Software Testing, Validation, and Verification

Professor Mark Gabel
Slides courtesy of Professor W. Eric Wong
Department of Computer Science
University of Texas at Dallas

http://www.utdallas.edu/~mark.gabel/teaching/se6367f12

Subject and Objectives

• Software Quality
  – what it means
  – how to discuss it
  – how to measure it
  – how to improve it
• Software Testing, Validation, and Verification
• Specific objectives
  – Errors, Testing, debugging, test process, CFG, correctness, reliability, oracles.
  – Testing techniques
Course Structure

• No required textbook
• Lecture notes online
• Website and syllabus online
• http://www.utdallas.edu/~mark.gabel
• Grading
  – Quizzes (25%) [throughout]
  – Exam (25%) [end of semester]
  – Project (50%) [semi-guided, coming soon]

Motivation

• Bad software costs
  – time
  – money
  – opportunity
  – lives (safety critical)
• This very university is plagued with software integration problems
• Quality is of the utmost importance to software employers
Testing Concepts

Errors, Faults, Failures
Why test?
The first step is fully admitting that the code you write is riddled with errors. That is a bitter pill to swallow for a lot of people, but without it, most suggestions for change will be viewed with irritation or outright hostility. You have to want criticism of your code.

- John Carmack

If debugging is the art of removing bugs, then programming must be the art of inserting them.

- Unknown

Errors

Errors are a part of our daily life.

*Humans make errors* in their thoughts, actions, and in the products that might result from their actions.

Errors occur wherever humans are involved in taking actions and making decisions.

*These fundamental facts of human existence make testing an essential activity.*
Errors: Examples

<table>
<thead>
<tr>
<th>Area</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hearing</td>
<td>Spoken: He has a garage for repairing foreign cars.</td>
</tr>
<tr>
<td></td>
<td>Heard: He has a garage for repairing failed cars.</td>
</tr>
<tr>
<td>Medicine</td>
<td>Incorrect antibiotic prescribed.</td>
</tr>
<tr>
<td>Music</td>
<td>Incorrect note played.</td>
</tr>
<tr>
<td>performance</td>
<td>Incorrect algorithm for metric inversion.</td>
</tr>
<tr>
<td>Observation</td>
<td>Operator fails to recognize that a relief valve is stuck open.</td>
</tr>
<tr>
<td>Software</td>
<td>Operator used: $\neq$, correct operator: $&gt;$.</td>
</tr>
<tr>
<td></td>
<td>Identifier used: newline, correct identifier: nextLine.</td>
</tr>
<tr>
<td></td>
<td>Expression used: $a \land (b \lor c)$, correct expression: $(a \land b) \lor c$.</td>
</tr>
<tr>
<td></td>
<td>Data conversion from 64-bit floating point to 16-bit integer not</td>
</tr>
<tr>
<td></td>
<td>protected (resulting in a software exception).</td>
</tr>
<tr>
<td>Speech</td>
<td>Spoken: sugar instead, intent: sugar substitute.</td>
</tr>
<tr>
<td></td>
<td>Spoken: We need a new refrigerator, intent: We need a new washing</td>
</tr>
<tr>
<td></td>
<td>machine.</td>
</tr>
<tr>
<td>Sports</td>
<td>Incorrect call by the referee in a tennis match.</td>
</tr>
<tr>
<td>Writing</td>
<td>Written: What kind of pens did you use?</td>
</tr>
<tr>
<td></td>
<td>Intent: What kind of pens did you use?</td>
</tr>
</tbody>
</table>

Error, Faults, Failures

Programmer are used by Specifications

Program writes possibility of Error in thought or action

Fault may lead to

determines

Observable behavior tends to

Desired behavior are these the same?

Yes, Program behaves as desired.

No, Program does not behave as desired, a failure has occurred.
Software Quality

Static quality attributes: structured, maintainable, testable code as well as the availability of correct and complete documentation.

Dynamic quality attributes: software reliability, correctness, completeness, consistency, usability, and performance
Software Quality – Our Main Focus

Reliability: To be discussed

Correctness: A program is considered correct if it behaves as desired on all possible test inputs.
- valid inputs
- invalid inputs

Questions

Can we prove the correctness of a program by testing?

Correctness versus Reliability

Software Quality – Others

Completeness refers to the availability of all features listed in the requirements, or in the user manual. An incomplete software is one that does not fully implement all features required.

Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention.
Software Quality – Others

**Usability** refers to *the ease with which an application can be used*. This is an area in itself and there exist techniques for usability testing. *Psychology* plays an important role in the design of techniques for usability testing.

**Performance** refers to the time the application takes to perform a requested task. It is considered as a *non-functional requirement*. It is specified in terms such as “This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory.”

Requirements, Input Domain, Behavior, Correctness, Reliability
A standard test: Run a program with some input, and Verify that its behavior (output) is correct.

How do we know what “correct” is?

Which?

Requirements leading to two different programs:

**Requirement 1:** It is required to write a program that inputs two integers and outputs the maximum of these.

**Requirement 2:** It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence.
Requirements: Incompleteness

Suppose that program max is developed to satisfy Requirement 1. The expected output of max when the input integers are 13 and 19 can be easily determined to be 19.

Suppose now that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return, or on two separate lines with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question.

Requirements: Ambiguity

Requirement 2 is ambiguous. It is not clear whether the input sequence is sorted in ascending or descending order. The behavior of sort program, written to satisfy this requirement, will depend on the decision taken by the programmer while writing sort.
Input Domain (Input Space)

The set of all possible inputs to a program $P$ is known as the input domain or input space, of $P$.

Using Requirement 1 above we find the input domain of $\text{max}$ to be the set of all pairs of integers where each element in the pair integers is in the range $-32,768$ till $32,767$.

Using Requirement 2 it is not possible to find the input domain for the sort program.

Input Domain (Continued)

Modified Requirement 2:

It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order. The order of the output sequence is determined by an input request character which should be “A” when an ascending sequence is desired, and “D” otherwise.

While providing input to the program, the request character is input first followed by the sequence of integers to be sorted; the sequence is terminated with a period.
Input Domain (Continued)

Based on the above modified requirement, the input domain for sort is a set of pairs. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period.

Valid/Invalid Inputs

The modified requirement for sort mentions that the request characters can be “A” and “D,” but fails to answer the question “What if the user types a different character?”

When using sort it is certainly possible for the user to type a character other than “A” and “D.” Any character other than “A” and “D” is considered as invalid input to sort. The requirement for sort does not specify what action it should take when an invalid input is encountered.
Correctness vs. Reliability

Though correctness of a program is desirable, proving it is almost never the objective of testing.

To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish.

Thus correctness is established via mathematical proofs of programs.

Correctness and Testing

Correctness attempts to establish that the program is error free.

Testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.
Software Reliability: Two Definitions

**Software reliability** [ANSI/IEEE Std 729-1983]: is the *probability of failure free operation* of software over a given time interval and under *given conditions*.

**Software reliability** is the probability of failure free operation of software in *its intended environment*.

Operational Profile

An *operational profile* is a numerical description of how a program is used.

Consider a sort program which, on any given execution, allows any one of two types of input sequences (numbers only or alphanumeric strings). Sample operational profiles for sort follow.
## Operational Profile

### Operational profile #1

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers only</td>
<td>0.9</td>
</tr>
<tr>
<td>Alphanumeric strings</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### Operational profile #2

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers only</td>
<td>0.1</td>
</tr>
<tr>
<td>Alphanumeric strings</td>
<td>0.9</td>
</tr>
</tbody>
</table>
When testing reveals a bug, the process used to determine the cause of this bug and to remove it, is known as debugging.
Test Plan

A test cycle is often guided by a test plan.

Example: The sort program is to be tested to meet the requirements given earlier. Specifically, the following needs to be done.

- Execute sort on at least two input sequences, one with “A” and the other with “D” as the request character.
Test Plan (contd.)

- Execute the program on an *empty input* sequence.
  - Special case
- Test the program for *robustness against erroneous inputs* such as “R” typed in as the request character.
- *All failures* of the test program *should be recorded* in a suitable file using the Company Failure Report Form.

Test Case/Data

A *test case* is a pair consisting of test data to be input to the program and the *expected output*. The *test data* is a set of values, one for each input variable. A *test set* is a collection of zero or more test cases.

Sample test case for *sort*:

Test data: <“A” 12, -29, 32 >
Expected output: -29, 12, 32
Program Behavior

Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.

A state diagram specifies program states and how the program changes its state on an input sequence.

Behavior: Observation and Analysis

In the first step one observes the behavior.

In the second step one analyzes the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.

The entity that performs the task of checking the correctness of the observed behavior is known as an oracle.
Oracles can also be programs designed to check the behavior of other programs.

For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix $A$ and generates $B$ as the output matrix.
Oracle: Construction

Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, requires the determination of input-output relationship.

In general, the construction of automated oracles is a complex undertaking.

Testing and Verification

Program verification aims at proving the correctness of programs by showing that it contains no errors. This is very different from testing that aims at uncovering errors in a program.

Program verification and testing are best considered as complementary techniques. In practice, one can shed program verification, but not testing.
Testing and Verification (contd.)

Testing is *not a perfect technique* in that a program might contain errors despite the success of a set of tests.

Verification might appear to be perfect technique as it promises to verify that a program is free from errors. However, *the person who verified a program might have made mistake in the verification process*; there might be an *incorrect assumption* on the input conditions; incorrect assumptions might be made regarding the components that interface with the program, and so on.

*Verified and published programs have been shown to be incorrect.*

Program Representation: Control Flow Graphs
Program Representation: Basic Blocks

A basic block in program P is a sequence of consecutive statements with a single entry and a single exit point. Thus a block has unique entry and exit points.

Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.

---

Basic Blocks: Example

Example: Computing x raised to y

```plaintext
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
```
Basic Blocks: Example (contd.)

Basic blocks

<table>
<thead>
<tr>
<th>Block</th>
<th>Lines</th>
<th>Entry point</th>
<th>Exit point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4, 5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>11, 12</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Example: Computing x raised to y

```plaintext
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;
end
output(z);
end
```
Control Flow Graph (CFG)

A control flow graph (or flow graph) $G$ is defined as:

- a finite set $N$ of nodes, and
- a finite set $E$ of edges.

An edge $(i, j)$ in $E$ connects two nodes $n_i$ and $n_j$ in $N$.

We often write $G = (N, E)$ to denote a flow graph $G$ with nodes given by $N$ and edges by $E$.

Control Flow Graph (CFG)

In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

Blocks and nodes are labeled such that block $b_i$ corresponds to node $n_i$. An edge $(i, j)$ connecting basic blocks $b_i$ and $b_j$ implies that control can go from block $b_i$ to block $b_j$.

We also assume that there is a node labeled Start in $N$ that has no incoming edge, and another node labeled End, also in $N$, that has no outgoing edge.
CFG Example

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}

CFG Example

begin
int x, y, power;
float z;
input (x, y);
if (y<0)
    power=y;
else
    power=y;
z=1;
end

while (power! =0){
    z=z*x;
    power=power-1;
}
if (y<0)
z=1/z;
output(z);

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}
Paths

Consider a flow graph $G = (N, E)$. A sequence of $k$ edges, $k > 0$, $(e_1, e_2, \ldots, e_k)$, denotes a path of length $k$ through the flow graph if the following sequence condition holds.

Given that $n_p$, $n_q$, $n_r$, and $n_s$ are nodes belonging to $N$, and $0 < i < k$, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$

Paths (cont’d)

Two complete and feasible paths:

$p_1 = (\text{Start}, 1, 2, 4, 5, 6, 5, 7, 9, \text{End})$

$p_2 = (\text{Start}, 1, 3, 4, 5, 6, 5, 7, 9, \text{End})$

Specified unambiguously using edges:

$p_1 = ((\text{Start}, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, \text{End}))$

Bold edges: complete path (Connects Start to End)

Dashed edges: subpath (Contiguous)
Paths: Feasible Paths

A path \( p \) through a flow graph for program \( P \) is considered feasible if there exists at least one test case which when input to \( P \) causes \( p \) to be traversed.

\[ p_1 = (\text{Start}, 1, 3, 4, 5, 6, 5, 7, 8, 9, \text{End}) \]
\[ p_2 = (\text{Start}, 1, 1, 2, 4, 5, 7, 9, \text{End}) \]

Paths: Infeasible Paths (defensive code)

```c
#include <stdio.h>

void isOdd(int l) {
    switch (l & 2) {
    case 0:
        printf("The input value is even\n");
        break;
    case 1:
        printf("The input value is odd\n");
        break;
    default:
        printf("No such value\n");
        break;
    }
}

void main() {
    int l = 0;
    scanf("%d", &l);
    isOdd(l);
}
```

The compiler can help with some of these cases, but not all.
Paths: Infeasible Paths (program bug)

Contradictory Conditions

```java
if (k < 2) {
    if (k > 3) [should be: k > -3]
        x = x * k;
}
```

The problem of determining the feasibility of a path is **undecidable**.

An algorithm (or tool) may handle some cases, but not all.

Human ingenuity (Testing, Inspection) is crucial!

---

Number of Paths

There can be **many distinct paths** through a program.

A program with no conditions (branches, loops) contains **exactly one path** that begins at node Start and terminates at node End.

Each additional condition in the program can increases the number of distinct paths by **at least one**.

Depending on their location, conditions can have a multiplicative effect on the number of paths.

- two nested if-then-else
- while loop
- for loop
Test Generation
(Systematic Test Creation)

Test generation

Any form of test generation uses a source document. In the most informal of test methods, *the source document resides in the mind of the tester* who generates tests based on a knowledge of the requirements.

In most commercial environments, the process is a bit more formal. The tests are generated *using a mix of formal and informal methods* from the *requirements document* serving as the source. *In more advanced test processes, requirements serve as a source for the development of test plans.*
Test Generation Strategies

Model based: require that a subset of the requirements be **modeled using a formal notation (usually graphical)**. Models: Finite State Machines, Timed automata, Petri net, SDL, UML, etc.

Specification based: require that a subset of the requirements be **modeled using a formal mathematical notation**. Examples: B, Z, and Larch.

Code based: generate tests **directly from the code**.
Types of Software Testing

Types of testing

One possible classification is based on the following four classifiers:

C1: Source of test generation.

C2: Lifecycle phase in which testing takes place

C3: Goal of a specific testing activity

C4: Characteristics of the artifact under test
C1: Source of test generation

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements (informal)</td>
<td>Black-box</td>
<td>Ad-hoc testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boundary value analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Category partition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classification trees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cause-effect graphs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Equivalence partitioning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Partition testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predicate testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random testing</td>
</tr>
<tr>
<td>Code</td>
<td>White-box</td>
<td>Adequacy assessment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coverage testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data-flow testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Domain testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mutation testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Path testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Structural testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test minimization using coverage</td>
</tr>
<tr>
<td>Requirements and code</td>
<td>Black-box and White-box</td>
<td></td>
</tr>
<tr>
<td>Formal model: Graphical or mathematical specification</td>
<td>Model-based Specification</td>
<td>Statechart testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FSM testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pairwise testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Syntax testing</td>
</tr>
<tr>
<td>Component interface</td>
<td>Interface testing</td>
<td>Interface mutation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pairwise testing</td>
</tr>
</tbody>
</table>

C2: Lifecycle Phase in Which Testing Takes Place

<table>
<thead>
<tr>
<th>Phase</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding</td>
<td>Unit testing</td>
</tr>
<tr>
<td>Integration</td>
<td>Integration testing</td>
</tr>
<tr>
<td>System integration</td>
<td>System testing</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Regression testing</td>
</tr>
<tr>
<td>Post system, pre-release</td>
<td>Beta-testing</td>
</tr>
</tbody>
</table>
**C3: Goal of Specific Testing Activity**

<table>
<thead>
<tr>
<th>Goal</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advertised features</td>
<td>Functional testing</td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>Security testing</td>
<td></td>
</tr>
<tr>
<td>Invalid inputs</td>
<td>Robustness testing</td>
<td></td>
</tr>
<tr>
<td>Vulnerabilities</td>
<td>Vulnerability testing</td>
<td></td>
</tr>
<tr>
<td>Errors in GUI</td>
<td>GUI testing</td>
<td>Capture/plaback</td>
</tr>
<tr>
<td></td>
<td>System testing</td>
<td>Event sequence graphs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complete Interaction Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transactional-flow</td>
</tr>
<tr>
<td>Operational correctness</td>
<td>Operational testing</td>
<td></td>
</tr>
<tr>
<td>Reliability assessment</td>
<td>Reliability testing</td>
<td></td>
</tr>
<tr>
<td>Resistance to penetration</td>
<td>Penetration testing</td>
<td></td>
</tr>
<tr>
<td>System performance</td>
<td>Performance testing</td>
<td>Stress testing</td>
</tr>
<tr>
<td>Customer acceptability</td>
<td>Acceptance testing</td>
<td></td>
</tr>
<tr>
<td>Business compatibility</td>
<td>Compatibility testing</td>
<td>Interface testing</td>
</tr>
<tr>
<td>Peripherals compatibility</td>
<td>Configuration testing</td>
<td>Installation testing</td>
</tr>
</tbody>
</table>

**Functional Testing**

- Testing either an *element* of or the *complete* product to determine whether it will function as planned.

- The *system testing* of an *integrated, black-box system* against its operational (i.e., functional) requirements.

- Testing the *advertised features* of a system for correct operation.

- Geared towards verifying that a product/application *conforms to all functional specifications*.

- Entail the following tasks: *test generation* from requirements or some model of the requirements, *test execution*, and *test assessment*. 
**C4: Artifact Under Test**

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application component</td>
<td>Component testing</td>
</tr>
<tr>
<td>Client and server</td>
<td>Client-server testing</td>
</tr>
<tr>
<td>Compiler</td>
<td>Compiler testing</td>
</tr>
<tr>
<td>Design</td>
<td>Design testing</td>
</tr>
<tr>
<td>Code</td>
<td>Code testing</td>
</tr>
<tr>
<td>Database system</td>
<td>Transaction-flow testing</td>
</tr>
<tr>
<td>OO software</td>
<td>OO testing</td>
</tr>
<tr>
<td>Operating system</td>
<td>Operating system testing</td>
</tr>
<tr>
<td>Real-time software</td>
<td>Real-time testing</td>
</tr>
<tr>
<td>Requirements</td>
<td>Requirement testing</td>
</tr>
<tr>
<td>Software</td>
<td>Software testing</td>
</tr>
<tr>
<td>Web service</td>
<td>Web service testing</td>
</tr>
</tbody>
</table>

**Test methodology**

- **Program Specifications**
- **Clues**
- **Expected behavior**
- **Oracle**
- **Actual behavior**
- **Program output is correct**
  - or
  - Program has failed; make a note and proceed with testing or get into the debug mode.

Until specs. exhausted.
Test Requirements Checklist

• Obtaining clues and deriving test requirements can become a tedious task.

• To keep it from overwhelming us it is a good idea to make a checklist of clues.

• This checklist is then transformed into a checklist of test requirements by going through each clue and deriving test requirements from it.

Test Specifications

• A test requirement indicates “how” to test a program. But it does not provide exact values of inputs.
  – “Send an email with a valid address”, vs
  – “Subj: test, Body: test, Address: test@example.com”

• A test requirement is used to derive a test specification, which is the exact specification of values of input and environment variables.
  – In essence, a Test Case
Test Specifications (cont’d)

• There may *not be a one-to-one correspondence* between test requirements and test specifications.

• A test requirement checklist might contain 50 entries. These might result in only 22 test specifications.
  – Example: sending an email can test address book, header generation, SMTP format, and network code simultaneously

• The fewer the tests the better---but only if these tests are of *good quality*!
  – We will discuss *test quality* when discussing *test assessment*.

Test Requirements to Specifications

• The test requirements *checklist* guides the process of deriving test specifications.
• Initially all entries in the checklist are unmarked or set to 0.
• Each time a test is generated from a requirement it is marked or the count incremented by 1.
• Thus, at any point in time, one *could assess the progress* made towards the generation of test specifications.
• One could also determine how many tests have been generated using any test requirement.
• Once a test requirement has been marked or its count is more than 0 we say that it has been satisfied.
Test Requirements to Specifications (cont’d)

Some rules of thumb to use:
- Try to satisfy multiple requirements using only one test.
- Try to satisfy each requirement by more than one test.
- Satisfy all test requirements.
- Avoid reuse of same values of a variable in different tests. Generating new tests by varying an existing one is likely to lead to tests that test the same part of the code as the previous one.
  
  "Poor test coverage"

In testing, variety helps!

Test Requirements to Specifications (cont’d)

• Though we try to combine several test requirements to generate one test case, this process is very subtle when considering error conditions.
• Example
Test Requirements to Specifications (cont’d)

• For example, consider the following:
  – speed_dial must be in the interval [0, 120]
    • if speed_dial < 0, error
    • if speed_dial > 120, error
  – zone, an interval [5, 10]
    • if zone < 5, error
    • if zone > 10, error

• One test specification obtained by combining the two requirements above is:
  – speed_dial = -1 (outside interval [0,120])
  – zone = 4 (outside interval [5, 10])

• Now, if the code to handle these error conditions is:

        if (speed_dial < 0 || speed_dial > 120)
            error_exit (“Incorrect speed_dial”)
        if (zone < 4 || zone > 10)
            error_exit (“Incorrect zone”);

• For our test, the program will exit before it reaches the second if statement. Thus, it will miss detecting the error in coding the test for zone.
We have dealt with some of the most basic concepts in software testing.