Chapter 10
Software Test Metrics

Contents

- Test concepts, definitions and techniques
- Estimating number of test case
- Allocating test times
- Test case management
- Decisions based on testing
- Test coverage measurement
- Software testability measurement
- Remaining defects measurement

Definitions

- Run: The smallest division of work that can be initiated by external intervention on a software component. Run is associated with input state (set of input variables) and runs with identical input state are of the same run type.
- Direct input variable: is a variables that controls the operation directly
  - Example: arguments, selection menu, entered data field.
- Indirect input variable: is a variable that only influences the operations or its effects are propagated to the operation
  - Example: traffic load, environmental variable.

What is a Test Case?

- Definition: A *test case* is a partial specification of a run through the naming of its direct input variables and their values.
- Better Definition: A *test case* is an instance (or scenario) of a use-case composed of a set of test inputs, execution conditions and expected results.
- Prior to specifying test cases one should document the use-cases.

Types of Software Test

- Certification Test: Accept or reject (binary decision) an acquired component for a given target failure intensity.
- Feature Test: A single execution of an operation with interaction between operations minimized.
- Load Test: Testing with field use data and accounting for interactions among operations
- Regression Test: Feature tests after every build involving significant change, i.e., check whether a bug fix worked.

Basic Testing Methods

- Two widely recognized testing methods:
- White Box Testing
  - Reveal problems with the internal structure of a program
- Black Box Testing
  - Assess how well a program meets its requirements

White Box Testing

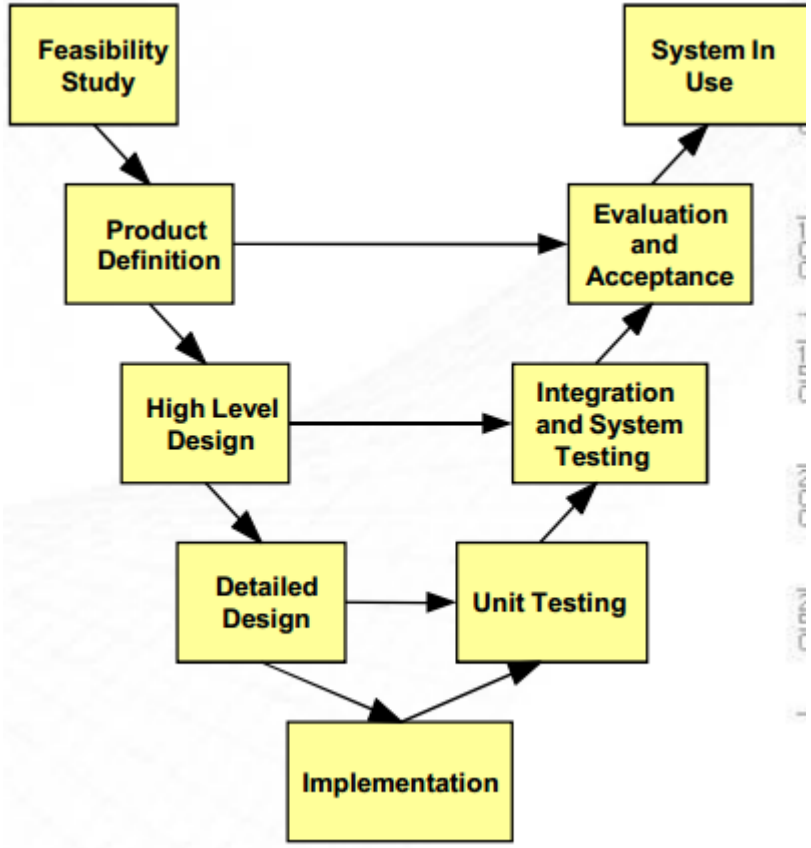
- Checks internal structure of a program
- Requires detailed knowledge of structure
- Common goal is Path Coverage
  - How many of the possible execution paths are actually tested?
  - Effectiveness often measured by the fraction of code exercised by test cases

Black Box Testing

- Checks how well a program meets its requirements
  - Assumes that requirements are already validated
  - Look for missing or incorrect functionality
  - Exercise system with input for which the expected output is known
  - Various methods:
    - Performance, Stress, Reliability, Security testing

Testing Levels

- Testing occurs throughout software lifecycle:
  - Unit
  - Integration & System
  - Evaluation & Acceptance
  - Installation
  - Regression (Reliability Growth)



1. Unit Testing

- White box testing in a controlled test environment of a module in isolation from others
  - Unit is a function or small library
  - Small enough to test thoroughly
  - Exercises one unit in isolation from others
  - Controlled test environment
  - White Box

2. Integration Testing

- Units are combined and module is exercised
- Focus is on the interfaces between units
- White Box with some Black Box
- Three main approaches:
  - Top-down
  - Bottom-up
  - Big Bang

Integration Testing: Top-Down

- The control program is tested first
- Modules are integrated one at a time
- Major emphasis is on interface testing
  - Interface errors are discovered early
  - Forms a basic early prototype
  - Test stubs are needed
  - Errors in low levels are found late

Integration Testing: Bottom-Up

- Modules integrated in clusters as desired
- Shows feasibility of modules early on
- Emphasis on functionality and performance
  - Usually, test stubs are not needed
  - Errors in critical modules are found early
  - Many modules must be integrated before a working program is available
  - Interface errors are discovered late

Integration Testing: ‘Big Bang’

- Units completed and tested independently
- Integrated all at once
  - Quick and cheap (no stubs, drivers)
  - Errors:
    - Discovered later
    - More are found
    - More expensive to repair
- Most commonly used approach

3. External Function Test

- Black Box test
- Verify the system correctly implements specified functions
- Sometimes known as an *Alpha* test
- In-house testers mimic the end use of the system

4. System Test

- More robust version of the external test
- Difference is the test platform
- Environment reflects end use
  - Includes hardware, database size, system complexity, external factors
- Can more accurately test nonfunctional system requirements (performance, security etc.)

5. Acceptance Testing

- Also known as *Beta* testing
- Completed system tested by end users
- More realistic test usage than ‘system’ phase
- Validates system with user expectations
- Determine if the system is ready for deployment

6. Installation Testing

- The testing of full, partial, or upgrade install/uninstall processes
- Not well documented in the literature

7. Regression Testing

- Tests modified software
- Verify changes are correct and do not adversely affect other system components
- Selects existing test cases that are deemed necessary to validate the modification
- With bug fixes, four things can happen
  - Fix a bug; add a new bug; damage program structure; damage program integrity
  - Three of them are unwanted

Estimate No. of Test Cases

- How many test cases do we need?
- Affected by two factors: time and cost
- Method:
- Compute the number of test cases for the given
  - Time:

    (available time × available staff) / (average time to prepare a test case)
  - Cost:

    (available budget) / (average preparation cost per test case)
- Select the minimum number of the two

Example

- The development budget for a project is $4 million and %10 of which can be spent on preparing test cases. Each test case costs $250 or 4 hours to prepare. The duration of project is set to be 25 weeks (of 40 hours each) and a staff of 5 is assigned to prepare the test cases. How many test cases should be prepared?

- From cost point of view:

  $N1 = (4{,}000{,}000 \times 0.1) / 250 = 1{,}600$

- From time point of view:

  $N2 = (25 \times 40 \times 5) / 4 = 1{,}250$

  $N = \min(N1, N2)$, therefore, $N = 1{,}250$

How to Specify Test Cases?

- For a given operation, quantify the value of direct input variables based on *levels* for which the same behavior will be expected.
- A test case for a given operation will be specified using a combination of levels of direct input variables.
- If the number of combinations exceed the number of test cases assigned to that operation only a portion of combinations must be selected to represent all.

Example: Test Case

- Suppose that we are going to build test cases for an ATM system. The operation that we would like to test is *"Withdraw from Checking account"*. Suppose that an input variable named "Withdraw_amount" with the following specification is given.

Variable name/ definition	*Specification*
Name: `Withdraw_amount` **Definition: The amount of cash that can be withdrawn in a single transaction**	1. 3 digit numbers are accepted (withdrawal only between 100$ and 1,000$ cash) 2. The number cannot start with 0 3. The rightmost digit must be 0 (10$, 20$, 50$ and 100$ bills only) 4. 4 digit numbers, only 1000 is acceptable 5. Any other number of digits is not acceptable

Example: Test Case (Cont'd)

- Specify the "minimum" set of test-cases to test the operation for this variable. Note that we should avoid redundant test cases.

Case	*Value of the variable* "`Withdraw_amount`"	*Criteria (pass, fail)*
1	i where $0 \le i \le 9$	fail
2	ij where $0 \le i \le 9$ and $0 \le j \le 9$	fail
3	ijk where $i = 0$ and $0 \le j \le 9$ and $0 \le k \le 9$	fail
4	ijk where $1 \le i \le 9$ $0 \le j \le 9$ and $k \ne 0$	fail
5	ijk where $1 \le i \le 9$ $0 \le j \le 9$ and $k = 0$	pass
6	$ijkl$ where $i = 1$ and $j = k = l = 0$	pass
7	$ijkl$ where $0 \le j, k, l \le 9$ and $i \ne 1$	fail

How to Create Test Cases?

1. Test case creation based on equivalent classes.

- Prepare only one test case for the representative class.

2. Test case creation based on boundary conditions.

- Programs that fail with non-boundary values fail at the boundaries, too.

3. Test case creation based on visible state transitions.

1. Equivalence Classes

A group of tests cases are “equivalent” if:

- They all test the same operation.
- If one test case can catch a bug, the others will probably do.
- If one test case does not catch a bug, the others probably will not do.
- They involve the same input variables.
- They all affect the same output variable.

Example: Equivalent Test Cases

- A test case that covers case 2 will also cover case 1. Therefore, one test case based on 2 is enough to check both. Note that case 1 does not cover both.

Case	*Value of the variable "`Withdraw_amount`"*	*Criteria (pass, fail)*
1	i where $0 \le i \le 9$	fail
2	ij where $0 \le i \le 9$ and $0 \le j \le 9$	fail
3	ijk where $i = 0$ and $0 \le j \le 9$ and $0 \le k \le 9$	fail
4	ijk where $1 \le i \le 9$ $0 \le j \le 9$ and $k \neq 0$	fail
5	ijk where $1 \le i \le 9$ $0 \le j \le 9$ and $k = 0$	pass
6	$ijkl$ where $i = 1$ and $j = k = l = 0$	pass
7	$ijkl$ where $0 \le j, k, l \le 9$ and $i \neq 1$	fail

2. Boundary Conditions

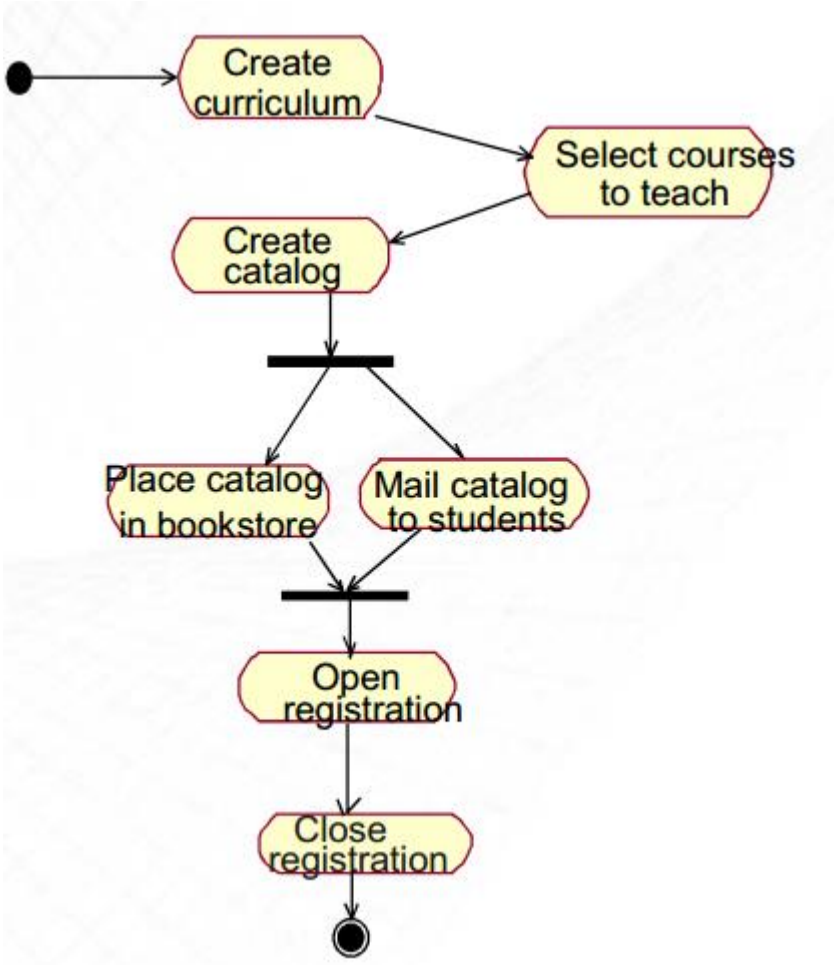
- Check if the boundary conditions for variables are set correctly.
- Check if the inequality boundary conditions can be changed to equality or not.
- Check if the counter input variables allow departure from the loop correctly or not.
- etc.

3. Visible State Transitions

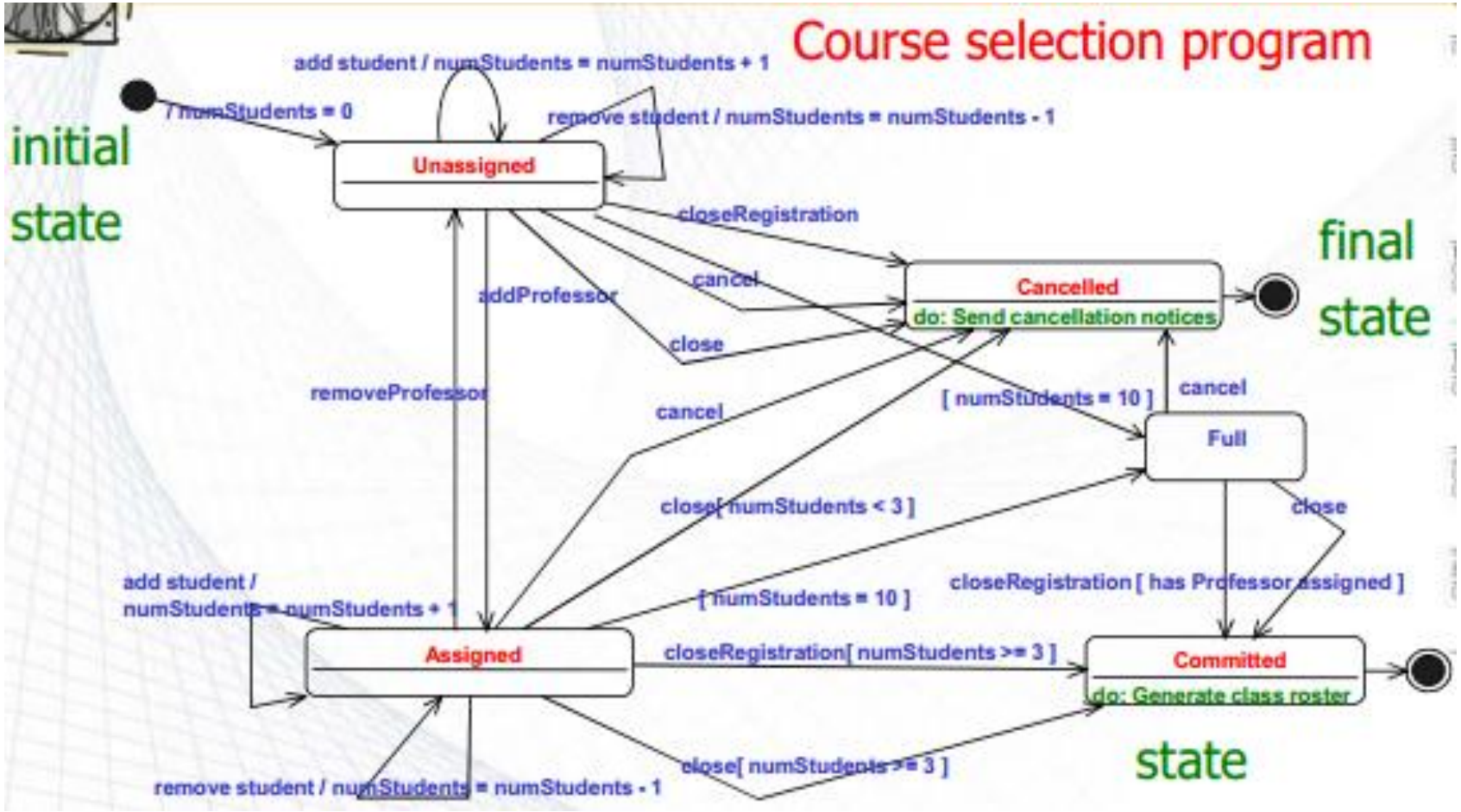
- Every interaction with the program (i.e., setting an input variable, selecting a menu item, etc.) makes the program state move to another state (using UML *activity* or *statechart* diagram).
- Test all paths that are likely to be followed by ordinary users under normal conditions (activity diagram).
- Test any setting in one place whose effects are suspected to be propagated to elsewhere in the program (statechart diagram).
- Test a few other random paths through the program.

Example: Activity Diagram

- An activity diagram shows the flow of events within the use-case.



Example: State chart Diagram

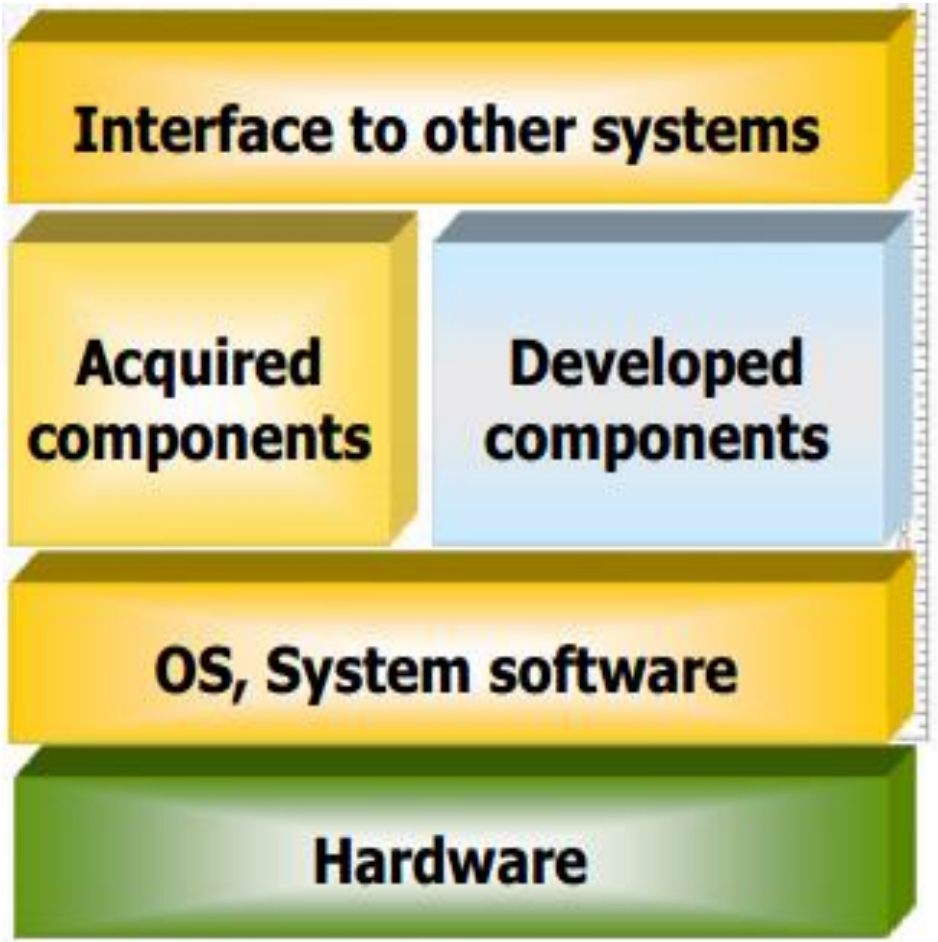


Allocating Test Time (group 6)

- How to manage test time?
- How to allocate test time among
- *system components* (e.g., acquired components, developed components)
- *test type* (e.g., certification test, feature test,load test, regression test) and
- *operation modes* (e.g., peak, prime and off hours modes)?

Test Time: Systems /1

- Allocate time to interface to other systems based on estimated risk.
- Allocate less than 10% of the remaining time to certification test of acquired components.
- Allocate time to the developed components based on their significance.



1. Test Time: Systems /2

Example:

- Total test time: 340 h (8.5 weeks)

Associated System Components 220 h

- Test case proportion:
  - Associated system components: 64.7%
  - Operating system component: 23.5%

2. Test Time: Test Type

- For each developed system components during reliability growth test:
  - for all new test cases: Allocate time to *feature test* (first release)
  - for all new test cases: Allocate time to *regression test* (subsequent releases)
- In this way testing all critical new operations will be guaranteed.
- Example: set aside 20 hours for feature test.
- The remaining time goes to *load test*.

3. Test Time: Load Test

- Allocate time for load test based on the proportion of the occurrences in operational modes
- Example: Web-based data transaction system

Operational mode	Proportion of transactions
Peak hours	0.1
Prime hours	0.7
Off hours	0.2

Operational mode	Test Time (h)		
	Interface	Product	OS, etc.
Peak hours	4	18	8
Prime hours	28	126	56
Off hours	8	36	16
Total (h)	40	180	80

The remaining 20 hours goes to feature and/or regression tests.

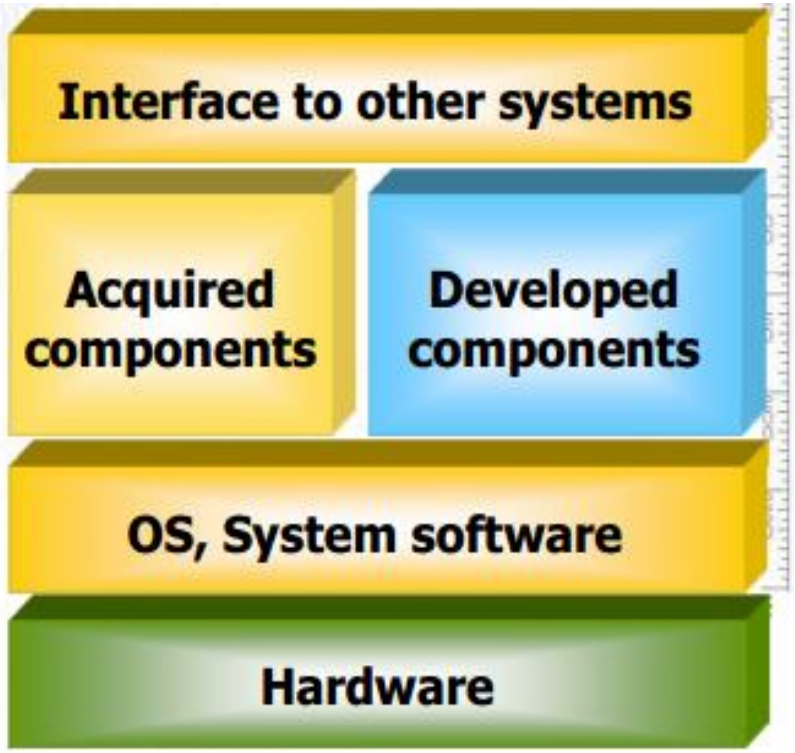
Test Cases Management

The procedure for preparing test cases involves:

1. Estimate the number of new test cases needed for the current release
2. Allocate the number of new test cases among the subsystems to be tested (system level)
3. Allocate the number of new test cases for each subsystem among its new operations (operation level)
4. Specify the new test cases
5. Adding the new test cases to the ones already available (may be from a previous release)

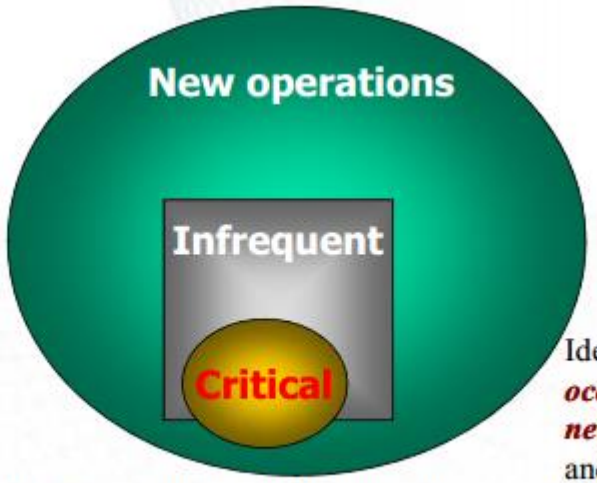
Test Case Allocation: System

- Allocate the bulk of the test cases to the developed product itself.



Test Case Allocation: Operations

Determine the threshold occurrence probability: 0.5 divided by the number of new test cases. Assign one test case to each ***infrequent new operation***.



Identify ***rarely occurring critical new operations*** and assign 2-4 test cases to each.

Assign the remaining test cases to the remaining new operations in accordance with the occurrence probabilities.

Example /1

- Total number of test cases: 500
- First release: All test cases are new.
- Suppose that we have one critical operation. Assign 2 test cases to it.
- Threshold occurrence probability: 0.5 / 500 = 0.001
- Suppose that the number of infrequent operations with occurrence probabilities below threshold is 2.
- Assign 1 test case to each infrequent operation.
- Distribute the remaining 500 - (2+2) = 496 test cases among the rest of operations based on their occurrence probabilities.

Example contd. /2

- Example: Occurrence probabilities for normal operation mode.

Operation	Occurrence probability
Process voice call, no pager, answer	0.18
Process voice call, no pager, no answer	0.17
Process voice call, pager, answer	0.17
Process fax call	0.15
Process voice call, pager, answer on page	0.12
Process voice call, pager, no answer on page	0.10
Phone number entry	0.10
Audit section of phone number database	0.009
Add subscriber	0.0005
Delete subscriber	0.0005
Recover from hardware failure	0.000001
Total	1.0

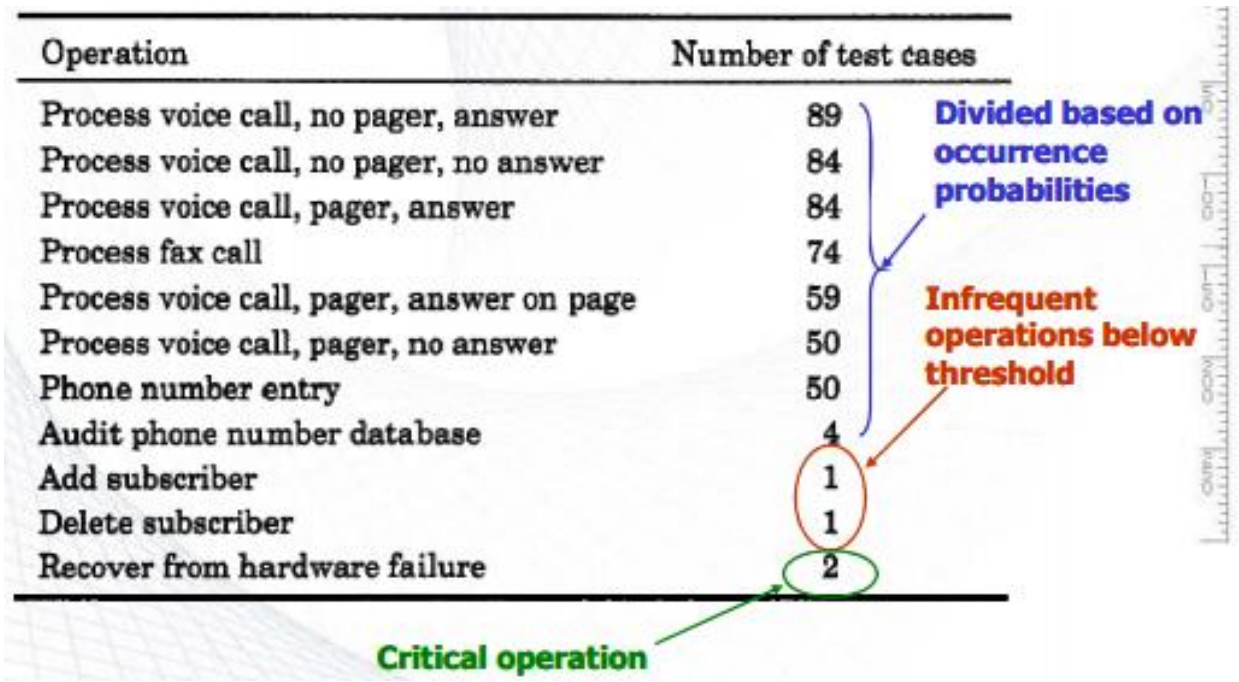
Infrequent operations below threshold

Critical operation

Table from Musa's Book

Example contd. /3

Operation	Number of test cases
Process voice call, no pager, answer	89
Process voice call, no pager, no answer	84
Process voice call, pager, answer	84
Process fax call	74
Process voice call, pager, answer on page	59
Process voice call, pager, no answer	50
Phone number entry	50
Audit phone number database	4
Add subscriber	1
Delete subscriber	1
Recover from hardware failure	2



Test Case Invocation

- In what order the system should be tested?
  - Recommended sequence of system test:
  - Acquired components (certification test only)
  - Developed product
    - Feature test and then load test for a new product
    - Feature test, and then regression test for subsequent releases
  - Other systems, OS, etc. (load test only)

Software Test Metrics:
Test Coverage Metrics

Software Test Metrics:

Test Coverage Metrics

Test Coverage Metrics /1

- Coverage of what?
- Statement coverage
- Branch coverage
- Component/Module coverage
- Specification coverage
- GUI coverage

Test Coverage /1

- Statement coverage (*CVs*)
  - Portion of the statements tested by at least one test case.

$$CV_s = \left(S_t / S_p \right) \times 100\%$$

S_t : number of statements tested

S_p : total number of statements

- Branch coverage (CV_b)
  - Portion of the branches in the program tested by at least one test case.

$$CV_b = \left(n_{bt} / n_b \right) \times 100\%$$

n_{bt} : number of branches tested

n_b : total number of branches

Test Coverage /2

- Component coverage (CV_{cm})
  - Portion of the components in the software covered and tested by at least one test case.

$$CV_{cm} = \left(n_{cmt} / n_{cm} \right) \times 100\%$$

n_{cmt} : number of components tested

n_{cm} : total number of components

- GUI coverage (CV_{GUI})
  - Portion of the GUI elements (e.g., menus, buttons, multiple selections, text fields, etc.) in the software covered and tested by at least one test case.

$$CV_{GUI} = \left(n_{GUIt} / n_{GUI} \right) \times 100\%$$

n_{GUIt} : number of GUI elements tested

n_{GUI} : total number of GUI elements

Other Coverage Metrics

- Path coverage
  - Test every possible path through the program. The number may be very large and not all the paths thoroughly tested.
- Boundary coverage
  - Every input/output and internal variables have their boundary values tested.
- Data coverage
  - Providing at least one test case for each data type or variable in the program.
- Test Pass Rate (*Rtp*)
  - Portion of the test cases that were executed successfully (i.e., produced expected output).

$$R_{tp} = \left(n_{t\,pass} / n_{t\,case} \right) \times 100\%$$

$n_{t\,pass}$: number of test cases passed

n : total number of test cases

Test Failure Rate (R_{tf})

- Portion of the test cases that were not executed successfully (i.e., produced different output than expected).

$$R_{tf} = \left(n_{t\,fail} / n_{t\,case} \right) \times 100\%$$

$n_{t\,fail}$: number of test cases failed

$n_{t\,case}$: total number of test cases

- Test Pending Rate (*Rtpend*)

  Portion of the test cases that were pending (i.e., couldn't be executed or correctness of the output couldn't be verified).

$$R_{tpend} = \left(n_{t\,pend} / n_{t\,case} \right) \times 100\%$$

$n_{t\,pend}$: number of test cases pending

$n_{t\,case}$: total number of test cases

Software Testability Metrics

What is a testable software?

- A software system is testable if there are built-in test (BIT) mechanisms implemented and described explicitly and those BITs can be activated from the external interface to the software.
- BITs can be implemented at different stages (design, coding, etc.) and levels (BCS level, component / module level, system / framework level)

Independently Determinable BCS

- A BCS is *independently determinable* (ID_BCS) if values of its Boolean variables or expressions are dependent only on the external inputs of the component that contains the BCS
- *Controllability* for ID_BCS

$$CA_{ID\_BCS} = f(I)$$

- Where {I} is the set of direct inputs of the component.

Non-Independently Determinable BCS

- A BCS is non-independently determinable if values of its Boolean variables or expressions are dependent on the external inputs of the component and a set of variables of the component that contains the BCS
- Controllability for BCS

$$CA_{BCS} = f(I, V)$$

{I} is the set of direct inputs of the component.

{V} is set of internal variables of the component.

- Example: path-sensitivity in a program (next path is selected based on the executed paths prior to BCS)

Test Controllability of a BCS

- Test controllability of a BCS (TCBCS) in a component is the capability to directly determine the control variables of the BCS by {I} the set of direct inputs of the component.

TCBCS = 1 if the BCS is independently determinable

TCBCS = 0 otherwise

Test Controllability of a Component

- Test controllability of a component (TC) is the mathematical mean of the TCBCS of its BCSs.

$$TC = \frac{1}{n}\sum_{i=1}^{n} TC_{BCSi}$$

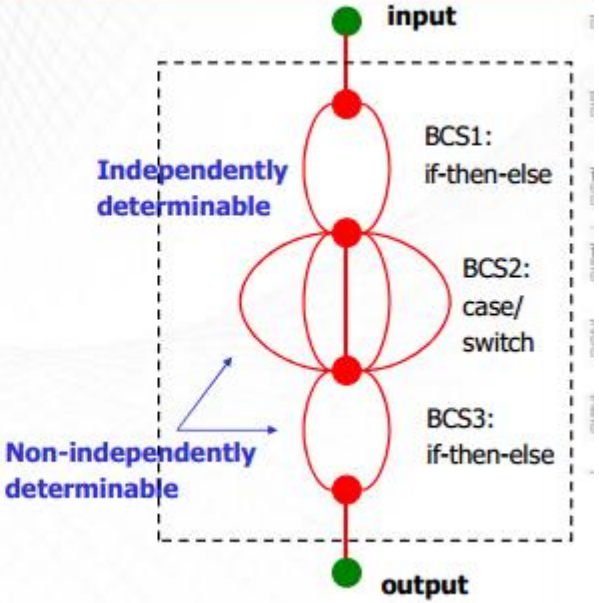
- A component with TC=1 means that testing the component is *fully controllable*.

- Example 1

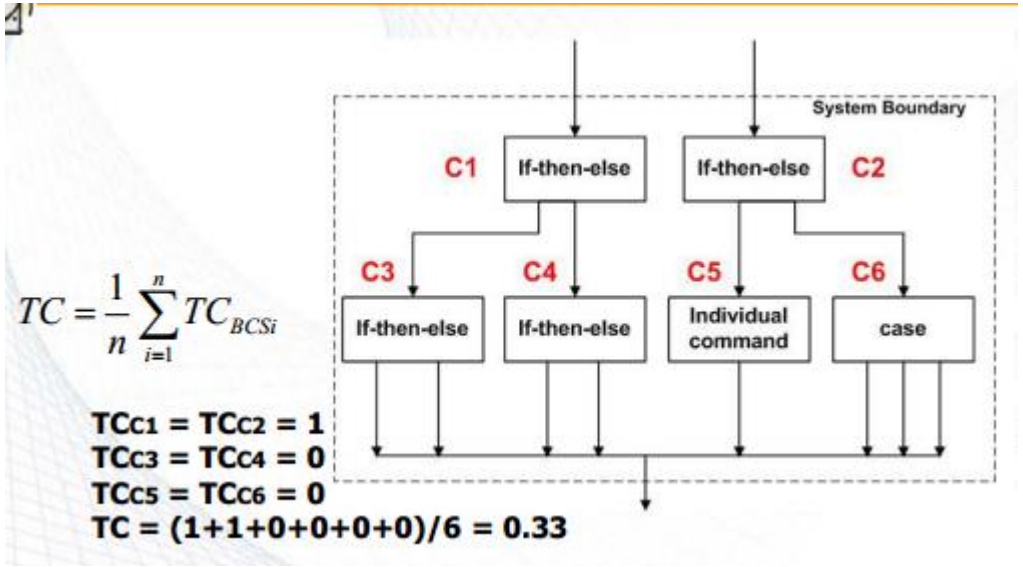
$$TC = \frac{1}{n}\sum_{i=1}^{n} TC_{BCSi}$$

$$TC = \frac{1}{3}(1+0+0)$$

$$TC = 0.33$$



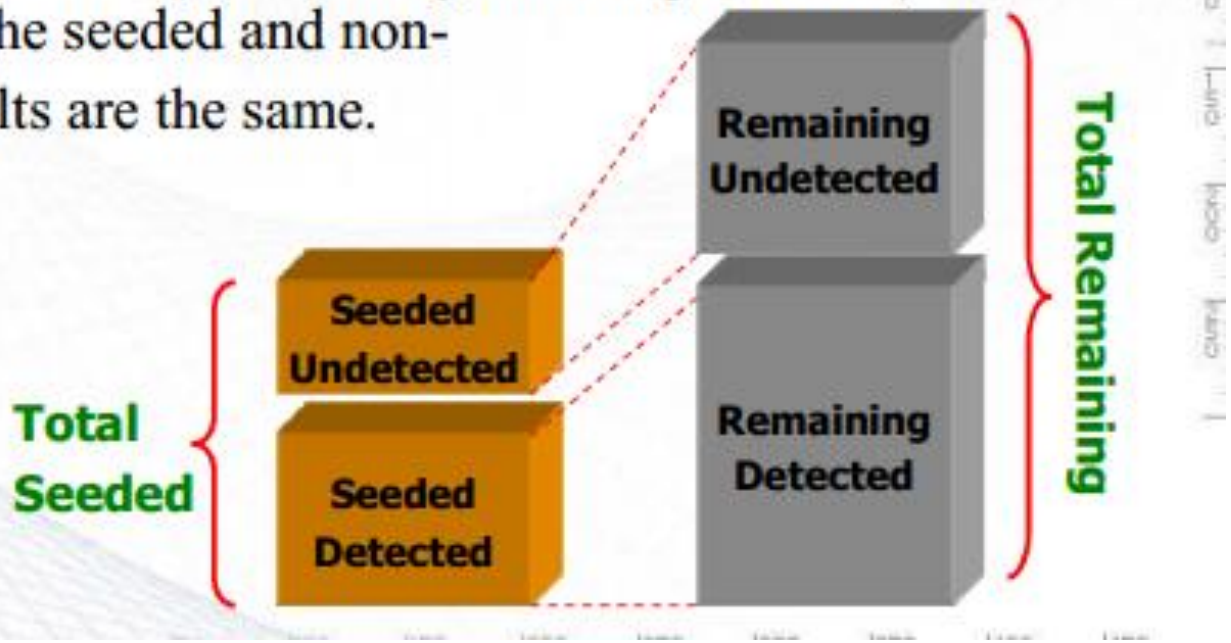
Example 2


System Boundary
C1
If-then-else
If-then-else
C2
C3
C4
C5
C6
If-then-else
If-then-else
Individual command
case
$TC = \frac{1}{n}\sum_{i=1}^{n} TC_{BCSi}$
TCc1 = TCc2 = 1
TCc3 = TCc4 = 0
TCc5 = TCc6 = 0
TC = (1+1+0+0+0+0)/6 = 0.33

Remaining Defects Metrics /1

Q: How to determine number of remaining bugs?

The idea is to inject (seed) some faults in the program and calculate the remaining bugs based on detecting the seeded faults [Mills 1972]. Assuming that the probability of detecting the seeded and non-seeded faults are the same.



Remaining Defects Metrics /2

$$\frac{n_s}{N_s} = \frac{n_d}{N_d} \quad \text{or} \quad N_d = \frac{n_d}{n_s} \times N_s$$

n_s detected seeded faults

N_s total seeded faults

n_d detected remaining faults

N_d total remaining faults

N_r undetected remaining faults

$$N_r = (N_d - n_d) + (N_s - n_s)$$

- The total injected faults (N_s) is already known; n_d and n_s are measured for a certain period of time.
- **Assumption:** all faults should have the same probability of being detected.

Example

- Assume that

$$N_s = 20 \qquad n_s = 10 \qquad n_d = 50$$

$$N_d = \frac{n_d}{n_s} \times N_s = \frac{50}{10} \times 20 = 100$$

$$N_r = (N_d - n_d) + (N_s - n_s)$$

$$N_r = (100 - 50) + (20 - 10) = 60$$

Comparative Remaining Defects /1

- Two testing teams will be assigned to test the same product.

$$N_d = \frac{d_1 d_2}{d_{12}} \qquad N_r = N_d - (d_1 + d_2 - d_{12})$$

Defects detected by Team 1 : d_1 ; by Team 2 : d_2

Defects detected by both teams: d_{12}

N_d total remaining defects

N_r undetected remaining defects

- Example

Defects detected

by Team 1 : $d_1 = 50$; by Team 2 : $d_2 = 40$

Defects detected by both teams: $d_{12} = 20$

$$N_d = \frac{d_1 d_2}{d_{12}} = \frac{50 \times 40}{20} = 100$$

$$N_r = N_d - (d_1 + d_2 - d_{12})$$

$$N_r = 100 - (50 + 40 - 20) = 30$$

Example 2

- According to Dr. Stephen Kan the "phase containment effectiveness" (PCE) in the software development process is:

$$PCE = \frac{\text{Defects removed (at the step)} \times 100\%}{\text{Defects existing on step entry} + \text{Defects injected during the step}}$$

- Higher PCE is better because it indicates better response to the faults within the phase. A higher PCE means that less faults are pushed forward to later phases.

- Using the data from the table below, calculate the phase containment of the requirement, design and coding phases.

Phase	*Number of defects*		
	Introduced	*Found*	*Removed*
Requirements	12	9	9
Design	25	16	12
Coding	47	42	36

$$PCE_{req} = \frac{9 \times 100\%}{0 + 12} = \%75 \qquad PCE_{design} = \frac{12 \times 100\%}{3 + 25} = \%42.85$$

$$PCE_{coding} = \frac{36 \times 100\%}{(13+3) + 47} = \%57.14$$

End …