driver and the stubs are never discarded after the unit test because they may need to be reused for regression test.
  - Update the driver and stubs if the unit is updated.
  - Put them under CM.

- Each unit should have its own test driver and stubs, Otherwise:
  - The driver may be too complicated
  - Side effects

- Each driver should have its own segregated input data set.
- Test drivers should automatically determine fail/pass.
Select Test Data for Unit Testing

- Control flow testing
- Data flow testing
- Domain testing
- Functional Program Testing
Control Flow Testing

- Draw a control flow graph from the unit under test
- Select a few control flow testing criteria
- Identify paths in the graph to satisfy the selected criteria
- Derive path predicate expressions from the selected paths
- Generate test inputs by solving the predicate expressions
Control Flow Graph (CFG): Symbols

- Computation
- Decision
- True
- False
- Sequential computation
- Decision point
- Merge point
A basic block in a program $P$ is a sequence of consecutive statements with a single entry and a single exit point.

- it is impossible to exit or halt at any point inside a basic block except at its exit point.
- when the block contains only one statement, the entry and exit points coincide.
```cpp
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;
```

OR

```cpp
cin >> x >> y << z;
if ( x > 1 && y == 0 )
    start
    z = z / x;
true
if ( z == 2 || y > 1 )
    true
false
return z;
end```

CFG: Example
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;

it considers the short-circuit evaluation!

But for most coverage criteria, we still use the previous one!
This one is only used for condition coverage.
begin
  int x, y, power;
  float z;
  input (x, y);
  if (y<0)
    power=-y;
  else
    power=y;
  z=1:
  while (power! =0){
    z=z*x;
    power=power−1;
  }
  if (y<0)
    z=1/z;
  output(z);
end
CFG Exercise:

```c
1   begin
2       int x, y, power;
3       float z;
4       input (x, y);
5       if (y<0)
6           power=-y;
7       else
8           power=y;
9       z=1:
10      while (power!=0){
11         z=z*x;
12         power=power-1;
13      }
14      if (y<0)
15          z=1/z;
16      output(z);
17   end
```
Control Flow Coverage Criteria

- Function coverage: make sure all functions in a module are executed. (easy)
- Statement coverage
- Decision coverage
- Condition coverage
- Condition/Decision coverage (C/DC)
- Modified Condition/Multiple Decision coverage (MC/DC)
- All-path coverage
cin >> x >> y >> z;
if (x > 1 && y == 0)
    z = z / x;
if (z == 2 || y > 1)
    z = z + 1;
return z;

Input | Output | Path
--- | --- | ---
2 0 4 3 1,2,3,4,5

predicate: (x>1&&y==0)&&(z/x==2||y>1)
Statement coverage exercise

Path1: 1 2 4 5 6 5 7 8 9
Predicate1:
\[ y < 0 \land -y \neq 0 \land -y - 1 = 0 \]
solution: \( y = -1 \)

Path2: 1 3 4 5 7 9
Predicate2:
\[ y \geq 0 \land y = 0 \]
solution: \( y = 0 \)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;

Each branching point has two decisions:
True or False!
A.K.A. Branch coverage

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 0 4</td>
<td>3</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>2 -1 4</td>
<td>4</td>
<td>1,3,5</td>
</tr>
</tbody>
</table>

predicates:
1: (x>1&&y==0) && (z/x==2 || y>1)
2: !(x>1&&y==0) && !(z==2 || y>1)
**Decision coverage exercise**

Path1: 1 2 4 5 6 5 7 8 9
Predicate1:
  \[ y < 0 \land -y \neq 0 \land -y - 1 = 0 \]
solution: \( y = -1 \)

Path2: 1 3 4 5 7 9
Predicate2:
  \[ y \geq 0 \land y = 0 \]
solution: \( y = 0 \)

Input | Output
--- | ---
x | y | z
1 | -1 | 1
1 | 0 | 1
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;

Each condition need to be tested for both True and False!
A.K.A. Predicate coverage
cin >> x >> y >> z;
if (x > 1 && y == 0)
    z = z / x;
if (z == 2 || y > 1)
    z = z + 1;
return z;

Each condition need to be tested for both True and False!
A.K.A. Predicate coverage

<table>
<thead>
<tr>
<th>Input</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y z</td>
<td></td>
</tr>
<tr>
<td>2 0 4</td>
<td>1,2,3,4,5,7,8</td>
</tr>
<tr>
<td>2 -1 4</td>
<td>1,2,3,5,6,8</td>
</tr>
<tr>
<td>-1 2 4</td>
<td>1,2,5,6,7,8</td>
</tr>
</tbody>
</table>
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;

Each condition need to be tested for both True and False!
A.K.A. Predicate coverage

- **Condition coverage cannot guarantee decision coverage!**
  - e.g., if ( A && B )

- All leaf-level Boolean expressions are considered as conditions.
### Condition/Decision Coverage

- Combines the requirements of both Condition and Decision coverage.

<table>
<thead>
<tr>
<th></th>
<th>A v B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A v B == True</td>
<td>// true decisions</td>
</tr>
<tr>
<td>2</td>
<td>A v B == False</td>
<td>// false decisions</td>
</tr>
<tr>
<td>3</td>
<td>A == True</td>
<td>// condition</td>
</tr>
<tr>
<td>4</td>
<td>A == False</td>
<td>// condition</td>
</tr>
<tr>
<td>5</td>
<td>B == True</td>
<td>// condition</td>
</tr>
<tr>
<td>6</td>
<td>B == False</td>
<td>// condition</td>
</tr>
</tbody>
</table>
Modified Condition/Multiple Decision coverage

- **MC/DC** enhances the Condition/Decision Coverage testing by requiring each Condition (A, B) to independently influence the Decision (True or False).
- There are eight conditions that must be met to ensure MC/DC.

<table>
<thead>
<tr>
<th>A  v  B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A  v  B == True // true decisions</td>
</tr>
<tr>
<td>2</td>
<td>A  v  B == False // false decisions</td>
</tr>
<tr>
<td>3</td>
<td>A == True // condition</td>
</tr>
<tr>
<td>4</td>
<td>A == False // condition</td>
</tr>
<tr>
<td>5</td>
<td>B == True // condition</td>
</tr>
<tr>
<td>6</td>
<td>B == False // condition</td>
</tr>
<tr>
<td>7</td>
<td>There is a B value such that A  v False == A  v B</td>
</tr>
<tr>
<td>8</td>
<td>There is an A value such that B  v False == A  v B</td>
</tr>
</tbody>
</table>
Modified Condition/Multiple Decision coverage

- **MC/DC** enhances the Condition/Decision Coverage testing by requiring each Condition (A, B) to independently influence the Decision (True or False).
- There are eight conditions that must be met to ensure MC/DC.

<table>
<thead>
<tr>
<th></th>
<th>A (\land) B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A (\land) B == True // true decisions</td>
</tr>
<tr>
<td>2</td>
<td>A (\land) B == False // false decisions</td>
</tr>
<tr>
<td>3</td>
<td>A == True // condition</td>
</tr>
<tr>
<td>4</td>
<td>A == False // condition</td>
</tr>
<tr>
<td>5</td>
<td>B == True // condition</td>
</tr>
<tr>
<td>6</td>
<td>B == False // condition</td>
</tr>
</tbody>
</table>

*Show that each condition independently affect the decision*

7. There is a B value such that \(A \land \text{True} == A \land B\)
8. There is an A value such that \(B \land \text{True} == A \land B\)
### MC/DC Example

- Just for the “**independent affect the decision**”:

  \[ A \land B \land C \]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
MC/DC Example

- Just for the “independent affect the decision”: $A \land B \land C$

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
MC/DC Example

- Just for the "independent affect the decision":

\[ A \land B \land C \]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
### MC/DC Example

- Just for the “**independent affect the decision**”:

\[ A \land B \land C \]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
## MC/DC Example

### A ^ B ^ C

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
MC/DC Exercise 1

- Generate a set of test cases of the following code to satisfy MC/DC coverage:

  if (A || B && C) then do something

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;

<table>
<thead>
<tr>
<th>Input</th>
<th>MC/DC?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x  y  z</td>
<td></td>
</tr>
<tr>
<td>2 0 4</td>
<td>False</td>
</tr>
<tr>
<td>1 0 4</td>
<td>True</td>
</tr>
<tr>
<td>2 2 4</td>
<td>False</td>
</tr>
</tbody>
</table>
All-Path Coverage

- Only possible for extremely small programs.
- In most cases, not feasible because of the large number of possible paths.
  
  — consider a program with a while loop.
  
  — how many possible path for the following code:

    if(a &&b&&c) then ...
    if(a||b||c) then ...
    if(a&&d||e) then ...
Data Flow Testing

1. Draw a data flow graph from the unit under test
2. Select a few data flow testing criteria
3. Identify paths in the graph to satisfy the selected criteria
4. Derive path predicate expressions from the selected paths
5. Generate test inputs by solving the predicate expressions
Data Flow Coverage Motivations

```
int xValue, yValue, zValue;
int count = 0;
cin >> xValue >> yValue;
xValue = 10;
yValue *= xValue;
if ( yValue < zValue )
    yValue = zValue;
```

- A variable is defined but never used! \texttt{count}
- A variable is used before defined! \texttt{zValue}
- A variable is defined twice before used! \texttt{xValue}
Data Flow Graph

- **Objective:** identify data definitions and their uses.

- **Definition** of a variable: when a value is moved into the memory location of that variable.
  
  e.g. `cin >> x; i = x;

- **Use** of a variable: when the value is fetched from the memory location of the variable.
  
  — **c-use** (computation use): `i++;`
  
  — **p-use** (predicate use): `while (i < 100)`

- a **def-clear subpath** for a definition `d` of `v` and a use `u` of `v` is a subpath in the DFG between `d` and `u` on which `v` is not redefined
Exercise

- How many def, c-uses and p-uses?

```plaintext
input(x);
z=x+1;
func(x*x*x);
if(A[x+1]>0)
    output(x);
for( i=0; i<z; i++)
    A[i] = B[i]+1;
```
Data Flow Graph

- DFG is a directed graph constructed as follows:
  - a sequence of definitions and c-uses is associated with each node
  - a set of p-uses is associated with each edge
DFG: def-c-use table

<table>
<thead>
<tr>
<th>Node i</th>
<th>def(i)</th>
<th>c-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x, y, z}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{z}</td>
<td>{x, z}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{z}</td>
<td>{z}</td>
</tr>
<tr>
<td>5</td>
<td>{}</td>
<td>{z}</td>
</tr>
</tbody>
</table>

```cpp
cin >> x >> y << z;
if (x > 1 && y == 0)
    z = z / x;
if (z == 2 || y > 1)
    z = z + 1;
return z;
end
```
DFG: predicate-p-use table

<table>
<thead>
<tr>
<th>edge (i,j)</th>
<th>predicate</th>
<th>p-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>x&gt;1&amp;&amp; y==0</td>
<td>{x,y}</td>
</tr>
<tr>
<td>(1,3)</td>
<td>!(x&gt;1&amp;&amp;y==0)</td>
<td>{x,y}</td>
</tr>
<tr>
<td>(2,3)</td>
<td>true</td>
<td>{}</td>
</tr>
<tr>
<td>(3,4)</td>
<td>z==2</td>
<td></td>
</tr>
<tr>
<td>(3,5)</td>
<td>!(z==2</td>
<td></td>
</tr>
<tr>
<td>(4,5)</td>
<td>true</td>
<td>{}</td>
</tr>
</tbody>
</table>

```c++
cin >> x >> y << z;
if ( x > 1 && y == 0 )
    start
    z = z/x;
true
    z = z + 1;
false
    return z;
end
```
DFG: labels on nodes

1. {d1(x), d1(y), d1(z)}
   cin >> x >> y << z;
   if ( x > 1 && y == 0 )
      start
   true
   false

2. {d2(z), c2(x), c2(z)}
   z = z/x;
   true
   false

3. {p3(z), p3(y)}
   if( z == 2 || y > 1 )
   true
   false

4. {d4(z), c4(z)}
   z = z + 1;
   true

5. {c5(z)}
   return z;

end
Exercise

- Construct DFG based on the given CFG.
Rapps and Weyuker’s Data Flow Criteria

- All-defs
- All-p-uses
- All-c-uses
- All-p-uses/Some-c-uses
- All-c-uses/Some-p-uses
- All-uses
- All-du-pairs
All-defs coverage

- Some **definition-clear subpath** from **each definition** to some use reached by that definition.

```c
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    start
    z = z/x;
    if ( z == 2 || y > 1 )
        if ( false )
            z = z + 1;
        return z;
else
    return z;
end
```
Exercise: All-def coverage

\{d_1(x), d_1(y), p_1(y)\}

\{d_2(\text{power}), c_2(y)\}

\{d_3(\text{power}), c_3(y)\}

\{d_4(z)\}

\{p_5(\text{power})\}

\{d_6(z), d_6(\text{power}), c_6(x), c_6(z), c_6(\text{power})\}

\{p_7(y)\}

\{d_8(z), c_8(z)\}

\{c_9(z)\}
Some **definition-clear subpath** for each variable from each definition to each c-use reached by that definition.

- `d1(x)-c2(x)`
- `d1(z)-c2(z)`
- `d1(z)-c4(z)`
- `d1(z)-c5(z)`
- `d2(z)-c4(z)`
- `d2(z)-c5(z)`
- `d4(z)-c5(z)`

Variables and their definitions:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d1(x)</code></td>
<td><code>{d1(x), d1(y), d1(z)}</code></td>
</tr>
<tr>
<td><code>d1(y)</code></td>
<td><code>{p1(x), p1(y)}</code></td>
</tr>
<tr>
<td><code>d1(z)</code></td>
<td><code>{d2(z), c2(x), c2(z)}</code></td>
</tr>
<tr>
<td><code>d2(z)</code></td>
<td><code>{p3(z), p3(y)}</code></td>
</tr>
<tr>
<td><code>d4(z)</code></td>
<td><code>{d4(z), c4(z)}</code></td>
</tr>
<tr>
<td><code>c5(z)</code></td>
<td><code>{c5(z)}</code></td>
</tr>
</tbody>
</table>

Code snippet:
```
cin >> x >> y << z;
if ( x > 1 && y == 0 )
start
true
z = z/x;
true
if( z == 2 || y > 1 )
false
z = z + 1;
true
return z;
false
end
```
Exercise: All-c-use coverage

\{d_1(x), d_1(y), \ p_1(y)\}

\{d_2(\text{power}), \ c_2(y)\}

\{d_3(\text{power}), \ c_3(y)\}

\{d_4(z)\}

\{d_5(\text{power})\}

\{d_6(z), \ d_6(\text{power}), \ c_6(x), \ c_6(z), \ c_6(\text{power})\}

\{p_7(y)\}

\{d_8(z), \ c_8(z)\}

\{c_9(z)\}
Some **definition-clear subpath** for each **variable** from each **definition** to each **p-use** reached by that definition and each successor node of the use.

- `d1(x)` to `p1(x)`: (1,2) (1,3)
- `d1(y)` to `p1(y)`: (1,2) (1,3)
- `d1(y)` to `p3(y)`: (1,3,4) (1,3,5)
- `d1(z)` to `p3(z)`: (1,3,4) (1,3,5)
- `d2(z)` to `p3(z)`: (2,3,4) (2,3,5)

The code snippet is:

```cpp
cin >> x >> y << z;
if ( x > 1 && y == 0 )
    start
true
false
z = z/x;  
if( z == 2 || y > 1 )
    2
false
true
false
if( z == 2 || y > 1 )
    3
true
false
true
false
z = z + 1;
return z;
end
```
Exercise: All-p-use coverage

{d1(x), d1(y), p1(y)}

{d2(power), c2(y)}

{d3(power), c3(y)}

{d4(z)}

{p5(power)}

{d6(z), d6(power),
c6(x), c6(z), c6(power)}

{p7(y)}

{d8(z), c8(z)}

{c9(z)}
Other coverage criteria

- **All-p-uses/Some-c-uses** is identical to all-p-use except when a variable doesn’t have any p-use:
  - in that case, use the c-use of that variable instead.

- **All-c-uses/Some-p-uses** is the same: when a variable doesn’t have any c-use, use its p-use instead.

- **All-uses**: combination of all-c-uses and all-p-uses.

- **All-du-paths**: for each definition of each variable, all def-clear paths that are cycle-free or simple-cycles to each use of the variable.
Relationship Among Coverage Criteria
Domain Testing

- Two classes of errors [Howden]:
  - **computation error**: a specific input data causes the program to execute the desired path but wrong output.
    
    e.g. result = f(b,a); where actually it should be f(a,b).
  - **domain error**: a specific input data causes the program to execute the wrong path.
    
    e.g. faults in the conditions. if(x>0) instead of if(x<0)

- **Domain Testing** aims at selecting test data to detect domain errors.
  - Equivalence class partitioning
  - Boundary value analysis
Functional Testing

- Black box testing
- Derive test cases from the software specifications.
- Each program is treated as a function
  - three core elements: input, output, and transformation.

- Key questions:
  - How to identify the input domain and output domain?
  - How to select test data from the huge pool of input domain?

- Random testing
- Boundary value analysis
- Equivalence class partitioning
- All-pairs
**Oracles – Truth Teller**

- **What is the expected output?**
  
  An oracle is a mechanism for determining whether a test has passed or failed.
  
  - specifications
  - previous version of a system
  - other similar products
  - domain experts
  - etc...
Random Testing

- Profile the inputs that users actually make.
- Randomly generate test inputs based on this profile.

- Pros:
  - inputs are not biased by testers/developers.
  - no need to design test cases.
  - quick and easy.

- Cons:
  - oracles are difficult to identify.
  - not efficient in detecting defects.
Equivalence Class Partitioning (EC)

- **Equivalence class:**
  - Two input values are in the same equivalence class if they are treated in the same way according to the specification, i.e. they cause the same path to be executed.

- **Idea:**
  - Partition the entire input domain into a finite number of subdomains (equivalence classes).
  - At least one test input from one EC.

- **Key question: how to define EC?**
  - requirement specifications
Equivalence Class Testing

- EC testing objectives:
  - To tell how many tests are necessary to test a given piece of software to a known level
  - To spot and eliminate redundant tests

- Ensure **completeness** and **non-redundancy**

**Completeness:**
\[ A = A_1 \cup A_2 \cup \ldots \cup A_7 \]

**Non-redundancy:**
\[ i \neq j \Rightarrow A_i \cap A_j = \emptyset \]
Define Equivalence Classes

- Myers guidelines for an individual input:
  1. An input condition specifies range \([a, b]\):
     - one EC for \(X < a\), one EC for \(a \leq X \leq b\), one EC for \(X > b\).
  2. An input condition specifies a set of values:
     - one EC for each value and one EC for an invalid value.
  3. An input condition specifies a “must-be” value:
     - one EC for the “must-be” value and one EC for other values.

How about multiple inputs?
A car insurance company provides insurance cover for motor vehicles.

When a customer applies for a quote on insurance, the agent requests the following information:

1. Vehicle type
2. Age of the driver

The system outputs a quote for insurance.

As a potential customer, aren’t you curious about how the quote is calculated?
Q: "Do you treat all types of vehicles alike?"
A: "No. We insure sport cars, trucks, sedans/vans, and motorcycles. We have a different way of getting a quote for each type."

Q: "Do you treat all ages the same way?"
A: "Well, not exactly. There are three classes of age: the under 21 kids, the great middle aged group from 21 to 65, and the Golden Oldies who are over 65. We do it differently for each group."
# Care Insurance Example

<table>
<thead>
<tr>
<th>Vehicle Type</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sport Cars</td>
<td>&lt; 21</td>
</tr>
<tr>
<td>Trucks</td>
<td>21 - 65</td>
</tr>
<tr>
<td>Sedans/Vans</td>
<td>&gt; 65</td>
</tr>
<tr>
<td>Motorcycles</td>
<td></td>
</tr>
</tbody>
</table>

These are only normal ECs.

How many test cases should we generate?
EC Testing for Multiple Inputs

- Weak Normal EC
- Strong Normal EC
- Weak Robust EC
- Strong Robust EC
Weak Normal EC Testing

- **Single Fault Assumption:**
  - Failures are *rarely* the result of the simultaneous occurrence of two (or more) faults

- Define normal ECs for each input variable

- Cover ECs for each variable

How many test-cases are needed?
What is the minimal number of tokens that are needed to be put in an $m \times n$ grid such that each row and column contains at least one token?

$\max(m,n)$
How many test cases should we generate?

4 = max (4, 3) for weak normal EC testing.
Strong Normal EC Testing

- Cover the all combinations of equivalence classes for the domain of all variables
- Multiple fault assumption

How many test-cases are needed?

\[ m \times n \]
Vehicle Type | Age
---|---
Sport Cars | < 21
Trucks | 21 - 65
Sedans/Vans | > 65
Motorcycles | > 65

How many test cases should we generate?

**12 for strong normal EC testing.**
Weak Robust EC Testing

- Include weak normal;
- Add one test case from each abnormal EC for each variable
Care Insurance Example

Vehicle Type       Age
Sport Cars         < 21
Trucks             21 - 65
Sedans/Vans       > 65
Motorcycles        > 65

How many test cases should we generate?

7 = max (4, 3) + 3 for weak robust EC testing.
Strong Robust EC Testing

- Similar as strong normal;
- Include all abnormal ECs
Care Insurance Example

Vehicle Type       Age

Sport Cars  < 21
Trucks       21 - 65
Sedans/Vans  > 65
Motorcycles  

How many test cases should we generate?
25 for robust normal EC testing.
Exercise: NextDate()

- Given a date: Month/Day/Year, decide the next date.
- Variables:
  - Month(M): 1-12
  - Year(Y): 1-3000
  - Day(D): 1-31

- How many ECs for each variable?
- Generate test case using strong normal EC.
Boundary Value Analysis (BVA)

- **Idea:**
  - select test data near the boundary of a data domain so that data both within and outside an EC are selected.
  - single fault assumption.

- **Goal:**
  - detect failures caused by incorrect implementation of boundaries.

Robust Worst Case BVA
Guidelines For Robust/Standard BVA

- For each input variable, select 7/5 values:
  1. Just below the minimum
  2. Minimum
  3. Just above the minimum
  4. Normal
  5. Just below the maximum
  6. Maximum
  7. Just above the maximum

Other input variables should use normal values.

Can we apply BVA to the car insurance example?
BVA Limitations

- Does not work well for Boolean variables
- Does not work well for logical variables
  - PIN, transaction type, vehicle type
Worst Case BVA Testing

- Disregard the single fault assumption.
- Consider all possible combinations.
- Cartesian product of the 5-tuple set (min, min+, nom, max-, max)

\[ X \times Y = \{ (x, y) \mid x \in X \land y \in Y \} \]
- If N independant input variables, then \(5^N\) tests

- Robust Worst Case BVA Testing:
  - Cartesian product of the 7-tuple set.
Triangle Example

inputs: integer a, b, c in the range [1,200]; output: triangle type

Standard BVA:

\[
\begin{align*}
\text{min} & = 1 \\
\text{min}+ & = 2 \\
\text{nom} & = 100 \\
\text{max}^- & = 199 \\
\text{max} & = 200 \\
\end{align*}
\]

How many test cases for the worst-case BVA?

For the robust worst-case BVA?

<table>
<thead>
<tr>
<th>Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>1</td>
<td>Isosceles</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>100</td>
<td>2</td>
<td>Isosceles</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>Equilateral</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>199</td>
<td>Isosceles</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>100</td>
<td>200</td>
<td>Not a Triangle</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>199</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>200</td>
<td>100</td>
<td>Not a Triangle</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>12</td>
<td>199</td>
<td>100</td>
<td>100</td>
<td>Isosceles</td>
</tr>
<tr>
<td>13</td>
<td>200</td>
<td>100</td>
<td>100</td>
<td>Not a Triangle</td>
</tr>
</tbody>
</table>
When an EC is based on ranges of a variable, we can combine BVA with EC testing.

How?
Exercise

- Consider a program that accepts the name of a grocery item and a list of the different sizes the item comes in, specified in ounces.
- The specifications state that the item name is to be alphabetic characters 2 to 15 characters in length.
- Each size may be a value in the range of 1 to 99, integer numbers only. The sizes are to be entered in ascending order (smaller sizes first). A maximum of five sizes may be entered for each item.
- The item name is to be entered first, followed by a comma, then followed by a list of sizes. A comma will be used to separate each size. Spaces (blanks) are to be ignored anywhere in the input.

List all ECs you can detect.
Apply equivalence class partitioning and boundary value analysis.
The Bitter Fact:
  - There’s too much to do in too little time.

Question:
  - How can we “test everything” without really testing everything?

All-Pair Testing / Pairwise Testing / Combinatorial Method
  - maximizing coverage of input combinations while minimizing the number of tests performed.
All Combinations – Strong EC

- Assume 2 input variables A and B; each has 5 equivalence classes of inputs:
  - \{a1,a2,a3,a4,a5\}, \{b1,b2,b3,b4,b5\},...
- How many test cases to cover all combinations?
  - 25 (Cartesian Product!)

<table>
<thead>
<tr>
<th>Variable A</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>a5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable B</td>
<td>a1 x b1</td>
<td>a2 x b1</td>
<td>a3 x b1</td>
<td>a4 x b1</td>
<td>a5 x b1</td>
</tr>
<tr>
<td>b1</td>
<td>a1 x b1</td>
<td>a2 x b1</td>
<td>a3 x b1</td>
<td>a4 x b1</td>
<td>a5 x b1</td>
</tr>
<tr>
<td>b2</td>
<td>a1 x b2</td>
<td>a2 x b2</td>
<td>a3 x b2</td>
<td>a4 x b2</td>
<td>a5 x b2</td>
</tr>
<tr>
<td>b3</td>
<td>a1 x b3</td>
<td>a2 x b3</td>
<td>a3 x b3</td>
<td>a4 x b3</td>
<td>a5 x b3</td>
</tr>
<tr>
<td>b4</td>
<td>a1 x b4</td>
<td>a2 x b4</td>
<td>a3 x b4</td>
<td>a4 x b4</td>
<td>a5 x b4</td>
</tr>
<tr>
<td>b5</td>
<td>a1 x b5</td>
<td>a2 x b5</td>
<td>a3 x b5</td>
<td>a4 x b5</td>
<td>a5 x b5</td>
</tr>
</tbody>
</table>

- How about 3 input variables? 125 = 5^3
- How about 5 input variables? 3125 = 5^5
All-Pair Strategy

- If there are 5 inputs and \textbf{a1} and \textbf{b2} will result in a failure, how many test cases will fail with all-combination testing? \(125 = 5^3\)

- Fact: Very few failures are caused by a combination of more than 2 input variables!
  - 3 / 109 in a medical software device failure study (Wallace 2000)

- Idea: Tests all possible discrete combinations for \textit{each pair} of input variables.

- How many test cases if there are 5 input variables? \(250 = \binom{5}{2} \times 25\)

\[
\binom{n}{k} = \frac{n(n-1) \ldots (n-k+1)}{k(k-1) \ldots 1},
\]
Savings with All-Pair Strategy

- The larger the number of input variables, the larger the savings in using All Pairs Testing.

<table>
<thead>
<tr>
<th>Number of Input Variables</th>
<th>Equivalence Classes per input</th>
<th>Total Combinations</th>
<th>All Pairs Combinations</th>
<th>% of tests savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>25</td>
<td>25</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>125</td>
<td>3 * 25 = 75</td>
<td>40%</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3125</td>
<td>10 * 25 = 250</td>
<td>92%</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>9,765,625</td>
<td>45 * 25 = 1,125</td>
<td>1-0.01152%</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>95,367,431,640,625</td>
<td>190 * 25 = 4,750</td>
<td>1-0.0000000004980736%</td>
</tr>
</tbody>
</table>
Probability of Catching a Failure

- Assume we have 10 input variables A, B, C, D, E, F, G, H, I and J each with 5 ECs.
- Assume there is one defect in the system and it occurs when we have both i5 and j5.

What is the probability of detecting this defect if we only have time to run one random combination of inputs?
  - Each combination of A, B, C, D, E, F, G, and H with the combination of i5 and j5 will fail: \( 390625 = 5^8 \)
  - The probability of catching a failure is \( \frac{5^8}{5^{10}} = 4\% \)

What would be the probability of detecting a defect if we only have time to run one random all-pair combination?

\[ \frac{1}{1125} = 0.089\% \]
CRUD Testing

- Create, Read, Update and Delete
- Often used to test database system and UI
- Use CRUD matrix to trace testing of object usage

Order Processing System CRUD Diagram 1

<table>
<thead>
<tr>
<th>Action</th>
<th>Customer</th>
<th>Customer Order</th>
<th>Customer Account</th>
<th>Customer Invoice</th>
<th>Vendor Invoice</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Customer Order</td>
<td>R</td>
<td>C</td>
<td>CR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process Customer Order</td>
<td>CRU</td>
<td></td>
<td></td>
<td>RU</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>Maintain Customer Order</td>
<td>U</td>
<td></td>
<td>U</td>
<td></td>
<td>RU</td>
<td></td>
</tr>
<tr>
<td>Terminate Customer Order</td>
<td>U</td>
<td></td>
<td>U</td>
<td></td>
<td>RU</td>
<td></td>
</tr>
<tr>
<td>Fill Customer Order</td>
<td>RU</td>
<td>RU</td>
<td></td>
<td>RU</td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Ship Customer Order</td>
<td>U</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate Vendor Invoice</td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pay Vendor Invoice</td>
<td>RU</td>
<td></td>
<td></td>
<td>RU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invoice Customer</td>
<td>RU</td>
<td>RU</td>
<td></td>
<td>C</td>
<td></td>
<td>CRUD</td>
</tr>
<tr>
<td>Maintain Inventory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CRUD Testing

- **Create**
  - Creating a brand **new** item one at a time
  - Creating a **duplicate** item
  - Creating **more than one** items in a single shot (if applicable)

- **Read**
  - Reading an **existing** item one at a time
  - Reading **more than one** existing items at a time
  - Reading a **non-existing** item
  - Reading in a **sorted** order (such as sort a grid in ascending or descending order)
  - Read an item based on **wild cards** (such as search or filter items)

- **Update**
  - Update an **existing** item one at a time
  - Update **more than one** existing items
  - Update a **non-existing** item

- **Delete**
  - Delete an **existing** item one at a time
  - Delete **more than one** existing items
  - Delete a **non-existing** item
Broken Box Testing

- A form of “gray-box” testing: with knowledge of algorithms, but not necessarily the exact implementation.

- Select input values to test different conditions of the algorithms.

- Why?
  - Some algorithms have known weaknesses: accuracy, performance, exceptions, etc.
Integration Testing

- tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems.
  - Big bang
  - Top-down
  - Bottom-up
  - Functional Incremental
Big Bang Integration Testing

- Test when all components or modules are integrated simultaneously

http://istqbexamcertification.com/what-is-integration-testing/
Top-Down Integration Testing

- Testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu).
- Components or systems are substituted by stubs.

http://istqbexamcertification.com/what-is-integration-testing/
Botton-Up Integration Testing

- Testing takes place from the bottom of the control flow upwards.
- Components or systems are substituted by drivers.

http://istqbexamcertification.com/what-is-integration-testing/
Incremental Integration Testing

- All components are integrated one by one, as developed.
- A test is carried out after each integration.
System Testing

- functionality testing
- security testing
- robustness testing
- compatibility testing
- load testing
- stability testing
- performance testing
- reliability testing
- ...

Security Testing

- **Objective:** to verify whether the system meets the security requirements

<table>
<thead>
<tr>
<th>Security Requirement</th>
<th>Data and processes are protected from</th>
</tr>
</thead>
<tbody>
<tr>
<td>confidentiality</td>
<td>unauthorized disclosure</td>
</tr>
<tr>
<td>integrity</td>
<td>unauthorized modification</td>
</tr>
<tr>
<td>availability</td>
<td>the denial of service to authorized users</td>
</tr>
</tbody>
</table>

- **Real story:**
  - *Excuse me while I turn off your pacemaker*
Robustness Testing

- Robustness: how sensitive a system is to erroneous input and changes in its operational environment.

- Approach: deliberately break the system/environment and see how it works.
  - Boundary value analysis
  - Power cycling test: create a power glitch
  - Online insertion and removal (OIR) test: of modules in both idle and heavy load operations
  - Fault tolerance test: can switch to standby modules
A program must run on a wide range of computers.

Even on the same brand of computer there may be different configurations of memory, printers, bios, dlls, or other peripherals.

A program should work well on common configurations of computers.
Compatibility Testing

- **Compatibility** – checks the way one product works with another.
  - They may both use the *same file* or *communicate* with each other.
  - **Forward compatibility** – old products can handle newer files/program versions; this is generally NOT expected.
  - **Backward compatibility** – newer products can work with older files/program versions; this is very desirable.

- **Multiplatform** – programs that run on multiple OS, DBs, hardware, etc. should be compatible to the degree advertised. *(Port Testing)*

- **Conversion Tests**
  - if a program is not directly compatible, is there an intermediate program available to convert data into a form that can work?
  - This happens when database scheme or file formats change between versions.
Performance Testing

- To verify whether the performance of the actual system meets the performance requirement.

- Performance metrics depend on specific application types.
  - response time
  - memory
Load Testing

- **Volume Tests**: study the largest and smallest tasks the program can handle. For example,
  - feed a compiler a huge program
  - feed empty input files to a system

- **Soak Tests**: run a system at normal load for a long time;
  - monitor system resources, e.g., memory, disk, queue length, overheating, bandwidth...

- **Stress Tests**: burst rate activity
  - start with small load and ramp it up till the program fails.
Background Testing

- Programs that run on multitasking environments need to be tested while other typical programs are active.

- Need to make sure the program is not starved for memory, CPU, disk or communications.
Reliability Testing

- To measure the ability of the system to remain operational for long periods of time.

- **MTTF (mean time to failure)**
  - difference of time in two consecutive failures
  - recorded during system testing process.
  - often use random testing for measurement.

- **MTTR (mean time to repair/recovery)**: the average time required to fix a failure
Installation Testing

- Installation is the users’ first exposure to a software product.
  - Does the install program configure the program for software end hardware environment?
  - Can experienced user customize the installation?
  - Grandma Test: how long does it take for an inexperienced user to install the system?
Alpha and Beta Testing

- Both are considered as acceptance testing.
- **Alpha test:**
  - At least one system-level test is done.
  - There may be known critical defects in the system, but all major features are working.
  - The code is sent to a very limit number of “friendly” users to test under real-world conditions.
- **Beta test:**
  - All features are working.
  - Very few (if known) defects still exist.
  - Real world users test the system in real world conditions.

- **Apple:** very little and private beta testing.
  - iphone antenna problem
- **Microsoft:** very huge and public beta testing
  - windows developer preview
Summary

Testing

Static
- formal reviews
- peer reviews

Dynamic
- white box
  - control flow coverage
  - data flow coverage
- black box
  - function testing
  - domain testing

unit test
integration test
system test
acceptance test
regression test

EC
BVA
All-pair
CRUD
int someFunction(int a, int b) {
    int result = 0;
    if (a < b)
        system.exit(0);
    else {
        int c = a + b;
        int I = 0;
        while (i < c) {
            result = (result + a) / b;
            i++;
        }
    }
    return result;
}