Software Testing

Module - 1 : Basics of software Testing

Basic definitions :

→ **Error**: Synonym of error is Mistake. Mistakes made while coding are called ~~errors~~ bugs. Errors tend to Propogate.
Ex:- Requirement error may be magnified during design & amplified more during coding.

→ **Fault**: Result of an error. Also considered as representation of an error. Synonym of fault is Defect.
Ex: When designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation.

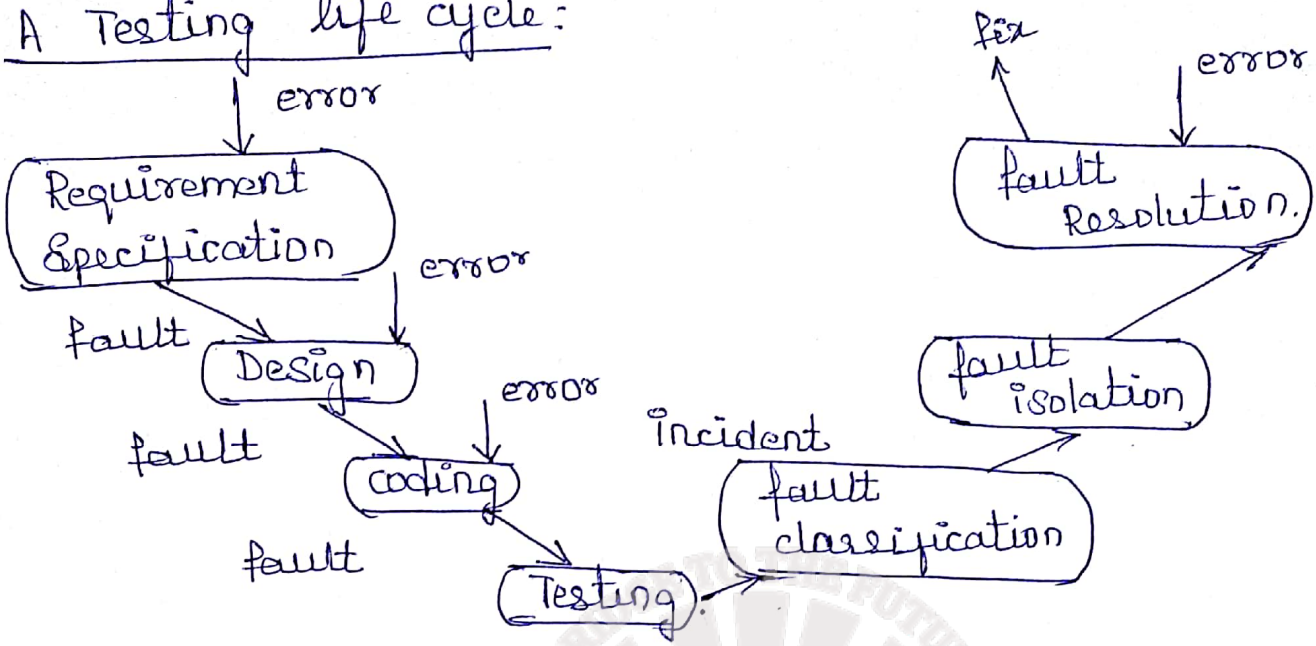
→ **failure**: failure occurs when a fault executes.
↳ failures only occur in an executable representation. which is usually taken to be source code.
↳ failure definition relates only to faults of commission.

→ **Incident**: Symptom associated with a failure that alerts the user to the occurence of a failure.

→ **Test**: Testing is concerned with errors, faults, failures and incidents. It is an act of exercising software with test cases.
2 distint goals: < find failures
demonstrate correct execution

→ **Testcase**: It has an identity and is associated with a program behaviour. It has a set of i/p and expected o/p

A Testing life cycle:



→ In the development phases, three opportunities arise for errors to be made, resulting in faults that propogate through the remainder of the development process.

→ Testing life cycle can be summarised as: The first 3 phases are putting Bugs In; the testing phase is finding Bugs; and the last 3 phases are getting Bugs out.

→ The fault Resolution step is another opportunity for errors.

→ Test cases occupy a central position in testing.

→ The process of testing can be subdivided into seperate steps:

- Test planning
- Test case development
- running test cases
- evaluating test results.

Software Quality:

The below attributes define quality of Software.

Quality attributes:

There exist several Measures of Software quality. These can be divided into static and dynamic quality attributes. * static quality attributes refer to the actual code and related documentation.

* Dynamic quality attributes relate to the behaviour of the application while in use.

→ Static quality attributes include structured, Maintainable, and testable code as well as the availability of correct and complete documentation

→ Dynamic quality attributes include Software reliability, correctness, completeness, consistency, usability and Performance.

* Reliability refers to the probability of failure-free operation. ~~and is considered~~ in

* correctness refers to the correct operation of an application and is always with reference to some artifact.

* Completeness refers to the availability of all the features listed in the requirements or in the user Manual. An incomplete Software is one that doesnot fully implement all features required.

* consistency refers to adherence to a common set of conventions and assumptions.

Ex: All buttons in the UI Might follow a common color-coding convention.

* Usability refers to the ease with which an application can be used.
* Performance refers to the time the application takes to perform a requested task.

Reliability:

ANSI/IEEE STD 729-1983: Reliability

→ Software reliability is the probability of failure free operation of software over a given time interval and under given conditions:

The Probability referred to in this definition depends on the distribution of the inputs to the program. Such input distribution is often referred to as the operational Profile. According to this definition, software reliability could vary from one operational Profile to another.

An implication is that one user might say "this Program is lousy" while another might sing praises for the same program. The following is an alternate definition of reliability.

→ Software reliability is the Probability of failure free operation of software in its intended environment.

This definition is independent of "Who uses what features of the software and how often". Instead it depends exclusively on the correctness of its features. As there is no notion of operational profile, the entire input domain is considered as uniformly distributed. The term environment refers to the software and hardware elements needed to execute the application. These elements include the OS,

hardware requirements, and any other application needed for communication.

Requirements, Behavior and correctness:

→ Products, software in particular, are designed to respond to requirements. Requirements specify the functions that a product is expected to perform. Once the product is ready, it is the requirements that determine the expected behavior.

Example 1.3 Two requirements are given below, each of which leads to a different program.

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.

→ Suppose that program max is developed to satisfy req1 above. 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 2 integers are to be input to the program on one line followed by a carriage return, or on 2 separate lines with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question. This example illustrates the incompleteness Requirement 1

→ The second Requirement in the above example is ambiguous. It is not clear from this requirement whether the input sequence is to be sorted in ascending or descending order.

The behaviour of Sort program, written to satisfy this requirement will depend on the decision taken by the Programmer while writing sort.

→ Regardless of the nature of the requirements, testing requires the determination of the expected behaviour of the Program under test. The observed behaviour of the Program is compared with the expected behaviour to determine if the Program function as desired.

Input Domain and Program correctness:

A Program is considered correct if it behaves as desired on all possible test inputs. Usually, the set of all possible inputs is too large for the Program to be executed on each input.

→ Testing a Program on all possible input is known as exhaustive testing.

Input Domain:

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

Example 1.4: using requirement 1 from example 1.3, 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 from -32,768 to 32,767

Example 1.5: using requirement 2 it is not possible to find the input domain for the sort prog -ram.

Let us therefore assume that the requirement was Modified to be the following:

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 entered 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.

Ex:- < A - 3 15 12 55 . >
< D 23 78 . >
< A . >

Correctness: A program is considered correct if it behaves as expected on each element of its input domain.

Valid and Invalid Inputs:

→ The input domains are derived from the requirements. However due to the incompleteness of requirements one might think harder to determine the input domain.

→ To illustrate why, consider the Modified Requirement 2 [example 1.5]. The requirement Mentions that the request characters can be "A" or "D", but it fails to answer the question "What if the user types a different characters?". When using Sort, it is certainly possible for the user to type a character other the "A" or "D". Any characters other than "A" or "D" is considered as an invalid input to Sort. The requirement for sort doesnot specify what action it should take when an invalid input is encountered.

→ Identifying the set of invalid inputs and testing the program against these inputs are import -ant parts of the testing activity. Even when the requirements fail to specify the program behaviour on invalid inputs, the programmer does treat these in one way or another.

→ Testing a program against invalid inputs might reveal errors in the program.

Correctness Versus Reliability:

Correctness:

Correctness attempts to establish that the Program is error-free, testing attempts to find if there are any errors in it.

As testing progresses, error Might be revealed. Removal of errors from the Program usually improves the chances, or the Probability of the Program executing without any failure.

Example: This example illustrates why the Probability of Program failure Might not change upon error removal. Consider the following program that inputs 2 integers x and y and Prints the value of $f(x, y)$ or $g(x,y)$ depending on the condition $x < y$.

```
integer x, y
input x, y
if (x <y) ← This Condition should be x ≤ y.
  { Print f(x,y) }
else
  { Print g(x,y) }
```

→ The above Program uses 2 functions f and g, not defined here. Let us suppose that function f Produces incorrect result Whenever it is invoked with $x=y$ and that $f(x,y) \neq g(x,y)$, $x=y$.

→ In its Present form the Program fails When tested with equal input values because function g is invoked instead of function f. When the error is removed by changing the condition $x<y$ to $x \leq y$, the Program fails again When the input values are the Same.

→ The latter failure is due to the error in function f. In this Program, when the error in f is also removed, the Program will be correct assuming that all other code is correct.

Reliability:

The reliability of a Program P is the Probability of its Successful execution on a randomly Selected element from its input domain.

→ A comparison of Program correctness and reliability reveals that while correctness is a binary metric, reliability is a continuous metric over a scale from 0 to 1.

→ A program can be either correct or incorrect; its reliability can be anywhere between 0 & 1. Intuitively, when an error is removed from a Program, the reliability of the Program is obtained is expected to be higher than that of the one that contains the error.

→ The below example illustrates how to compute Program reliability in a simplistic manner.

Consider a Program P which takes a pair of integers as input. The input domain of this Program is the set of all pairs of integers. Suppose now that in actual use these are only three pairs that will be input to P. These are as follows:

$$\{<(0,0)\ (-1,1)\ (1,-1)>\}$$

The above set of three pairs is a subset of the input domain of P and is derived from a

knowledge of the actual use of P, and not solely from its requirements.

Suppose also that each pair in the above set is equally likely to occur in Practice. If it is known that P fails on exactly one of the three possible input pairs then the frequency with which P will function correctly is 2/3. This number is an estimate of the Probability of the Successful operation of P and hence is the reliability of P.

Program use and the Operational Profile:

Operational Profile: An operational profile is a numerical description of how a program is used, in accordance with the above definition, a Program might have several operational Profiles depending on its users.

Example: Consider a sort program which, on any given execution, allows any one of 2 types of input sequences. One sequence consists of numbers only and the other consists of alphanumeric strings. One operational Profile for sort is Specified as follows:

Operational Profile 1

Sequence	Probability
Numbers only	0.9
Alphanumeric strings	0.1

} used Mostly for Sorting sequences of numbers

Another operational profile for sort is Specified as follows:

Operational Profile 2

Sequence	Probability
Numbers only	0.1
Alphanumeric strings	0.9

} used Mostly for sorting alphanumeric strings.

Testing and Debugging:

Testing is the Process of determining if a Program behaves as expected. In the Process one may discover errors in the Program under test.

When Testing reveals an error, the Process used to determine the cause of this error & to remove it is known as debugging.

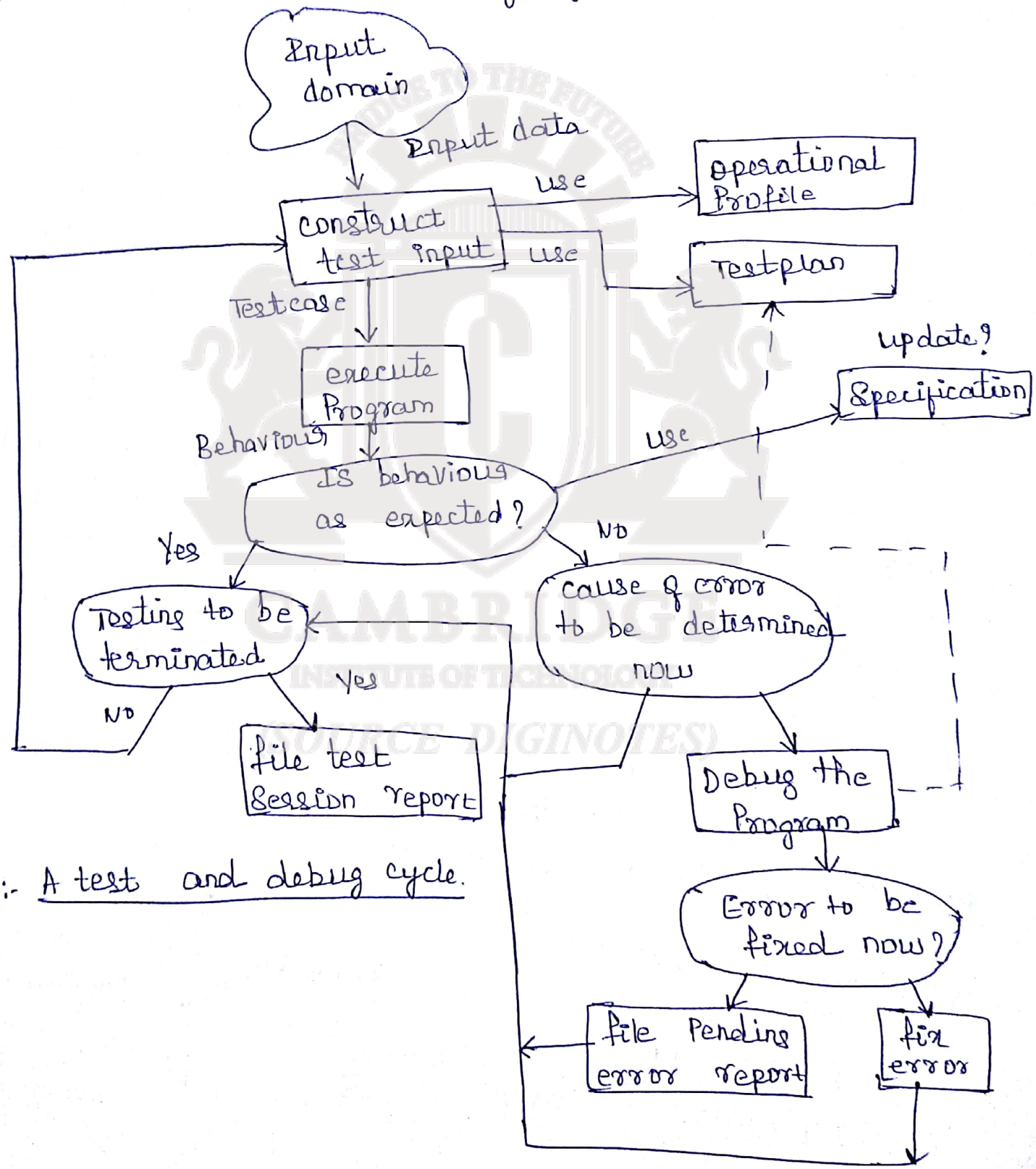


fig:- A test and debug cycle.

1. Preparing a test plan:

A test cycle is often guided by a test plan. When relatively small Programs are being tested, a test plan is usually informal and in the testers mind, or there may be no plan at all. A sample test plan for testing the Sort Program is shown in Prev fig:

2. Constructing test data:

→ A test case is a pair consisting of test data to be input to the Program and the expected o/p. 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.

→ Program requirements and the test plan help in the construction of test data. Execution of the program on test data might begin after all or a few test cases have been constructed.

3. Executing the Program:

Execution of a program under test is the next significant step in testing. Execution of this step for the Sort Program is most likely a trivial exercise.

→ The complexity of actual Program execution is dependent on the Program itself.

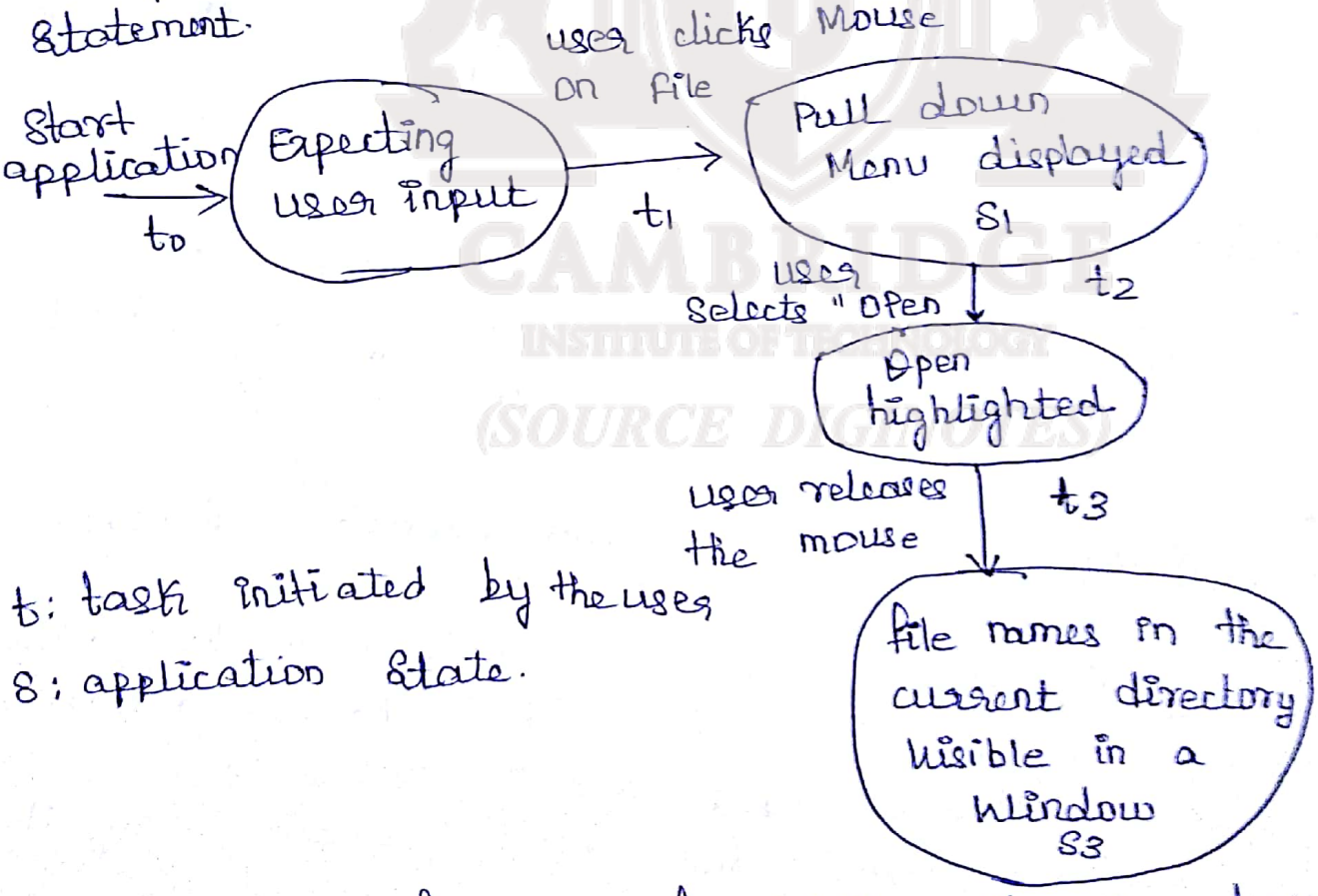
Often a tester might be able to construct a test harness to aid in Program execution. The harness initializes any global variables, inputs a test case, and execute the Program. The o/p generated by the Program may be saved in a file for subsequent examination by a tester.

4. Specifying Program Behaviour:

→ The simplest way is to specify the behaviour in a natural language such as English. ~~However, this is more~~ State diagram can be used to specify program behaviour.

→ The state of a program is the set of current values of all its variables and an indication of which statement in the program is to be executed next. One way to encode the state is by collecting the current values of program variables into a vector known as the state vector.

→ An indication of where the control of execution is at any instant of time can be given by using an identifies associated with the next program statement.



t: task initiated by the user

S: application state.

Fig:- A state sequence for myapp showing how the application is expected to behave when the user selects the Open option under the file

5. Assessing the correctness of Program Behaviour:

→ An important step in testing a program is the one wherein the tester determines if the observed behaviour of the program under test is correct or not.

This step further divided into 2 smaller steps
- One observes the behaviour
- Analyses the observed behaviour to check if it is correct or not.

→ The entity that performs the task of checking the correctness of the observed behaviour is known as an Oracle.

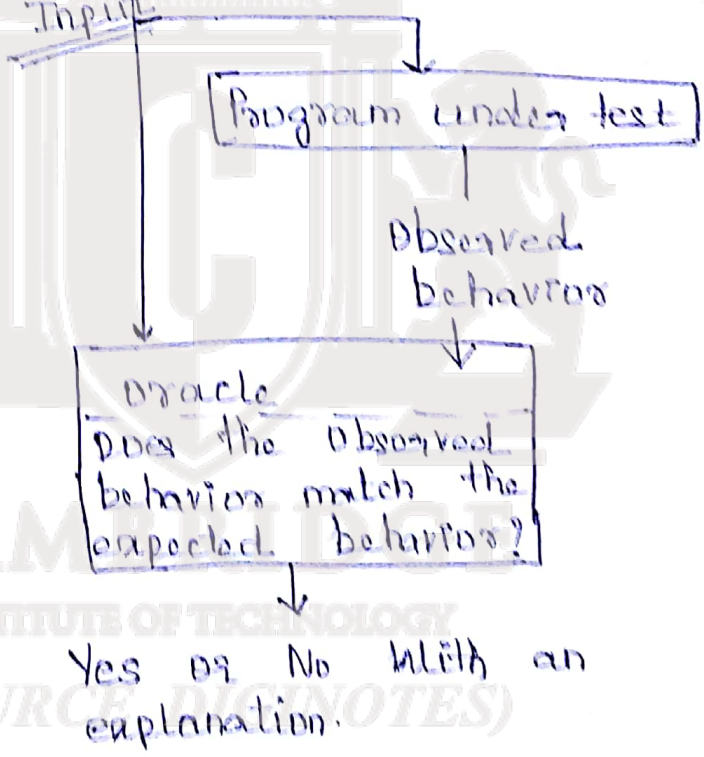


Fig: Relationship b/w the Program under test & the Oracle.

→ A tester often assumes the role of an Oracle & thus serves as a human Oracle.

→ checking Program behaviour by humans has several disadv:
1) It is error prone as the human Oracle might make error in analysis.

2) It may be slower than the speed with which the program computed the results.

3) it might result in the checking of only trivial I/o behaviours.

→ using programs as oracles has the advantage of speed, accuracy and the ease with which complex computations can be checked.

6. Construction of oracles:

construction of automated oracles, such as the one to check a matrix Multiplication program or a sort program, requires the determination of I/o relationship.

Test cases:

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

→ Inputs (2 types)
- → Preconditions (circumstances that hold prior to test case execution)
- → Actual Inputs (identified by testing Method)

→ Outputs (2 types)
- → Post conditions
- → Actual outputs.

→ The Act of testing entails establishing the necessary Pre conditions, Providing the testcase 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.

→ The remaining information in a well developed testcase: Primarily supports testing management. Testcases should have an identity and a reason for being. It is also useful to record the execution history of a testcase, including when and by whom it was run, the Pass/fail result of each execution, and the version on which it was run.

Test case ID
Purpose
Preconditions
Inputs
Expected outputs
post conditions
Execution History
Date Result Version RunBy

Fig:- typical test case Information.

→ Test cases need to be developed, reviewed, used, Managed and saved.

Insights from a Venn Diagram:

→ Testing is fundamentally concerned with behavior & behavior is orthogonal to the structural view common to software developers.

A quick differentiation is that the structural view focuses on what it is and the behavioral view considers What it does.

→ A simple Venn diagram clarifies several questions about testing.
Consider a universe of Program behaviors. Given a Prog and its Specification, consider the Set S of Specified behaviors and Set P of Programmed behaviors.

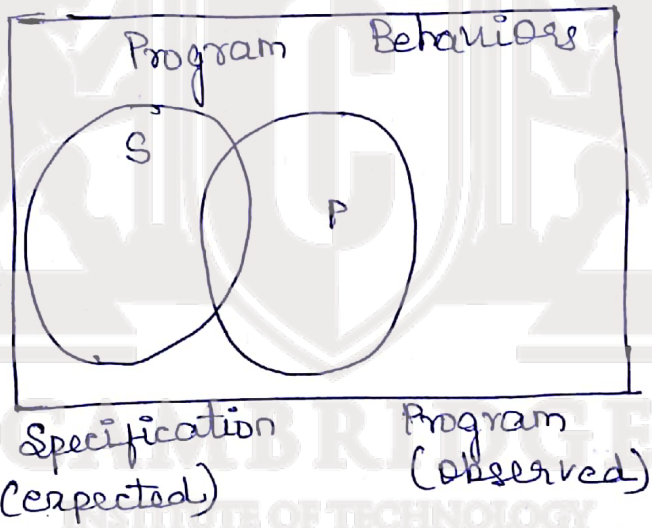


Fig:- Specified and implemented Program behaviors.

→ The above figure shows the relationship among our universe of discourse as well as the Specified Programmed behaviors. Of all the possible Programmed behaviors, the Specified ones are in the circle labled S, and all those behaviours actually Programmed are in P.

→ With this diagram, We can see more clearly the Problems that confront a tester. What if certain Specified behaviors have not been Programmed?

→ Similarly, What if certain Programmed behaviours 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 is the "correct" Portion, that is, behaviors that are both Specified and implemented.

→ The new circle in the below figure is for testcases.

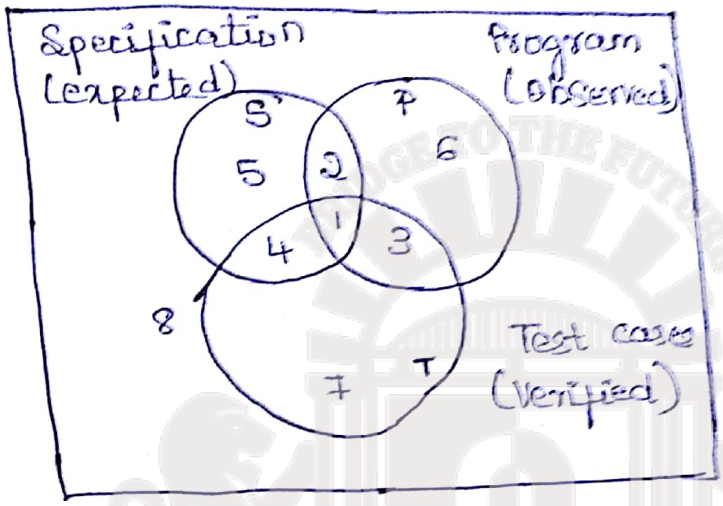


fig 1.4 : Specified, implemented, and tested behaviors.

→ Now, consider the relationships among the sets S, P and T. There may be Specified behaviors that are not tested (regions 2 & 5). Specified behaviors that are tested (regions 1 & 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).

Identifying Test cases:

Two fundamental approaches are used to identify test cases known as functional and structural testing.

Functional 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.

→ Systems considered as blackbox and hence the name blackbox testing.

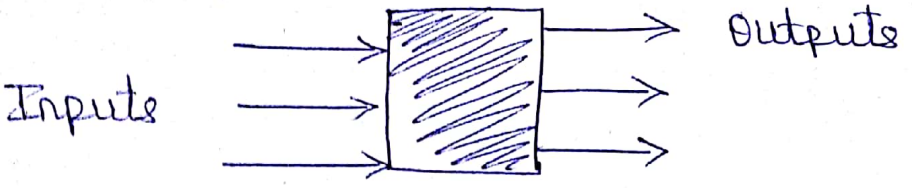


fig:- An engineer's blackbox.

→ With the functional approach to testcase identification, the only information used is the specification of the software.

Advantages:

↳ They are independent of how the software is implemented. So if the implementation changes, the testcases are still useful.

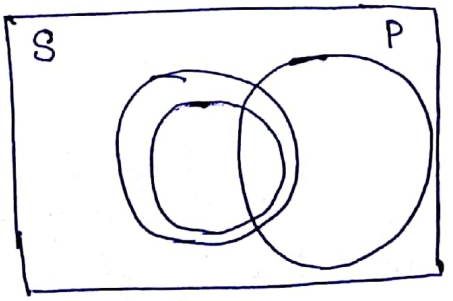
↳ Test case development can occur in Parallel with the implementation, thereby reducing the overall Project development interval.

Disadvantages:

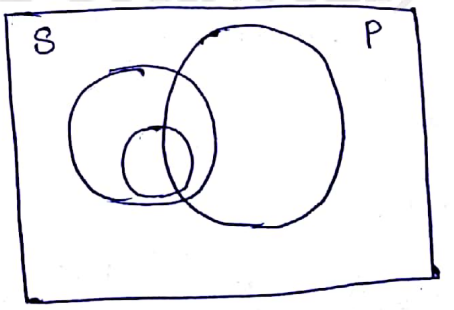
↳ Significant redundancies may exist among test cases

↳ Compounded by the possibility of gaps of untested software.

→ The foll figure shows the results of testcases identified by 2 functional methods.



Test case
(Method A)



Testcase
(Method B)

→ Method A identifies larger set of test cases that does method B. The set of cases for both methods is completly contained within the set of Specified Behavior.

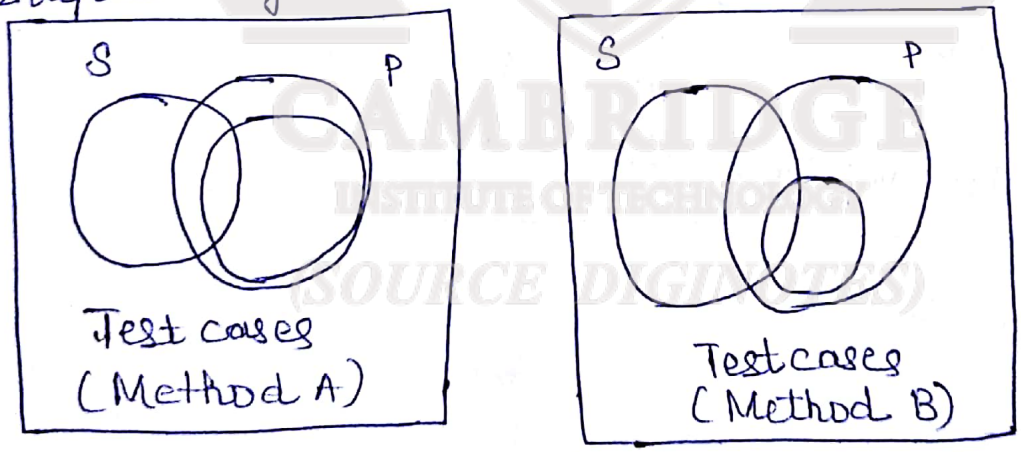
Because functional methods are based on the Specified Behavior, it is hard to imagine these methods identifying behaviors that are not Specified.

Structural Testing:

→ Sometimes called White Box Testing. Difference between black box is the ~~essential difference~~ implementation is known and is used to identify test cases. The ability to "See inside" the blackbox allows the tester to identify testcases based on how the function is actually implemented.

→ The tester can rigourously describe exactly what is tested. Test coverage metrics provide a way to explicitly state the extent to which a Software item has been tested.

→ The foll figure shows the result of test cases identified by 2 structural methods.



Eg:- For both methods, the Set of testcases is completely contained within the Set of Programmed behavior. Because Structural methods are based on the Program it is hard to imagine these methods identifying behaviours that are not Programmed.

The functional Versus Structural Debate:

→ Both approaches is to identify testcases. functional testing uses only the specification to identify testcases, while structural testing uses the program source code as the basis of testcase identification

→ functional testing often suffers from twin problems of redundancies and gaps. When functional testcases are executed in combination with structural test coverage metrics, both of these problems can be recognized and resolved.

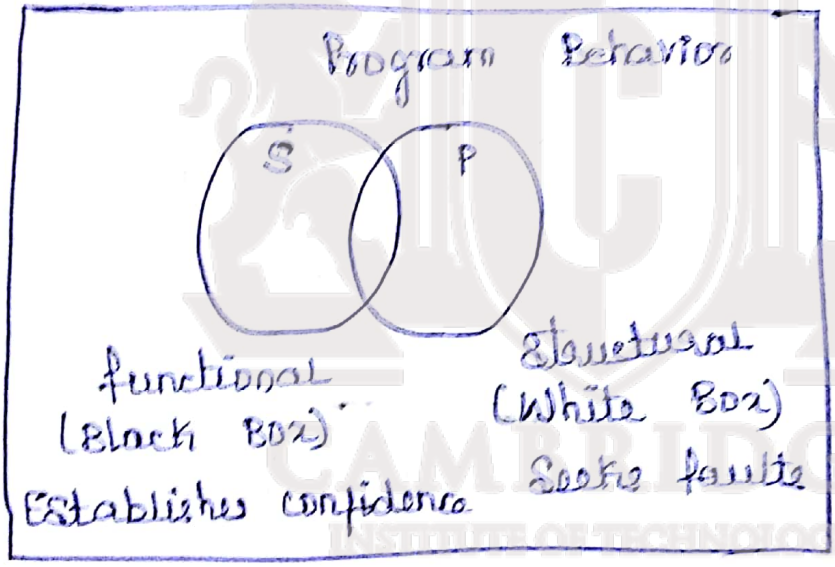


Fig:- Sources of testcases.

Errors and fault Taxonomies:

→ Process: refers to how we do something.
Product: end results of a Process.
SE&A (Software quality assurance) tries to improve the Product by improving the Process.

→ SQA is more concerned with ~~discovering~~ reducing errors endemic in the development Process, while testing is more concerned with discovering faults in a Product.

Test Generation Strategies

→ Any form of test generation uses a source document. In the most informal of testmethods, the source document resides in the mind of the tester who generates tests based on a knowledge of the requirements. In some organizations, tests are generated using a mix of formal and informal methods often directly from the requirements document serving as the source.

→ The following figure summarizes several strategies for test generation.

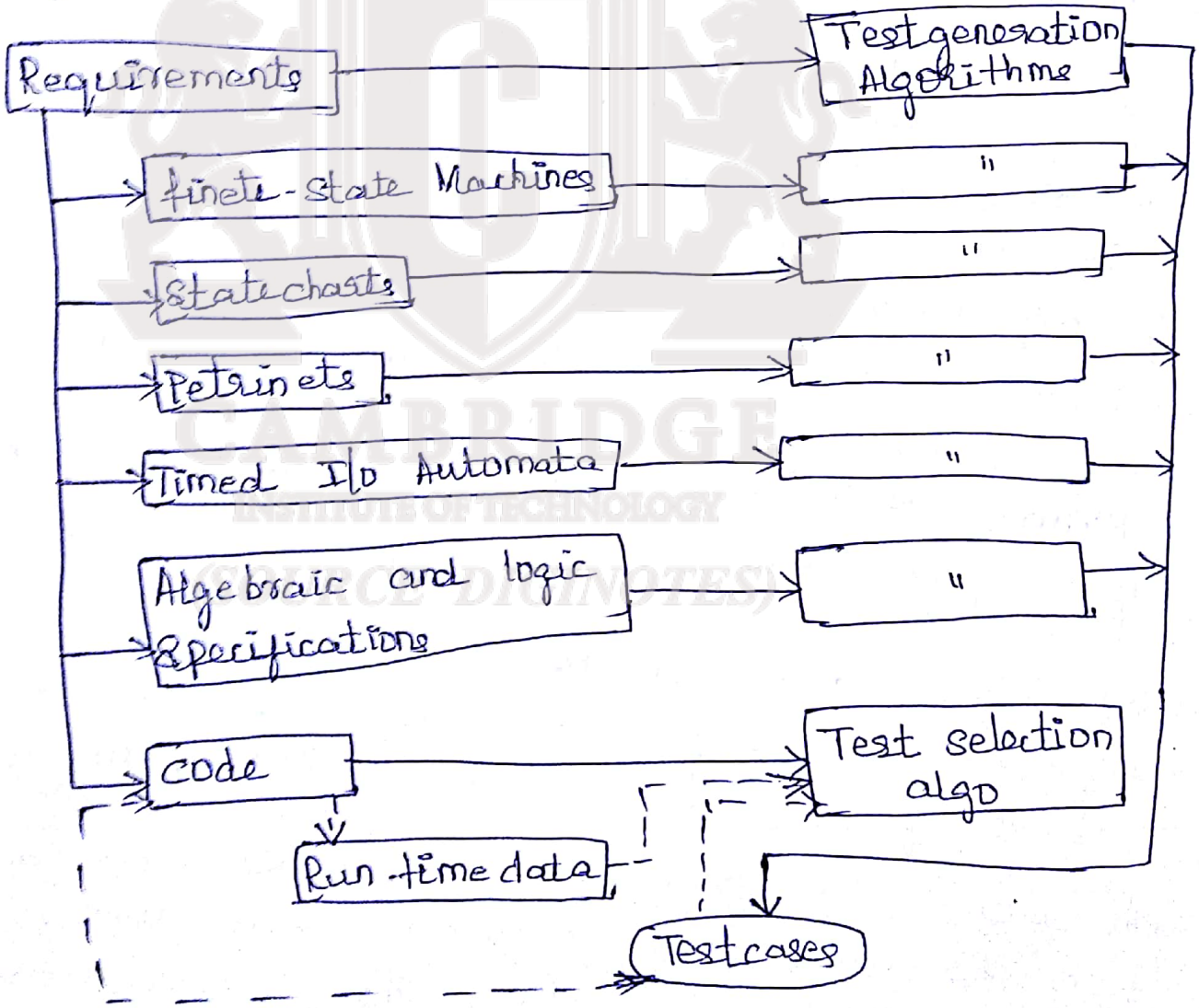


Fig:- Requirements, Models & test generation algorithms.

→ The top row in the figure captures techniques that are applied directly to the requirements. These may be informal techniques that assign values to input variables without the use of any rigorous or formal Methods.

→ These could also be techniques that identify input variables, capture the relationship among these variables & use formal techniques for test generation such as random test generation & cause-effect graphing.

→ Another set of strategies falls under the category of model-based test generation. These strategies require that a subset of the requirements be modeled using a formal notation. Such a model is also known as a specification of the subset of requirements.

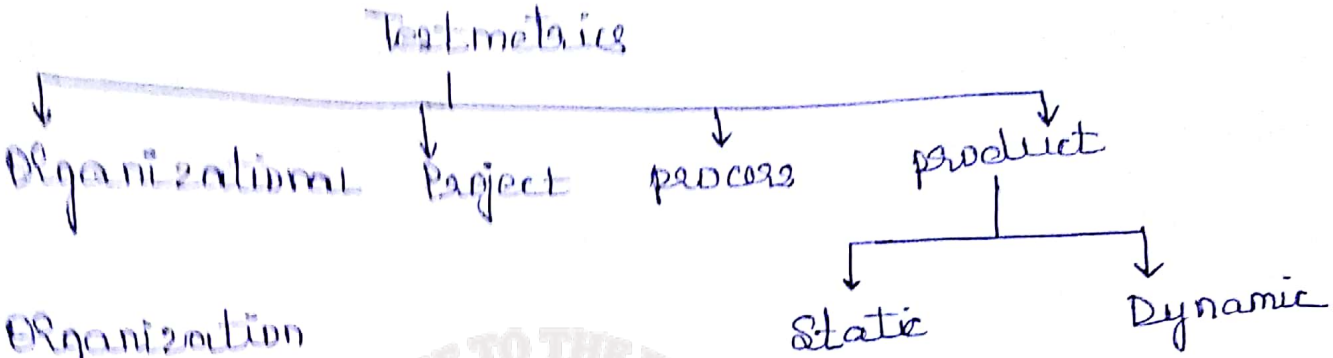
→ FSMs, statecharts, Petrinets, and Timed I/O automata are some of the well-known and used formal notations for modelling various subsets of requirements.

→ Languages based on Predicate logic as well as algebraic languages are also used to express subsets of requirements in a formal manner.

→ There also exist techniques to generate tests directly from the code. Such techniques fall under code-based test generation. These techniques are useful when enhancing existing tests based on test adequacy criteria.

Test Metrics:

→ Metric refers to a standard of Measurement.



Organization
↓
establishes test processes
↓
used in projects
↓
To test products.

fig: Types of metrics used in software testing and their relationships.

1) Organizational Metrics:

Metrics at the level of an organization are useful in overall project planning & Management. Some of these metrics are obtained by aggregating compatible metrics across Multiple Projects.

→ Computing this metric at regular intervals and over all products released over a given duration shows the quality trend across the organization.

→ Organization-level metrics allow Senior Mgmt to monitor the overall strength of the organization & points to areas of weakness. Thus, these metrics help Senior mgmt in setting new goals & plan for resources needed to realize these goals.

2) Project Metrics:

→ Project Metrics relate to Specific Project. These are useful in monitoring & control of a specific project. The ratio of actual-to-planned system test effort is one project metric. Test effort could be measured in terms of tester-man-months.

→ Another project metric is the ratio of the number of successful tests to the total number of tests in the system test phase.

3) Process Metrics:

→ The goal of a process Metric is to assess the goodness of the Process.

→ When a test process consists of several phases, for example unit test, integration test and system test, one can measure how many defects were found in each phase. It is well known that the later a defect is found, the costlier it is to fix.

4) Product Metrics: Generic

Product metrics relate to a specific product such as a compiler for a programming language. These are useful in making decision related to the Product.

→ Product complexity related metrics abound. 2 metrics are introduced → The cyclomatic complexity
→ Halstead Metrics.

→ The cyclomatic complexity proposed by Thomas McCabe in 1976 is based on the control flow of a Program. Given the CFG G_1 of program P containing N nodes, E edges and P connected Procedures, the cyclomatic complexity $V(G_1)$ is computed as follows:

$$V(G_1) = E - N + 2P.$$

The term P in $V(G_1)$ counts only procedures that are reachable from the main function. $V(G_1)$ is the complexity of a CFG G_1 that corresponds to a Procedure reachable from the main Procedure.

Also $V(G_1)$ is not the complexity of the entire program, instead it is the complexity of a procedure in P that corresponds to G_1. Larger values of $V(G_1)$ tend to imply higher program complexity and hence a prog more difficult to understand and test than one with a smaller values. $V(G_1)$ of the values 5 or less are Recommended.

→ Halstead complexity Measures - using Program size (S) and effort (E), the following estimator has been Proposed for the number of errors (B) found during a sw development effort:

$$B = 7.6\, E^{0.667}\, S^{0.33}.$$

→ Halstead Measures of program complexity & effort.

Measure	Notation	Definition.
Operator count	N_1	Num of Operators in prog
operand count	N_2	" " Operand " "
unique Operators	η_1	" " unique Operators "

→ An advantage: Allows Management to plan for testing resources.

5. Product Metrics : OO software:

A number of empirical studies have investigated the correlation between Product complexity & quality. The below table lists a sample of product metrics for OO. Product reliability is a quality metric & refers to the probability of product failure for a given Operational profile.

Metric	Meaning
Reliability	- Probability of failure of a SW product with respect to a given OP in a given env.
Defect density	- Num of defects per KLOC
Defect Severity	- Distribution of defects by their level of Severity.

6. Progress Monitoring & Trends:

Metrics are often used for Monitoring Progress. This requires making measurements on a regular basis over time. Such measurements offer trends.

For ex: Suppose that a browser has been coded, unit tested and its components integrated. It is now in the system-testing phase.

→ One could Measure the cumulative Number of defects found and plot these over time. Eventually, such a plot will rise over time. Eventually, it is likely to show a saturation indicating that product is Reaching Stability.

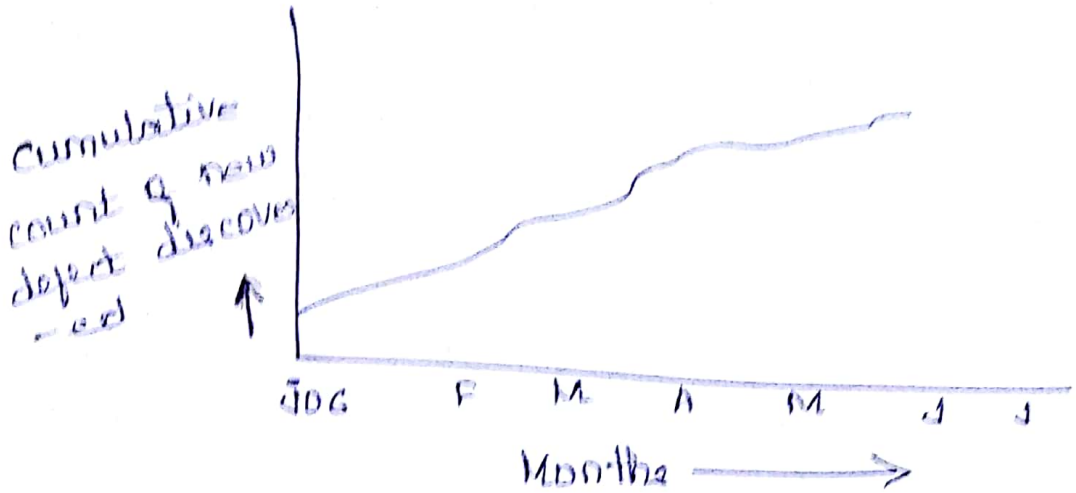


fig: A sample plot of cumulative count of defects found over seven consecutive months in a software project.

7. Static and Dynamic Metrics:

Static Metrics are those computed without having to execute the product. Number of testable entities in an application is an example of a static product metric.

Dynamic Metrics require code execution. For example, the number of testable entities actually covered by a test suite is a dynamic product metric.

8. Testability:

According to IEEE, testability is 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.

→ Different ways to measure testability of a product can be categorized into static and dynamic testability metrics.

→ Software complexity is one static testability metric. The more complex an application, the lower the testability, that is, the higher the effort required to test it. Dynamic metrics for testability include various code-based coverage criteria.

→ High testability is a desirable goal. This is best done by knowing well what needs to be tested and how well in advance.

Error and fault taxonomies (contd):

→ faults can be classified in several ways: the development phase in which the corresponding error occured, the consequences of corresponding failures, difficulty to resolve, risk of no resolution & so on.

→ The foll table contains a fault taxonomy that distinguishes faults by the severity of their consequences.

1. Mild - Misspelled Word
2. Moderate - Misleading or redundant info
3. Annoying - Truncated names, bill for $0.00
4. Disturbing - Some transactions not processed.
5. Serious - lose a transaction.
6. Very serious - Incorrect transaction execution
7. extreme - frequent "very serious" errors
8. Intolerable - Database corruption.

9. Catastrophic System Shutdown.
10. Infectious Shutdown that spreads to others

→ The IEEE standard defines a detailed anomaly resolution process built around 4 phases recognition, investigation, action and disposition. Some of the useful anomalies are given in table below.

Table : Input/output faults :

Type	Instances
Input	Correct input not accepted.
	Incorrect input accepted.
	Description wrong or missing
	Parameters wrong or Missing
Output	Wrong format
	Wrong result
	correct result at wrong time.

Table : logic faults :

Missing case (s)
Duplicate case (s)
Misinterpretation
Missing condition

Table : Interface faults

Incorrect interrupt handling
I/o timing
Incompatible types
parameters Mismatch

Table : Computation faults :

Incorrect algorithm
Missing computation
Incorrect Operand
Incorrect Operator

Table : Data faults :

Incorrect initialization.
Incorrect Storage/access
Scaling or units error
Incorrect type.

Levels of testing:

→ A diagrammatic variation of the waterfall Model is given below. This variation emphazies the correspondance between testing and design levels

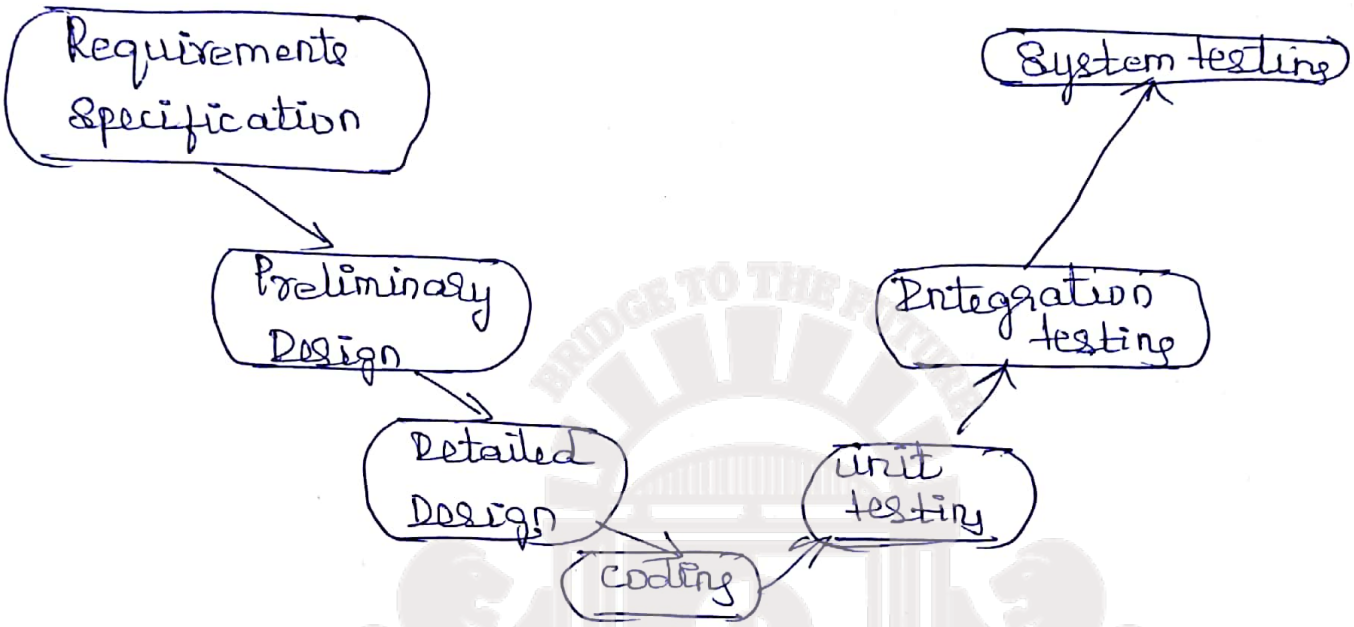


fig:- levels of abstraction & testing in the waterfall Model.

→ In terms of functional testing the 3 levels of definition (Specification, Preliminary design & detailed design) correspond directly to 3 levels of testing System, Integration & unit testing.

→ A Practical relationship exists blw levels of testing Vs functional & Structural testing.

Structural testing is most appropriate at the unit level, while functional testing is most appropriate at the System level.

Testing and Verification

Testing :- Aims at uncovering errors in a Program

Verification :- Aims at showing that a given Program work for all possible inputs that satisfy a set of conditions. Testing aims to show that the given Program is reliable in that no errors of any significance were found.

→ Testing is not a perfect process in that a Prog might contain errors despite the success of a set of tests. However, it is a process with direct impact on our confidence in the correctness of an application increases when an application passes a set of thoroughly designed & executed tests.

→ Verification might appear to be a perfect process as it promises to verify that a program is free from errors. However a close look at verification reveals that it has its own weakness. The person who verified a program might have made mistakes in verification process. there might be incorrect assumption on the i/p cond^n; incorrect assumptions might be made regarding the components that interface with the program & so on.

→ Thus, neither verification nor testing is a perfect technique for proving the correctness of programs

Static Testing:-

→ Carried out without executing the appln under test.

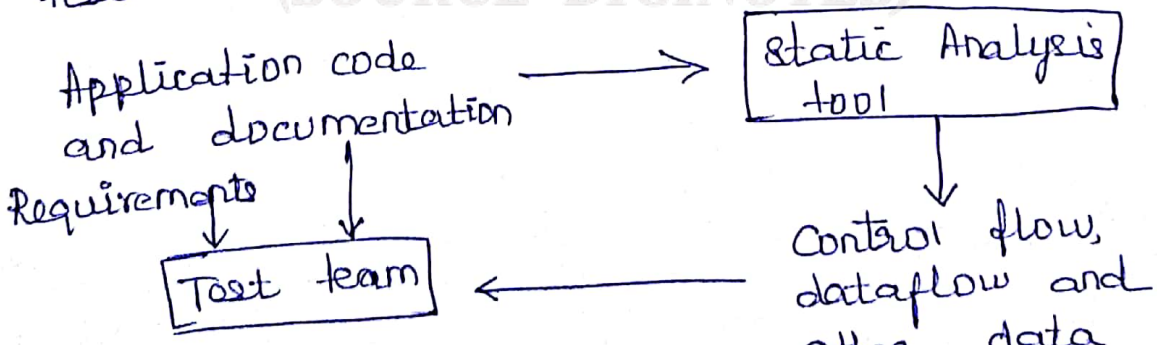
→ Dynamic testing: requires one or more executions of the application under test.

→ Static testing is useful in that it may lead to the discovery of faults in the appln, as well as ambiguities & errors in requirements & other appln-related documents, at a relatively low cost.

→ dynamic testing is expensive.

→ A Sample Process of Static testing is illustrated in figure below. The test team responsible for Static testing has access to requirements doc, appln and all associated documents such as design document & user manuals. The team also has access to one or more static testing tools.

→ A static testing tool takes the application code as input & generates a variety of data useful in the test Process.



Walkthroughs:

→ Walkthroughs and inspections are an integral part of static testing. Walkthrough is an informal process to review any application-related document. For example, requirements are reviewed using a process termed requirements Walkthrough. Code is reviewed using code Walkthrough, also known as peer code review.

→ A Walkthrough begins with a review plan agreed upon by all members of the team. Each item of the document, for ex a source code module is reviewed with clearly stated objectives in view. A detailed report is generated that lists items of concern regarding the document reviewed.

Inspections:

→ Inspections is more formally defined process than a Walkthrough. This term is usually associated with the code.

→ Code inspection is carried out by a team. The team works according to an inspection plan that consists of the foll elements:

(a) Statement of purpose.
(b) Work product to be inspected. [code & documents]
(c) Team formation, roles & tasks to be performed.
(d) rate at which the inspection task is to be completed
(e) data collection forms, where the team will record its finding such as defects discovered, coding standard violations and time spent in each task.

→ Members of the inspection team are assigned roles of Moderator, reader, recorder and author. The Moderator is in charge of the process and leads the review. Actual code is read by the reader, perhaps with the help of a code browser and with large monitors for all in the team to view the code. The recorder records any errors discovered or issues to be looked into. The author is the actual developer whose primary task is to help others understand code.

→ Use of static code Analysis tools in Static Testing:

Static code Analysis tools can provide control-flow and dataflow information. The control-flow info, presented in terms of a CFG, is helpful to the inspection team in that it allows the determination of the flow of control under different cond?. A CFG can be annotated with dataflow info to make a data flow graph.

↳ Several such commerical as well as open source tools are available. Purify from IBM Rational & Klocwork from Klocwork, Inc. are 2 of the many commercially available tools for Static analysis of C and Java Programs. Lightweight analysis for program security in eclipse (LAPSE) is an open source tool for the Analysis of Java programs.

Software Complexity and Static Testing

→ A more complex module is likely to have more errors & must be accorded higher priority during inspection than a lower Priority Module.

→ Static Analysis tool often compute complexity metrics using one or more complexity metrics

Generalized Pseudocode: Pseudocode Provides a "language neutral" way to express Program source code.

Ex:-

language element	Generalized Pseudocode construct.
Comment	<text>
Datastructure declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <Variable> As <type>
Assignment Statement	<Variable> = <expression>
Input	Input (<Variable list>)

The Triangle Problem:

Problem Statement:

Simple Version: Input: Three integers A, b & c. (Taken as sides of a triangle).

O/P: Type of triangle (equilateral, isosceles, Scalene or not a triangle).

Improved Version: Input: Three integers a, b, c. Three integers must satisfy following conditions:

c1: $1 \leq a \leq 200$

c2: $1 \leq b \leq 200$

c3: $1 \leq c \leq 200$

c4. $a < b+c$

c5. $b < a+c$

c6. $c < a+b$.

Output: Type of triangle

1. If all three sides are equal, the Program o/p is equilateral
2. If exactly one pair of sides is equal, the Program o/p is isosceles.
3. If no pair of sides is equal the Program o/p is Scalene.
4. If any of conditions c4, c5 & c6 is not met, the Program o/p is not a triangle.

Improved version: The triangle program accepts three integers, a, b, and c as input. These are taken to be sides of triangle. The integers a, b , and c must satisfy the following conditions:

C1. $1 \leq a \leq 200$ C4. $a < b + c$
C2: $1 \leq b \leq 200$ C5. $b < a + c$
C3: $1 \leq c \leq 200$ C6. $c < a + b$

The output of the program is the type of triangle determined by the three sides: Equilateral, Scalene, Isosceles or not a triangle. If values of a, b, and c satisfy conditions C1, C2, and C3, one of four outputs is given:

1.If all three sides are equal, they constitute an equilateral triangle.
2.If exactly one pair of sides is equal, they form an isosceles triangle.
3.If no pair of sides is equal, they constitute a scalene triangle.
4.If any of conditions C4, C5, and C6 is not met, output is not a triangle.

Discussion

Perhaps one of the reasons for the longevity of this example is that, among other things, it typifies some of the incomplete definition that impair communication among customers, developers, and testers.

Traditional Implementation

The "traditional" implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. The flowchart box numbers correspond to comment numbers in the (FORTRAN-like) pseudocode program given next.

```
Program triangle1 'Fortran-like version
'
Dim a,b,c,match As INTEGER
'
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
match = 0
If a = b
   Then match = match + 1
EndIf

If a = c
   Then match = match + 2
EndIf
If b = c
   Then match = match + 3
EndIf
If match = 0
   Then   If (a+b)<=c
      Then     Output ("NotATriangle")
      Else   If (b+c)<=a
         Then   Output ("NotATriangle")
```

```
        Else        If  (a+c)<=b
                Then        Output  ("NotATriangle")
                Else        Output  ("Scalene")
                        EndIf
                EndIf
            EndIf
Else  If match=1
    Then        If  (a+c)  <=b
            Then        Output  ("NotATriangle")
            Else        Output  ("Isosceles")
        EndIf
    Else        If match=2
                Then   If  (a+c)<=b
                            Then   Output  ("NotATriangle")
                            Else   Output  ("Isosceles")
                       EndIf
                Else   If match=3
                            Then   If  (b+c)<=a
                                        Then   Output  ("NotATriangle")
                                        Else   Output  ("Isosceles")
                                   EndIf
                            Else   Output  ("Equilateral")
                       EndIf
                EndIf
        EndIf
EndIf
'
End Triangle1
```

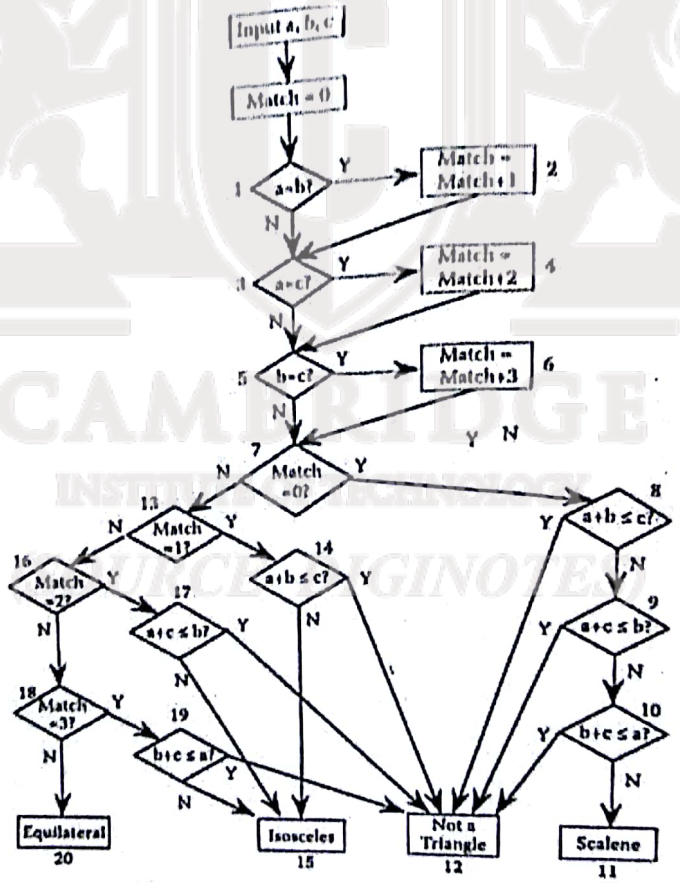


Figure 2.1 Flowchart for the traditional triangle program implementation.

Structured Implementation

The Below figure is a dataflow diagram description of the triangle program

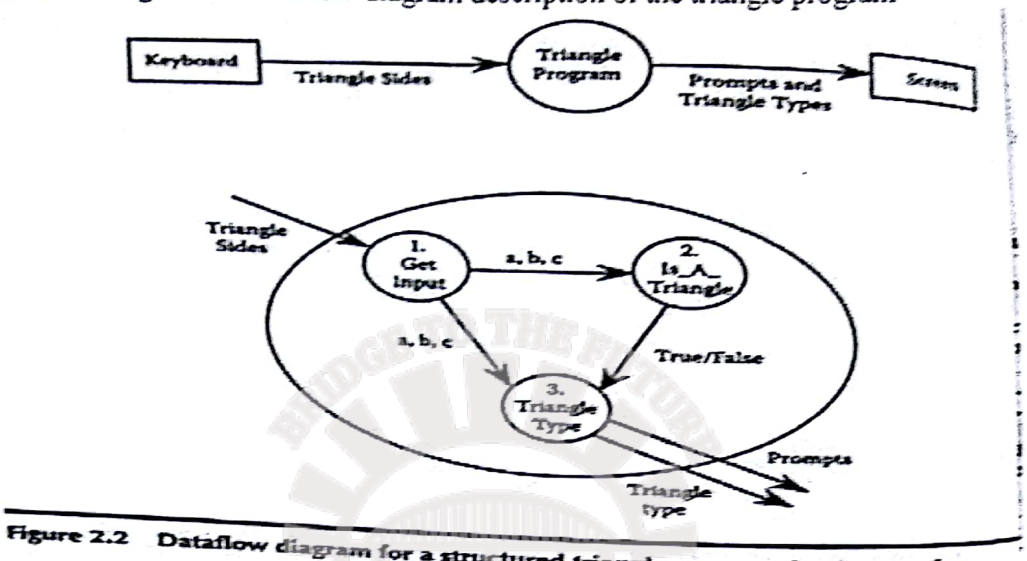


Figure 2.2 Dataflow diagram for a structured triangle program implementation.

```
Program triangle2 'Structured programming version of simple
specification
'
Dim a,b,c As Integer
Dim IsATriangle As Boolean
'
'Step 1: Get Input
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
      Then IsATriangle = True
      Else IsATriangle = False
EndIf
'
'Step 3: Determine Triangle Type
If IsATriangle
     Then   If (a = b) AND (b = c)
              Then Output ("Equilateral")
              Else   If (a = b) AND (a = c) AND (b = c)
                       Then    Output ("Scalene")
                       Else    Output ("Isosceles")
                     EndIf
            EndIf
     Else       Output ("Not a Triangle")
EndIf
'
End triangle2

Program triangle3 'Structured programming version of improved
specification
'
Dim a,b,c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'
'Step 1: Get Input
Do
     Output("Enter 3 integers which are sides of a triangle")
     Input(a,b,c)
     c1 = (1 <= a) AND (a <= 200)
     c2 = (1 <= b) AND (b <= 200)
     c3 = (1 <= c) AND (c <= 200)
```

```
      If NOT(c1)
        Then  Output("Value of a is not in the range of
        permitted values")
      EndIf
      If NOT (c2)
        Then  Output("Value of b is not in the range of
        permitted values")
      EndIf
      If NOT(c3)
        Then  Output ("Value of c is not in the range of
        permitted values")
      EndIf
Until c1 AND c2 AND c3
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < (b + c)) AND (b < (a + c)) AND (c < (a + b))
     Then IsATriangle = True
     Else IsATriangle = False
EndIf
  '
  'Step 3: Determine Triangle Type
  If IsATriangle
       Then  If (a = b) AND (b = c)
                Then Output ("Equilateral")
                Else  If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                         Then  Output ("Scalene")
                         Else  Output ("Isosceles")
                      EndIf
             EndIf
       Else  Output ("Not a Triangle")
  EndIf
  '
  End triangle3
```

The NextDate Function

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.

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.

C1. $1 \leq month \leq 12$ C2. $1 \leq day \leq 31$ C3. $1812 \leq year \leq 2012$

Discussion

There are two sources of complexity in the NextDate function: the just mentioned complexity of the input domain, and the rule that distinguishes common years from leap years. Since a year is 365.2422 days long, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, there would be a slight error. The Gregorian Calendar (instituted by Pope Gregory in 1582) resolves this by adjusting leap years on century years. Thus 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, so 1992, 1996, and 2000 are leap years, while the year 1900 is a common year.

Implementation

```
Program NextDate1        'Simple version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
'
Output ("Enter today's date in the form MM DD YYYY")
Input (month,day,year)
```

```
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
   If day < 31
     Then tomorrowDay = day + 1
     Else
       tomorrowDay = 1
       tomorrowMonth = month + 1
   EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
   If day < 30
     Then tomorrowDay = day + 1
     Else
       tomorrowDay = 1
       tomorrowMonth = month + 1
   EndIf
Case 3: month Is 12: 'December
   If day < 31
     Then tomorrowDay = day + 1
     Else
       tomorrowDay = 1
       tomorrowMonth = 1
       If year = 2012
          Then Output ("2012 is over")
          Else tomorrow.year = year + 1
       EndIf
   EndIf
Case 4: month is 2: 'February
   If day < 28
     Then tomorrowDay = day + 1
     Else
       If day = 28
         Then
           If ((year is a leap year)
             Then tomorrowDay = 29 'leap year

                  Else    'not a leap year
                    tomorrowDay = 1
                    tomorrowMonth = 3
             EndIf
           Else   If day = 29
                    Then tomorrowDay = 1
                    tomorrowMonth = 3
                  Else Output ("Cannot have Feb.", day)
                EndIf
           EndIf
       EndIf
  EndCase
  Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay,
  tomorrowYear)
  '
  End NextDate

  Program NextDate2        Improved version
  '
  Dim tomorrowDay,tomorrowMonth, tomorrowYear As Integer
  Dim day,month,year As Integer
  Dim c1, c2, c3 As Boolean
  '
  Do
     Output ("Enter today's date in the form MM DD YYYY")
     Input (month,day,year)
     c1 = (1 <= day) AND (day <= 31)
     c2 = (1 <= month) AND (month <= 12)
     c3 = (1812 <= year) AND (year <= 2012)
     If NOT(c1)
       Then       Output ("Value of day not in the range 1..31'
     EndIf
     If NOT (c2)
     EndIf
       Then       Output ("Value of month not in the range 1..12
     If NOT(c3)
       Then   Output ("Value of year not in the range 1812..2012
     EndIf
Until c1 AND c2 AND c3
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec
   If day < 31
     Then tomorrowDay = day + 1
     Else
        tomorrowDay = 1
        tomorrowMonth = month + 1
   EndIf
```

```
Case 2: month Is 4,6,9, Or 11 '30 day months
   If day < 30
     Then tomorrowDay = day + 1
     Else
        If day =30
          Then  tomorrowDay = 1
                tomorrowMonth = month + 1
          Else  Output ("Invalid Input Date")
        EndIf
   EndIf
Case 3: month Is 12: 'December
   If day < 31
     Then tomorrowDay = day + 1
     Else
        tomorrowDay = 1
        tomorrowMonth = 1
        If year = 2012
          Then Output ("Invalid Input Date")
          Else tomorrow.year = year + 1
        EndIf
   EndIf
Case 4: month is 2: 'February
   If day < 28
     Then tomorrowDay = day + 1
     Else
       If day = 28
         Then
           If (year is a leap year)
             Then tomorrowDay = 29 'leap day
             Else  'not a leap year
                tomorrowDay = 1
                  tomorrowMonth = 3
             EndIf
           Else
             If day = 29
               Then
                 If (Year is a leap year)
                    Then tomorrowDay = 1
                         tomorrowMonth = 3
                 Else
                    If day > 29
                      Then Output ("Invalid Input Date")
                    EndIf
                 EndIf
             EndIf
         EndIf
   EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay
tomorrowYear)

End NextDate2
```

The Commission Problem

Our third example is more typical of commercial computing. It contains a mix of computation and decision making, so it leads to interesting testing questions.

Problem Statement

A Rifle salespersons in the Former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45.00, stocks cost $30.00, and barrels cost $25.00. Salesperson had to sell at least one complete rifle per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. After each town visit the salesperson

sent a telegram to the Missouri gunsmith with the number of locks, stocks and barrels sold in that town. At the end of each month, the salesperson sent a very short telegram showing -1 lock sold. The gunsmith then knew the sales for the month were completed and computed the salesperson's commission as follows: 10% on sales up to $1000, 15% on the next $800, and 20% on any sales in excess of $1800. The commission program produced a month sales report that gave the total number of locks, stocks, and barrels sold, the salesperson told $ sales and finally, the commission.

Implementation

```
Program Commission (INPUT,OUTPUT)
'
Dim locks, stocks, barrels As Integer
Dim lockPrice, stockPrice, barrelPrice As Real
Dim totalLocks, totalStocks,totalBarrels As Integer
Dim lockSales, stockSales, barrelSales As Real
Dim sales,commission : REAL
'
lockPrice = 45.0
stockPrice = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
totalBarrels = 0
'
Input(locks)
While NOT(locks = -1)  'Input device uses -1 to indicate end of
data
   Input(stocks, barrels)
   totalLocks = totalLocks + locks
   totalStocks = totalStocks + stocks
   totalBarrels = totalBarrels + barrels
   Input(locks)
EndWhile
'
Output("Locks sold: ", totalLocks)
Output("Stocks sold: ", totalStocks)
Output("Barrels sold: ",totalBarrels)
'
lockSales = lockPrice * totalLocks
stockSales = stockPrice * totalStocks
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output("Total sales: ", sales)
'
If (sales > 1800.0)
   Then
      commission = 0.10 * 1000.0
      commission = commission + 0.15 * 800.0
      commission = commission + 0.20 *(sales-1800.0)
   Else If (sales > 1000.0)
         Then
            commission = 0.10 * 1000.0
            commission = commission + 0.15*(sales-1000.0)
         Else commission = 0.10 * sales
        EndIf
EndIf
Output("Commission is $",commission)
'
End Commission
```

The SATM System

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.

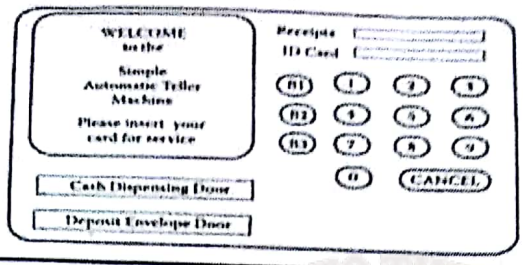


Figure 2.3 The SATM terminal.

Problem statement

The SATM system communicates with bank customers via the fifteen screens shown in Figure 2.4. Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries, and these can be done on two types of accounts, checking and savings.

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/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 system adds two pieces of information to the customer's account file: the current date, and an increment to the number of ATM sessions. The customer selects the desired transaction from the options shown on screen 5; then the system immediately displays screen 6,where the customer chooses the account to which the selected transaction will be applied.

If balance is requested, the system checks the local ATM file for any unposted transactions, and reconciles these with the beginning balance for that day from the customer account file. Screen 14 is then displayed.

If 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 there is a problem 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 deposit amount is entered as an unposted amount in the local ATM file, and the count of deposits per month is incremented. Both of these (and other information) are processed by the Master ATM (centralized) system once per day. The system then displays screen 14.

If 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 money to dispense.

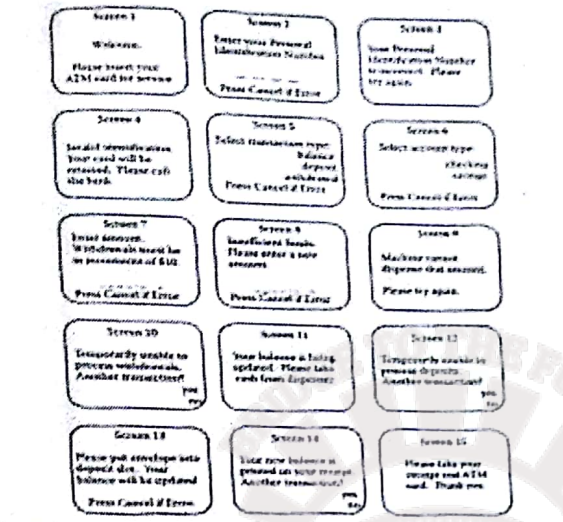


Figure 2.4 SATM screens.

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), and if there are insufficient funds, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed, and the money is dispensed. The withdrawal amount is written to the unposted local ATM file, and the count of withdrawals per month in incremented. 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.

Discussion

There is a surprising amount of information "buried" in the system description just given. For instance, if you read it closely, you can infer that the terminal only contains ten dollar bills (see screen 7). This textual definition is probably more precise than what is usually encountered in practice.

The Currency Coverter

The currency conversion program is another event-driven program that emphasizes code associated with a graphical user interface (GUI). A sample GUI built with Visual Basic is shown in Figure 2.5.

The application converts U.S. dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen. Currency selection is governed by the radio buttons (Visual Basic 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 U.S. 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

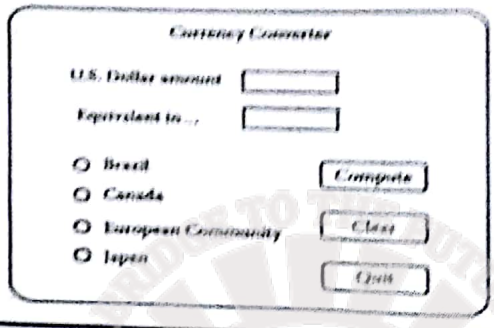


Figure 2.5 Currency converter GUI.

results in the conversion of the U.S. dollar amount to the equivalent amount in the selected currency. Clicking on the Clear button resets the currency selection, the U.S. 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 some Saturn automobiles is controlled by a lever 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.

c1.	Lever	OFF	INT	INT	INT	LOW	HIGH
c2.	Dial	n/a	1	2	3	n/a	n/a
a1.	Wiper	0	4	6	12	30	60