**Department of Information Science and Engineering**

**18IS62-Software Testing Notes**

**Text Books**

1. Paul C. Jorgensen: Software Testing, A Craftsman’s Approach, 3rd Edition, Auerbach

Publications, 2008. (Listed topics only from Chapters 1, 2, 5, 6, 7, 9, 10, 12, 13)

2. Mauro Pezze, Michal Young: Software Testing and Analysis – Process, Principles and

Techniques, Wiley India, 2009. (Listed topics only from Chapters 3, 4, 16, 17, 20,21,

22,24)

3. Aditya P Mathur: Foundations of Software Testing, Pearson Education, 2008.( Listed topics only from Section 1.2 , 1.3, 1.4 ,1.5, 1.8,1.12,6. 2.1,6. 2.4 )

**Unit I- Basics of Software Testing (**T.B1:Chapter1, T.B3:Chapter1, T.B1:Chapter2) **Basics of Software Testing:** Basic definitions, Software Quality , Requirements, Behavior and Correctness, Correctness versus Reliability, Testing and Debugging, Test cases, Insights from a Venn diagram, Identifying test cases, Test-generation Strategies, Test Metrics, Error and fault taxonomies ,

Levels of testing, Testing and Verification, Static Testing.

**Problem Statements:** Generalized pseudo code, the triangle problem, the NextDate function, the commission problem, the SATM (Simple Automatic Teller Machine) problem, the currency converter, Saturn windshield wiper

**Basic Definitions:**

**Error**: People make errors. A good synonym for error is “mistake”. When people make mistakes while coding, we call these mistakes “bugs”. Errors tend to propagate; a requirements error may be magnified during design, and amplified still more during coding

e.g developer's coding error like Syntactic Error, User interface error, Flow control error, Error

handling error, Calculation error, Hardware error, Testing Error

**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, dataflow diagrams, hierarchy charts, source code, and so on. “Defect” is a good synonym for fault; so is “bug”. Faults can be elusive. A fault may happen in a program because of the following reasons:

• Lack of resources

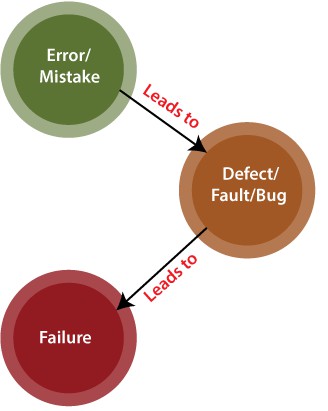
• An invalid step

• Inappropriate data definition

e.g Business Logic Faults, Functional and Logical Faults, faulty GUI, , performance Faults

Security Faults, Software/ hardware fault

**Failure**: A failure occurs when a faulty code 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 code. The second subtlety is that this definition relates failures only to faults of commission



**Fault of Omission:** refers to something which we fail to enter correct information . Very difficult to detect and solve.

• How can we deal with failures that correspond to faults of omission?

• What about faults that never happen to execute, or perhaps do not execute for a long time?

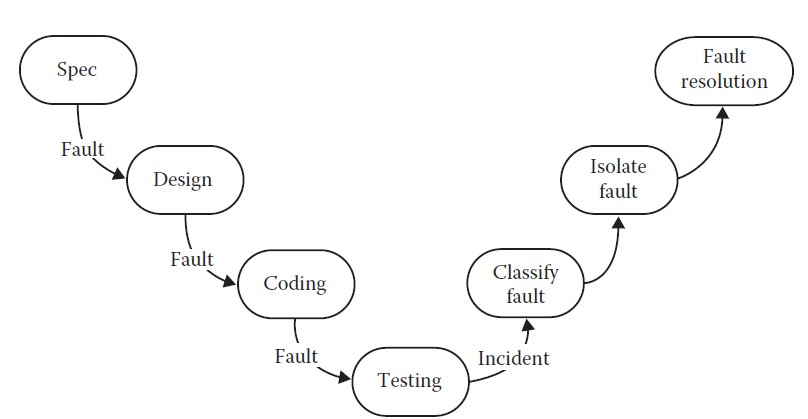
**Fault of Commission:** refers to which we enter wrong in the representation of something

**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(s) associated with a failure that alerts the user to the occurrence of f**a**ilure.

**Test**: Testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. There are two distinct goals of a test: either to find failures, or to demonstrate correct execution.

**Test Case**: The essence of software testing is to determine a set of test cases for the item to be tested. A test case is (or should be) a recognized work product. A test case has an identity, and is associated with a program behavior. A test case also has a set of inputs, a list of expected outputs

**Software Testing Life Cycle:**



**Fig 1:Software Testing Life Cycle**

• Notice that, in the development phases, three opportunities arise for errors to be made, resulting in faults that may propagate through the remainder of the development process.

• The fault resolution step is another opportunity for errors(and new faults). When a fix causes formerly correct software to misbehave, the fix is deficient.

• From this sequence of terms, we see that test cases occupy a central position in testing.

• The process of testing can be subdivided into separate steps:

o test planning,

o test case development, o running test cases, and o evaluating test results.

**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.

**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. An example of inconsistency would be when a database application displays the date of birth of a person in the database without regard for the user's preferences.

**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, Behavior, Correctness**

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 to sorted in ascending or in 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 **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.

**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. 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.

Test Case 1

Test data: <''A'’ 12 -29 32 > Expected output: -29 12 32

Test Case 2

Test data: <'‘D'’ 12 -29 32. > Expected output: 32 12 -29

Test Case 3

Test data: <''A'’ .>

Expected output: No input to be sorted in ascending order

Test Case 4

Test data: <'‘D'’. >

Expected output: No input to be sorted in descending order

Test Case 5

Test data: <'‘R'’ 12 -29 32. >

Expected output: Invalid request character; valid characters ‘A’ and ‘D’

Test Case 6

Test data: <'‘D'’ c,17,2 . >

Expected output: Invalid number

**Correctness**

Though correctness of a program is desirable, 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.

**Correctness and Testing**

While correctness attempts to establish that the program is error free, testingattempts to find if there are any errors in it.

Thus, completeness of testing does not necessarily demonstrate that a program is error free.

**Software reliability: two definitions**

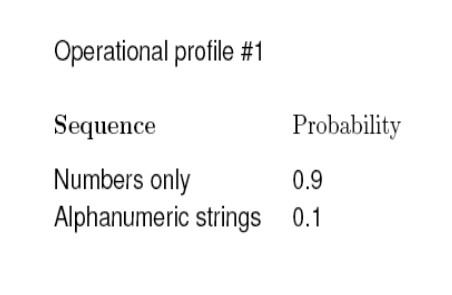
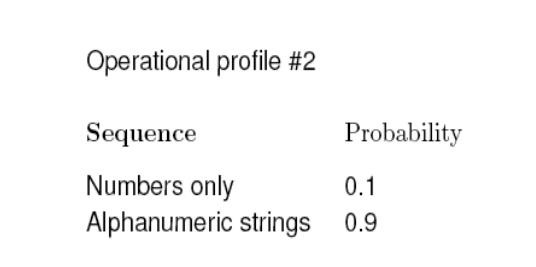
**Software reliability [**ANSI/IEEE Std 729-1983]: is the probability of failure free operation of software over a given time interval and under given conditions.

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

**Operational profile**

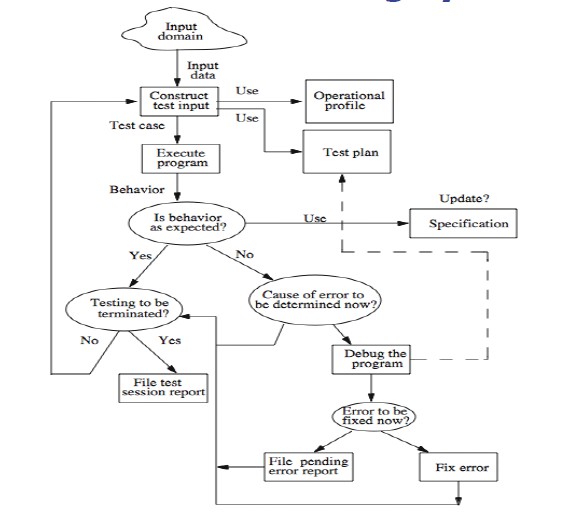
An operational profile is a numerical description of how a program is used.

Consider a sort program which, on any given execution, allows any one of two types of input sequences. Sample operational profiles for sort follow.



**Testing and debugging**

Testing is the process of determining if a program has any errors. When testing reveals an error, the process used to determine the cause of this error and to remove it, is known as debugging. Test or debug cycle is depicted in Fig 2



**Test plan**

**Fig 2: Test or Debug Cycle**

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 request characters.

• Execute the program on an empty input sequence.

• 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.

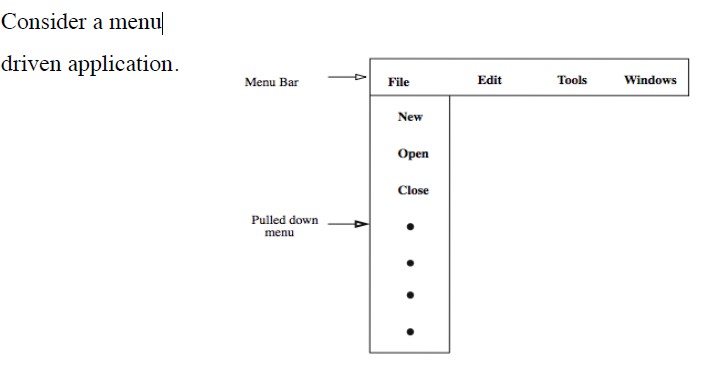
A test set is a collection of zero or more test cases. Sample test case for sort:

Test data: <''A'’ 12 -29 32 > Expected output: -29 12 32

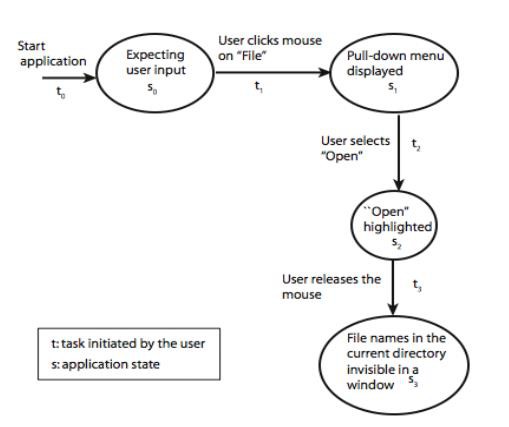
**Program behavior:**

• 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. inputs.



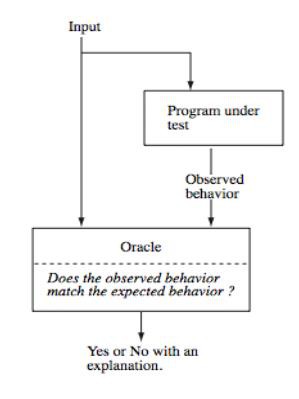
**Program Behaviour Example:**



**Behavior: observation and analysis**

The entity that performs the task of checking the correctness of the observed behavior is known as an oracle. In the first step one observes the behavior. In the second step one analyzes the observed behavior to check if it is corrector not. Both these steps could be quite complex for large commercial programs.

**Oracle Example:**



**Oracle: Programs**

Oracles can also be programs designed to check the behavior of other programs.

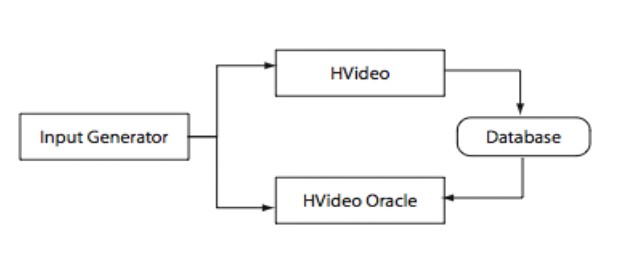
For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix A and generates B as the output matrix.

**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.

**Oracle Construction Example:**



**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, program verification is often avoided, and the focus is on testing.

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

• Verification promises to verify that a program is free from errors. However, the person/tool who verified a program might have made a 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.

**Test Cases**

• The essence of software testing is to determine a set of test cases for the item to be tested. A

test case is (or should be) a recognized work product.

• A complete test case will contain a test case identifier, a brief statement of purpose (e.g., a business rule), a description of preconditions, the actual test case inputs, the expected outputs, a description of expected post conditions, and an execution history.

• The execution history is primarily for test management use—it may contain the date when the test was run, the person who ran it, the version on which it was run, and the pass/fail result.

• Test case execution entails establishing the necessary preconditions, providing the test case inputs, observing the outputs, comparing these with the expected outputs, and then ensuring that the expected post conditions exist to determine whether the test passed.

• From all of this, it becomes clear that test cases are valuable—at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved.

**Insights from a Venn Diagram**

• Testing is fundamentally concerned with behavior, and behavior is orthogonal to the code- based view common to software (and system) developers.

• A quick distinction is that the code-based view focuses on what it is and the behavioral view considers what it does.

• One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers;

• the emphasis is therefore on code-based, instead of behavioral, information.

Venn diagram clarifies several questions about testing.

Consider a universe of program behaviors Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors. Figure shows the relationship between the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labelled S and all those behaviors actually programmed are in P.

• What if certain specified behaviors have not been programmed? In our earlier terminology, these are faults of omission.

• Similarly, what if certain programmed (implemented) behaviors have not been specified?

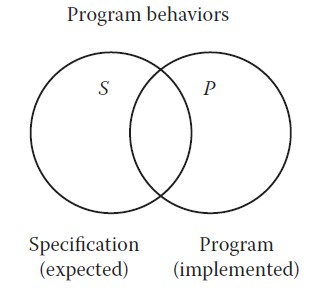
These correspond to faults of commission and to errors that occurred after the specification was complete.

• The intersection of S and P (the football-shaped region) is the “correct” portion, that is,

behaviors that are both specified and implemented.

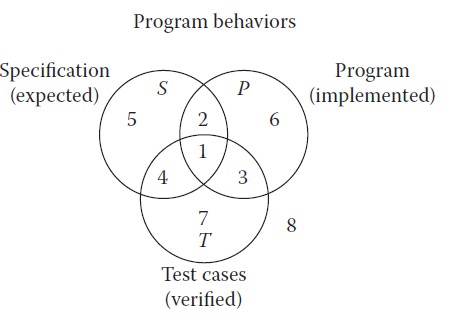
• A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented.

**Specified and implemented program behaviors**



The new circle in the following is for test cases.

**Specified, implemented, and tested behaviors**



Now, consider the relationships among sets S, P, and T.

• There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested(regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).

• Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to behaviors that were not implemented (regions 4 and 7).

• Each of these regions is important. If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete. If certain test cases correspond to unspecified behaviors, either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified non-behavior does not occur.

**Identifying Test Cases**

There are two fundamental approaches to identifying test cases;

• Functional testing or Specification Based Testing

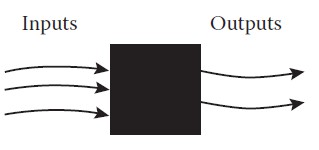
• Structural testing. or Code Based Testing

***Specification-Based Testing***

• Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range.

• This is also called black box testing, in which the content (implementation) of the black box is

not known, and the function of the black box is understood completely in terms of its inputs and outputs.

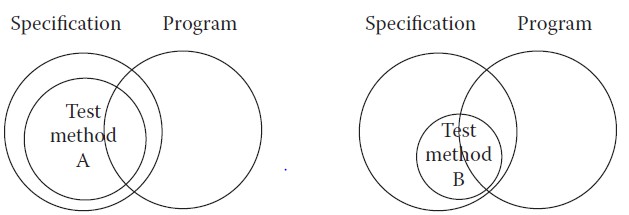


• With the specification-based approach to test case identification, the only information used is the specification of the software.

• Therefore, the test cases have two distinct advantages: (1) they are independent of how the software is implemented, so if the implementation changes, the test cases are still useful; and (2) test case

development can occur in parallel with the implementation, thereby reducing the overall project development interval.

• On the negative side, specification based test cases frequently suffer from two problems: significant redundancies may exist among test cases, compounded by the possibility of gaps of untested software



**Comparing specification-based test case identification methods.**

• Method A identifies a larger set of test cases than does method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Because specification based methods are based on the specified behavior

***Code-Based Testing***

• Code-based testing is the other fundamental approach to test case identification.

• To contrast it with black box testing, it is sometimes called white box (or even clear box) testing.

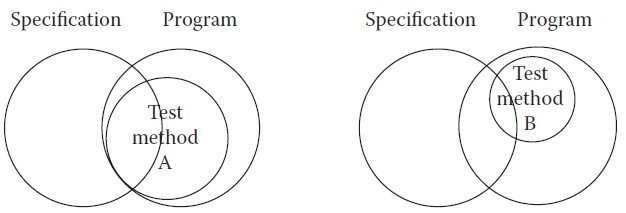
• the essential difference is that the implementation(of the black box) is known and used to identify test cases.

• The ability to “see inside” the black box allows the tester to identify test cases on the basis of how the function is actually implemented.

• Code-based testing and use lends itself to the definition of test coverage metrics.

• Test coverage metrics provide a way to explicitly state the extent to which a software item has been tested, and this in turn makes testing management more meaningful.

**Comparing code-based test case identification methods.**

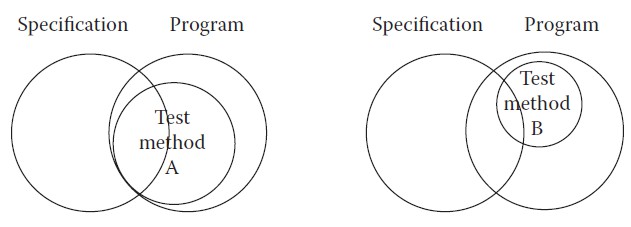


• Method A identifies a larger set of test cases than does method B.

• For both methods, the set of test cases is completely contained within the set of programmed behavior.

Because code-based methods are based on the program, it is hard to imagine these methods identifying behaviors that are not programmed.

• It is easy to imagine, however, that a set of code-based test cases is relatively small with respect to the full set of programmed behaviors



• Specification-based testing uses only the specification to identify test cases, while code-based testing uses the program source code (implementation) as the basis of test case identification.

• Neither approach by itself is sufficient.

• Consider program behaviors: if all specified behaviors have not been implemented, code-based test cases will never be able to recognize this. Conversely, if the program implements behaviors that have not been specified, this will never be revealed by specification-based test cases. (A Trojan horse is a good example of such unspecified behavior.)

**Test-generation Strategies**

• Key Task in any software Test activity is generation of test cases. How to generate Test cases?

• 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.

• The tests are generated using a mix of formal and informal methods either directly from the requirements document serving as the source.

• In more advanced test processes, requirements serve as a source for the development of formal models.

• Several strategies are there for test case generation

• In the figure, top row captures techniques that are applied directly to the requirements. These are informal techniques that assign values to input variables without the use of formal methods.

• These techniques identify input variables and use formal techniques for test generation and cause effect graphing.

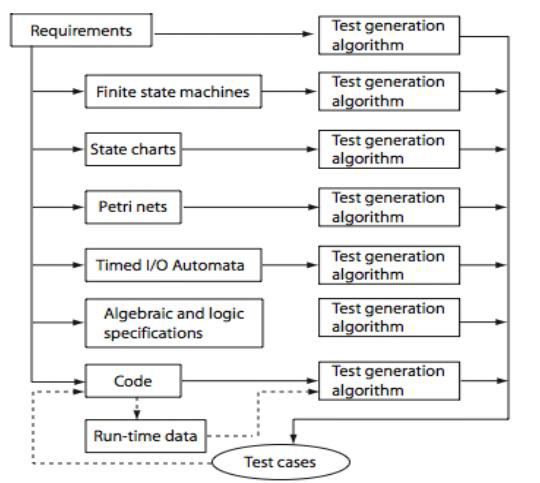
• Another way is use of model based testing

• They need subset of requirements to be modeled using a formal notation which is called as specification.

• The tests are generated from specification using FSMs, Statecharts, Petri Nets and Timed I/O Automata notations for modeling.

• Unified modeling language can also used for modeling the requirements into proper specification for test case generation.

• Model can also be built using predicate Logic and algebraic languages. Each model has its own strengths and weaknesses



• Code based Test case generation techniques are also available.

• Existing tests can be enhanced based on adequacy criteria using code test case generation techniques.

• Code based techniques can be used generate tests, or modify existing ones, to generate new tests that force a condition to evaluate to true or false.

• Two techniques: Program mutation and control flow coverage techniques

• Code based test case generation used during regression testing when they need to reduce the size of the test suite or prioritize tests against which regression test is to be performed.

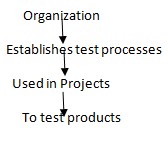
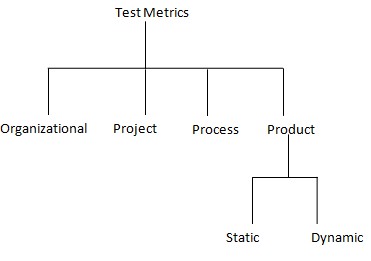
**Test Metrics:**

• Quantitative measurement determining the extent to which a software process, product or project possesses a certain attribute (used for tracking purposes)

• Goal for the metric is to quantify the progress of the product toward a specified quality objective

• standard measurement.

• Variety of metrics in Software Testing



• Each set of metrics has its value in Monitoring, planning and control

• There are 4 metrics which are core areas

• Schedule -measures actual completion times of various activities and compare these with estimated time to completion

• Quality- measures quality of a product or process

• resources-measures cost in $, Manpower and tests executed

• Size-measures size of the various objects like source code and number of tests in a test suite

**Organizational Metrics**

o Metrics at organizational level are useful in project planning and management.

o obtained aggregating compatible metrics across multiple projects

o useful metric of product quality- number of defects reported after product release averaged over a set of products developed and marketed by an organization.

o Allow senior management to monitor the overall strength of the organization and points to areas od weaknesses

o Helps in setting new goals plan for resources needed to realize these goals

o Examples:

o - Defects per thousand lines of code (defects/KLOC)

o - Testing cost per KLOC

o - Actual schedule of system testing

o - Delivery schedule slippage

**Project Metrics**

• Project Metrics relates to a specific project. Useful in the monitoring and control of the project.

Project Metrics used to track project progress and allocate resources to the projects

Examples:

- The ratio of actual to expected system test effort

- Ratio of number of successful tests to the total number of tests.

**Process Metrics**

• Every project uses the process. The goal of Process Metrics to measure the goodness of the process.

• When a test process consists of several phases (Unit test, integration test and system test) one can measure how many defects were found in each phase. Defect is to be fixed at the earlier phase. Otherwise it is costlier.

• feedback to improve the process, productivity

• A metric classifies defects according to the phase that assists in evaluating the process itself.

**Product Metrics**

• Product metrics relate to a specific product such as a compiler for a programming language. Product metrics are useful in making decisions related to the product. used to track quality of product (Static, Dynamic)

• For example should the product be released to the client?

Examples:

Given the CFG G of program P,

- Cyclomatic complexity: V(G) = E - N + 2p

• V(G) is the complexity of the control flow graph. E is the edges, N

nodes, p the connected procedures

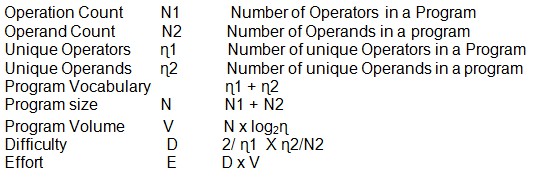
• V(G) of the value 5 or less are preferred

• V(G) is a complexity of procedure p that corresponds to G not entire program. V(G) counts only procedures that are reachable from main function.

**Halstead metrics:**

• B = 7.6 E0.667S0.333

B represents the number of errors found during software development. S the program size and E is the effort.



**Product Metrics : OO Software**

**Reliability**: Probability of failure of a software product with respect to a given operational profile in a given environment

**Defect Density**: Number of defects per KLOC

**Defect Severity**: Distribution of defects by their level of severity **Test Coverage**: Fraction of Testable items e.g. Basic blocks covered **Cyclomatic Complexity**: Measures complexity of a program based on its CFG

**Weighted Methods per class:** ∑i=1 to n Ci, Ci is the complexity of method i in the class

**Class Coupling**: Measures the number of classes to which a given class is coupled

**Response set**: Set of all methods that can be invoked directly or indirectly, when a message is sent to object O

**Number of Children**: Number of immediate descendents of a class in the class hierarchy

**Static and Dynamic Metrics**

• **Static metrics** are those computed without executing the product. E.g., Number of testable entities in an application. Average number of tester working on a project is a static metric

• **Dynamic metrics** require code execution e.g., Number of Testable entities actually covered by a test suite. Number of defects remaining to be fixed is dynamic attribute because it can be computed accurately only after a code change has been made and the product retested.

**Testability:** The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

**Error and fault taxonomies**

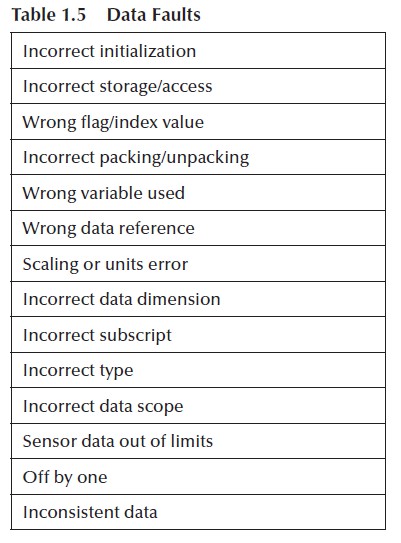
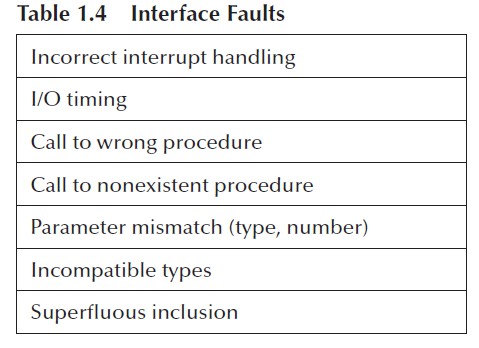
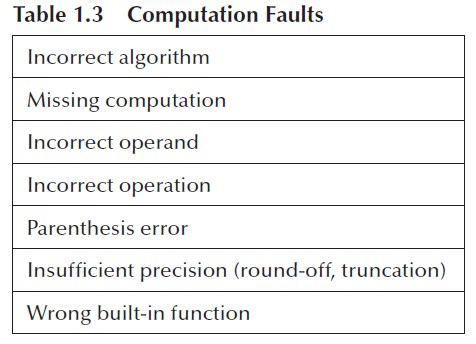
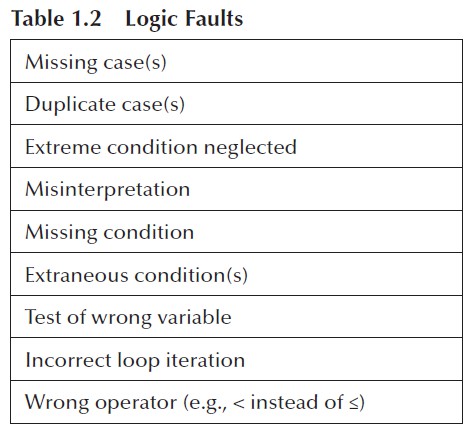
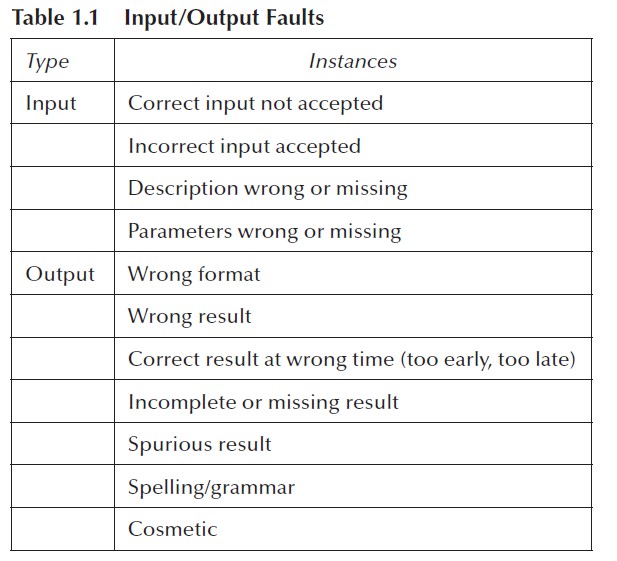
• distinction between process and product: process refers to how we do something, and product is the end result of a process. The point at which testing and Software Quality Assurance (SQA) meet is that SQA typically tries to improve the product by improving the process. In that sense, testing is clearly more product oriented.

• SQA is more concerned with reducing errors endemic in the development process, whereas testing is more concerned with discovering faults in a product.

• Both disciplines benefit from a clearer definition of types of faults.

• Faults can be classified in several ways: the development phase in which the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on. My favorite is based on anomaly (fault) occurrence: one time only, intermittent, recurring, or repeatable.

• The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition, investigation, action, and disposition. Some of the more useful anomalies are given in Tables 1.1 through 1.5;



**Levels of testing**

• Levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle. Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level.

• A diagrammatic variation of the waterfall model, known as the V-Model

• is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels.

• Notice that, especially in terms of specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing—system, integration, and unit testing.

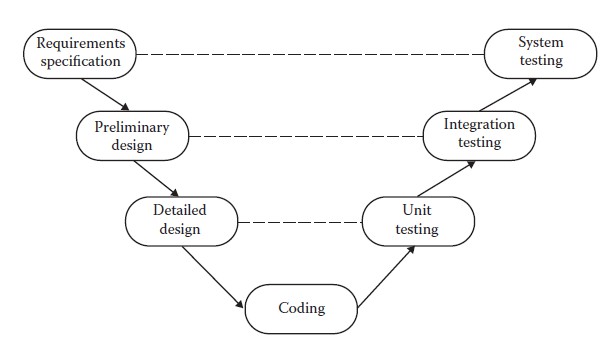
• A practical relationship exists between levels of testing versus specification-based and code based testing.

• Most practitioners agree that code-based testing is most appropriate at the unit level, whereas specification-based testing is most appropriate at the system level.

• This is generally true; however, it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases.

• The constructs defined for code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing.

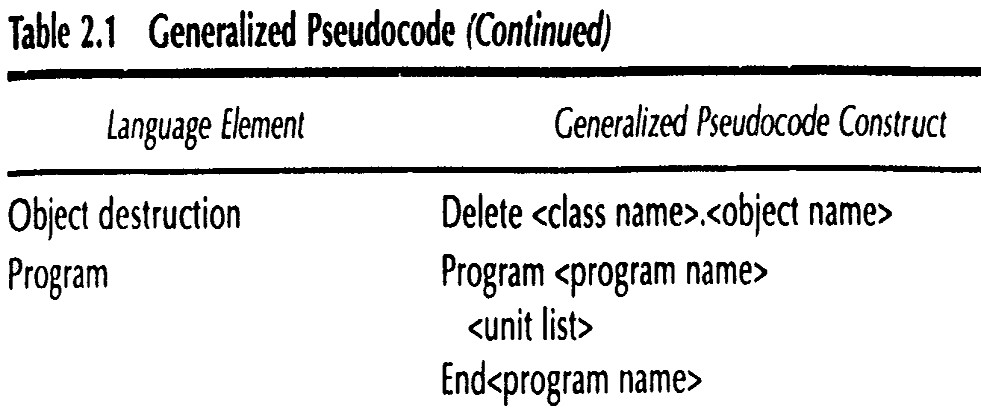
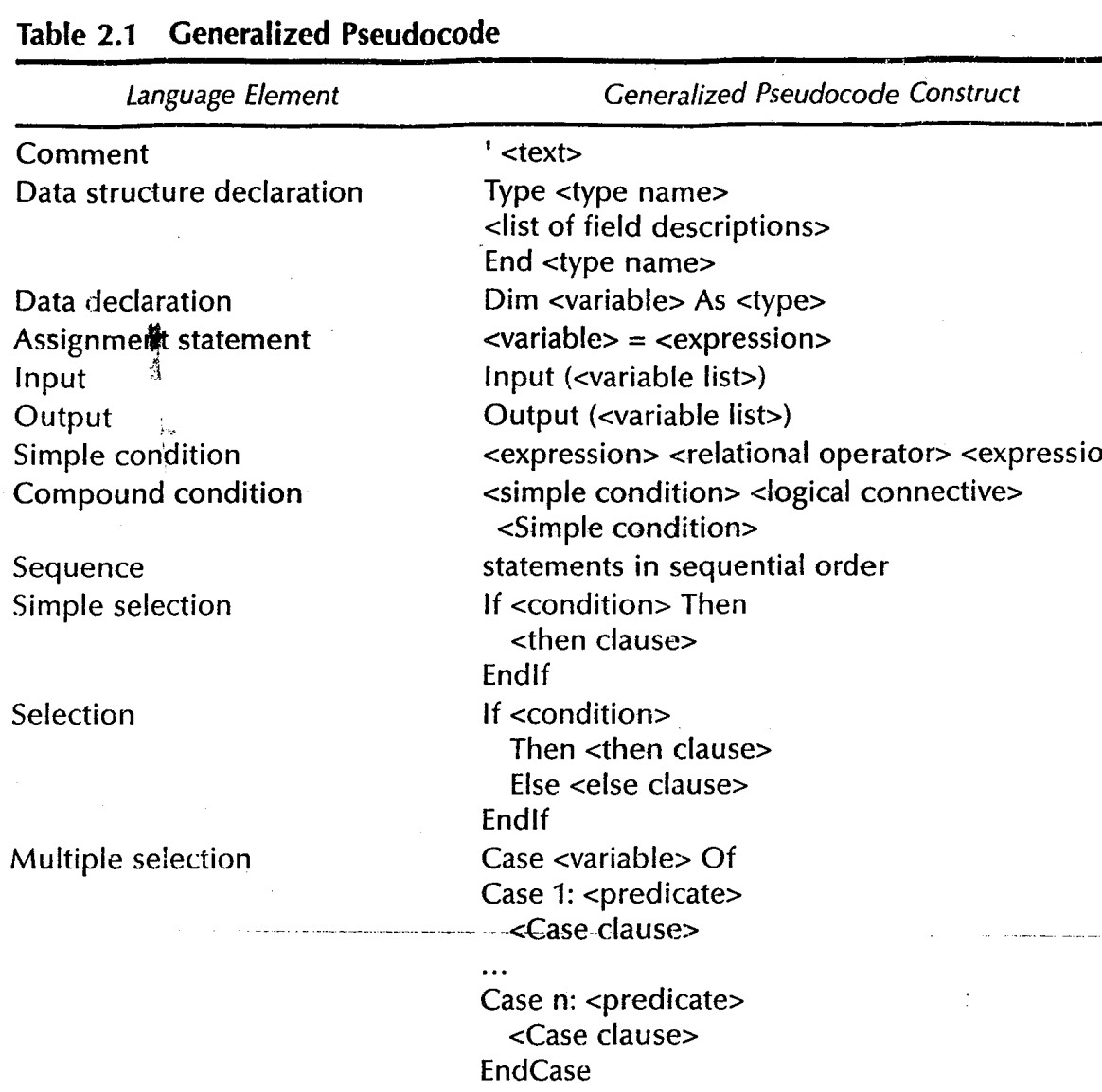
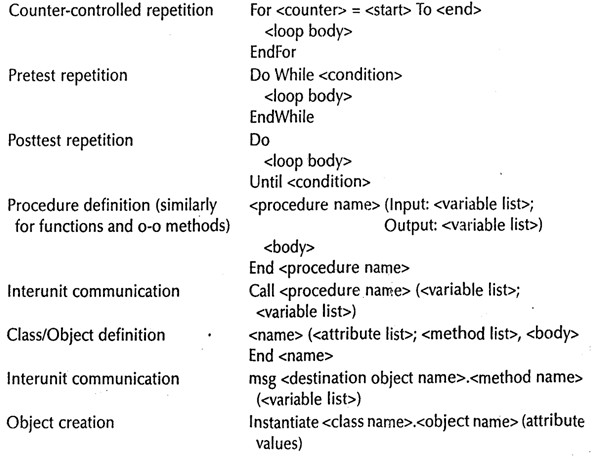
**Levels of abstraction and testing in waterfall model**



**Generalized Pseudo Code**

 Pseudo code provides a *“language neutral”* way to express program source code.

 Pseudo code given here is based on visual basic.



**Triangle problem:**

The Triangle Program accepts three integers as input; these are taken to be sides of a triangle. The

output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. Sometimes this problem is extended to include right triangles as a fifth type.

The triangle program accepts three integers, a, b, and c, as input. These

are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

The output of the program is the type of triangle determined by the three sides: Equilateral,Isosceles, Scalene, or NotATriangle. If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message, for example, “Value of b is not in the range of permitted values.” If values of a, b, and c satisfy conditions c4, c5, and c6, one of four mutually exclusive



outputs is given:

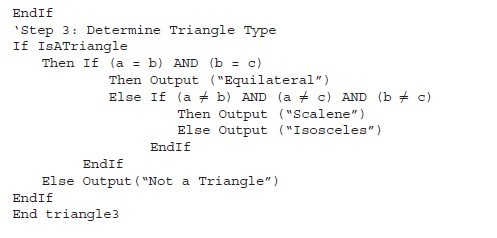
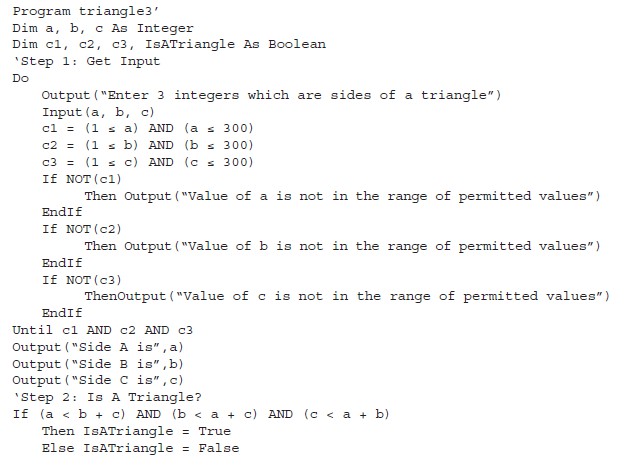
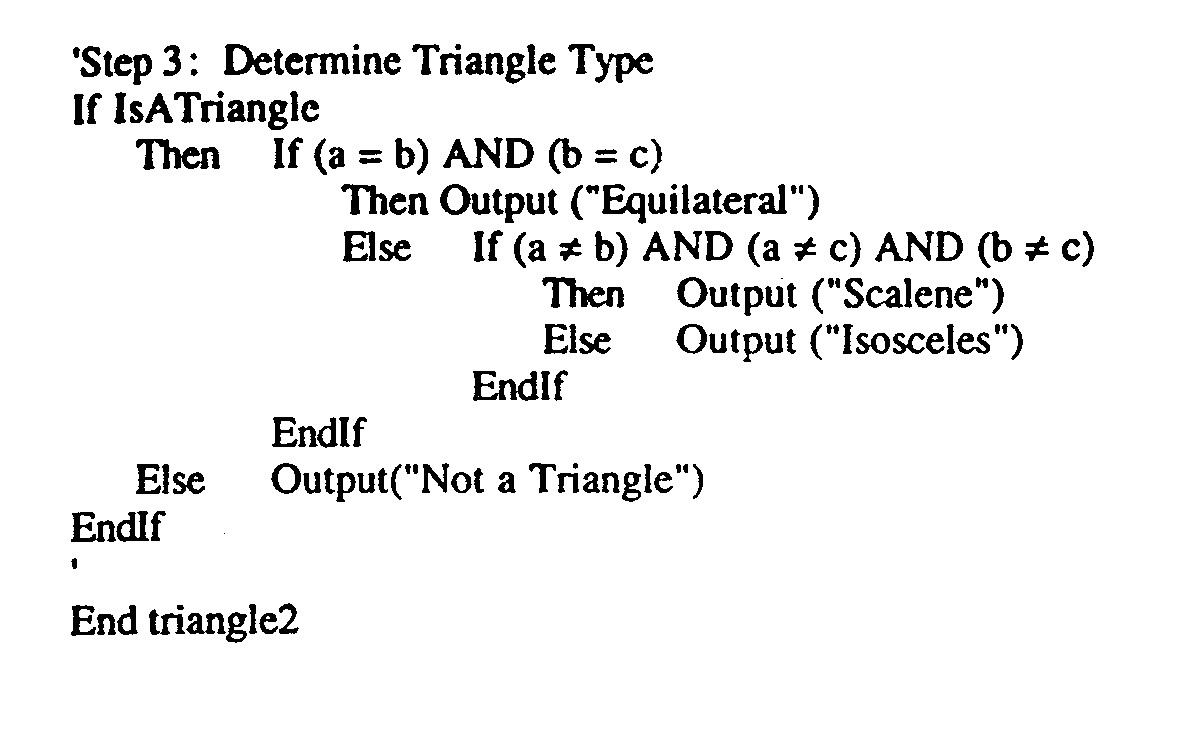
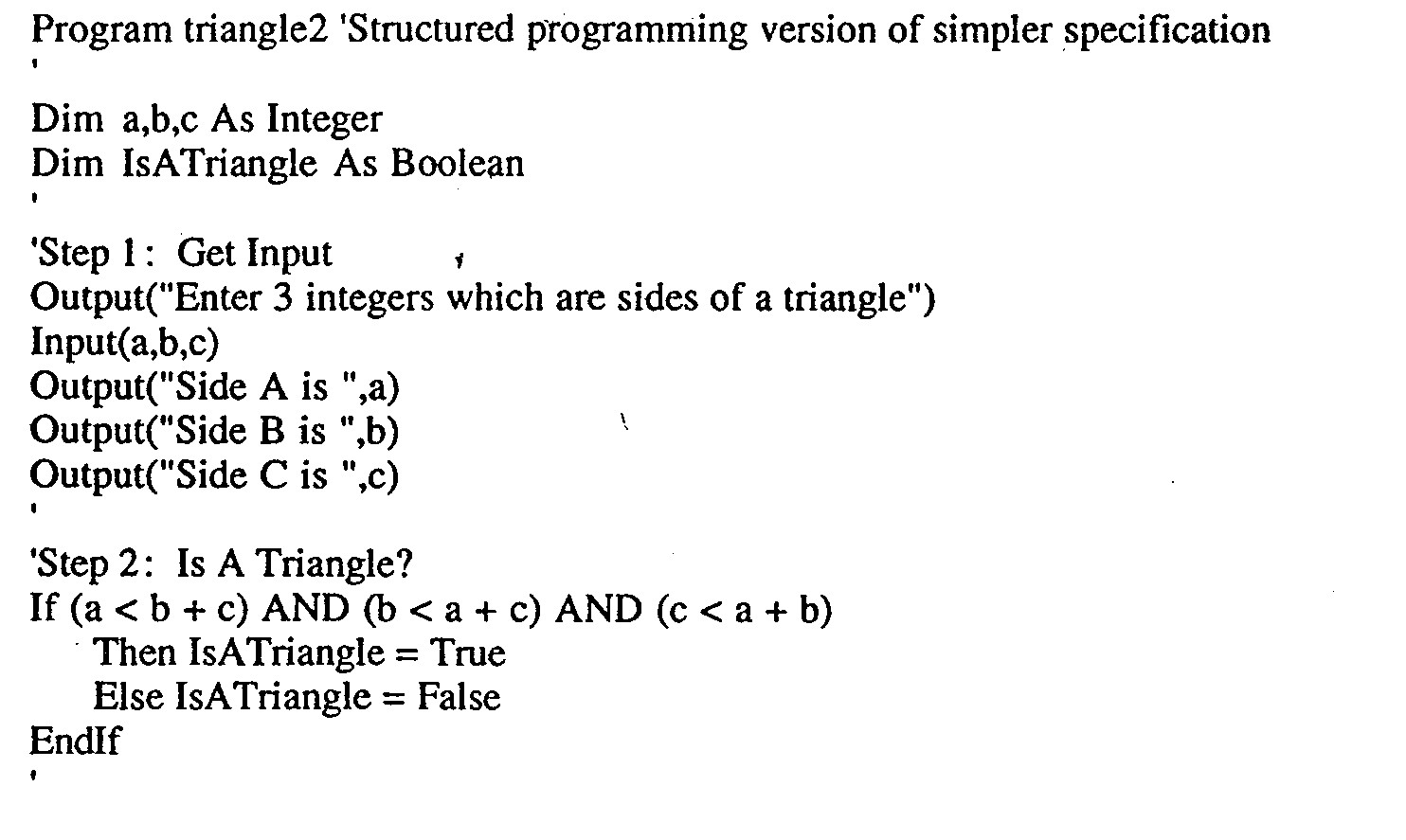
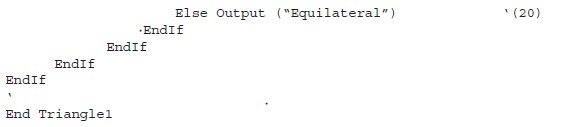
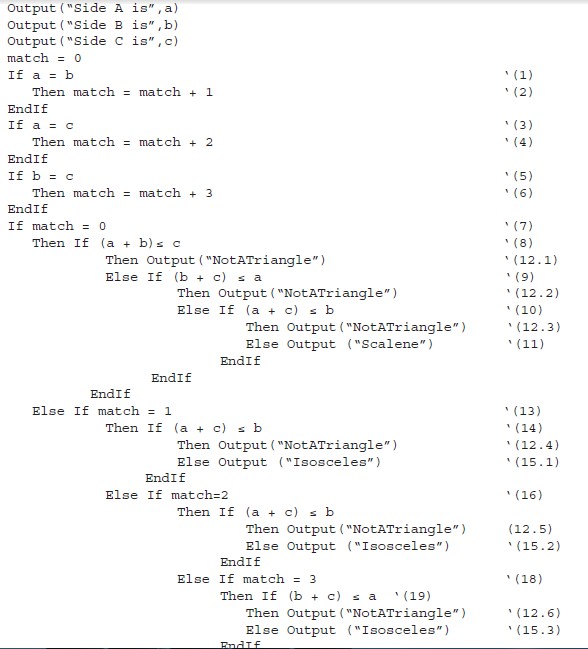
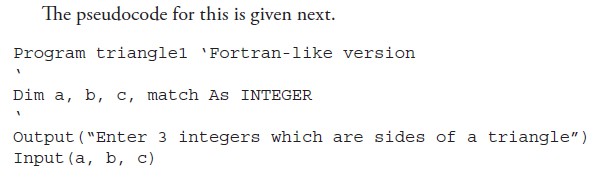
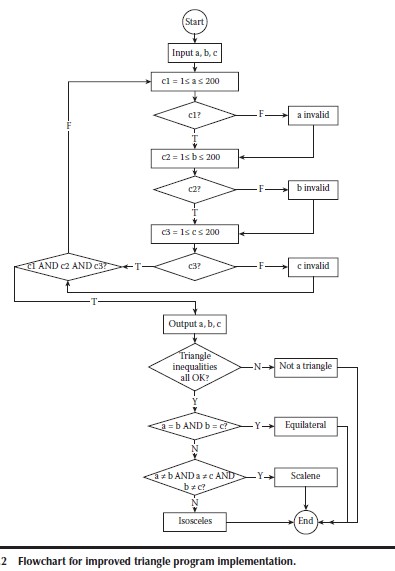
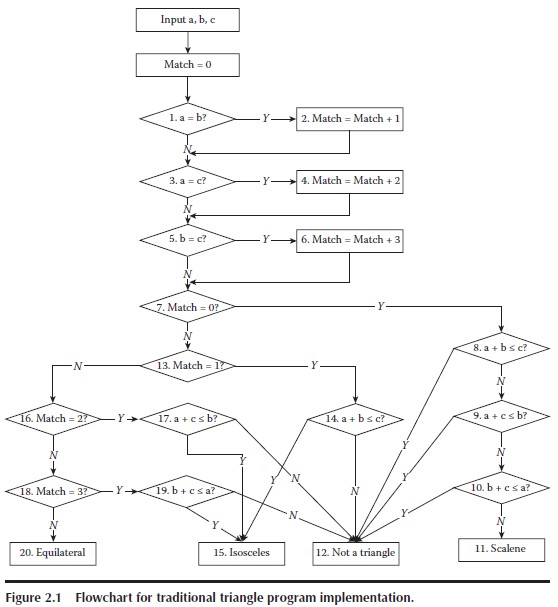
1. If all three sides are equal, the program output is Equilateral.

2. If exactly one pair of sides is equal, the program output is Isosceles.

3. If no pair of sides is equal, the program output is Scalene.

4. If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.

Triangle Property: the sum of any pair of sides must be strictly greater than the third side. If a, b, and c denote the three integer sides, then the triangle property is mathematically stated as three inequalities: a < b + c, b < a + c, and c < a + b. If any one of these fails to be true, the integers a, b, and c do not constitute sides of a triangle. If all three sides are equal, they constitute an equilateral triangle; if exactly one pair of sides is equal, they form an isosceles triangle; and if no pair of sides is equal, they constitute a scalene triangle



**The NextDate Problem**

The complexity in the Triangle Program is due to relationships between inputs and correct outputs. We will use the NextDate function to illustrate a different kind of complexity — logical relationships among the input variables themselves.

**Problem Statement**

NextDate is a function of three variables: month, day, and year. It returns the date of the day after

the input date. The month, day, and year variables have numerical values: with 1 ≤ month ≤ 12, 1

≤ day ≤ 31, and 1812 ≤ year ≤ 2012.

c1. 1 ≤ month ≤ 12 c2. 1 ≤ day ≤ 31

c3. 1812 ≤ year ≤ 2012

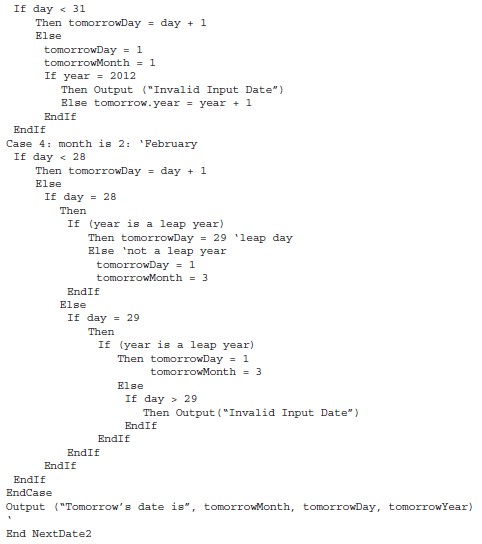
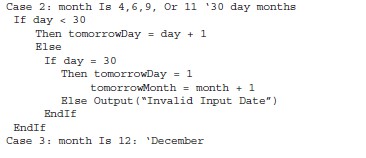
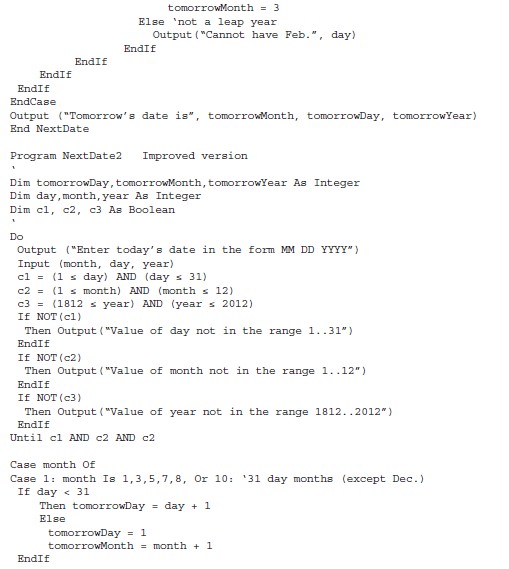
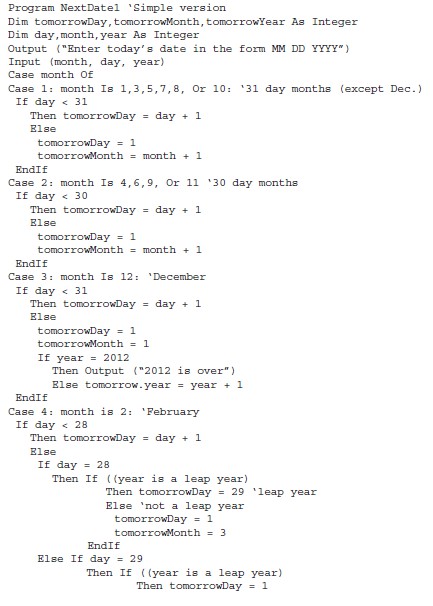
• We can make our problem statement more specific. This entails defining responses for invalid values of the input values for the day, month, and year. We can also define responses for invalid combinations of inputs, such as June 31 of any year. If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, “Value of month not in the range 1...12.” Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message:“Invalid Input Date.”

• Two sources of complexity exist in the NextDate function: the complexity of the input domain discussed previously, and the rule that determines when a year is a leap year. A year is

365.2422 days long; therefore, leap years are used for the “extra day” problem. If we declared a leap year every fourth year, a slight error would occur. A year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400.

• Notice how much of the source code is devoted to leap year considerations. In the second

implementation, notice how much code is devoted to input value validation.



**The Commission Problem**

• It contains a mix of computation & decision making.

• A rifle salesperson in the former Arizona territory sold rifle lock’s, stocks, & barrel’s made of

a gunsmith in Missouri.

• Locks cost $45, stocks cost $30, Barrel Cost $ 25.

• Sales person has to sell at least 1 complete rifle per month

• Production limitation such that 1 sales man can sell 70 locks, 80 stocks, 90 barrels per month.

• After each town visit salesperson update sale of no of locks, stocks, barrels through a telegram to gunsmith

• At the end of month salesperson sent a shot telegram showing -1 locks sold.

• Gunman knew sales for month are over & compute the commission of sales person

• 10% on sales up to $1000

• 15% on the next $800

• 20% on any sales in excess of $1800

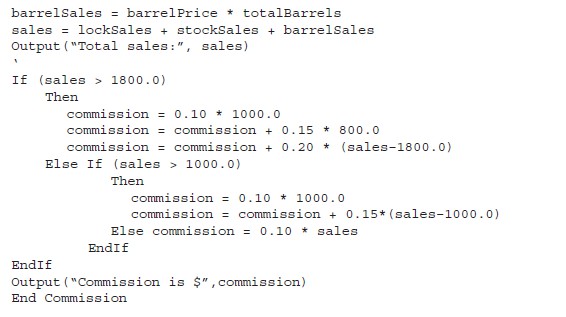
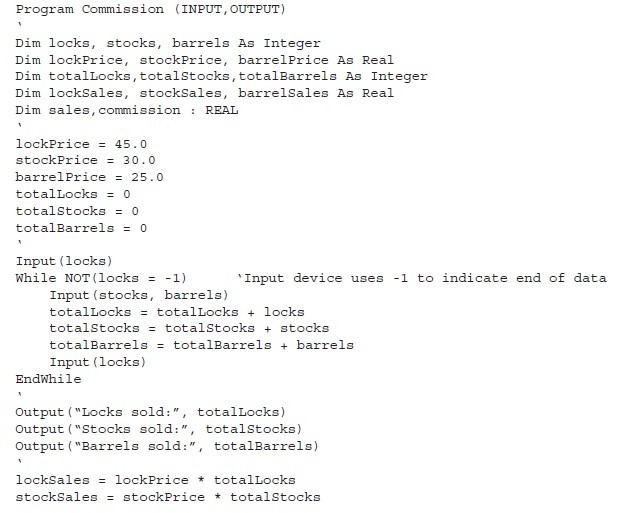
• The commission program produces a monthly sales report that gave total no. of locks, barrels, stocks sold. Sales persons total dollar sale & commission.

• This problem separates into 3 distinct pieces

o The input data portion( data validation) ignore here

o Sales calculation

o Commission calculation problem.



**The SATM Problem**

• To better discuss the issues of integration and system testing, we need an example with larger scope. The automated teller machine contains an interesting variety of functionality and interactions. Although it typifies real-time systems, practitioners in the commercial EDP domain are finding that even traditional COBOL systems have many of the problems usually associated with real-time systems.

• The SATM system communicates with bank customers via the 15 screens shown in Figure 1.

• Using a terminal with features as shown in Figure 2, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. For simplicity, these transactions can only be done on a checking account.

• When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer’s name and account information. If the customer’s PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer’s

PAN is not found, screen 4 is displayed, and the card is kept.

• At screen 2, the customer is prompted to enter his or her personal identification number (PIN).

If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept. On entry to screen

5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed.

• If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the

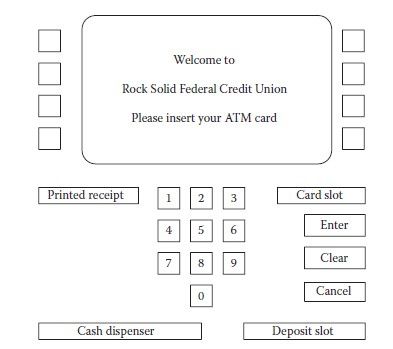
transaction amount. If a problem occurs with the deposit envelope slot, the system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The system then displays screen 14. If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed.

• The system checks the customer balance (as described in the balance request transaction); if the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14.

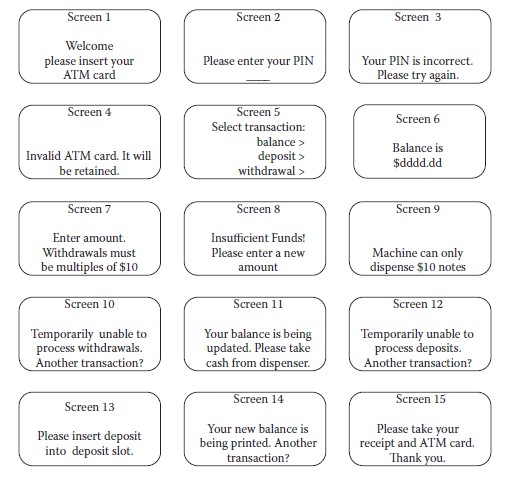
• When the “No” button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer’s ATM card. Once the card is removed from the card slot, screen 1 is

displayed. When the “Yes” button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

**SATM Terminal**



**SATM SCREENS**



**Currency Converter**

• Another event driven program that emphasizes code associated with a GUI. A sample GUI

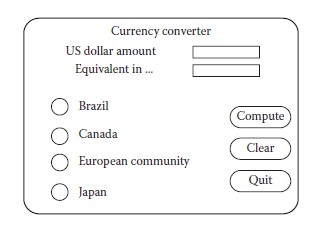
built with visual basic is shown

• The application converts US dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen.

• Currency selection is governed by the radio butons (option buttons), which are mutually exclusive. When a country is selected, the system responds by completing the label; for example, “Equivalent in …” becomes “Equivalent in Canadian dollars” if the Canada button is clicked. Also, a small Canadian flag appears next to the output position for the equivalent currency amount.

• Either before or after currency selection, the user inputs an amount in US dollars. Once both tasks are accomplished, the user can click on the Compute button, the Clear button, or the Quit button.

• Clicking on the Compute button results in the conversion of the US dollar amount to the equivalent amount in the selected currency. Clicking on the Clear button resets the currency selection, the US dollar amount, and the equivalent currency amount and the associated label. Clicking on the Quit button ends the application.



**Saturn Windshield Wiper Controller**

• The windshield wiper on the Saturn automobile (at least on the 1992 models) is controlled by alever with a dial. The lever has four positions, OFF, INT (for intermittent), LOW, and HIGH, and the dial has three positions, numbered simply 1, 2, and 3.

• The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in

wipes per minute) for the lever and dial positions.

