Software Testing

Lecture Notes 1 (of 4)

No issue is meaningful unless it can be put to the test of decisive verification.

C.S. Lewis, 1934

Contents

Outline of the Lecture

- Triangle program
- Introduction
- Basic definitions
- Black-box testing

Triangle program (simple version)

- The triangle problem is the most widely used example in software testing literature.
- The program accepts three integers, *a*, *b*, and *c* as input. The three values are interpreted as representing the lengths of sides of a triangle. The program prints a message that states whether the triangle is scalene (oregelbunden), isosceles (likbent) or equilateral (liksidig).
- On a sheet of paper, write a set of test cases (i.e., specific sets of data) that you feel would adequately test this program.

Evaluation of set of test cases (one point for each "yes" answer)

1. Do you have a test case that represents a valid scalene triangle? (note that test cases such as 1,2,3 and 2,5,10 do not warrant a "yes" answer, because there does not exist a triangle having such sides.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (2,2,4 would not be counted)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (e.g., 3,3,4; 3,4,3; and 4,3,3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1,2,3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g., 1,2,3; 1,3,2; and 3,1,2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third? (e.g., 1,2,4 or 12,15,30)
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1,2,4; 1,4,2; and 4,1,2)?
11. Do you have a test case in which all sides are 0 (i.e., 0,0,0)?
12. Do you have at least one test case specifying noninteger values?
13. Do you have at least one test case specifying the wrong number of values (e.g., two rather than three, integers)?
14. For each test case, did you specify the expected output from the program in addition to the input values.

Triangle program (improved version)

The program accepts three integers, *a*, *b*, and *c* as input. The three values are interpreted as representing the lengths of sides of a triangle. The integers *a*, *b*, and *c* must satisfy the following triangle property (the sum of any pair of sides must be greater than the third side).

$a < b + c$

Testing a ballpoint pen

- Does the pen write in the right color, with the right line thickness?
- Is the logo on the pen according to company standards?
- Is it safe to chew on the pen?
- Does the click-mechanism still work after 100 000 clicks?
- Does it still write after a car has run over it?

What is expected from this pen?
Intended use!!

Goal: develop software to meet its intended use!
But: human beings make mistake!


bridge


automobile

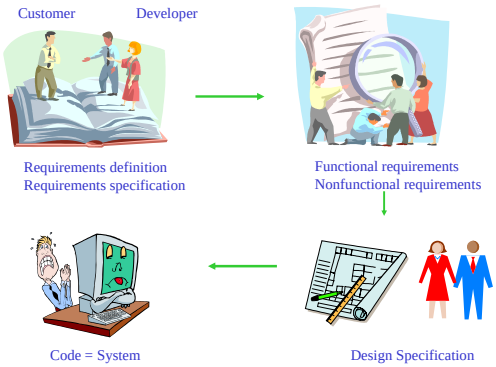

television


word processor

⇒ Product of any engineering activity must be verified against its requirements throughout its development.

- Verifying bridge = verifying design, construction, process,…
- Software must be verified in much the same spirit. In this lecture, however, we shall learn that verifying software is perhaps more difficult than verifying other engineering products.

We shall try to clarify why this is so.

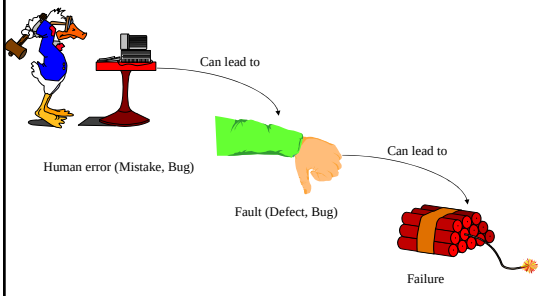


Basic Definitions

The terminology here is taken from standards developed by the institute of Electronics and Electrical Engineers (IEEE) computer Society.

- **Error**: people make errors. A good synonym is mistake. When people make mistakes while coding, we call these mistakes bugs. Errors tend tend to propagate; a requirements error may be magnified during design and amplified still more during coding.
- **Fault**: a fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. Defect is a good synonym for fault, as is bug. Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. We might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.
- **Failure**: a failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission?

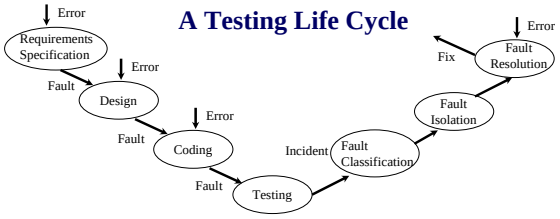
Error, Fault, Failure



Basic Definitions (cont.)

- **Incident**: when a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.
- **Test**: testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution.
- **Test Case**: test case has an identity and is associated with a program behavior. A test case also has a set of inputs and a list of expected outputs.

A Testing Life Cycle

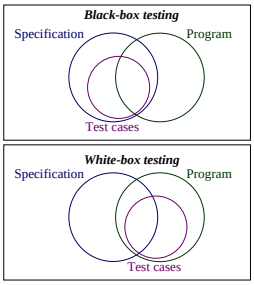


Typical Test case Information

Test Case ID:
Purpose:
Preconditions:
Inputs:
Expected Outputs:
Postconditions:
Execution History:
Date Result Version Run By

Types (strategy) of testing

- Black-box: a strategy in which testing is based on requirements and specifications.
- White-box: a strategy in which testing is based on internal paths, structure, and implementation.
- Gray-box: peek into the "box" just long enough to understand how it has been implemented.



Testing level

- Unit testing
- Integration testing
- System testing
- Acceptance testing

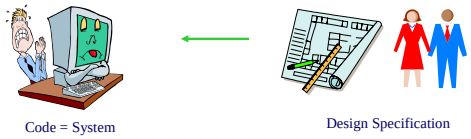
Types of Faults

(dep. on org. IBM, HP)

- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: speed in req.
- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other

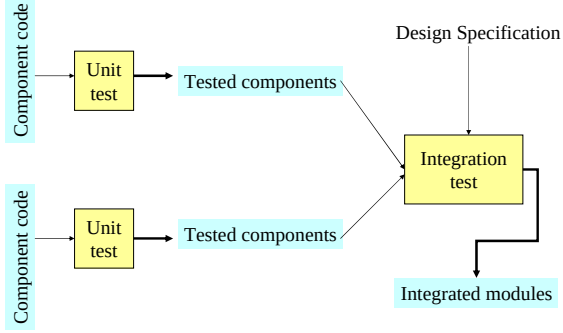
Unit & Integration Testing

Objective: to ensure that code implemented the design properly.



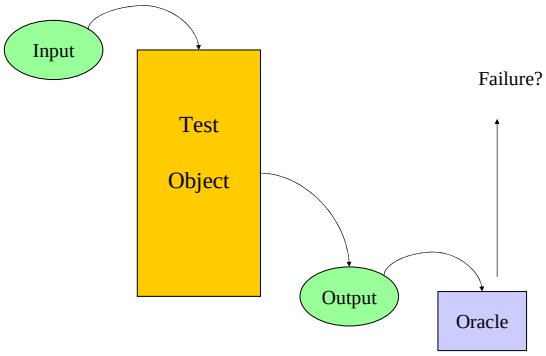
Code = System

Design Specification



Unit Testing

- Black-box Testing
- White-box Testing



Two Types of Oracles

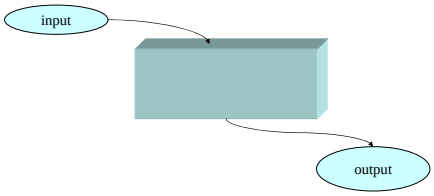
- **Human**: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.
- **Automated**: a system capable of performing the above task.

Proving code correct

- Formal proof techniques
- Symbolic execution
- Automated theorem proving

Black-box / Closed-box Testing

- incorrect or missing functions
- interface errors
- performance error



Block-Box Testing Techniques

- Definition: a strategy in which testing is based on requirements and specifications.
- Applicability: all levels of system development
  - Unit
  - Integration
  - System
  - Acceptance
- Disadvantages: never be sure of how much of the system under test (SUT) has been tested.
- Advantages: directs tester to choose subsets to tests that are both efficient and effective in finding defects.

Black-box Testing

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
4. Decision table testing
5. Use case testing

Exhaustive testing

- **Definition**: testing with every member of the input value space.
- Input value space: the set of all possible input values to the program.

Equivalence Class Testing

- Equivalence Class (EC) testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.
- Each EC consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is ***equivalent***, in terms of testing, to any other value.

Identifying the Equivalence Classes

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - range of values x: 1-50
- Valid equivalence class
  - 1< x < 50
- Invalid equivalence classes
  - x < 1
    x > 50

Guidelines

1. If an input condition specifies a *range* of values; identify one valid EC and two invalid EC.
2. If an input condition specifies the *number* (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).
3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.
4. If an input condition specifies a "must be" situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter)
5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.

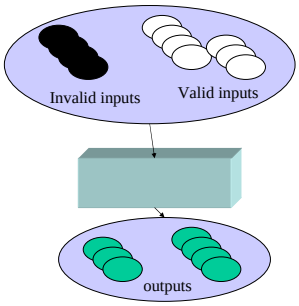
Identifying the Test Cases

1. Assign a unique number to each EC.
2. Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered valid ECs as possible.
3. Until all invalid ECs have been covered by test cases, write a test case that cover one, and only one, of the uncovered invalid ECs.

Applicability and Limitations

- Most suited to systems in which much of the input data takes on values within ranges or within sets.
- It makes the assumption that data in the same EC is, in fact, processed in the same way by the system. The simplest way to validate this assumption is to ask the programmer about their implementation.
- EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system's requirements.

Equivalence partitioning



Specification: the program accepts four to eight inputs which are 5 digit integers greater than 10000.

Specification: the program accepts four to eight inputs which are 5 digit integers greater than 10000.

Input values

Less than 10000	Between 10000 and 99999	More than 99999

Number of input values

Less than 4	Between 4 and 8	More than 8

Boundary Value Testing

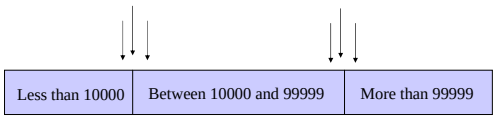
Boundary value testing focuses on the boundaries simply because that is where so many defects hide. The defects can be in the requirements or in the code.

The most efficient way of finding such defects, either in the requirements or the code, is through inspection (Software Inspection, Gilb and Graham's book).

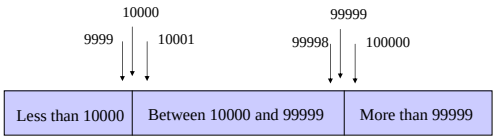
Technique

1. Identify the ECs.
2. Identify the boundaries of each EC.
3. Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.

Boundary value analysis



Boundary value analysis



Applicability and Limitations

Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system's requirements.

Decision Table Testing

Decision tables are an excellent tool to capture certain kinds of system requirements and to document internal system design. They are used to record complex business rules that a system must implement. In addition, they can serve as a guide to creating test cases.

Technique

The general format of a decision table:

	Rule 1	**Rule 2**	**…**	**Rule P**
Conditions				
Condition-1				
Condition-2				
…				
Condition-m				
Actions				
Action-1				
Action-2				
…				
Action-n				

Technique (cont.)

A decision table converted to a test case table:

	Test Case 1	Test Case 2	…	Test Case P
Inputs				
Condition-1				
Condition-2				
…				
Condition-m				
Expected Results				
Action-1				
Action-2				
…				
Action-n				

A decision table with "don't care" entry

	Rule 1	Rule 2	Rules 3,4	Rule 5	Rule 6	Rules 7,8
C1	T	T	T	F	F	F
C2	T	T	F	T	T	F
C3	T	F	_	T	F	_
A1	X	X		X		
A2	X				X	
A3		X		X		
A4			X			X

- _ : "don't care" entry. The don't care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes the "n/a" symbol for this latter interpretation.
- Limited entry decision tables: all the conditions are binary.
- Extended entry decision tables: conditions are allowed to have several values.
- Decision tables are deliberately declarative (as opposed to imperative); no particular order is implied by the conditions, and selected actions do not occur in any particular order.

Triangle program (new conditions)

The program accepts three integers, *a*, *b*, and *c* as input. The three values are interpreted as representing the lengths of sides of a triangle. The integers *a*, *b*, and *c* must satisfy the following conditions:

- C1: $1 <= a <= 200$
- C2: $1 <= b <= 200$
- C3: $1 <= c <= 200$
- C4: $a < b + c$
- C5: $b < a + c$
- C6: $c < a + b$

Technique

Decision table for the Triangle problem

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9
C1: a, b, c form a triangle?	N	Y	Y	Y	Y	Y	Y	Y	Y
C2: a = b?	_	Y	Y	Y	Y	N	N	N	N
C3: a = c?	_	Y	Y	N	N	Y	Y	N	N
C4: b = c?	_	Y	N	Y	N	Y	N	Y	N
A1: Not a triangle	X								
A2: Scalene									X
A3: Isosceles					X		X	X	
A4: Equilateral		X							
A5: **Impossible**			X	X		X			

- If the integers a, b, and c do not constitute a triangle, we do not even care about possible equalities (rule 1).
- If two pairs of integers are equal, by transitivity, the third pair must be equal; thus the negative entry (N) makes these rules impossible (rule3, 4, and 6).

Refined Decision table for the Triangle problem

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9	Rule 10	Rule 11
C1: a < b + c ?	F	T	T	T	T	T	T	T	T	T	T
C2: b < a + c ?	_	F	T	T	T	T	T	T	T	T	T
C3: c < a + b ?	_	_	F	T	T	T	T	T	T	T	T
C4: a = b?	_	_	_	T	T	T	T	F	F	F	F
C5: a = c?	_	_	_	T	T	F	F	T	T	F	F
C6: b = c?	_	_	_	T	F	T	F	T	F	T	F
A1: Not a triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: **Impossible**					X	X		X			

Choice of conditions: Here we expand the old condition (C1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.

Decision table for the Refined Decision table with Rule Counts

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9	Rule 10	Rule 11
C1: a < b + c ?	F	T	T	T	T	T	T	T	T	T	T
C2: b < a + c ?	_	F	T	T	T	T	T	T	T	T	T
C3: c < a + b ?	_	_	F	T	T	T	T	T	T	T	T
C4: a = b?	_	_	_	T	T	T	F	F	F	F	F
C5: a = c?	_	_	_	T	T	F	F	T	T	F	F
C6: b = c?	_	_	_	T	F	T	F	T	F	T	F
Rule Count	**32**	**16**	**8**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
A1: Not a triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: **Impossible**					X	X		X			

- If *n* conditions exist, there must be *2 power n* rules (e.g., above 6 conditions; 64 rules).
- Rules in which no don't care entries occur count as one rule. Each don't care entry in a rule doubles the count of that rule.

Test Cases for the Triangle Problem

	1	2	3	4	5	6	7	8	9	10	11
C1: a < b + c ?	F	T	T	T	T	T	T	T	T	T	T
C2: b < a + c ?	_	F	T	T	T	T	T	T	T	T	T
C3: c < a + b ?	_	_	F	T	T	T	T	T	T	T	T
C4: a = b?	_	_	_	T	T	T	F	T	T	F	F
C5: a = c?	_	_	_	T	T	F	T	T	F	T	F
C6: b = c?	_	_	_	T	F	T	F	T	F	T	F
A1: Not a triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: **Impossible**					X	X		X			

Case ID	a	b	c	Expected output
DT1				Not a Triangle
DT2				Not a Triangle
DT3				Not a Triangle
DT4				Equilateral
DT5				Impossible
DT6				Impossible
DT7				Isosceles
DT8				impossible
DT9				Isosceles
DT 10				Isosceles
DT 11				Scalene

Test Cases for the Triangle Problem

	1	2	3	4	5	6	7	8	9	10	11
C1: a < b + c ?	F	T	T	T	T	T	T	T	T	T	T
C2: b < a + c ?	_	F	T	T	T	T	T	T	T	T	T
C3: c < a + b ?	_	_	F	T	T	T	T	T	T	T	T
C4: a = b?	_	_	_	T	T	T	F	F	F	F	F
C5: a = c?	_	_	_	T	F	F	T	T	F	F	F
C6: b = c?	_	_	_	T	F	T	F	T	F	T	F
A1: Not a triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: **Impossible**					X	X		X			

Case ID	a	b	c	Expected output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	impossible
DT9	2	3	2	Isosceles
DT 10	3	2	2	Isosceles
DT 11	3	4	5	Scalene

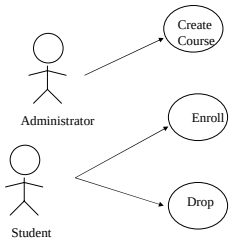
Applicability and Limitations

Decision table testing can be used whenever the system must implement complex business rules when these rules can be represented as a combination of conditions and when these conditions have discrete actions associated with them.

Use Case Testing

- Test cases that exercise a system's functionalities from start to finish.
- Use cases were created by Ivar Jacobsen in his book Object-Oriented Software Engineering: A use case driven approach.

Unified Modeling Language Notation



- Actor
- Scenario

Applicability and Limitations

- System and acceptance testing.