Software Testing

- Fault--a defect in the system specification, design, or implementation
- Failure--Software does not conform to requirements.
- Testing--process of identifying faults.

Possible Sources of Failures

- Incorrect or incomplete specification
- Incorrect design
- Incorrect coding
- Incorrect interfaces with other systems.
- Incorrect behavior of externally supplied components.
A Basic Tenet of Testing:

- You cannot guarantee the absence of faults in a system via testing.
- Thus the objective of testing should not be viewed as a demonstration of correctness of the system.
- Rather, the objective of testing should be viewed as a process of exposing defects.

Types of Faults--A Very Partial List
- algorithmic faults
- Computation and Precision faults
- documentation faults
- stress of overload faults
- capacity or boundary faults
- timing or coordination faults
- throughput or performance faults
- recovery faults
- hardware and system software faults
- standards and procedures faults
Hewlett-Packard Fault Classification

Origin: Where?
- Specification/requirements
- Design
- Code
- Environment/support
- Documentation
- Other

Type: What?
- Requirements or specifications
- HW interface
- SW interface
- Functional description
- Interprocess comm.
- Data def.
- Logic desc.
- Error checking
- Standards

Mode: Why?
- Logic
- Computation
- Data handling
- Module interface/implementation
- Standards
- Test HW
- Test SW
- Integration SW
- Development tools

Missing Unclear Wrong Changed Better Way

HP Fault Type Breakdown:

- Logic 32%
- Computation 18%
- Documentation 19%
- Data handling 6%
- Requirements 5%
- Hardware 4%
- Process/interprocess communication 5%
- Other code 11%
Stages of Testing

- Component code
  - Unit test
    - Integration test
      - Design Specs.
      - System functional requirements
      - Other software requirements
      - Customer requirements specs.
      - User environment
    - Function test
      - Integrated modules
      - Functioning system
      - Verified, validated software
      - Accepted system
    - Performance test
      - In-use system
    - Acceptance test
    - Installation test

Testing Philosophies

- Black-box testing--test with no knowledge of internal structure of the component/system.
- Gray-box testing--knowledge of design and algorithms, but not of coding.
- White-box testing(clear box testing)--testing based upon full knowledge of internal structure and coding.
  - e.g. attempt to exercise all control paths
Unit Testing Principles

- Code Examination (code review)
  - code walk-through
  - code inspection
- Success of code review:
  - Fagan (1976)--Controlled experiment
    - Formal inspection found 67% of faults prior to unit test
    - Informal walkthrough considerably less effective.
  - Jones (1977)-empirical study (10 M LOC)
    - Inspections found 85% of faults

Unit Testing Principles--Continued

- Formal Proof of Correctness--use mathematical logic to prove that the code conforms to certain logical assertions.
- Limitations
  - expensive, difficult, and error-prone process
  - can typically prove only simple functional properties.
Unit Test Principles--Continued

- Choosing test cases:
  - Determine test objectives
  - Determine “equivalence classes” of inputs
    - Every possible input belongs to one of these classes
    - Classes are disjoint
    - Likely that all inputs from a class exhibit the same
      behavior for faults related to the test objective.
  - Choose test cases
- Typically test cases employ both black-box and white-box notions.

Choosing Unit Test Cases--An Example

- Consider tests for a component that calculates the roots of a quadratic equation:
  - \[ ax^2 + bx + c = 0 \]
  - Black-box approach:
    - all possible combinations of positive, zero, and negative values for a, b, and c
    - \( a > b > c \)
    - \( b > c > a \)
    - etc.
  - White box notion (actually Grey box):
    - logic depends on value of discriminator\((b^2 - 4ac)\)
    - test positive, zero, and negative values of discriminator.
Unit Test--Test Thoroughness

• Statement testing--every statement in component exercised at least once.
• Branch testing--each branch of every decision point is exercised
• Path testing--each distinct path through code is exercised
• Definition-use path testing--each path from declaration to use of each variable is tested
• etc.

Black-Box (Functional) Testing

• Testing strategy depends on nature of the system specification
  – syntax driven testing
    • applicable to systems whose behavior is defined by a certain grammar
    • Strategy: generate test cases to insure that each production rule is applied at least once
  – decision table testing
    • useful when requirements have been specified as “if-then” rules
Example of Decision Table-based Testing

- Consider a process control application with the following specified behavior:
  - if sensor s1 is active, then open valve v1
  - if sensor s2 is active, then open valve v2
  - if both sensors are active, then open v2 and send a warning message.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 0 0 1 1</td>
<td>open v1 0 0 1 0</td>
</tr>
<tr>
<td>s2 0 1 0 1</td>
<td>open v2 0 1 0 1</td>
</tr>
<tr>
<td></td>
<td>warning 0 0 0 1</td>
</tr>
</tbody>
</table>

Decision Table

Functional Testing--Cause-Effect Graphs

- Show relationships between specific input combinations (causes) and system outputs (effects).

Causes:
- C1: credit command
- C2: debit command
- C3: account # valid
- C4: transaction amt. valid

Effects
- E1: “invalid command”
- E2: “invalid account #”
- E3: “invalid amount”
- E4: debit account
- E5: credit account
Cause-Effect Graph Example--Continued

Equivalent Decision Table:

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>x</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Functional Testing--Testing Boundary Conditions

• Consider a requirement stated as:
  – if the value x of sensor S1 exceeds the value y of sensor S2 then perform action A1, else perform action A2

• Two equivalence classes of inputs:
  – \( E_1 = \{(x,y) \mid x > y\} \)
  – \( E_2 = \{(x,y) \mid x \leq y\} \)

• The criteria of randomly selecting a representative from each equivalence class will not test the boundary condition \( x = y \)
Boundary Conditions--Another Example

- Consider a specification that requires a system to be able to store any number of sensor reading between 1 and 16,383 \((2^{14} - 1)\)

- Equivalence classes:
  - E1: zero readings
  - E2: 1 <= # of readings <= 16,383
  - E3: # of readings > 16,383

- Boundary condition test cases:

White Box (Structural) Testing

- Statement coverage
  - require that each statement be executed at least once
  - limitations: Consider the following code:
    
    ```
    begin
    if (y >= 0) then y = 0 - y;
    abs = y
    end
    ```
  - A single test case, \(y = 0\), exercises all statements but doesn’t exhibit the error.
Structural Testing--Continued

- Branch Testing (Decision Coverage)
  - insure that each branch is executed at least once.
  - In previous example:

```
if (x < upper) && (y > lower) {
    z = compute (x,y);
}
else {
    z = compute_alternative(x,y);
}
```

```plaintext
Structural Testing--Continued

- Condition/Branch Coverage
  - Each branch executed at least once and every combination of conditions in compound conditions must be exercised
  - Example:
  - if (x < upper) && (y > lower) {
    z = compute (x,y);
  }
  else {
    z = compute_alternative(x,y);
  }
```
Condition/Branch Test Example--Continued

Note that there is a fourth case: $x >$ upper, $y >$ lower.

Limitations of Condition/Branch Testing

Test cases:
- $x=2, z=6$
- $x=0, z=12$

But consider the case: $x=0, z=10$. Results in a divide by zero condition.
Structural Testing--Continued

- Path Testing
  - exercise all logical paths in a program
  - Problem: a program with n branch points can have up to $2^n$ paths.
  - Loops further complicate matters.
  - Meyers (Software Engineering Journal, Jan. 1992) showed an example of a simple program with a loop that iterates $\leq 18$ times. He calculated the number of paths as:
    - $4.77 \times 10^{12}$

Structural Testing--continued

  - Statement testing can be effective for small modules (< 5000 LOC), but 100% statement coverage is needed.
  - In general, statement testing is only about half as effective as branch testing.
  - Branch testing is most effective for flow control testing at module level. A coverage of at least 85% is needed
  - Path testing is much more complex (typically 8 to 10 times) than branch testing. Use should be limited to mission or life critical functions.