

MODULE - 1

BASICS OF SOFTWARE TESTING

1.1 Basic Definitions

Much of testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades and via scores of writers. The terminology here (and throughout this book) is taken from standards developed by the Institute of Electronics and Electrical Engineers Computer Society. To get started let's look at a useful progression of terms

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.

Error

People make errors. A good synonym 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.

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. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. This suggests a useful refinement; to borrow from the Church, we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve

Failure

A failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code. The second subtlety is that this definition relates failures only to faults of commission.

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 failure.

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

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.

A TESTING LIFE CYCLE

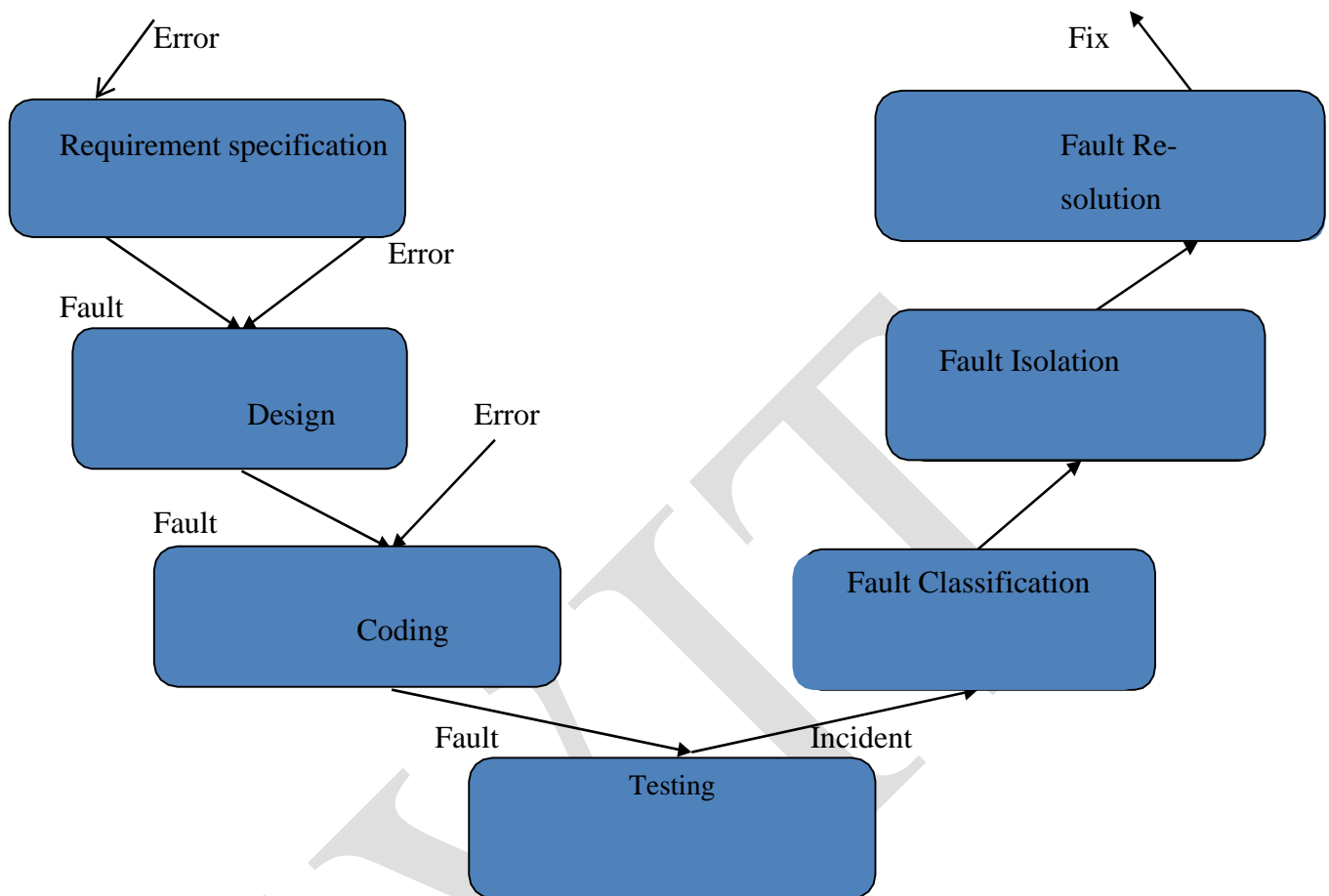


Figure portrays a life cycle model for testing. Notice that, in the development phases, there are three opportunities for errors to be made, resulting in faults that propagate through the remainder of the development. One prominent tester summarizes this life cycle as follows: the first three phases are –Putting Bugs IN, the testing phase is Finding Bugs, and the last three phases are –Getting Bugs OUT.

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. We'll revisit this when we discuss regression testing. 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: test planning, test case development, running test cases, and evaluating test results

1.2 SOFTWARE QUALITY

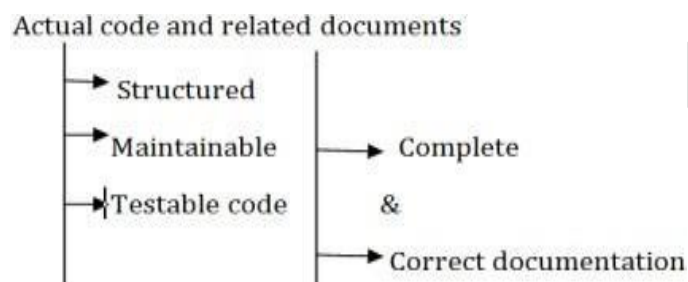
- Software quality is a multidimensional quantity and is measurable.

Quality Attributes

- These can be divided to static and dynamic quality attributes

Static quality attributes

- It refers to the actual code and related documents.



Example:

- A poorly documented piece of code will be harder to understand and hence difficult to modify.
- A poorly structured code might be harder to modify and difficult to test.

Dynamic quality Attributes:

- Reliability
- Correctness
- Completeness
- Consistency
- Usability
- Performance

Reliability:

- It refers to the probability of failure free operation.

Correctness:

- Refers to the correct operation and is always with reference to some artifact
- For a Tester, correctness is w.r.t to the requirements
- For a user correctness is w.r.t the user manual

Completeness:

- Refers to the availability of all the 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.
- Ex: All buttons in the user interface might follow a common-color coding convention.

Usability:

- Refer to 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.
- Usability testing is a testing done by its potential users.
- The development organization invites a selected set of potential users and asks them to test the product.
- Users in turn test for ease of use, functionality as expected, performance, safety & security
- Users thus serve as an important source of tests that developers or testers within the organization might not have conceived.
- Usability testing is sometimes referred to as user-centric testing.

Performance:

- Refers to the time the application takes to perform a requested task.
- Performance is considered as a non-functional requirement.

1.3 Requirements, Behaviour and Correctness:

- Product(or) software are designed in response to requirements. (Requirements specify the functions that a product is expected to perform.) During the development of the product, the requirement might have changed from what was stated originally. Regardless of any change, the expected behaviour of the product is determined by the tester's understanding of the requirements during testing.

- Example 1.3 : Two requirements are given below ,each of which leads to different program

Requirement 1: It is required to write a program that inputs 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 the program max is developed to satisfy requirement 1 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 two integers are to be input to the program on one line followed by a carriage return typed in after each number. This example illustrates the **incompleteness** requirements 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 behavior of sort program, written to satisfy this requirement, will depend on the decision taken by the programmers while writing sort. Testers are often faced with **incomplete/ambiguous requirements**. In such situations a testers may resort to a variety of ways to determine what behaviour to expect from the program under test).
- 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 functions as desired

1.3.1 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.
- For integer value, -32,768 to 32,767. This requires 2³² executions.
- Testing a program on all possible inputs is known as –exhaustive testing
- If the requirements are complete and unambiguous, it should be possible to determine the set of all possible inputs.

Definition:**Input Domain**

- The set of all possible inputs to 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 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 from example 1.3 ,it is not possible to find the input domain for sort program, let is find 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, For Example, Following are three element in the input domain of sort:

<A – 3 15 12 55. .>

<D 23 78. >

< A . >

The first element contains a sequence of four integers to be sorted in ascending order, the second one has a sequence to be sorted in descending order and third one has an empty sequence to be sorted in ascending order

Definition

Correctness

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

1.3.2 Valid and Invalid Inputs:

- The input domains are derived from the requirements. It is difficult to determine the input domain for incomplete requirements.
- Identifying the set of invalid inputs and testing the program against these inputs are important parts of the testing activity. Even when the requirements fail to specify the program behavior 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.

Ex: sort program < E 7 19...>

The sort program enters into an infinite loop and neither asks the user for any input nor responds to anything typed by the user. This observed behavior points to a possible error in sort.

1.4 Correctness Versus Reliability

1.4.1 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, which is impossible to accomplish in most cases that are encountered in practice, thus correctness is established via mathematical proofs of programs, The proof uses the formal specification of requirements and the program text to prove or disprove that the program will behave as intended

While correctness attempts to establish that the program is error-free testing attempts to find if there are any errors in it. Thus completeness of testing does not necessarily demonstrate that a program is error free. Removal of errors from the program usually improves the chances or the probability of the program executing without any failure, Also testing, debugging ,and the errors-removal process together increase our confidence in the correct functioning of the program under test

Example : This example illustrates why the probability of program failure might not change upon error removal, Consider the following program that inputs two 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  $x \leq y$ 
    {print f(x,y) }
else
    {print g( x ,y )}
```

1.4.2 Reliability

The probability of a program failure is captured more formally in the term reliability, Consider –The reliability of a program P is the probability of its successful execution on a randomly selected element from its input domainl

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 and 1,Intititively,when an error is removed from program, the reliability of the program so obtained is expected to be higher than that of the one that contains the error, As illustrated in example.

Example: 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 there are only three pairs that will be input to P, There are as follows

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

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

1.4.3 Program Use and 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 the sort program which, on any given execution, allows any one of two types of input sequences, One sequence consists of number 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

Operational profile 2	
Sequence	probability
Numbers only	0.1
Alphanumeric strings	0.9

The two operational profiles above suggest significantly different uses of sort, in one case it is used mostly for sorting sequences of numbers and in the other case it is used mostly for sorting alphanumeric strings

1.5 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.
- However, when testing reveals an error, the process used to determine the cause of this error and to remove it is known as debugging.

As illustrated in figure, testing and debugging are often used as two related activities in a cyclic manner.

Steps are

1. Preparing a test plan
2. Constructing test data
3. Executing the program
4. Specifying program behaviour
5. Assessing the correctness of program behaviour
6. Construction of oracle

1.5.1 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. **Example test plan:**
- Consider following items such as the method used for testing, method for evaluating the adequacy of test cases, and method to determine if a program has failed or not.

Test plan for sort : The sort program is to be tested to meet the requirements given in example

1. Execute the program on at least two input sequence one with -A|| and the other with -D|| as request characters.
2. Execute the program on an empty input sequence
3. Test the program for robustness against erroneous input such as -R|| typed in as the request character.
4. All failures of the test program should be recorded in a suitable file using the company failure report form.

A test/debug cycle

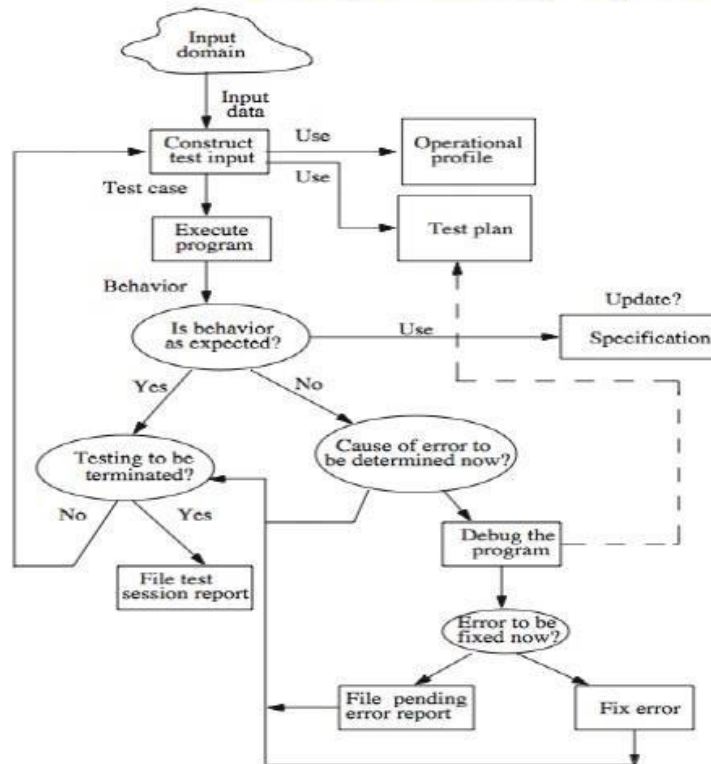


Figure: Test and Debug cycle

1.5.2 Constructing Test Data

A test case is a pair consisting of test data to be input to 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

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. Based on the results obtained, the testers decide whether to continue the construction of additional testcases or to enter the debugging phase.

The following test cases are generated for the sort program using the test plan in the previous figure

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" 3 17.>

Expected output: Invalid request character;

valid characters: "A" and "D"

Test case 6:

Test data: <"A" c 17.>

Expected output: Invalid number

1.5.3 Executing the program:

- Execution of a program under test is the next significant step in the 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.
- Testers might be able to construct a test harness to aid in program execution. The harness initializes any global variables, inputs a test case, and executes the program. The output generated by the program may be saved in a file for subsequent examination by a tester

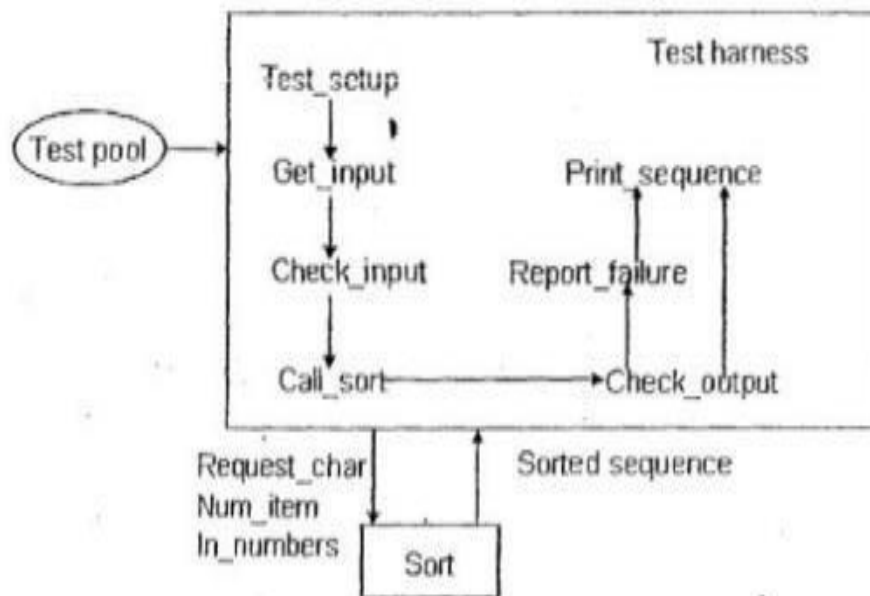


Figure: A simple test harness to test the *sort* program.

In preparing this test harness assume that:

- Sort is coded as a procedure
- The get-input procedure reads the request character & the sequence to be sorted into variables request char, num_items and in_number, test_setup procedure-invoked first to set up the test includes identifying and opening the file containing tests.
- Check output procedure serve as the oracle that checks if the program under test behaves correctly
- Report_failure: output from sort is incorrect. May be reported via a message(or) saved in a file
- Print_sequence: prints the sequence generated by the sort program. This also can be saved in file for subsequent examination

1.5.4 Specifying program behavior:

There are several ways to define and specify program behavior, the notion of program state can be used to define program behavior and how the state transition diagram, or simply state diagram can be used to specify program behavior

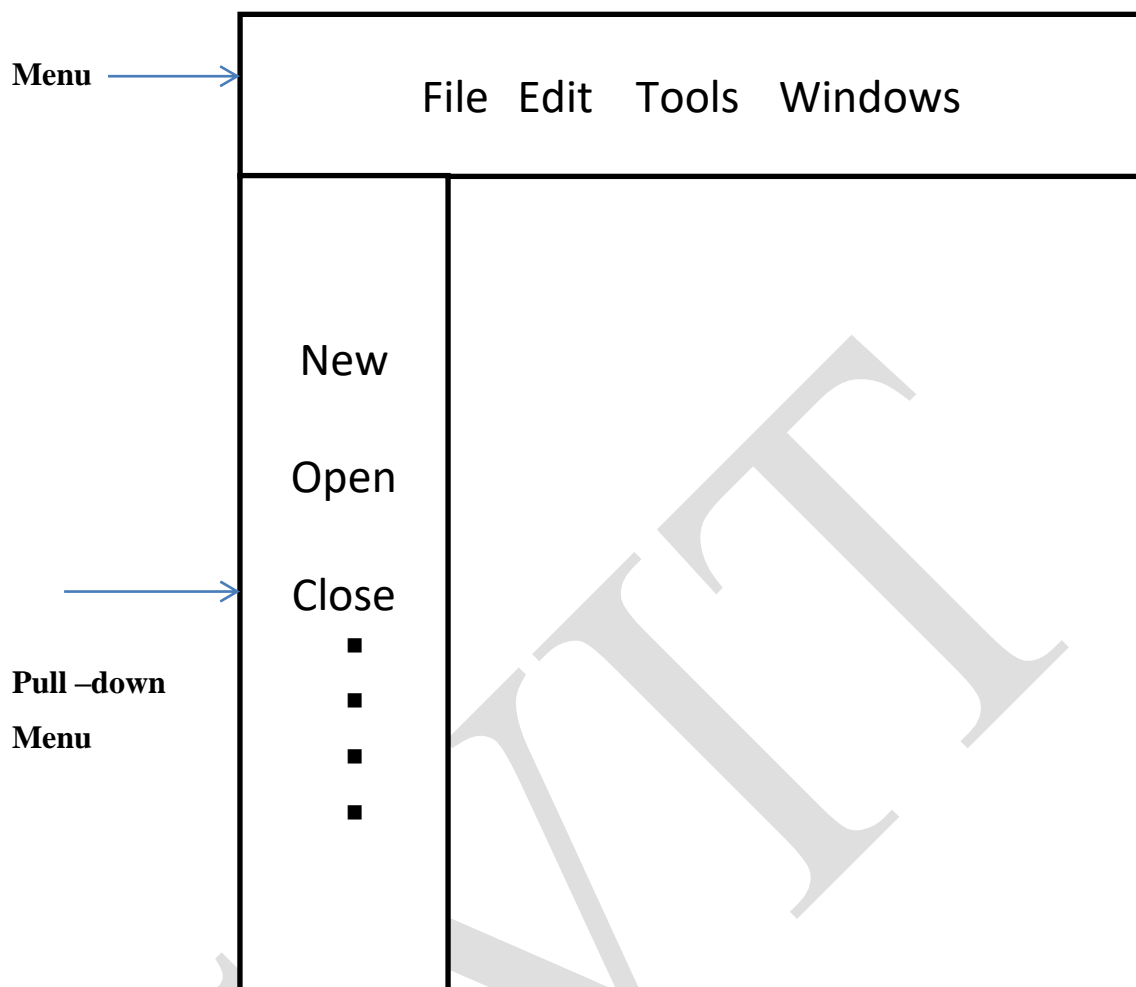


Figure: Menu bar displaying four menu items when application myapp is started

Consider a menu driven application named myapp Figure shows Menu bar for this application. it allows a user to position and click the mouse on any one of a list of menu items displayed in the menu bar on the screen. This results in pulling down of the menu and list of options is displayed on the screen. one of the items on the menu bar is labeled File. When File is pulled down, it displays open as one of several options. When the open option is selected, by moving the cursor over it, it should be highlighted, when the mouse is released, indicating that the selection is complete. A window displaying name of files in the current directory should be displayed

State vector: collecting the current values of program variables into a vector known as the statevector. An indication of where the control of execution is at any instant of time can be given by using an identifier associated with the next program statement.

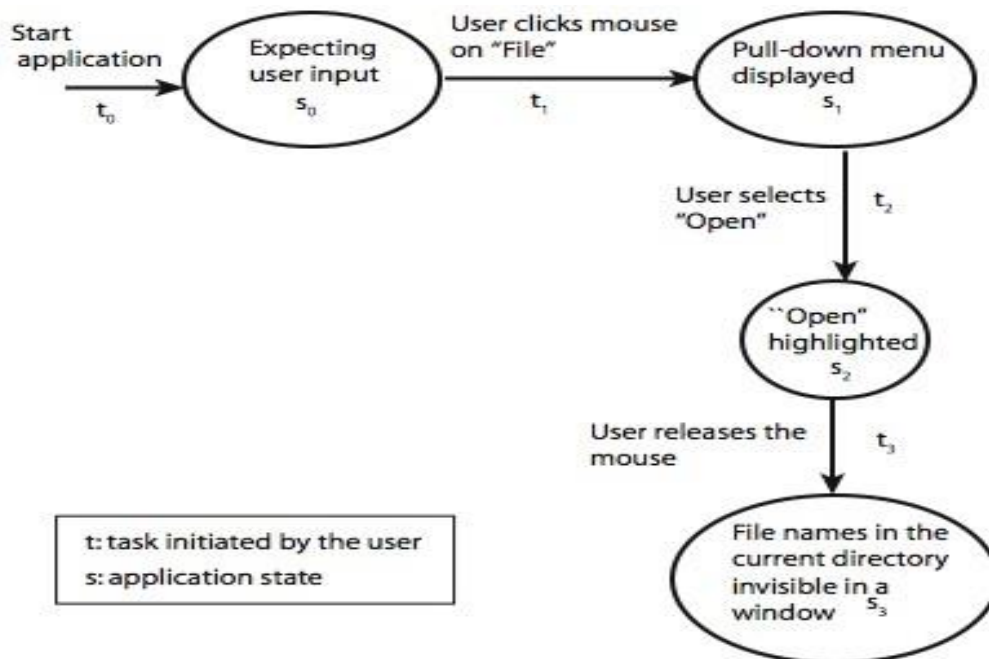


Figure depicts the sequence of states that my app is expected to enter when the user actions described are performed, when started, the application enters the initial states wherein it displays the menu bar and waits for the user to select a menu item. This state diagram depicts the expected behavior of myapp in terms of state sequence

As shown in figure myapp moves from state s_0 to s_3 after the sequence of action t_0 , t_1 , t_2 , t_3 , has been applied. To test myapp, the tester could apply the sequence of actions depicted in this state diagram

State sequence diagram can be used to specify the behavioral requirements. This same specification can then be used during the testing to ensure if the application confirms to the requirements

1.5.5 Assessing the correctness of program Behavior:

It has two steps:

- Observes the behaviour
- Analyzes the observed behaviour.

Above task, extremely complex for large distributed system

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

Oracle can also be programs designed to check the behaviour of other programs

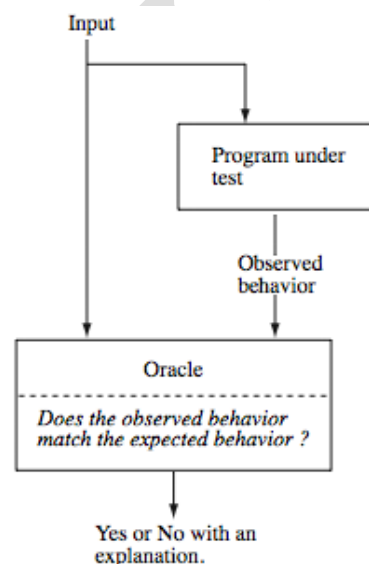


Figure: Relationship between the program under test and the oracle

A tester often assumes the role of an oracle and thus serves as a human oracle, for example to verify if the output of matrix multiplication program is correct or not, a tester might input 2x2 matrices and check if the output produced by the program matches the results of hand calculation

Checking the program behavior by humans has several disadvantages First it is error prone as the human oracle might make error in analysis, Second ,it may be slower than the speed with which the program computed results,

Using program as oracles has the advantages of speed ,accuracy, and the ease with which the complex computations can be checked, thus a matrix multiplication program, when used as an oracle for matrix inversion program can be faster ,accurate.

1.5.6 Construction of oracles:

Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, requires determination of I/O relationship. When tests are generated from models such as finite-state machines (FSMs) or state charts, both inputs and the corresponding outputs are available. This makes it possible to construct an oracle while generating the tests.

Example:

Consider a program named Hvideo that allows one to keep track of home videos. In the data entry mode, it displays a screen in which the user types in information about a DVD. In searchmode, the program displays a screen into which a user can type some attribute of the video being searched for and set up a search criterion.

- To test Hvideo we need to create an oracle that checks whether the program function correctly in data entry and search nodes. The input generator generates a data entry request. The input generator now requests the oracle to test if Hvideo performed its task correctly on the input given for data entry.

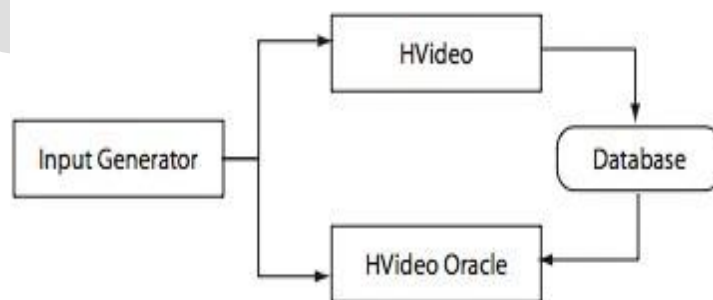
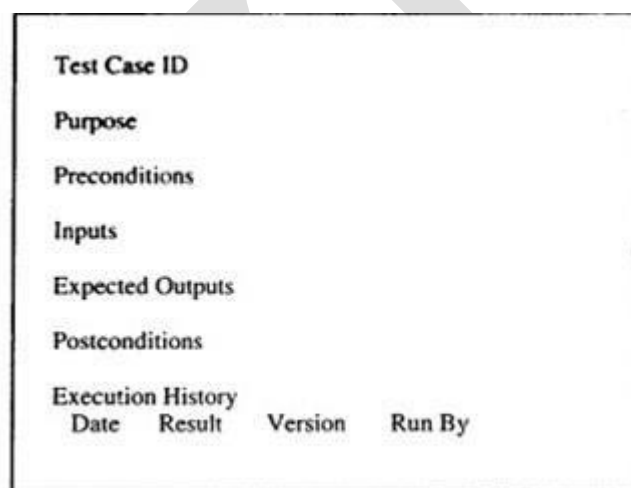


Figure: Relationship between an input generator ,Hvideo ,and its oracle

- The oracle uses the input to check if the information to be entered into the database has been entered correctly or not.
- The oracle returns a pass or no pass to the input generator.

1.6 Test Cases

The essence of software testing is to determine a set of test cases for the item being tested. Before going on, we need to clarify what information should be in a test case. The most obvious information is inputs; inputs are really of two types: pre-conditions (circumstances that hold prior to test case execution) and the actual inputs that were identified by some testing method. The next most obvious part of a test case is the expected outputs; again, there are two types: post conditions and actual outputs. The output portion of a test case is frequently overlooked.



Execution History			
Date	Result	Version	Run By

Typical test case information

Suppose, for example, you were testing software that determined an optimal route for an aircraft, given certain FAA air corridor constraints and the weather data for a flight day.

The act of testing entails establishing the necessary pre-conditions, providing the test case inputs, observing the outputs, and then comparing these with the expected outputs to determine whether or not the test passed.

The remaining information in a well-developed test case primarily supports testing management. Test cases should have an identity, and a reason for being (requirements tracing is a fine reason). It is also useful to record the execution history of a test case, including when and by whom it was run, the pass/fail result of each execution, and the version (of software) on which it was run. From all of this, it should be clear that test cases are valuable — at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved.

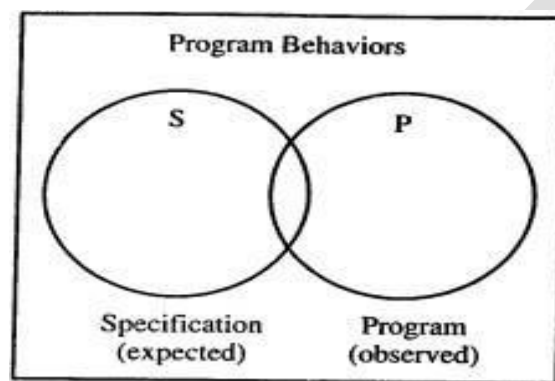


Figure: Specified and Implemented Program Behavior

1.7 Insights from a Venn diagram

Testing is fundamentally concerned with behavior; and behavior is orthogonal to the structural view common to software (and system) developers. A quick differentiation is that the structural 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, and therefore the emphasis is on structural, rather than behavioral, information. We develop a simple Venn diagram which clarifies several nagging questions about testing.

- Consider a Universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set S of specified behaviors, and the set P of programmed behaviors.
- Figure 1.3 shows the relationship between our universe of discourse and the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S; and all those behaviors actually programmed (the slight difference between P and U, the Universe) are in P.

- With this diagram, we can see more clearly the problems that confront a tester. What if there are specified behaviors that have not been programmed?
- In our earlier terminology, these are faults of omission. Similarly, what if there are programmed (implemented) behaviors that have not been specified?
- These correspond to faults of commission, and to errors which 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. (As a sidelight, note that –correctness only has meaning with respect to a specification and an implementation. It is a relative term, not an absolute.)

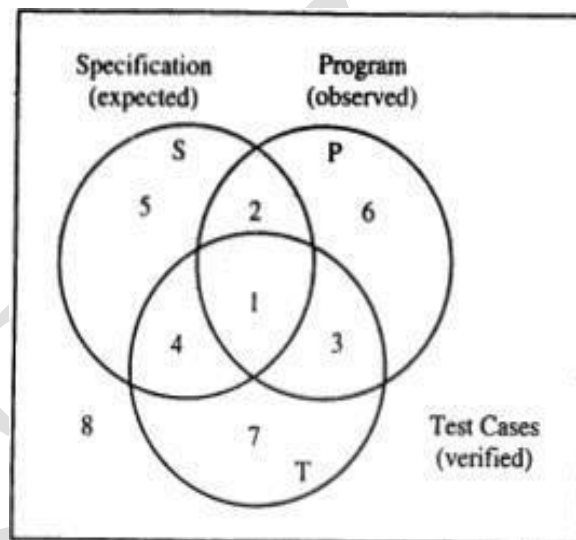


Figure: Specified, Implemented, and Tested Behavior

- The new circle in Fig. 1.3 is for Test Cases. Notice there is a slight discrepancy with our Universe of Discourse, the set of program behaviors. Since a test case causes a program behavior, the mathematicians might forgive us. Now, consider the relationships among the 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 un-programmed behaviors (regions 4 and 7).
- Each of these regions is important. If there are specified behaviors for which there are no test cases, the testing is necessarily incomplete.
- If there are test cases that correspond to unspecified behaviors, two possibilities arise: either such a test case is unwarranted, or the specification is deficient
- This is a fine reason to have good testers participate in specification and design reviews.) We are already at a point where we can see some possibilities for testing as a craft: what can a tester do to make the region where these sets all intersect (region 1) be as large as possible? Another way to get at this is to ask how the test cases in these sets T are identified. The short answer is that test cases are identified by a testing method. This framework gives us a way to compare the effectiveness of diverse testing methods.

1.8 Identifying Test Cases

There are two fundamental approaches to identifying test cases; these are known as functional and structural testing. Each of these approaches has several distinct test case identification methods, more commonly called testing methods.

1.8.1 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. (Function, domain, and range are defined in Chapter 3.)
- This notion is commonly used in engineering, when systems are considered to be –black boxes. This leads to the term Black Box Testing, in which the content (implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs.
- In *Zen and The Art of Motorcycle Maintenance*, Pirsig refers to this as –romantic comprehension. Many times, we operate very effectively with black box knowledge; in fact this is central to object orientation. As an example, most people successfully operate automobiles with only black box knowledge.

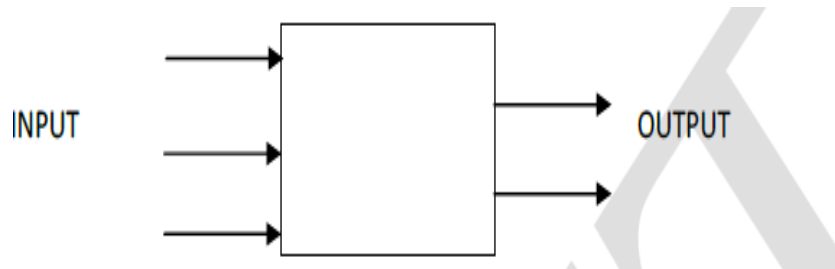


Figure: An Engineer's Black Box

- With the functional approach to test case identification, the only information that is used is the specification of the software.
- There are two distinct advantages to functional test cases: they are independent of how the software is implemented, so if the implementation changes, the test cases are still useful, and test case development can occur in parallel with the implementation, thereby reducing overall project development interval.
- On the negative side, functional test cases frequently suffer from two problems: there can be significant redundancies among test cases, and this is compounded by the possibility of gaps of untested software.
- Figure shows the results of test cases identified by two functional 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. Since functional methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified.

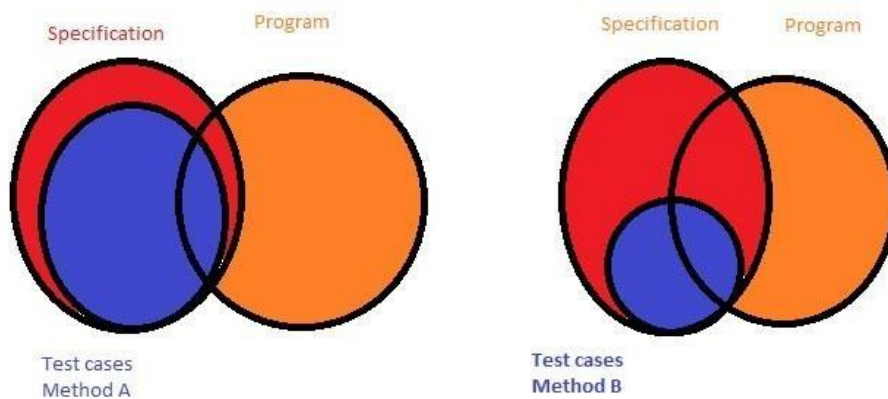


Figure: Comparing functional test case identification methods

1.8.2 Structural Testing

- Structural testing is the other fundamental approach to test case identification. To contrast it with Functional Testing, it is sometimes called White Box (or even Clear Box) Testing.
- The clear box metaphor is probably more appropriate, because the essential difference is that the implementation (of the Black Box) is known and used to identify test cases. Being able to –see inside the black box allows the tester to identify test cases based on how the function is actually implemented.
- Structural Testing has been the subject of some fairly strong theory. To really understand structural testing, the concepts of linear graph theory (Chapter 4) are essential. With these concepts, the tester can rigorously describe exactly what is being tested.
- Because of its strong theoretical basis, structural testing lends itself to the definition and use 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

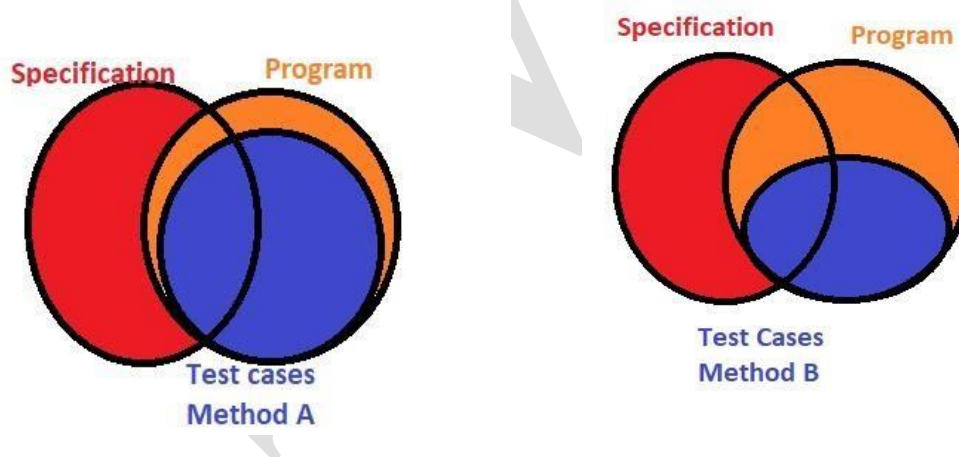


Figure: Comparing Structural test case identification methods

1.8.3 The Functional Versus Structural Debate

- Functional testing uses only the specification to identify test cases
- Structural testing uses the program source code (implementation) as the basis of test case identification
- Consider program behaviors, if all specified behaviors have not been implemented, structural 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 functional test cases
- Both the approaches are needed, a judicious combination will provide the confidence of functional testing and the measurement of structural testing.
- Functional testing often suffers from twin problems of redundancies and gaps
- When functional test cases are executed in combination both of these problems can be recognized and resolved

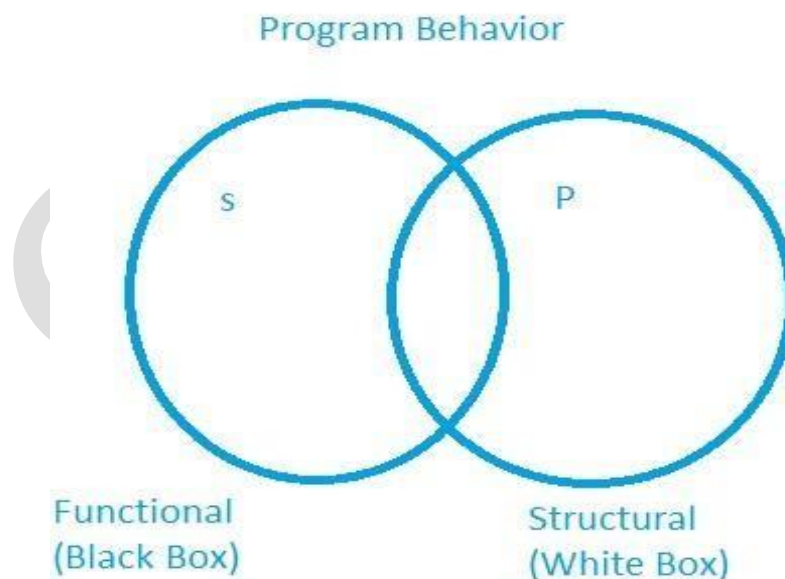


Figure: Source of test cases

1.9 Test –Generation Strategies

One of the key tasks in any software test activity is the generation of 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 test based on knowledge of the requirements. In some organizations, tests are generated using mix of formal and informal methods often directly from the requirements document serving as the source.

In some test processes requirements serve as a source for the development of formal models used for test generation

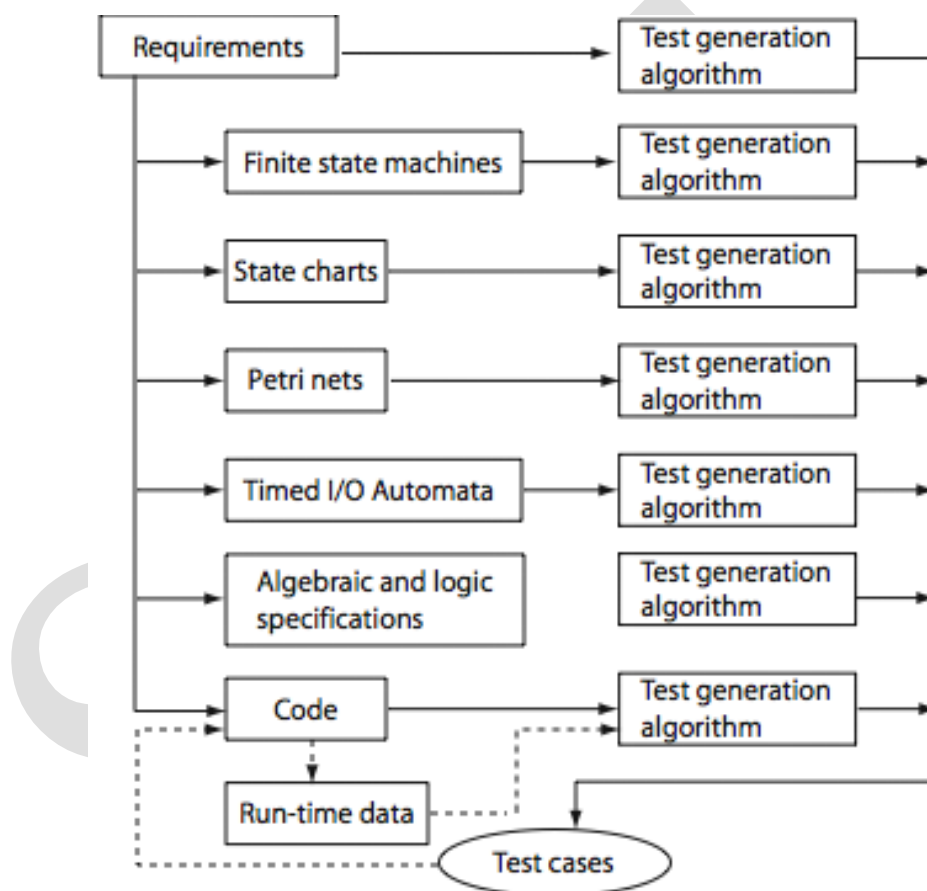


Figure: Requirements ,Models ,and test generation algorithms

- Figure summarizes several strategies for test generation. The top row in 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 and use formal techniques for test generation such as random test generation and cause –effect graphing
- Another set of strategies falls under the category of model based test generation. These strategies require that subset of the requirements be modeled using a formal notation. Such a model is also known as a specification of the subset of requirements.
- The tests are then generated with specification serving as the source. FSMs, state charts, petrinets, and timed I/O automata are some of the well-known and used formal notation for modeling various subsets of requirements.
- The notation falls under the category of graphical notation, through textual equivalents also exist, several other notations such as sequence and activity diagrams in Unified Modeling Language(UML)also exist and are used as models of 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.
- Each of these notational tools have their strengths and weakness. Usually for any large application and generate tests.
- There also exist techniques to generate tests directly from the code. Such techniques falls under code based test generation. These techniques are useful when enhancing existing test based on test adequacy criteria.

For example ,Suppose that program P has been tested against tests generated from a state chart specification, After the successful execution of all tests, one finds that some of the branches in P have not been covered, that is ,there are some conditions that have never been evaluated to both true and false. One could now use code based test generation techniques to generate tests, or modify existing ones, to generate new tests that force a condition to evaluate to true or false, assuming that the evaluations are feasible. Two such techniques, one base on program mutation and other on control –flow coverage.

1.10 TEST METRICS

The term metric refers to a standard of measurement. In software testing, there exist a variety of metrics.

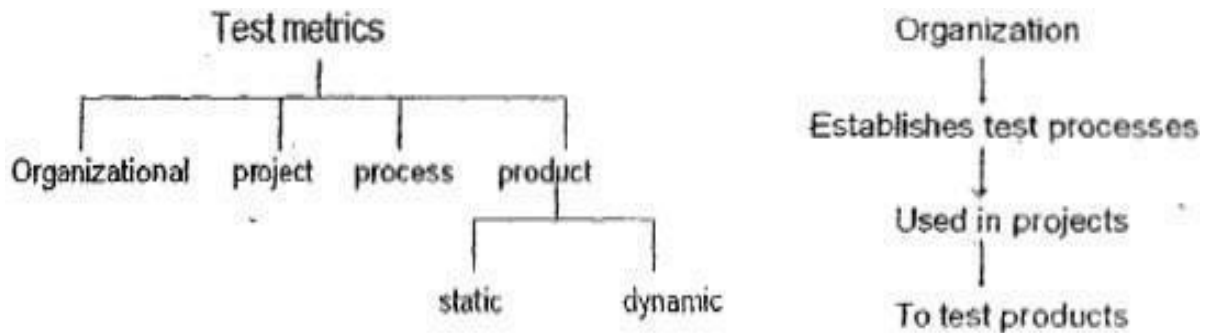


Figure: Types of metrics used in software testing and their relationship

There are four general core areas that assist in the design of metrics

1. Schedule
2. Quality
3. Resources
4. Size

Schedule related metrics:

- Measure actual completion times of various activities and compare these with estimated time to completion.

Quality related metrics:

- Measure quality of a product or a process

Resource related metrics:

- Measure items such as cost in dollars, man power and test executed.

Size-related metrics:

- Measure size of various objects such as the source code and number of tests in a test suite

1.10.1 Organizational metrics:

- Metrics at the level of an organization are useful in overall project planning and management.

Ex: the number of defects reported after product release, averaged over a set of products developed and marketed by an organization, is a useful metric of product quality at the organizational level.

- Organizational metrics allow senior management to monitor the overall strength of the organization and points to areas of weakness. Thus, these metrics help senior management in setting new goals and plan for resources needed to realize these goals.

1.10.2 Project metrics:

- Project metrics relate to a specific project, for example the I/O device testing project or a compiler project. These are useful in the monitoring and 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 the tester –man –months. At the start of the system test phases for example the project manager estimates the total system test effort. The ratio of actual to estimated effort is zero prior to the system test phase.

1.10.3 Process metrics:

- Every project uses some test process. Big-bang approach well suited for small single person projects. The goal of a process metric is to assess the goodness of the process.
- Test process consists of several phases like unit test, integration test, 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. Hence a metric that classifies defects according to the phase in which they are found assists in evaluating the process itself.

1.10.4 Product metrics: Generic

- Cyclomatic complexity
- Halstead metrics

Cyclomatic complexity

$$V(G) = E - N + 2P$$

Program p containing N node, E edges and p connected procedures.

Larger value of V(G) higher program complexity & program more difficult to understand & test than one with a smaller values

V(G) value 5 or less are recommended

Halstead complexity

Number of error(B) found using program size(S) and effort(E) $B = 7.6E0.667 S0.33$

Measure	Notation	Definition
Operator count	N1	Number of operators in a program
Operand count	N2	Number of operands in a program
Unique operators	n1	Number of unique operators in a program
Unique operands	n2	Number of unique operands in a program
Program vocabulary	n	$n1 + n2$
Program size	N	$N1 + N2$
Program volume	V	$N * \log_2 n$
Difficulty	D	$2/n1 * 2n/N$
Effort	E	$D * V$

Table: Halstead measures of program complexity and effort

1.10.5 Product metrics: OO software

Metrics are reliability, defect density, defect severity, test coverage, cyclomatic complexity, weighted methods/class, response set, number of children.

1.10.6 Static and dynamic metrics:

Static metrics are those computed without having to execute the product.

Ex: no. of testable entities in an application. Dynamic metric requires code execution. Ex: no. of testable entities actually covered by a test suite is a dynamic quality.

One could apply the notion of static and dynamic to organization and project, For example the average number of testers working on project is statics project metric, Number of defects remaining to be fixed could be treated as dynamic metrics as it can be computed accurately only after a code change has been made and the product retested.

1.10.7 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 metl.
- Two types:
 1. static testability metrics
 2. dynamic testability metrics
- **Static testability metric:** Software complexity is one static testability metric. more complex an application, the lower the testability, that is higher the effort required to test it.
- **Dynamic testability metrics** for testability includes various code based coverage criteria.Ex: when it is difficult to generate tests that satisfy the statement coverage criterion is considered to have low testability than one for which it is easier to construct such tests.

1.11 Error and Fault Taxonomies

- Our definitions of error and fault hinge on the 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 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, while 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 where the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so . My favorite is based on anomaly occurrence: one time only, intermittent, recurring, or repeatable.

Table 1 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 (too early, too late)
	incomplete or missing result
	spurious result
	spelling/ grammar
	cosmetic

Table 2 Logic Faults missing case(s)

duplicate case(s)
extreme condition neglected
misinterpretation
missing condition
extraneous condition(s)

test of wrong variable
incorrect loop iteration
wrong operator (e.g., < instead ≤)

Table 3 Computation Faults incorrect algorithm

missing computation
incorrect operand
incorrect operation
parenthesis error
insufficient precision (round-off, truncation)
wrong built-in function

Table 4 Interface Faults incorrect interrupt handling

I/O timing
call to wrong procedure
call to non-existent procedure
parameter mismatch (type, number)
incompatible types
superfluous inclusion

Table 5 Data Faults incorrect initialization

incorrect storage/access
wrong flag/index value
incorrect packing/unpacking
wrong variable used
wrong data reference
scaling or units error
incorrect data dimension
incorrect subscript
incorrect type
incorrect data scope
sensor data out of limits
off by one
inconsistent data

1.12 Levels of Testing

Key concepts of testing — levels of abstraction. Levels of testing echo the levels of abstraction found in the Waterfall Model of the software development lifecycle. While 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

There is a practical relationship between levels of testing and functional and structural testing. Most practitioners agree that testing is most appropriate at the unit level, while functional testing is most appropriate at the system level. While this generally true, 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 structural 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. We develop such structures in Part IV to support structural testing at the integration and system levels for both traditional and object-oriented software.

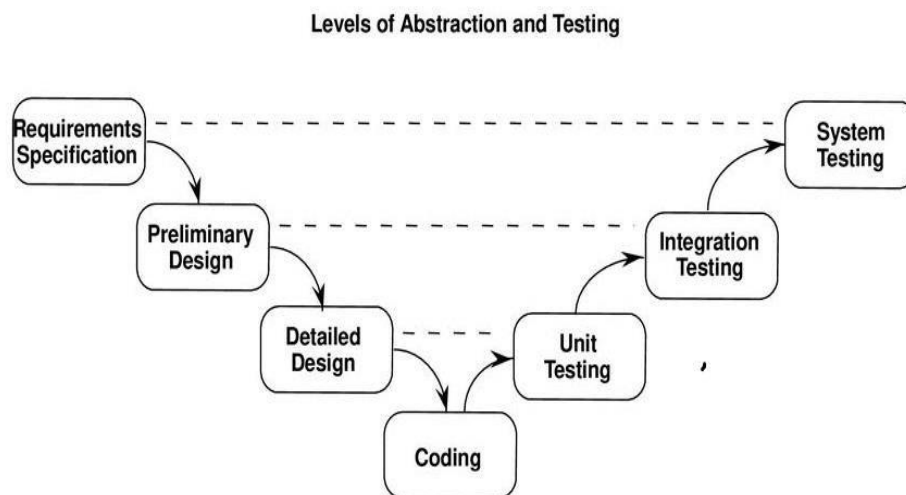


Figure: Levels of Abstraction and testing in the waterfall Model

1.13 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. While verification aims at showing that given program work for all possible inputs that satisfy a set of conditions ,testing aims to show that the given programs is reliable in that no errors of any significance were found.

Program verification and testing are best considered as complimentary techniques in Practice, one often sheds program verification, but not testing, However in the development of critical applications, Such as smart cards or control of nuclear plants ,one often make use of verification techniques to prove the correctness of some artifact created during the development cycle ,not necessarily the complete program

Testing is not a perfect process in that a program 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 the application under test.

Our confidence in the correctness of an application increases when an application passes a set of thoroughly designed and executed tests.

Verification might appear to be 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 the verification process, these 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. Thus neither verification nor testing is a perfect technique for proving the correctness of program

1.14 STATIC TESTING

- Static testing is carried out without executing the application under test.
- This is in contrast to dynamic testing that requires one or more executions of the application under test.
- It is useful in that it may lead to the discovery of faults in the application, ambiguities and errors in the requirements and other application-related document, at a relatively low cost,
- This is especially so when dynamic testing expensive.
- Static testing is complementary to dynamic testing.
- This is carried out by an individual who did not write the code or by a team of individuals.
- The test team responsible for static testing has access to requirements document, application, and all associated documents such as design document and user manual.
- Team also has access to one or more static testing tools. A static testing tool takes the application code as input and generates a variety of data useful in the test process

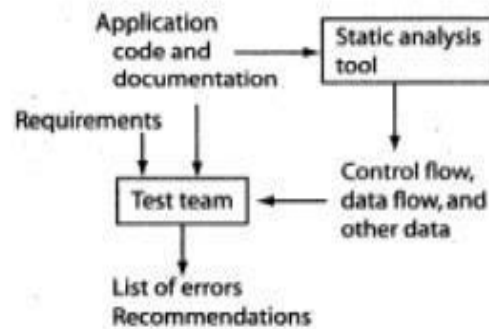


Fig. 1.13 Elements of static testing.

1.14.1 Walkthroughs

- Walkthroughs and inspections are an integral part of static testing.
- Walkthrough are an integral part of static testing.
- Walkthrough is an informal process to review any application-related document.
eg: requirements are reviewed----->requirements walkthrough
code is reviewed----->code walkthrough (or) peer code review
Walkthrough begins with a review plan agreed upon by all members of the team.

Advantages:

- improves understanding of the application
- both functional and non-functional requirements are reviewed.
- A detailed report is generated that lists items of concern regarding the requirements

1.14.2 INSPECTIONS

- Inspection is a more formally defined process than a walkthrough. This term is usually associated with code.
- Several organizations consider formal code inspections as a tool to improve code quality at a lower cost than incurred when dynamic testing is used.

Inspection plan:

1. statement of purpose
2. work product to be inspected this includes code and associated documents needed for inspection.
3. team formation, roles, and tasks to be performed.

4. rate at which the inspection task is to be completed
5. Data collection forms where the team will record its findings such as defects discovered, coding standard violations and time spent in each task.

Members of inspection team

1. Moderator: in charge of the process and leads the review.
2. Leader: actual code is read by the reader, perhaps with help of a code browser and with monitors for all in the team to view the code.
3. Recorder: records any errors discovered or issues to be looked into
4. Author: actual developer of the code.

Chapter -2

Problem Statement

2.1 Generalized Pseudo code

Language Element	Generalized Pseudocode Construct
✓ Comment	`<text>
✓ Data structure decln..	Type<type name><list of desc>End<type name>
✓ Data declaration	Dim<variable>As<type>
✓ Assignment statement	<variable>=<expression>
✓ Input	Input(<variable list>)
✓ Output	Output(<variable list>)
✓ Condition	<expression><relational operator><expression>
✓ Compound condition	<Condition><logical connective><Condition>
✓ Sequence	Statements in sequential order
✓ Simple selection	If<condition>Then<then clause> EndIf
✓ Selection	If<condition>
✓ Multiple Selection	Case <variable>Of
✓	Case 1:<predicate>
✓	<Case clause>
✓	...
✓	Case n:<predicate>
✓	<Case clause>
•	EndCase

Figure : Generalized Pseudocode(1)

- ✓ Counter controlled repetition -> For<counter>=<start> To <end>
- ✓ Pretest repetition -> While<condition> EndWhile
- ✓ Posttest repetition -> Do until<condition>
- ✓ Procedure definition -> <procedure name>(Input:<list of var>;Output:<list of var>)
- ✓ Interunit communication -> Call <procedure name>(<list of var>;<list of var>)
- ✓ Class/object definition -> <name>(<att list>;<method list>, <body>) End<name>
- ✓ Interunit communication -> msg <destination object name>.<method name>(<list of var>)
- ✓ Object creation -> Instantiate<classname>. <objectname>(list of attribute val)
- ✓ Object destruction -> Delete<classname>.<objectname>
- ✓ Program -> Program<program name>

Figure: Generalized Pseudocode (2)

2.2 The 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 Not A Triangle. Sometimes this problem is extended to include right triangles as a fifth type;

2.2.1 Problem Statement

Simple version: The triangle program accepts three integers a, b, and c as input. These are taken to be side of a triangle. The output of the program is the type of triangle determined by the three sides, Equilateral, Isosceles, Scalene, or Not a Triangle

Improved Version: 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.

- 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. $c < a + b$
- ✓ The output is the type of the triangle determined by the three sides.
- ✓ If an input value fails any of conditions c1, c2,c3 the program notes this with an output message.
- ✓ For example: —value of b is not in the range of permitted values||.

2.2.2 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 impairs communication among customers, developers, and testers. This specification presumes the developers know some details about triangles, in particular the 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. A good tester might further clarify the problem statement by putting limits on the lengths of the sides. What response would we expect if we presented the program with the sides -5, -4, -3? We will require that all sides be at least 1, and while we are at it, we may as well declare some upper , say 20,000. (Some languages, like Pascal, have an automatic limit, called MAXINT, which is the largest binary integer representable in a certain number of bits.)

2.2.3 Traditional Implementation

The -traditionall implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure.The flowchart

box numbers correspond to comment numbers in the (FORTRAN-like) Turbo Pascal program given next. (These numbers correspond exactly to those in [Pressman 82].) The variable match is used to record equality among pairs of the sides. There is a classical intricacy of the FORTRAN style connected with the variable match: notice that all three tests for the triangle property do not occur.

If two sides are equal, say a and c, it is only necessary to compare a+c with b. (Since b must be greater than zero, a + b must be greater than c, because c equals a.) This observation clearly reduces the amount of comparisons that must be made. The efficiency of this version is obtained at the expense of clarity (and ease of testing!).

The Traditional Implementation

- Program triangle1 _Fortran-like version`
- Dim a,b,c,match As INTEGER `
- Output(—Enter the 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+b)<=c
- Then Output(—NotATriangle||)
- Else Output(—Isoceses||)
- EndIf
- Else If match=2
- Then If (a+c)<=b
- Then Output (—NotATriangle||)
- Else Output(—Isoceses||)
- EndIf
- Else If match=3
- Then If (b+c)<=a
- Then Output(—NotATriangle||)
- Else Output(—Isoceses||)
- EndIf
- Else Output (-equilateral||)
- EndIf
- EndIf
- EndIf
- EndIf`
- End Triangle 1

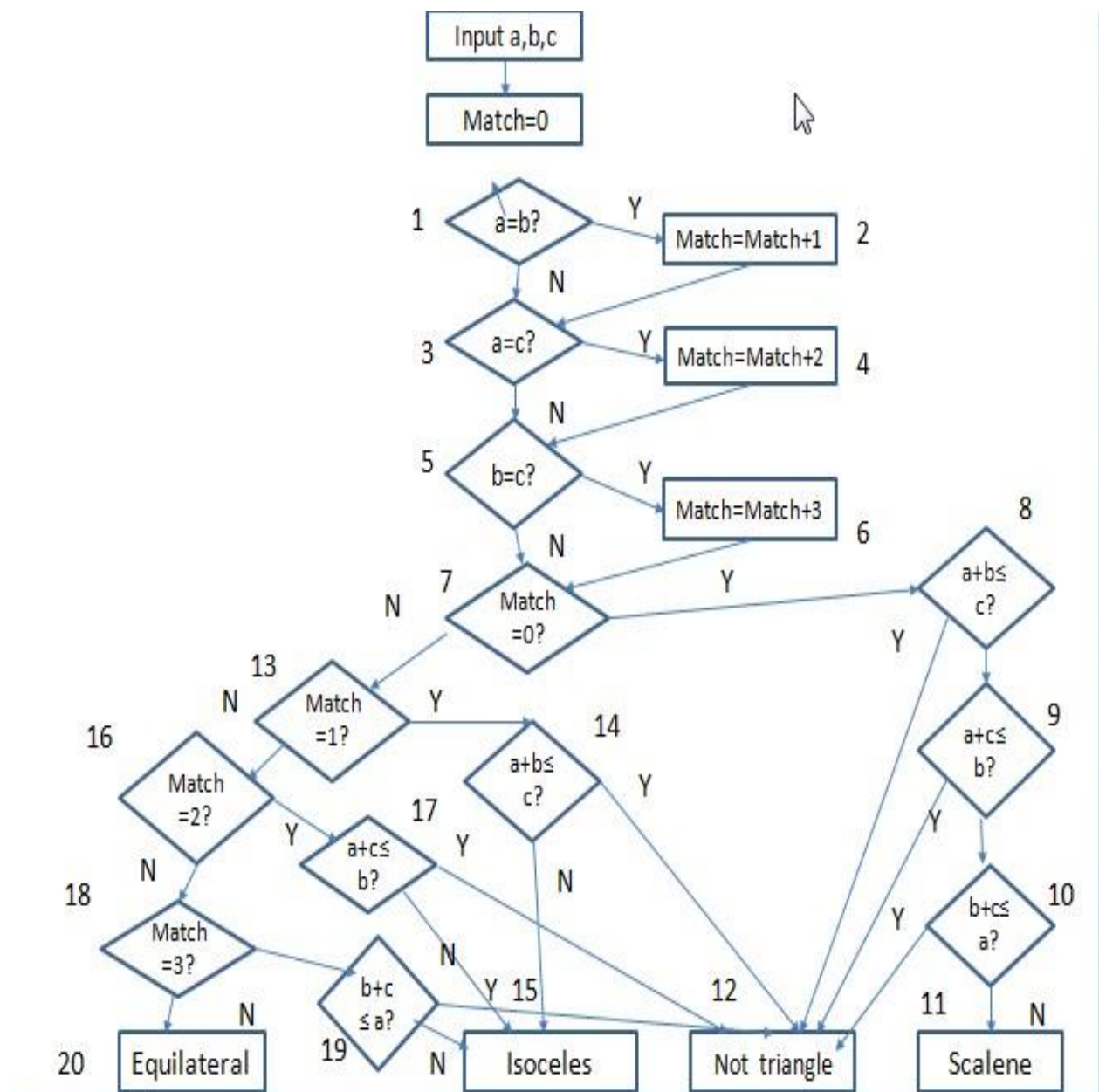


Figure: Flowchart for the traditional triangle program implementation

2.2.4 Structured Implementation

Dataflow diagram description of the triangle program. We could implement it as a main program with the four indicated procedures. the four procedures have been merged into one Turbo Pascal program

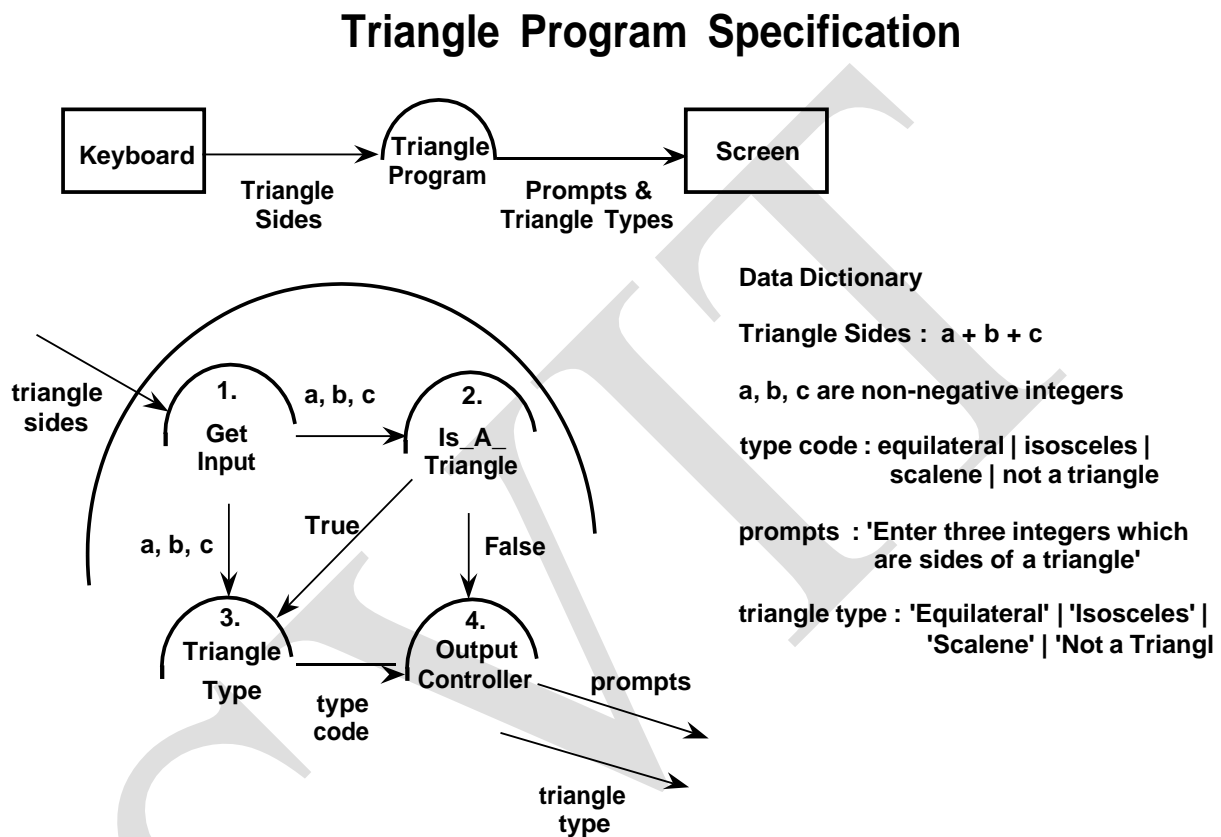


Figure 1.5 Dataflow Diagram for a Structured Triangle Program Implementation

Program triangle2 `Structured programming version of simpler specification `

```
Dim a,b,c As Integer
```

```
Dim IsATriangle As Boolean `
```

```
`Step1: Get Input
```

```
Output(—Enter the 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)`
`Step2:Is A Triangle?
If(a<b+c) AND (b<a+c) AND (c<a+b)
    Then IsATriangle=True
    Else IsATriangle=False
EndIf
`Step3:Determine Triangle Type
If IsATriangle
Then If(a=b) AND (b=c)
    Then Output (—Equilaterall)
    Else If( a!=b) AND (a!=c) AND (b!=c)
    Then Output (—Scalenell)
    Else Output (—Isocesesl)
    EndIf
EndIf
Else
    Output(—Not a Triangle)
EndIf
Endtriangle2

```

Program triangle3 `Structured programming version of improved specification`

```

Dim a,b,c As Integer
Dim c1,c2,c3,IsATriangle As Boolean `
`Step1: Get Input
Do
    Output(—Enter 3 integers which are the sides of the 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
    `Step3: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 (–Scalenell)
        Else Output (–Isoceles||)
        EndIf
    EndIf
    Else Output (–Not a Triangle||)
    EndIf `
    End triangle3
```

2.3 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 different kind of complexity — logical relationships among the input variables themselves

2.3.1 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 $1 \leq \text{month} \leq 12$, $1 \leq \text{day} \leq 31$, and $1812 \leq \text{year} \leq 2012$.

2.3.2 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 [Inglis 61], [ISO 91], so 1992, 1996, and 2000 are leap years, while the year 1900 is a common year. The NextDate function also illustrates a sidelight of software testing. Many times, we find examples of Zipf's Law, which states that 80% of the activity occurs in 20% of the space. Notice how much of the source code is devoted to leap year considerations.

2.3.3 Implementation

Program NextDate

Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer

Dim day,month,year As Integer

1. Output ("Enter today's date in the form MM DD YYYY")
2. Input (month,day,year)
3. Case month Of
4. Case 1: month Is 1,3,5,7,8,Or 10: '31 day months (except Dec.)
5. If day < 31
6. Then tomorrowDay = day + 1
7. Else
8. tomorrowDay = 1
9. tomorrowMonth = month + 1
10. EndIf
11. Case 2: month Is 4,6,9,Or 11 '30 day months
12. If day < 30
13. Then tomorrowDay = day + 1
14. Else
15. tomorrowDay = 1
16. tomorrowMonth = month + 1
17. EndIf
18. Case 3: month Is 12: 'December
19. If day < 31
20. Then tomorrowDay = day + 1
21. Else
22. tomorrowDay = 1
23. tomorrowMonth = 1
24. If year = 2012
25. Then Output ("2012 is over")
26. Else tomorrow.year = year + 1
27. EndIf
28. EndIf


```
29. Case 4: month is 2: 'February
30.   If day < 28
31.       Then tomorrowDay = day + 1
32.   Else
33.       If day = 28
34.           Then
35.               If ((year MOD 4)=0)AND((year MOD 400)≠0)
36.                   Then tomorrowDay = 29 'leap year
37.                   Else 'not a leap year
38.                       tomorrowDay = 1
39.                       tomorrowMonth = 3
40.               EndIf
41.           Else If day = 29
42.               Then tomorrowDay = 1
43.                   tomorrowMonth = 3
44.               Else Output("Cannot have Feb.", day)
45.           EndIf
46.       EndIf
47.   EndIf
48. EndCase
49. Output ("Tomorrow's date is", tomorrowMonth,
           tomorrowDay, tomorrowYear)
50. End NextDate
```

2.4 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.

2.4.1 Problem Statement

Rifle salesperson in the Arizona Territory sold rifle locks, stocks, barrels made by a gunsmith in Missouri. Locks cost \$45.00, stocks cost \$30.00, and barrels cost \$25.00. Salesperson had to sell least one complete rifle per month, and production limits are such that the most one salesperson could sell in a month is 70 locks, 80 stocks, and 90 barrels. Each rifle salesperson sent a telegram to the Missouri company with the total order for each town (s)he visits; salespersons visit at least one town per month, but travel difficulties made ten

towns the upper limit. At the end of each month, the company computed commissions as follows: 10% on sales up to \$1000, 15% on the next \$800, and 20% on any sales in excess of \$1800. The company had four salespersons.

The telegrams from each salesperson were sorted into piles (by person) and at the end of each month a data file is prepared, containing the salesperson's name, followed by one line for each telegram order, showing the number of locks, stocks, and barrels in that order. At the end of the sales data lines, there is an entry of --1|| in the position where the number of locks would be to signal the end of input for that salesperson. The program produces a monthly sales report that gives the salesperson's name, the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and finally his/her commission.

2.4.2 Discussion

This example is somewhat contrived to make the arithmetic quickly visible to the reader. It might be more realistic to consider some other additive function of several variables, such as various calculations found in filling out a US 1040 income tax form. This problem separates into three distinct pieces; the input data portion, in which we could deal with input data validation, the sales calculation, and the commission calculation portion.

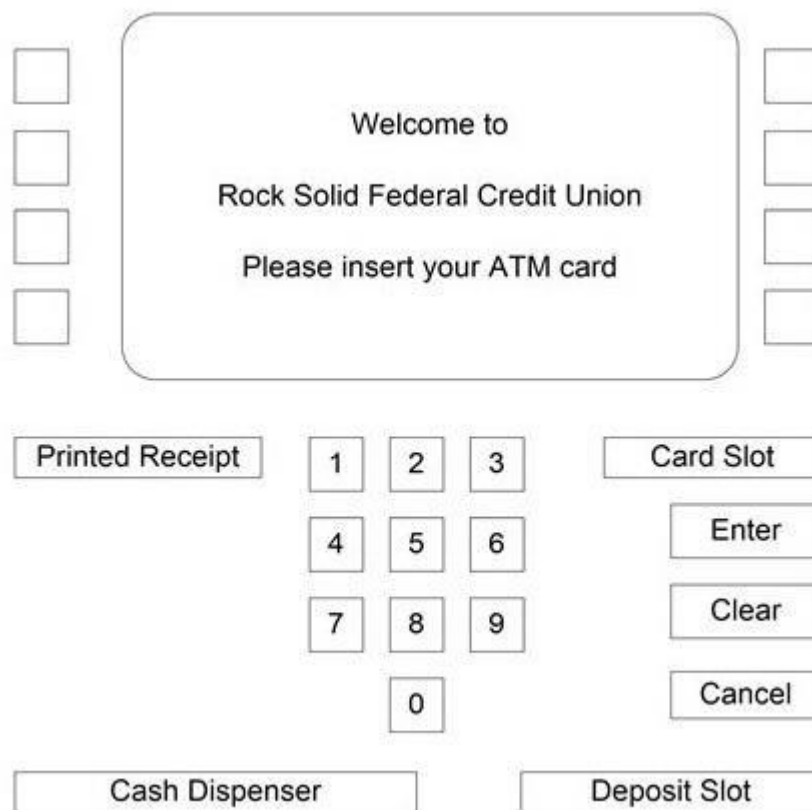
2.4.3 Implementation

1. Program Commission (INPUT,OUTPUT) `
2. Dim locks,stocks,barrels As Integer
3. Dim lockPrice,stockPrice,barrelPrice As Real
4. Dim totalLocks,totalStocks,totalBarrels As Integer
5. Dim lockSales,stockSales,barrelSales As Real
6. Dim sales,commission:REAL `
7. lockPrice=45.0
8. stockPrice=30.0
9. barrelPrice=25.0
10. totalLocks=0
11. totalStocks=0
12. totalBarrels=0`

```
13. Input(locks)
14. While NOT(locks=-1) `Input device uses -1 to indicate end of data
15.     Input (stocks,barrels)
16.     totalLocks=totalLocks+locks
17.     totalStocks=totalStocks+stocks
18.     totalBarrels=totalBarrels+barrels
19.     Input(locks)
20. EndWhile
21. Output(—Locks sold$,totalLocks)
22. Output(—Stocks sold$,totalStocks)
23. Output(—Barrels sold$,totalBarrels)`
24. lockSales=lockPrice*totalLocks
25. stockSales=stockPrice*totalStocks
26. barrelSales=barrelPrice*totalBarrels
27. sales=lockSales+stockSales+barrelSales
28. Output(—Total sales$,sales)
29. If (sales > 1800.0)
30.     Then
31.         Commission=0.10*1000.0
32.         Commission=commission+0.15*800.0
33.         Commission=commission+0.20*(sales-1800.0)
34. Else If(sales > 1000.0)
35.     Then
36.         Commission=0.10*1000.0
37.         Commission=commission+0.15*(sales-1000.0)
38. .Else
39. .commission=0.10*sales
40. EndIf
41. EndIf
42. Output (—Commission is $$,commission)`
43. End Commission
```

2.5 The SATM (Simple Automatic Teller Machine)

The ATM described here is minimal, yet it contains an interesting variety of functionality and interactions that typify the client side of client–server systems



2.5.1 Problem Statement

The SATM system communicates with bank customers via the 15 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. 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

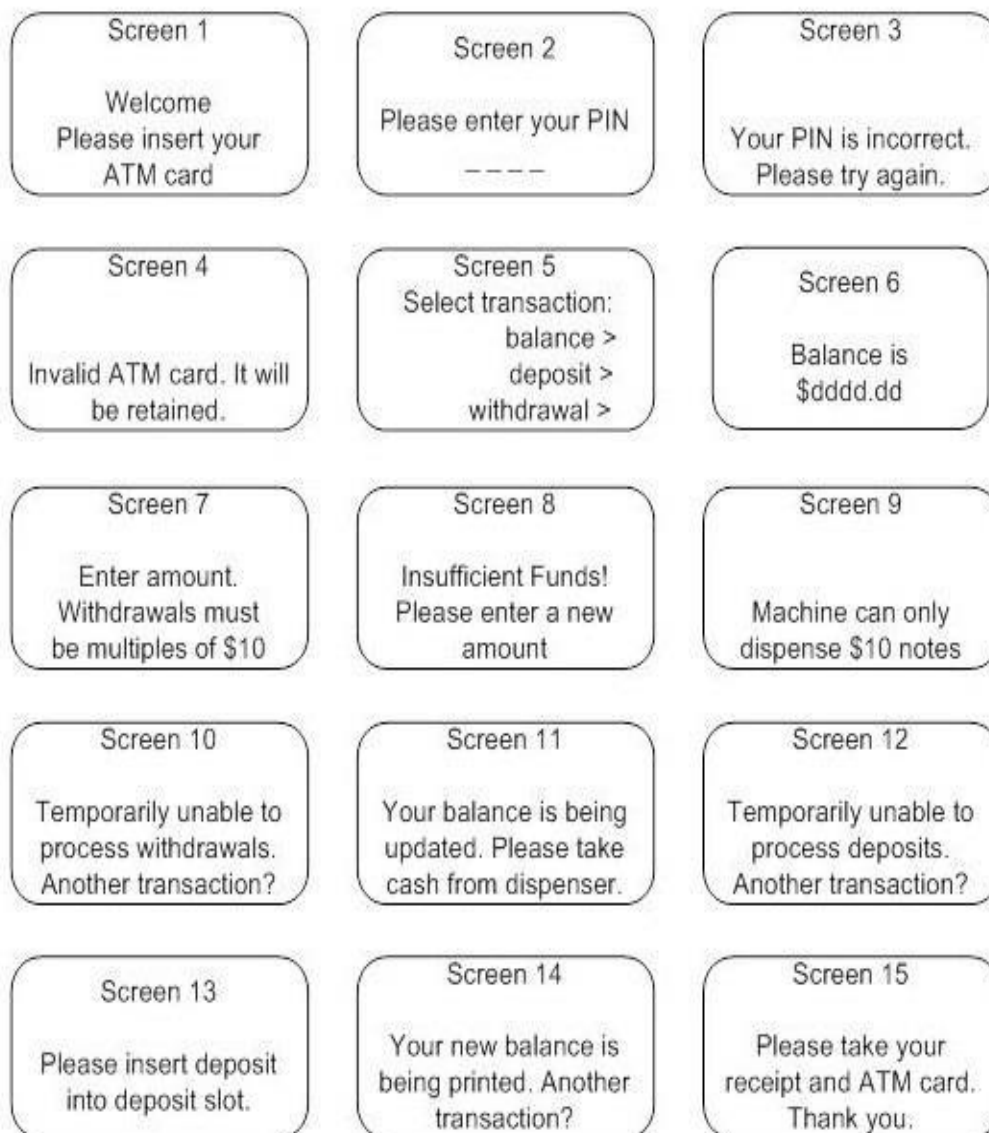


Figure: SATM Screens

2.6 The Currency Converter

The currency conversion program is another event-driven program that emphasizes code associated with a GUI. A sample GUI is shown in Figure

Currency Converter

US Dollar amount

Equivalent in...

- Brazil
- Canada
- European Community
- Japan

Figure: Currency converter graphical user interface

Figure: Currency Converter GUI

- 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 buttons (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. This example nicely illustrates a description with UML and an object-oriented implementation,

2.7 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