Advanced Software Engineering: Software Testing
COMP 3705(L3)

Sada Narayanappa

Anneliese Andrews
Thomas Thelin
Carina Andersson
Lecture

- Chapter 10 (Lab 3)
  - Software inspections
  - Fault content estimation

- Chapter 5 (Lab 4)
  - White-box testing techniques
Definitions

• **Software**
  • Source code
  • All related artifacts – design docs etc.

• **Testing**
  • Revealing defects
  • Dynamic - Execution
  • Static - Reviews
Why Review?

- Main objective
  - Detect faults

- Other objectives
  - Inform
  - Educate
  - Learn from (others) mistakes

- Faults affect software quality negatively
Types of Reviews

- Formal/Informal
- Technical/Managerial
- When does it occur
  - Colleague ask the other
  - Formally
- Types
  - Inspection
  - Walkthrough

Benefits of Reviews:
- Increased productivity
- Higher customer satisfaction
- Teaching tool for junior staff
- Identification of reusable assets
What to inspect?

- Analysis & Requirements
- Design
- Implementation
- System Test
- Integration & Function test
- Inspection
- Inspection
- Inspection
- Inspection
Relative cost of faults

How did it start?

- Fagan inspections (IBM, early 1970’s, Checklist)
- Phases
  - Overview, Preparation, Meeting, Rework, Follow-up
    
    Fault searching at meeting! – Synergy

- Roles
  - Author (designer), reader (coder), tester, moderator

- Classification
  - Major and minor
How did it start?

- Fagan 1976
  - Inspections detected 82% of the faults
  - Unit testing detected 18% of the faults
  - Productivity was increased by 23%
Getting the best from inspections

- The author
  - How do you react?
  - “…is in the hot seat”

- The development team
  - Better prepared
  - Feedback
  - Communication
Getting the best from inspections

- The inspection team
  - Critical thinking
  - Ability to detect omissions
  - Who should participate in inspections?

- Cost-effective verification
  - Minimising cost of correction
  - Is it cost-effective?
Inspections

- **Objective/Goals:**
  - Detect faults
  - Collect data
  - Communicate information

- **Roles**
  - Moderator
  - Reviewers (Inspectors)
  - Presenter
  - Author

- **Elements**
  - Planned, structural meeting
  - Preparation important
  - Team – 3-6 people

- **Disadvantages**
  - Short-term cost

- **Attentions to:**
  - Quality
  - Adherence to standards
  - Testability
  - Traceability
  - Anything else …?
Walkthroughs

- **Objective/Goals:**
  - Detect faults
  - Become familiar with the product

- **Roles**
  - Presenter (author)
  - Reviewers (Inspectors)

- **Elements**
  - Planned meeting
  - Team – 2-7 people
  - Brainstorm

- **Disadvantage**
  - Find fewer faults
Cognitive View

- Process
  - Comprehension – overview or preparation
  - Fault searching – preparation or meeting

- Inexperienced reviewers
  - Concrete representation (syntax)

- Experienced reviewers
  - Abstract representation (semantic)
  - Load less detailed information
Reading Techniques

- Ad hoc
- Checklist
- Active Design Review
- Defect-based reading
- Perspective-based reading
- Usage-based reading
Active design review

- Active design review (1990)
  - Forces reviewers to actively inspect
  - Question-based
    - e.g. how can this be implemented
Defect-based Reading

**Omission**
- Functionality
- Performance
- Interface
- Environment

**Commission**
- Ambiguity
- Inconsistency
- Incorrect
- Wrong

**Ad Hoc**
- Checklist

**Data Type Inconsistencies**
1. Identify all data objects mentioned...
2. Are all data objects mentioned...

**Incorrect Functionality**
1. For each functional requirement identify...
2. Are all values written to each input...

**Ambiguities or Missing Functionality**
1. Identify the required precision, response...
2. Are all required precisions indicated?
Perspective-based Reading

- Scenarios
- Purpose
  - Decrease overlap
  - Improved effectiveness
Usage-Based Reading

Model of usage

Design
Inspection Techniques

- **N-fold inspections (1993)**
  - N independent groups inspect the document
  - Small groups

- **Phased inspections (1985)**
  - Two or several inspection sessions
  - With or without fault correction in-between
Inspection Techniques

- **Stepwise abstraction (1982)**
  - Code
  - Decompose the code to subprograms
  - Each subprogram performs one function

- **Usability inspections**
  - Focus on end user
    - e.g. can learn, availability, can understand
  - Several sub-techniques
    - e.g. Cognitive walk-through
Inspections

Individual inspection

Meetin

Estimation
Capture-recapture
Capture - Recapture
Presenting the Information

Diagram representing

\[ N, D, \hat{N} - D, \text{ReviewersData} \]

\[ \hat{N} \text{ is the estimated number} \]
Capture-Recapture Models

- Degree of freedom
  - Models
  - Estimators

- Four basic models used for inspections

- Prerequisites for all models
  - All reviewers work independently of each other
  - It is not allowed to inject or remove faults during inspection
Prerequisites

Model 1, M0
Detection probabilities

Model 2, Mh
Detection probabilities

Model 3, Mt
Detection probabilities

Model 4, Mth
Detection probabilities
Testing Approaches

Black Box Testing

White Box Testing
Exhaustive testing

There are many possible paths!

Selective Testing
Selective testing

a selected path

SELECTIVE:

- Control flow testing
- Data flow testing
- Loop testing
- Fault-based testing

Advanced Software Engineering
## Control flow – coverage

<table>
<thead>
<tr>
<th></th>
<th>Statement coverage</th>
<th>Decision coverage</th>
<th>Condition coverage</th>
<th>Decision/condition coverage</th>
<th>Multiple condition coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Each statement is executed at least once</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>Each decision takes on all possible outcomes at least once</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>Each condition in a decision takes on all possible outcomes at least once</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>All possible combinations of condition outcomes in each decision occur at least once</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Statement coverage

- Execute each statement at least once

- Tools are used to monitor execution

- Requirement may be:
  - no dead code
**Decision/Branch coverage**

- **McCabe cyclomatic**
  The number of independent paths needed to cover all paths at least once in a program
  - Count number of conditional expressions. If compound conditional expressions, add one per compound item.
  - Visualize by drawing a flow graph
  - Alternative way: \( CC = \#(edges) - \#(nodes) + 2 \)
McCabe cyclomatic

Control complexity vs Data complexity, an example:

```
case A is
  when "One" => : =1;
  when "Two" => : =2;
  when "Three" =>
    : =3;
  when "Four" =>
    : =4;
  when "Five" =>
    : =5;
end Case

Strings : array (1..4) of STRING:=
  ("One","Two","Three","Four","Five");
i:=1;
loop
  exit when Strings(i)=A;
  i:=i+1;
end loop;

CC = 2
```

CC = 5
Condition Coverage

- Test all possible conditions in a program
- A condition in a program may contain:
  - Boolean operators and variables
  - Relational operators
  - Arithmetic expressions
  - …
Decision / Condition coverage

If $A < 10$ and $B > 250$ then …

**Decision**
1. $A = 2; B = 300$ (True)
2. $A = 12; B = 300$ (False)

**Condition (Decision/Condition)**
1. $A = 2; B = 300$ (True)
2. $A = 12; B = 200$ (False)

**Multiple condition**
1. $A = 2; B = 200$ (False)
2. $A = 12; B = 200$ (False)
3. $A = 2; B = 300$ (True)
4. $A = 12; B = 300$ (True)
Example (Decision coverage)

```c
void f() {
    if(count > 10){
        fixed_count = fixed_count + count;
        done = 1;
    } else if(count > 5){
        fixed_count --;
    } else {
        fixed_count = count * 4;
    }
}
```
void f()

if(count > 10)
    fixed_count = fixed_count + count;
    done = 1;

if(count > 5)
    fixed_count --;

fixed_count = count * 4;

return;
Decision Coverage

- **Independent paths:**
  1. (a) 1, 2, 3, 8
  2. (b) 1, 2, 4, 6, 7, 8
  3. (c) 1, 2, 4, 5, 7, 8

- **Test cases:**
  1. (a) count ← 13
  2. (b) count ← 8
  3. (c) count ← 2
The liability procedure:

procedure liability (age, sex, married, premium);

begin
    premium := 500;
    if ((age < 25) and (sex = male) and (not married)) then
        premium := premium + 1500;
    else
        begin
            if (married or (sex = female)) then
                premium := premium – 200;
            if ((age > 45) and (age < 65)) then
                premium := premium – 100;
        end;
    end;
end;
There are only two statements in this program, and any combination of inputs will provide coverage for both statements.

Different definitions!!!
## Decision/Branch coverage

<table>
<thead>
<tr>
<th>Decision coverage</th>
<th>Age</th>
<th>Sex</th>
<th>Married</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF-1</td>
<td>&lt; 25</td>
<td>Male</td>
<td>False</td>
<td>(1) 23 M F</td>
</tr>
<tr>
<td>IF-1</td>
<td>&lt; 25</td>
<td>Female</td>
<td>False</td>
<td>(2) 23 F F</td>
</tr>
<tr>
<td>IF-2</td>
<td>*</td>
<td>Female</td>
<td>*</td>
<td>(2)</td>
</tr>
<tr>
<td>IF-2</td>
<td>&gt;= 25</td>
<td>Male</td>
<td>False</td>
<td>(3) 50 M F</td>
</tr>
<tr>
<td>IF-3</td>
<td>&lt;= 45</td>
<td>Female</td>
<td>*</td>
<td>(2)</td>
</tr>
<tr>
<td>IF-3</td>
<td>&gt; 45, &lt; 65</td>
<td>*</td>
<td>*</td>
<td>(3)</td>
</tr>
</tbody>
</table>
## Condition coverage

<table>
<thead>
<tr>
<th>Decision coverage</th>
<th>Age</th>
<th>Sex</th>
<th>Married</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF-1</td>
<td>&lt; 25</td>
<td>Female</td>
<td>False</td>
<td>(1) 23 F F</td>
</tr>
<tr>
<td>IF-1</td>
<td>&gt;= 25</td>
<td>Male</td>
<td>True</td>
<td>(2) 30 M T</td>
</tr>
<tr>
<td>IF-2</td>
<td>*</td>
<td>Male</td>
<td>True</td>
<td>(2)</td>
</tr>
<tr>
<td>IF-2</td>
<td>*</td>
<td>Female</td>
<td>False</td>
<td>(1)</td>
</tr>
<tr>
<td>IF-3</td>
<td>&lt;= 45</td>
<td>*</td>
<td>*</td>
<td>(1)</td>
</tr>
<tr>
<td>IF-3</td>
<td>&gt; 45</td>
<td>*</td>
<td>*</td>
<td>(3) 70 F F</td>
</tr>
<tr>
<td>IF-3</td>
<td>&lt; 65</td>
<td>*</td>
<td>*</td>
<td>(2)</td>
</tr>
<tr>
<td>IF-3</td>
<td>&gt;= 65</td>
<td>*</td>
<td>*</td>
<td>(3)</td>
</tr>
</tbody>
</table>
Data Flow Testing

- Identifies paths in the program that go from the assignment of a value to a variable to the use of such variable, to make sure that the variable is properly used.

- Definitions
  - Def – assigned or changed
  - Uses – utilized (not changed)
    - Computation: \((c\text{-}use)\)
      e.g. right-hand side of an assignment, an index of an array, parameter of a function.
    - Predicate: \((p\text{-}use)\)
      branching the execution flow, e.g. in an if statement, while statement, for statement.

- Example: All def-use paths (DU) requires that each DU chain is covered at least once

- Goal is to cover all Def-Use Paths
Considering x, there are two DU paths:

(a) [2]-[4]
(b) [2]-[6]

We need therefore to derive test cases to match the following conditions:

(a) k() is executed and p(cond1) is true
(b) k() is executed and p(cond1) is false and q(cond2) is true
**Loop testing**

- Simple loop
- Nested loops
- Concatenated loops
- Unstructured loops

Advanced Software Engineering
Loop testing: simple loops

Minimum conditions - simple loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. $m$ passes through the loop $m < n$
5. $(n-1)$, $n$, and $(n+1)$ passes through the loop

where $n$ is the maximum number of allowable passes
**Nested loops**

- **Extend simple loop testing**
- **Reduce the number of tests:**
  - start at the innermost loop; set all other loops to minimum values
  - conduct simple loop test; add out of range or excluded values
  - work outwards while keeping inner nested loops to *typical* values
  - continue until all loops have been tested
Concatenated loops

- Same strategies as simple loops if the loops counters are independent
- Same strategies as nested loops if the loops counters depend on each other:

```c
int k;
for (k=0; k<10; k++)
{
    w();
    if p(m) break;
}
for (; k<10; k++)
{
    r();
}
```
**Mutation testing (fault-based testing)**

A method for evaluation of testing effectiveness

1. Mutate the program using the mutation operators
2. Run test suite on mutants
3. Obtain statistics on live and dead mutants
4. Create more test cases and iterate 2-4 until sufficient number of mutants are killed
Example mutation operators

- Relational operator replacement
- Logical operator replacement
- Arithmetic operator replacement
- Unary operator removal/insertion
- Comparable constant replacement
- Comparable variable replacement
- Comparable array replacement
Example

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i < range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}
Relational operator mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i < range; i++)
        if a[i] >= a[imax]
            imax := i;
    return imax;
}

Changed From:

a[i] > a[imax]
Variable operator mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i < range; i++)
        if i > a[imax]
            imax := i;
    return imax;
}

Changed From:

a[i] > a[imax]

<table>
<thead>
<tr>
<th></th>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Variable operator mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 0; i < range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}

Changed From:

<table>
<thead>
<tr>
<th></th>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Why is white-box testing not enough?

- Missing features
  - Missing code
- Timing and concurrency issues
- Different states of the software
  - Astronomical amount of different paths through the software
  - Different paths can reveal different defects
- Variations of input conditions
- Variations of data
- Qualities
  - Performance
  - Robustness
  - Reliability
  - Usability
  - …