CIS W338

SWE08
Testing the Program

How does software fail?

- Wrong requirement: not what the customer wants
- Missing requirement
- Requirement impossible to implement
- Faulty design
- Faulty code
- Improperly implemented design
Terminology

- Fault identification: what fault caused the failure?
- Fault correction: change the system
- Fault removal: take out the fault

Types of faults

- Algorithmic fault
- Syntax fault
- Computation and precision fault
- Documentation fault
- Stress or overload fault
- Capacity or boundary fault
- Timing or coordination fault
- Throughput or performance fault
- Recovery fault
- Hardware and system software fault
- Documentation fault
- Standards and procedures fault
Table 8.1. IBM orthogonal defect classification.

<table>
<thead>
<tr>
<th>Fault type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Fault that affects capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure</td>
</tr>
<tr>
<td>Interface</td>
<td>Fault in interacting with other components or drivers via calls, macros, control blocks or parameter lists</td>
</tr>
<tr>
<td>Checking</td>
<td>Fault in program logic that fails to validate data and values properly before they are used</td>
</tr>
<tr>
<td>Assignment</td>
<td>Fault in data structure or code block initialization.</td>
</tr>
<tr>
<td>Timing/serialization</td>
<td>Fault that involves timing of shared and real-time resources</td>
</tr>
<tr>
<td>Build/package/merge</td>
<td>Fault that occurs because of problems in repositories, management changes, or version control</td>
</tr>
<tr>
<td>Documentation</td>
<td>Fault that affects publications and maintenance notes</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Fault involving efficiency or correctness of algorithm or data structure but not design</td>
</tr>
</tbody>
</table>

Hewlett-Packard fault classification.

**ORIGIN:** WHERE?

**TYPE:** WHAT?

**MODE:** WHY?
Testing Issues

- Test organization
- Attitudes toward testing
- Who performs the tests?
- Views of the test objectives
Test Organization

- Module, Component, or Unit Testing
- Integration testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing
Attitudes Toward Testing

- New programmers not accustomed to viewing testing as a discovery process.
- Testing, by the student, generally designed to demonstrate that the program works under certain circumstances.
- May consider the presence of faults as a critique of ability.
- Customers are not interested in programs that work under certain circumstances, they want programs that work under all circumstances.

Egoless Programming

- Programs are viewed as components of a larger system.
- Programs are not the property of those who wrote them.
- When a fault is discovered or a failure occurs, the egoless development team is concerned with correcting the fault, not with placing the blame.
Who Performs the Tests?

- Even with the egoless approach, individuals have difficulty removing personal feelings from the testing process.
- Using an independent testing team avoids the conflict between personal responsibility for faults and the need to discover as many as possible.
- An independent team also provides an independent interpretation of the requirements.

Views of the Test Objectives

- Closed or Black Box Testing
  - The item is viewed from the outside. Testing feeds input and observes the output.
  - Goal is to exercise all possible inputs.
- Open or White Box Testing
  - Test data is constructed based upon a knowledge of the program’s internal structure.
  - Tests can be designed to exercise all statements or control paths within the component.
Example

- Consider a component that computes the roots to the equation:
  \[ ax^2 + bx + c = 0 \]
- Impossible to generate all possible values of \( a \), \( b \), and \( c \).
- Testing all combinations of positive, negative, and zero generates 27 possible cases.
- May want to consider cases where \( b^2 - 4ac \) are positive, negative, or zero.
  - But this makes an assumption about the implementation!
White Box Example

• The flowchart shows a program that executes $n \times m$ times.
• Would not want to pick large values of $n$ and $m$.
• Consider the cases where
  – $n > m$
  – $n = m$
  – $n < m$

Unit Testing

• Examining the Code
• Proving Code Correct
• Testing Program Components
Examining the Code

- **Code Walkthroughs**
  - Code and accompanying documentation is presented by the author to the review team.
  - Author leads and controls the discussion.
  - Atmosphere is informal.

- **Code Inspections**
  - Review team checks the code against a prepared list of concerns (checklist).
  - Several steps involved – team members generally review code individually and reconvene.
  - Trained moderator leads and controls the discussion.
  - Atmosphere is formal.

### Table 8.2. Typical inspection preparation and meeting times.

<table>
<thead>
<tr>
<th>Development artifact</th>
<th>Preparation time</th>
<th>Meeting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements document</td>
<td>25 pages per hour</td>
<td>12 pages per hour</td>
</tr>
<tr>
<td>Functional specification</td>
<td>45 pages per hour</td>
<td>15 pages per hour</td>
</tr>
<tr>
<td>Logic specification</td>
<td>50 pages per hour</td>
<td>20 pages per hour</td>
</tr>
<tr>
<td>Source code</td>
<td>150 lines of code per hour</td>
<td>75 lines of code per hour</td>
</tr>
<tr>
<td>User documents</td>
<td>35 pages per hour</td>
<td>20 pages per hour</td>
</tr>
</tbody>
</table>

### Table 8.3. Faults found during discovery activities.

<table>
<thead>
<tr>
<th>Discovery activity</th>
<th>Faults found per thousand lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements review</td>
<td>2.5</td>
</tr>
<tr>
<td>Design review</td>
<td>5.0</td>
</tr>
<tr>
<td>Code inspection</td>
<td>10.0</td>
</tr>
<tr>
<td>Integration test</td>
<td>3.0</td>
</tr>
<tr>
<td>Acceptance test</td>
<td>2.0</td>
</tr>
</tbody>
</table>
Proving code correct

• Formal proof techniques
• Symbolic execution
• Automated theorem-proving

Formal Proof

Based upon notation and concepts developed by C. A. R. Hoare.

\[ P(Q|R) \]

If the proposition \( P \) is true, execution of the program \( Q \) will result in the proposition \( R \) being true.

An alternative notation (adapted for C and C++)

\[ /* P */ Q /* R */ \]

or

//Precondition
{block of statements}
//Postcondition
Axiom of Assignment

Let $P$ be some proposition containing the symbol $x$. Let $e$ be some expression. (Note, the evaluation of $e$ is assumed to be without side-effects.) $P_0$ is the proposition $P$ in which every occurrence of $x$ is replaced by $e$. Note this is not every occurrence of $e$ in $P_0$ replaced by $x$. Or using our C notation

```
/* P0 */
x = e;
/* P */
```

Rules of Consequence

A rule of inference has this general form:

If $X$ and $Y$ then $Z$

which may be interpreted

If $X$ has been proven (or is an axiom) and $Y$ has been proven (or is an axiom) then, $Z$ is considered proven.

If $P \{Q\} R$ and $R \supset S$ then $P \{Q\} S$
If $P \{Q\} R$ and $S \supset P$ then $S \{Q\} R$
Rule of Composition

Programs generally consist of more than one statement. Using our C notation:

If
  /* P */ Q1 /* R1 */
and
  /* R1 */ Q2 /* R */
then
  /* P */ {Q1 Q2} /* R */

Rule of Iteration

Let P be a proposition whose truth is not changed by the execution of the statement S.
Let B be a proposition whose truth is necessary for the continued execution of the loop.
In the C notation
If
  /* P && B */
  S
  /* P */
then
  /* P */
  while (B) S
  /* !B && P */
Consider the following program:
\[
\begin{align*}
r &= x; \\
q &= 0; \\
\text{while } (y <= r) \\
&\quad \{ \\
&\quad \quad r = r - y; \ \\
&\quad \quad q = 1 + q; \\
&\quad \} \\
\end{align*}
\]
We wish to prove that this program terminates with \( q \) being the quotient of \( x \div y \) and \( r \) is the remainder. Formally:
// Precondition: none
\[
\begin{align*}
r &= x; \\
q &= 0; \\
\text{while } (y <= r) \\
&\quad \{ \\
&\quad \quad r = r - y; \ \\
&\quad \quad q = 1 + q; \\
&\quad \} \\
\end{align*}
\]
// Postcondition: (r < y) && (x == r + y*q)

<table>
<thead>
<tr>
<th>Line</th>
<th>Formal Proof</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{true} \supset x = x + y \times 0 )</td>
<td>Property of arithmetic</td>
</tr>
<tr>
<td>2</td>
<td>( x = x + y \times 0 { r = x; } x = r + y \times 0 )</td>
<td>Axiom of Assignment</td>
</tr>
<tr>
<td>3</td>
<td>( x = r + y \times 0 {q = 0;} x = r + y \times q )</td>
<td>Axiom of Assignment</td>
</tr>
<tr>
<td>4</td>
<td>( \text{true} { r = x; } x = r + y \times 0 )</td>
<td>Rule of consequence (1,2)</td>
</tr>
<tr>
<td>5</td>
<td>( \text{true} { r := x; \ q := 0;} x = r + y \times q )</td>
<td>Rule of composition (4,3)</td>
</tr>
<tr>
<td>6</td>
<td>( x = r + y \times q \land y \leq r \supset ) ( x = (r - y) + y \times (1 + q) )</td>
<td>Property of arithmetic</td>
</tr>
<tr>
<td>7</td>
<td>( x = (r - y) + y \times (1 + q) ) ({ r = r - y;} ) ( x = r + y \times (1 + q) )</td>
<td>Axiom of Assignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
| 8 | \[ x = r + y \times (1 + q) \]  
\{ q = 1 + q \}  
\[ x = r + y \times q \]  | Axiom of Assignment |
| 9 | \[ x = (r - y) + y \times (1 + q) \]  
\{ r = r - y; \ q = 1 + q; \}  
\[ x = r + y \times q \]  | Axiom of composition  
(7,8) |
| 10 | \[ x = r + y \times q \land y \leq r \]  
\{ r := r - y; \ q := 1 + q; \}  
\[ x = r + y \times q \]  | Rule of consequence  
(6, 9) |
| 11 | \[ x = r + y \times q \]  
while (y \leq r)  
\{ r = r - y; \ q = 1 + q; \}  
\(\neg (y \leq r) \land x = r + y \times q \)  | Rule of iteration |

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 12 | true  
\{ r = x; \ q = 0; \}  
while (y \leq r)  
\{ r = r - y; \}  
\{ q = 1 + q; \}  
\}  
\(\neg (y \leq r) \land x = r + y \times q \)  | Rule of composition |
Advantages and Disadvantages

- Formal proofs can discover algorithmic faults in the code.
- Regular use forces rigorous and precise specification.
- More successful when the program and the proof are developed simultaneously.
- Much work is involved in setting up and carrying out the proof.
- Proofs are not always correct. The history of mathematics is full of proofs that were later shown to be fallacious.

Symbolic Execution

- Code is simulated using symbols instead of data values.
- The program is viewed as having a state determined by the input data and conditions.
- As each line of the code is examined, the technique determines whether the state has changed.
- The final state is checked against the desired output.
Automated Theorem Proving

- Automated tools perform the proving process.
- Tool would accept as input
  - The input data and conditions
  - The output data and conditions
  - The source code.
- Such a tool is not trivial.
- Limited symbolic execution tools have been developed.
- General purpose tool is impossible, since it is equivalent to the "Turing halting problem."

Test thoroughness

- Statement testing
  - Every statement is executed at least once.
- Branch testing
  - At every decision point, each branch is chosen at least once.
- Path testing
  - Every path through the code is executed at least once.
- Definition-use testing
  - Every path from every definition of every variable to every use of that definition is exercised.
Test Thoroughness

• All-uses testing
  – The test set includes at least one path from every definition to every use that can be reached by that definition.

• All-predicate-uses/some-computational-uses
  – For every variable and every definition of that variable, a test includes at least one path from the definition to every predicate use.

• All-computational-uses/some-predicate-uses
  – For every variable and every definition of that variable, a test includes at least one path from the definition to every computational use.
1. Pointer = FALSE
2. X > K?  
   YES  POINTER = TRUE
   NO  
3. 
4. X = X + 1
5. CALL SUB (X, POINTER, RESULT)
6. RESULT > 0?  
   YES  PRINT RESULT
   NO 
7. 

Statement testing:
1-2-3-4-5-6-7

Branch testing:
1-2-3-4-5-6-7
1-2-4-5-6-1

Path testing:
1-2-3-4-5-6-7
1-2-3-4-5-6-1
1-2-4-5-6-7
1-2-4-5-6-1
Comparing techniques

Table 8.5. Fault discovery percentages by fault origin.

<table>
<thead>
<tr>
<th>Discovery technique</th>
<th>Requirements</th>
<th>Design</th>
<th>Coding</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototyping</td>
<td>40</td>
<td>35</td>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>Requirements review</td>
<td>40</td>
<td>15</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Design review</td>
<td>15</td>
<td>55</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Code inspection</td>
<td>20</td>
<td>40</td>
<td>65</td>
<td>25</td>
</tr>
<tr>
<td>Unit testing</td>
<td>1</td>
<td>5</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.6. Effectiveness of fault discovery techniques. (Jones 1991)

<table>
<thead>
<tr>
<th>Reviews</th>
<th>Requirements</th>
<th>Design faults</th>
<th>Code faults</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviews</td>
<td>Fair</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Good</td>
</tr>
<tr>
<td>Prototypes</td>
<td>Good</td>
<td>Fair</td>
<td>Fair</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Testing</td>
<td>Poor</td>
<td>Poor</td>
<td>Good</td>
<td>Fair</td>
</tr>
<tr>
<td>Correctness proofs</td>
<td>Poor</td>
<td>Poor</td>
<td>Fair</td>
<td>Fair</td>
</tr>
</tbody>
</table>

Integration testing

- Bottom-up
- Top-down
- Big-bang
- Sandwich testing
- Modified top-down
- Modified sandwich
Example program structure.

```
A
  B
  C
  D
    E
    F
    G
```

Bottom-up Testing

```
Test E
Test B, E, F
Test A, B, C, D, E, F, G
Test C
Test D, G
Test G
```
Top-Down Testing

Modified Top-Down
The Big Bang

Sandwich Testing
Modified Sandwich

![Diagram of Modified Sandwich]

Table 8.7. Comparison of integration strategies.

<table>
<thead>
<tr>
<th></th>
<th>Bottom-up</th>
<th>Top-down</th>
<th>Modified</th>
<th>Big-bang</th>
<th>Sandwich</th>
<th>Modified sandwich</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>Early</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
</tr>
<tr>
<td>Time to basic working program</td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
</tr>
<tr>
<td>Component drivers needed</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Stubs needed</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Work parallelism at beginning</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Ability to test particular paths</td>
<td>Easy</td>
<td>Hard</td>
<td>Easy</td>
<td>Easy</td>
<td>Medium</td>
<td>Easy</td>
</tr>
<tr>
<td>Ability to plan and control sequence</td>
<td>Easy</td>
<td>Hard</td>
<td>Hard</td>
<td>Easy</td>
<td>Hard</td>
<td>Hard</td>
</tr>
</tbody>
</table>
Testing Object-Oriented Systems

• Examine objects and classes:
  – Asymmetric associations
  – Disparate attributes and operations
  – One class playing two or more roles
  – An operation has no good target class
  – Two associations with the same name
• Traditional testing apply well to functions, but do not consider object states.
• Some things are harder and others easier.
Test planning

- Establish test objectives
- Design test cases
- Write test cases
- Test test cases
- Execute tests
- Evaluate test results
### Automated testing tools

- **Code analysis**
  - Static analysis
    - code analyzer
    - structure checker
    - data analyzer
    - sequence checker
  - Dynamic analysis
    - program monitor

- **Test execution**
  - Capture and replay
  - Stubs and drivers
  - Automated testing environments
  - Test case generators

### Static Analysis

- **Code analyzer**: The components are evaluated automatically for proper syntax and coding standards.
- **Structure checker**: This tool generates a graph from the components submitted as inputs.
- **Data analyzer**: The tool reviews the data structures, data declarations, and component interfaces, and then notes improper linkage among components, conflicting data definitions, and illegal data usage.
- **Sequence checker**: The tool checks sequences of events: if coded in the wrong sequence, the events are highlighted.
Dynamic Analysis

• Many systems are difficult to test because there are several parallel operations being performed concurrently.
• A program monitor is a tool that captures the state of events during the execution of a program.
• A monitor can list the number of times a component is called or a line of code is executed.
• Can verify 100% statement coverage.

Stubs and Drivers

Stubs and drivers are important part of integration testing. Tools are available to:
1. Set all appropriate state variables to prepare for a given test case.
2. Simulate keyboard input and other data-related responses
3. Compare actual output with expected
4. Track which paths have been traversed
5. Reset variables to prepare for the next test case
6. Interact with a debugging package.
When to Stop Testing

- When all of the faults have been found and removed.
- One might assume that the most difficult to find faults are the most difficult to correct.
  - This is not the case!
  - The most trivial faults are sometimes the most difficult to find, and have the most tragic consequences.
- Myers reports that as the number of detected faults increases, the probability of the existence of more undetected faults increases.

![Graph showing the relationship between probability of existence of additional faults and number of faults found to date.](image)

Probability of finding faults during development, based upon results by Myers.
Fault Seeding

A number of known faults is introduced into a program by one team.

The other team tries to find as many faults as possible.

We then assume:

\[
\frac{\text{detected seeded faults}}{\text{total seeded faults}} = \frac{\text{detected nonseeded faults}}{\text{total nonseeded faults}}
\]

This assumes that the seeded faults are similar to the real faults.