Module - 1
Basics of Software Testing

➢ Basic Definitions

- **Error:** people make errors. A good synonym is **mistake**. When people make mistakes while coding, we call these mistakes as **bugs**. Errors tend to propagate and requirements error may be **magnified** during **design** and **amplified** still more during **coding.**
- **Fault:** a fault is the **result of an error.** It is more precise to say that a fault is the **representation of an error**, where representation is the mode of **expression**, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. **Defect** is a good synonym for fault.

  Types of Fault:
  - **Fault of omission** occurs when we fail to enter correct information. (Some details are missed out in specifications and additional details are implemented)
  - **Fault of commission** occurs when we enter something into a representation that is incorrect. (Mentioned in Specifications but missed out while implementing) **or**

    Omission/commission:
    Omission - neglecting to include some entity in a module
    Commission - incorrect executable statement
- **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 loaded object code and 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 associated with a failure that alerts the user to the occurrence of failure.
- **Test:** testing is concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution.
- **Test Case:** test case has an identity and is associated with a program behavior. A test case also has a set of inputs and a list of expected outputs.

✓ **What is testing?**
- Software testing is a process used to identify the **correctness, completeness** and **quality** of developed computer software. **or**
- The process of **devising a set of inputs** to a given piece of software that will cause the software to **exercise some portion** of its code.

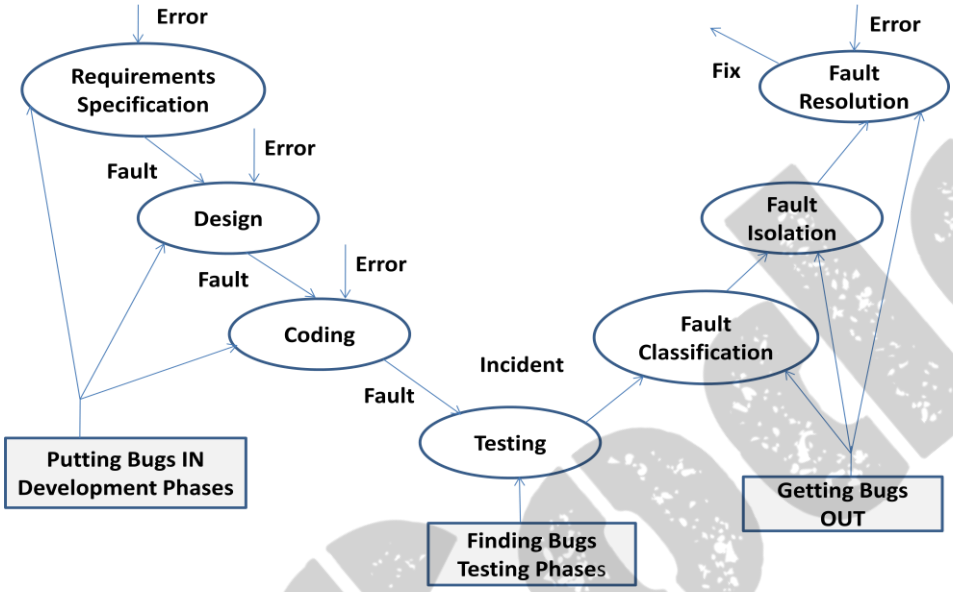
The developer of the software can then check that the results produced by the software are in accord with his or her expectations.

Note: Rigorous reviews are more effective, and more cost effective, than any other error-removal strategy, including testing. But they cannot and should not replace testing.

✓ **A Testing life cycle**

The **below figure** depicts life cycle model for testing. Three stages in which errors may be introduced resulting in faults that propagate through the remainder of the development process.

Figure: A Testing Life Cycle



Test cases occupy the central place in Testing. Testing can be subdivided into: **test planning, test case development, running test cases** and **evaluating test results.**

➢ Test cases

- Aim of testing is to determine a set of test cases. The below information should be in a test case.

Inputs: Pre-conditions (circumstances that hold prior to test case execution), Actual Inputs identified by some testing method.

Expected Outputs: Post- conditions and actual outputs

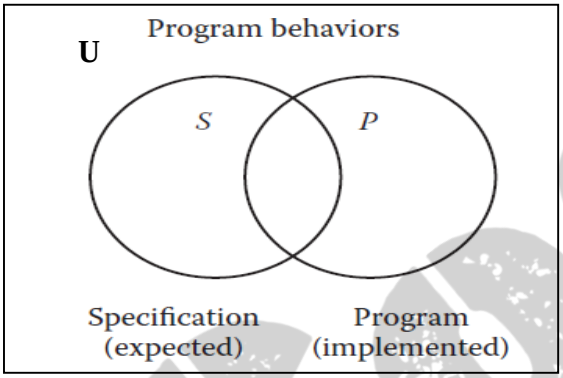
Typical Test Case Information (Contents of a Test Case)

Title, author, date,
Test case ID
Purpose
Pre-conditions
Inputs
Expected Outputs
Observed Outputs
Pass/Fail
Comments

➢ Insight from the Venn Diagram

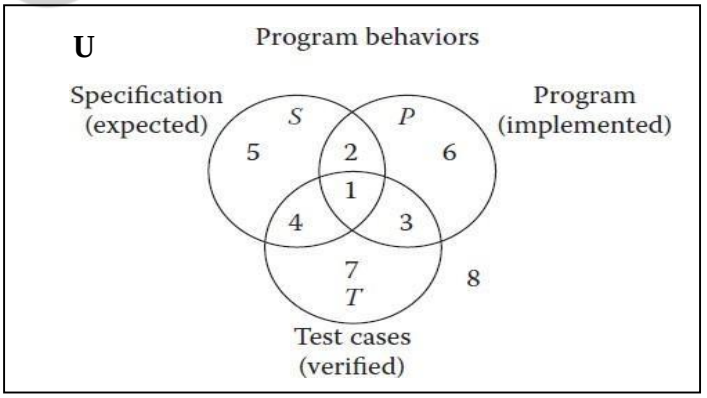
- A **Venn diagram** is made up of two or more overlapping circles. It is often used in mathematics to show relationships between sets. In software testing, Venn Diagrams are used to represent **input/output relationships** for the system.
- Testing is basically concerned with program behavior. Behavioral view considers what the **program does** and Structural view considers what the **program is.** Behavioral view is **orthogonal** (perpendicular) to structural view. One difficulty for testers is that documents are written for developers, so **emphasis** is on structural view instead of behavioral information. Simple **Venn** diagrams clarify several issues.

  Figure: Specified and Implemented program behaviors



- Consider a **universe U** of program behaviors in the above Venn diagram. Here, program and its specifications are given. We can see from the above diagram that **Set S** contains **specified behaviors** among all possible program behaviors and **set P** contains all those behaviors **actually programmed.** The intersection of **S and P** is the portion where behaviors that are both **specified and implemented.** This is the very good view of testing that determines the extent of program behavior.
- In the **figure below**, consider the relationships among **S, P, and T** (Test cases). We can find, there may be specified behaviors that are not tested correspond to the **regions 2 and 5,** specified behaviors that are tested correspond to the **regions 1 and 4,** and test cases that correspond to unspecified behaviors in **regions 3 and 7.**

  Figure: Specified, implemented and tested behaviors



- Similarly, there may be programmed behaviors that are not tested correspond to the **regions 2 and 6,** programmed behaviors that are tested correspond to the **regions 1 and 3**, and test cases that correspond to unprogrammed behaviors in **regions 4 and 7.**
- All regions are important. If specified behaviors exist for which **no test cases** are available, the testing is **incomplete.** If some test cases correspond to unspecified behaviors, then some possibilities arise: test case is **unwarranted or specification is deficient** or tester wishes to determine that specified nonbehavior does not occur.

➢ Identifying test cases

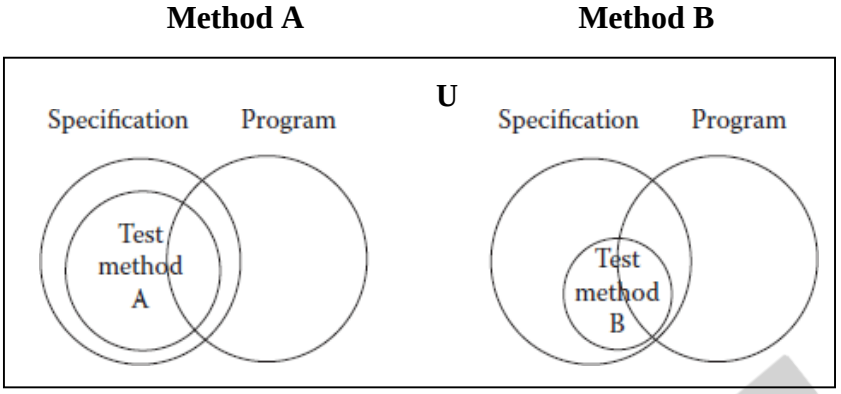
- **Two fundamental** approaches known as **Functional** and **Structural** testing are used to identify test cases.
- Each of these has different test case identification methods referred to as **testing methods.**

✓ Functional Testing

- Functional Testing is based on the view that **any program** can be considered to be a **function** that **maps values** from **input domain** to values in the **output range**. This notion is used when systems are considered to be **black boxes.** In black box testing, the content or **implementation of a black box is not known** and its function is understood completely in terms of its **inputs and outputs.**
- **Example:** Most people successfully **operate automobiles** with only black box knowledge. Here, only specifications of software are used to identify test cases.
  Figure: **An Engineer's black box.**

- **Figure below** shows result of **test cases** identified by two functional methods. **Method A** identifies **larger set** of test cases than M**ethod B**. For both the methods, the set of test cases is completely contained within the set of **specified behavior**. It is **hard** to identify behaviors that are **not specified.**
- **Black Box -** A strategy in which testing is based on **requirements and specifications**

Figure: Comparing functional test case identification methods



✓ **Advantages of Black Box Testing / Black Box Test cases**

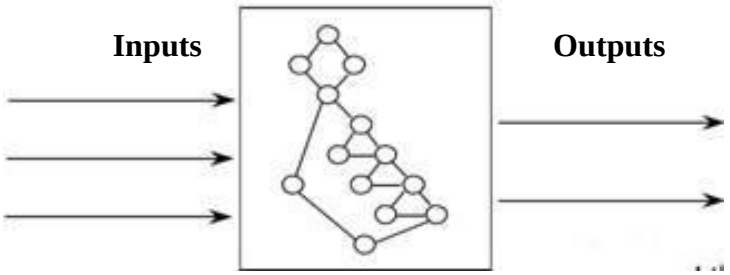
- The test is **unbiased** because the **designer** and the **tester** are independent of each other.
- Test cases are independent of how the software is implemented, so if the implementation changes, the **test cases are still useful.**
- The tester does not **need knowledge** of any specific **programming languages.**
- The test is done from the point of view of the user, **not the designer.**
- Test cases can be designed as soon as the **specifications are complete** or in parallelwith implementation.

✓ **Disadvantages of Black Box Testing/ Black Box Test cases**

- The test can be **redundant** if the software designer has already run a test case.
- Significant **redundancies** may exist among **test cases.**
- The test cases are difficult to design and compounded by the possibility of **gaps** of untested software.
- Testing every possible **input stream** is unrealistic because it would take an inordinate amount of time. Therefore, many **program paths** will go **untested.**

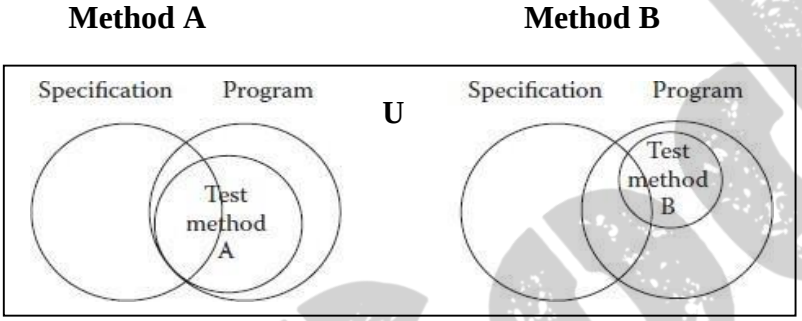
✓ Structural Testing

- Structural Testing is **another approach** to test case identification. It is also called as **White Box, Clear Box, Glass box** and **Open box** testing. Function is understood only in terms of its **implementation** and used to identify test cases.



- With the knowledge of **graph theory**, the tester can rigorously describe exactly what is testing. Structural testing provides the definition by itself and it uses the **test coverage metrics.** The test coverage metrics provide a way to explicitly state the **extent to which software has been tested** and this makes testing management more meaningful. The **figure above** depicts the **path flow** testing.
- **Figure below** shows the result of **test cases** identified by **two Structural methods**. **Method A** identifies **larger set** of test cases than **Method B**. In both methods, the set of test cases is completely contained within the set of **programmed behavior.**
- **White Box:** A strategy in which testing is based on **internal parts, structure,** and **implementation.**

Figure: Comparing structural test case identification methods

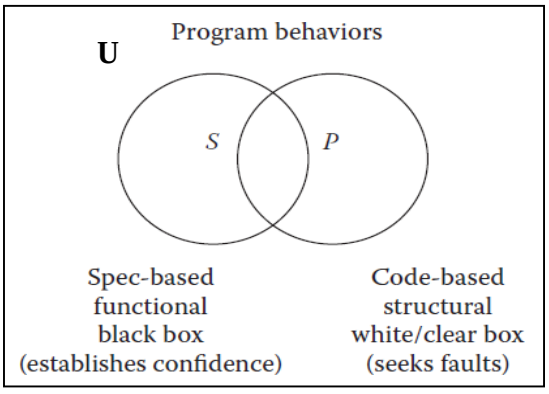


✓ The Functional versus Structural Debate

- Functional testing uses only the **specifications** to identify test cases.
- Structural testing uses the **program source code** as the basis of test case identification.
- Neither approach **alone** is sufficient.
- Consider program behaviors, if all **specified have not been implemented**, then structural test cases will **never be able** to recognize this.
- Program implements behaviors that **have not been specified**, this will never be revealed by **functional test cases**.

Both the approaches are needed to provide the **confidence. Here,** Problems can be **recognized** and **resolved.** The Venn diagram below provides final insight.

Figure: Sources of test cases



✓ **Advantages of White Box Testing are mentioned below:**

- All the features and functionality within the application can be tested. Testing can be started at the **very initial stage.** Tester **does not need** to **wait for interface** or GUI to be ready for testing.
- Can reduce to **number of test cases** to be executed during black box testing.
- Helps in checking **coding standards and optimizing** code.
- Extra code resulting in **hidden defects** can be removed.
- Reason of **failure** can be known.
- Identifying **test data is easy** because **coding knowledge.**

✓ **Disadvantages of White Box Testing are mentioned below:**

- Tester should be **highly skilled** because should have the knowledge of coding/implementation.
- **Cost** of tester is very **high**.
- White Box testing is very **complex**.
- It is **not possible** to look into each piece of code to find out **hidden errors**.
- Test cases **maintenance** can be **tough** if the implementation changes very frequently.
- Since White Box Testing is closely tied with the application being testing, **tools** to cater to every kind of implementation/platform may **not be readily available**.
- **Exhaustive testing** of larger system is **not possible**.

➢ Error and fault Taxonomies

Fault classified by severity

1	**Mild**	Misspelled word
2	**Moderate**	Misleading or redundant information
3	**Annoying**	Truncated names, bill for $0.00
4	**Disturbing**	Some transactions(s) 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

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

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 of <=)

Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Interface faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch(type, number)
Incompatible types
Superfluous inclusion

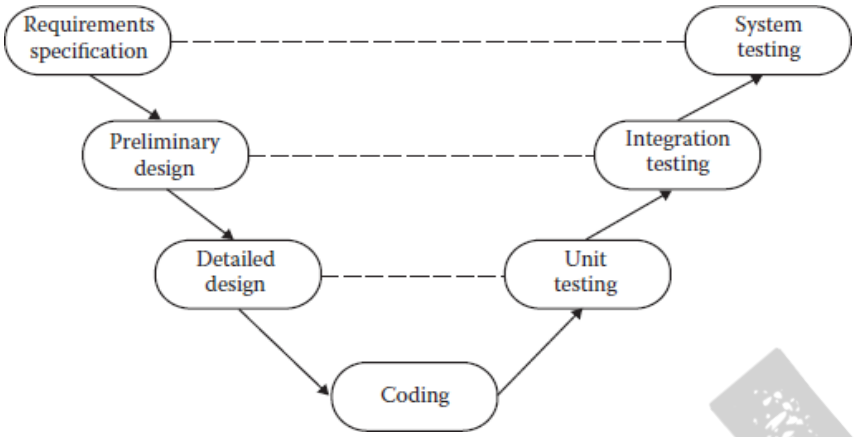
Data Faults

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

➢ Levels of Testing

- **Figure below** shows the Levels of abstraction and testing in the **waterfall model**. Levels of testing echo the levels of abstractions found in the waterfall model of the **Software Development Life Cycle (SDLC)**.
- It helps to **identify distinct levels of testing** and for **clarification of objectives** of each level. In terms of functional testing, the three levels of definition correspond directly to three levels of testing – **system, integration**, and **unit testing**.
- A practical relationship exists between **levels of testing** versus **functional and structural** testing.
- **Structural testing** is more appropriate at **unit level** and **functional testing** is at more appropriate at **system level.**

Figure: Levels of Testing



➢ Software quality

Software quality is a **multidimensional quantity** and is **measurable**.

✓ Quality attributes

There exist **several measures** of software quality. These can be divided into **static** and **dynamic** quality attributes.

- **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**.
  - **Correctness** refers to the **correct operation of an application** and is always with reference to some artifact.
  - **Completeness** refers to the **availability of all features** listed in the requirements, or in the user manual. Incomplete software is one that does not fully implement all features required.
  - **Consistency** refers to **adherence to a common set of conventions and assumptions.** For **example,** all buttons in the user interface might follow a common color coding convention.
  - **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

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

➢ Requirements, Behavior and Correctness

- Any software is designed in response to **requirements of the environment**. **Example:** Two Requirements are given below and each leads to two different programs.
  - **Requirement 1:** It is required to write a program that **inputs two integers** and **outputs the maximum** of these.
  - **Requirement 2**: It is required to write a program that inputs a **sequence of integers** and **outputs the sorted version** of this sequence.
- Consider **Requirement 1:** The expected output of **max** when the input integers are 13 and 19 is easily determined as 19. Suppose now that the tester wants to know if the **two integers to be on the same line** followed by a carriage return, or **on two separate lines** with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question. This requirement illustrates **Incompleteness.**
- Consider **Requirement 2**: It is not clear whether the input sequence is to be sorted in **ascending** or in **descending** order. The **behavior of sort program** written to satisfy this requirement will depend on the decision taken by the programmer. This is called as **ambiguity.**

✓ Input domain and Program Correctness

- A program is correct if it **behaves as desired** on all possible inputs. The set of all possible inputs to a program **P** is known as the **input domain** or **input space** of **P**.
- Using **Requirement 1**, we find the input domain of **max** to be the set of all pairs of integers where each element in the pair integers is in the range **-32,768 to 32,767.**
- Using **Requirement 2** it is not possible to find the input domain for the sort program.
- **Modified Requirement 2:** It is required to write a program that inputs a sequence of integers and outputs the integers sorted in either **ascending or descending** order. The order of the output sequence is determined by an input request character which should be **"A"** when an ascending sequence is desired, and **"D"** otherwise. While providing input to the program, the request character is input first followed by the sequence of integers to be sorted and 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. **Examples** are:

  <A -5 56 32 16 . >
  <D 79 103 -2 0 29 . >
  <A . >

✓ Valid and Invalid Inputs

- The modified requirement for sort mentions that the request characters can be **"A" and "D"**, but fails to answer the question **"What if the user types a different character?"** When using **sort** it is certainly possible for the user to type a character other than **"A"** and **"D"**. Any character other than **"A"** and **"D"** is considered as **invalid input** to **sort**.

The requirement for **sort** does not specify what action it should take when an **invalid** input is encountered. **Example:** < E 7 19 . > Now the sort program enters into infinite loop.

- Further the **modified input domain** consists of pair of values in which the first value is the **request character** and the second is the **sequence of integers combined** with **invalid characters** terminated by a period. **Example:** < D 7 **9F** 19 . >. From this it is assumed that invalid characters are possible inputs to sort program.
- In the next step, it is advisable to separate **valid and invalid inputs** and can be used for testing.

➢ Correctness versus Reliability

✓ Correctness

- **Correctness** is the process of **testing a program** on **all elements** in the input domain. In most cases this is impossible to accomplish. Thus, correctness is established via **mathematical proofs** of programs.
- While **correctness** attempts to establish that the program is **error free**, **testing** attempts to find if there are **any errors** in it.
- Testing, debugging and the error removal processes together increase our **confidence** in the correct functioning of the program under test.

✓ Reliability

- The **reliability** of a program **P** is the probability of its **successful execution** on a randomly selected element from its input domain. **Example:** Consider a program **P** whose inputs are {< (0, 0) (-1, 1) (1, -1)>}. 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 $\frac{2}{3}$.

✓ Program Use and the Operational profile

- An operational profile is a **numerical description** of how a program is executed and used.
- **Example:** Consider a **sort program**, on any given execution allows any one of two types of input sequences. Sample operational profiles for sort are as follows.

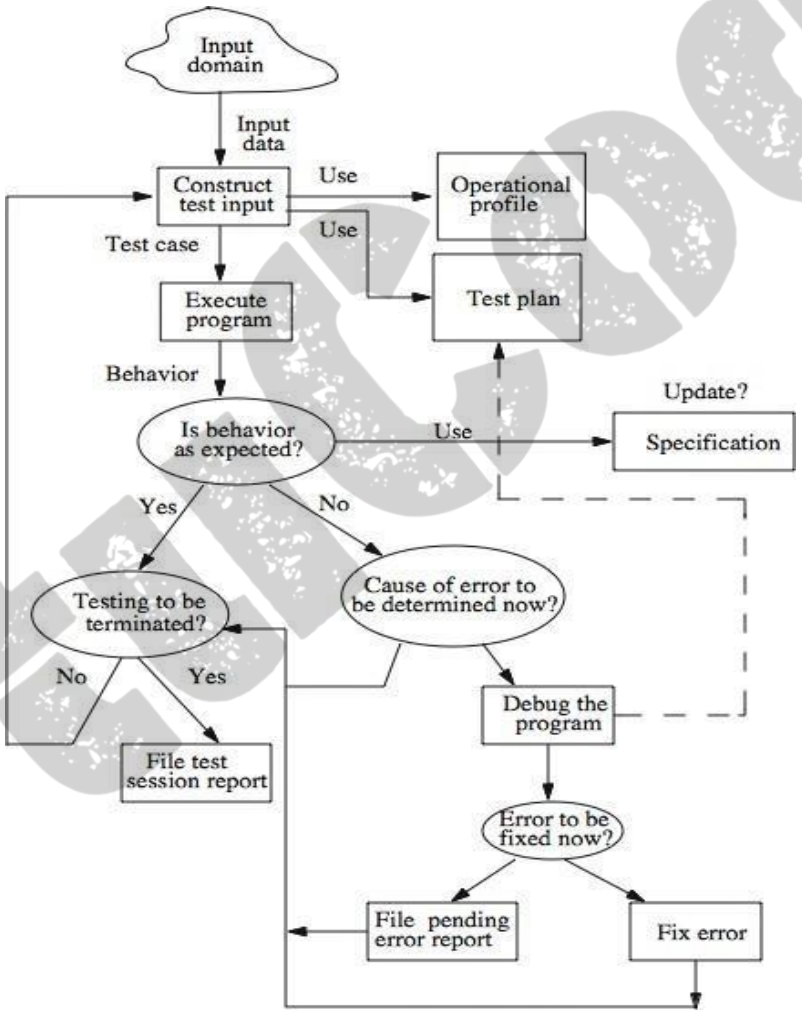
Operational Profile #1	
Sequence	Probability
Numbers Only	0.9
Alphanumeric strings	0.1

Operational Profile #2	
Sequence	Probability
Numbers Only	0.1
Alphanumeric strings	0.9

➢ Testing and debugging

Testing is the process of determining if a **program behaves as expected**. When testing reveals an error, the process used to **determine the cause of this error** and **to remove it**, is known as **debugging.** The **below figure** shows that the testing and debugging are often used as two related activities in a cyclic manner.

Figure: A test and debug cycle



✓ Preparing a Test plan

- A **test cycle** is often guided by a test plan. Test Plan for **sort program** is given below. The sort program is to be tested to meet the requirements given earlier.
- Specifically, the following needs to be done.

1. Execute sort on at least two input sequences, 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 inputs 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.

✓ Constructing Test Data

- A **test case** is a pair consisting of test data to be **input** to the program and the **expected output.** The **test data** is a set of values, one for each input variable. A **test set** is a collection of **zero or more** test cases. **Sample test cases** for **sort are,**

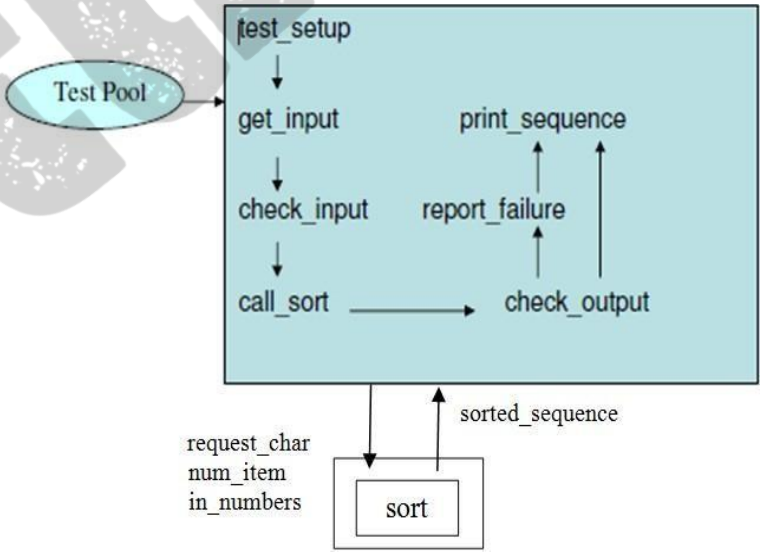
  Test data: <A 12 -29 32 >
  Expected output: -29 12 32

  Test data: <D 12 -29 32 >
  Expected output: 32 12 -29

✓ Executing the Program

- A tester will 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.
- **Example:** The test harness shown in **below figure reads** an input sequence, **checks** for its correctness, and then **calls** sort. The sorted array **sorted_sequence** returned by sort is printed using **print_sequence**. The test cases are assumed to be in the **Test Pool.**

Figure: A simple test harness to test the sort program



- In preparing this test harness the assumptions made are:
  (a) **sort** is coded as a **procedure.**
  (b) The **get_input** procedure **accepts/reads** the variables in the sequence as **request_char, num_items** and **in_numbers.**
  (c) The **input is checked** prior to calling **sort** by the **check_input** procedure.
- The **test_setup** procedure is invoked first to set up the test that includes **identifying and opening** the file containing tests.
- **check_output** procedure serves as the oracle that **checks** if the **program under test behaves correctly.**
- **report_failure** is invoked when the output from **sort is incorrect.**
- **print_sequence** prints the sequence generated by the sort program. This also can be saved in file for subsequent examination.

✓ Specifying Program behavior

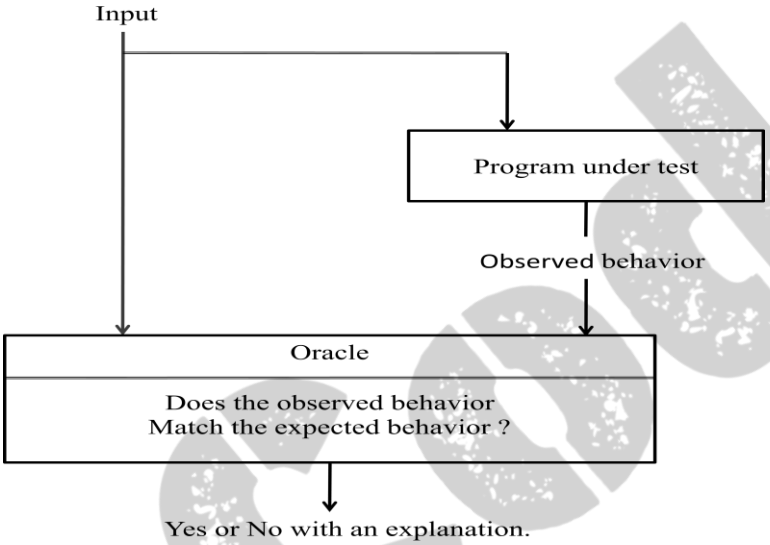
- There are several ways to define and specify program behavior such as **plain natural language** and a **state diagram.**
- The **state** of the program is the **set of current values** of all its **variables** and an **indication** of **which statement** in the program is to be **executed next.**
- One way is 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 **identifier associated** with the next program statement**.**
- In the case of **assembly language programs** the location of the control can be specified more precisely by giving the value of the **program counter**.
- **Example:** Consider a program given below to find maximum of two numbers and store in Z. Here if X is less than Y, line 4 will be executed, if not line 6 will be executed.

```
1. integer X, Y, Z;
2. input (X, Y);
3. if (X<Y)
4. {Z=Y;}
5. else
6. {Z=X;}
7. endif
8. output(Z);
9. end
```

✓ Assessing the Correctness of Program Behavior

- Assessing the correctness of program behavior includes **two steps**. In the **first step,** the **observation of the behavior** is done. In the **second step, analysis** of the **observed behavior** is done to check if the program is correct or not.
- The **entity** that performs the task of **checking the correctness** of the observed behavior is known as an **oracle.** B**elow figure** shows the **relationship** between the program **under test** and **the oracle**. An **oracle** is a testing **software** designed to check the **behavior** of other programs.

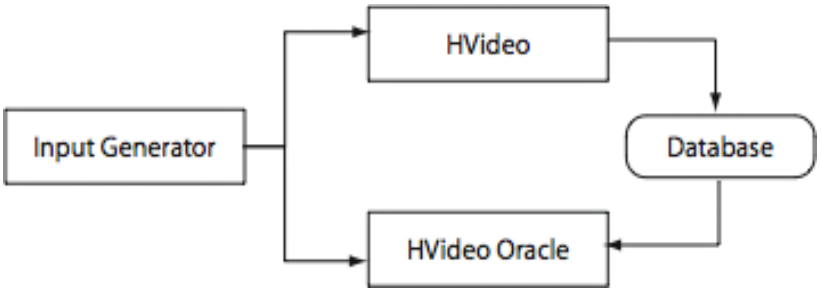
Figure: Relationship between the program under test and the oracle



✓ 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 HVideo** to keep track of home videos, which operates in two modes. In the **data entry mode** it displays a screen in which the user types in information about a **DVD** such as **title, comments** and **created date etc.,** and **stored** in **database.** In **search mode** the program displays a screen into which a user can type **some attribute** of the **DVD** to be searched.
- To test **HVideo program** we need to create an **oracle** that checks whether the **HVideo program** functions correctly in **data entry** and **search modes**. In **below figure**, the **input generator** generates a data entry request. After completion of request, HVideo returns **control** to the input generator. The input generator now **requests the oracle** to **test** if, HVideo (program) performed its **task correctly** on the input given.
- 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. Similarly it is carried for search mode also.

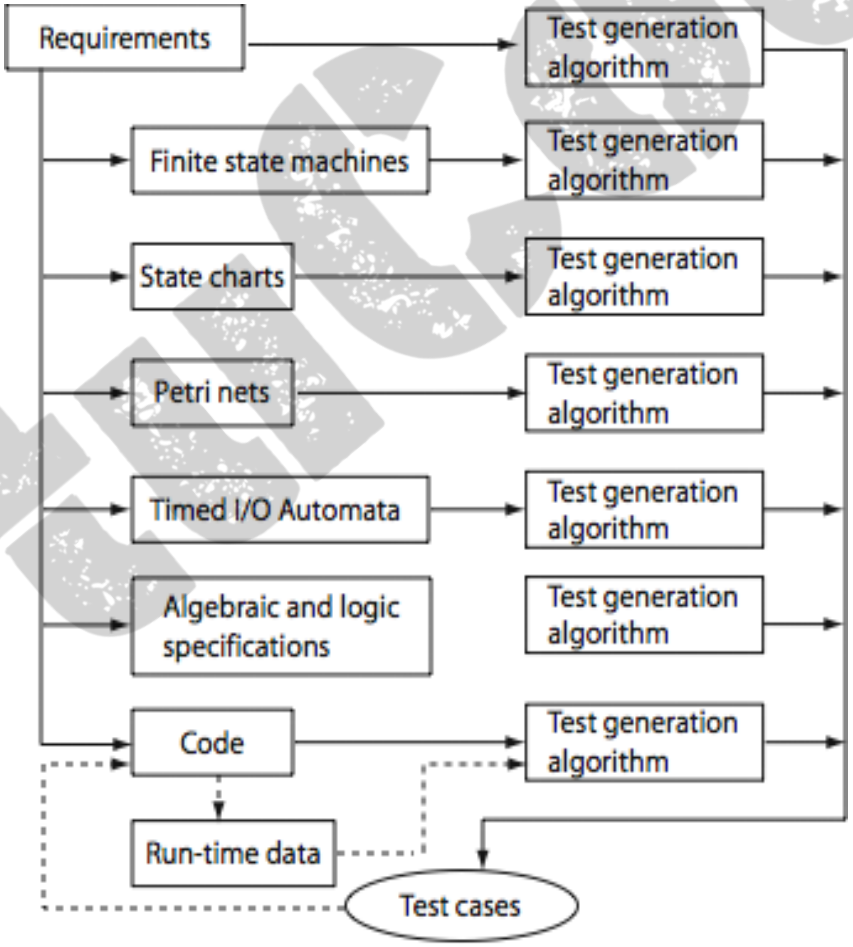
Figure: Relationship between an input generator, HVideo and its oracle



➢ Test- generation Strategies

- One of the key tasks in software testing is **generation of test cases**. Any form of test case generation uses a **source document/ requirement document**. In most of the test methods the source document resides in the mind of the tester who generates tests based on the knowledge of the requirements. The **below figure** summarizes several strategies for test case generation.

  Figure: Requirements, models and test generation algorithms



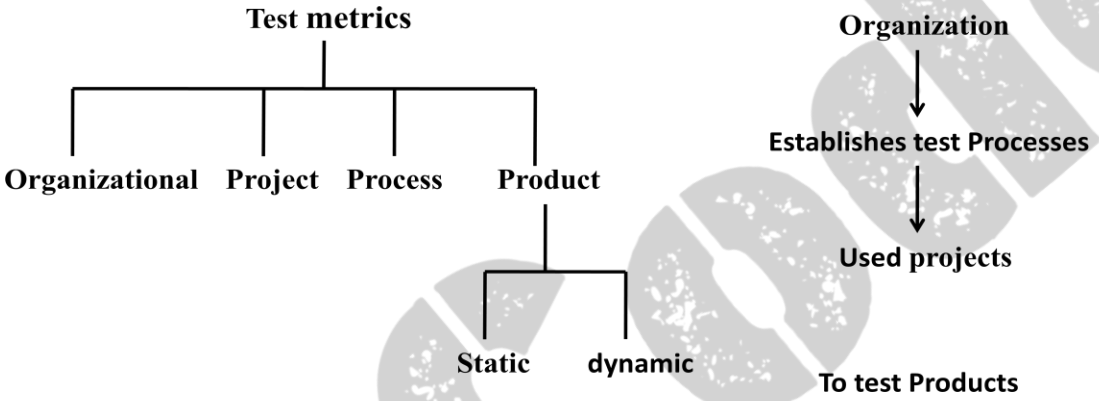
- The **top row** captures techniques that are applied directly to the requirements. Another set of strategies falls under the category of **model-based test generation**. These strategies require that a **subset of the requirements** be modeled using a **formal notation.**

- Languages based on **predicate logic** as well as **algebraic languages** are also used to express subsets of requirements. This model is also known as **specification** of the subset of requirements.
- **Finite State Machines, state charts, timed I/O automata** and **Petri nets** are some of the formal notations for modeling various subsets of the requirements. **Code based test generation** generate tests directly from the code.

➢ 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 relationships



- There are **four** general core areas that assist in the design of metrics. They are **schedule, quality, resources** and **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**, **man power** and **test execution**.
  - **Size-related metrics:** Measure size of **various objects** such as the **source code** and number of **tests in a test suite**.

✓ Organizational metrics

- Metrics at the level of an organization are **useful** in overall project **planning** and **management**. **Example:** the number of **defects** reported after product release, average of products **developed** and **marketed** by an organization is a useful metric of **product quality.**
- 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.

✓ Project metrics

- Project metrics relate to a **specific project**. The **I/O device testing** project or **compiler design projects** are the examples. 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**.
- Another project metric is the **ratio** of **number of successful tests to the total number of tests** in the system test phase.

✓ Process metrics

- Every project uses some test process. The b**ig-bang** approach is well suited for small single person projects. The **goal** of a process metric is to **assess** the **goodnes**s of the process.
- When test process consists of several phases like **unit test, integration test** and **system test,** one can measure how many defects were found in each phase. The defects should not be carried from one phase of testing to other phase. Otherwise the cost of testing will become high.

✓ Product metrics: Generic

- Product metrics relate to a **specific product**. They are useful in making decisions related to the product.
- For example "Should the product be released for use by the customer?"
- Product **complexity** can be measured with **two types** of metrics. They are **Cyclomatic complexity** and **Halstead** metrics.
- Given CFG (Control Flow Graph) **G** of Program **P** containing **N** nodes, **E** edges and **p** connected procedures, Cyclomatic complexity is computed as **V(G) = E-N+2p**
- **Larger values** of V(G) tends to **higher program complexity** and program is more difficult to understand and test.
- **Halstead complexity measures** are software metrics introduced by Maurice Halstead in a book titled **Elements of Software Science**.
- The **table below** lists some of the **software science metrics.** Using program size **S** and effort **E**, the following estimator has been proposed for number of errors **B** as **B= 7.6** $E^{0.667} S^{0.33}$

Halstead Measures of Program complexity and effort

Measure	Notation	Definition
Operator count	N_1	Number of operators in a Program
Operand count	N_2	Number of operands in a program
Unique operators	n_1	Number of unique operators in a program
Unique operands	n_2	Number of unique operands in a program

Program vocabulary	N	$n_1 + n_2$
Program size	N	$N_1 + N_2$
Program Volume	V	$N * \log_2 N$
Difficulty	D	$2/n_1 * n_2/N_2$
Effort	E	$D * V$

✓ Product metrics : Object-Oriented (OO) Software

- **Below table** lists a sample of product metrics for **Object-Oriented (OO)** and other applications.

Metric	Meaning
Reliability	Probability of failure of a software product with respect to a given operational profile in a given environment
Defect density	Number of defects per KLOC (thousands lines of code)
Defect severity	Distribution of defects by their level of severity
Test coverage	Fraction of testable items, e.g. basic blocks, covered. Also a metric for test adequacy or goodness of tests
Cyclomatic complexity	Measures complexity of a program based on its CFG
Weighted methods per class	$\sum_{i=1}^{n} c_i$ c_i is the complexity of method i in the class under consideration
Class coupling	Measures the number of classes to which a given class is coupled
Response set	Set of all methods that can be invoked, directly and indirectly, when a message is sent to object O
Number of children	Number of immediate descendants of a class in the class hierarchy

✓ Progress monitoring and trends

- **Metrics** are often used for monitoring **progress**. This requires making **measurements** on a regular basis over time. Such measurements offer **trends**.
- **Example:** When a **browser** is in system testing phase one could measure the **cumulative number** of **defects** found and plot these over time. Such plot will rise over time. It is also required to show the **saturation** indicating that the product is reaching **stability**.

✓ Static and dynamic metrics

- **Static** metrics are those computed **without** having to **execute** the product.
  Example: Number of testable entities in an application.
- **Dynamic** metric requires **code execution**. **Example:** Number of testable entities actually covered by a test suite.

✓ 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".
- **Two ways** to measure testability of a product are **static** testability metrics and **dynamic** testability metrics. Software complexity is one of the static testability metric. The **more complex** an application the **lower the testability,** i.e., **higher effort** is required to test it.
- Dynamic metrics for testability includes various **code based** coverage criteria. **Example:** when it is difficult to generate tests that satisfy the **statement coverage** criterion then it is considered to have **low testability.**

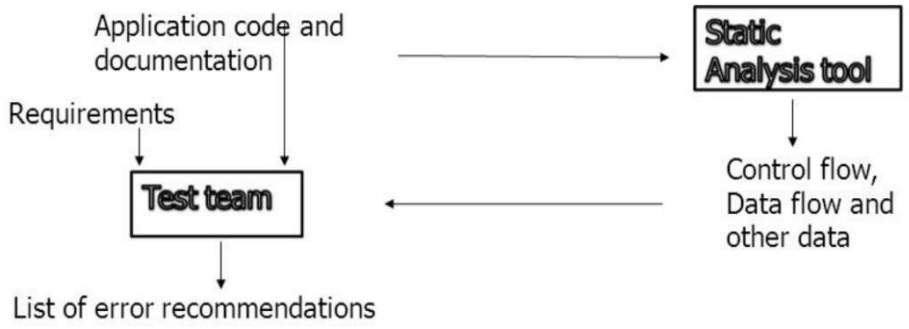
➢ Testing and Verification

- Program **verification** aims at proving the **correctnes**s of programs by showing that it contains **no errors**. Verification aims at showing that a given program works for **all possible inputs** that satisfy a **set of conditions** and **testing** aims to show that the given program is **reliable** i.e., it has **no errors**.
- Program **verification** and **testing** are considered as **complementary** techniques. In practice, one may **discard** program **verification** but **not testing**. Testing is not a perfect technique when a program still contains errors even after the successful execution of set of tests, but this increases the **confidence** about the **correctnes**s of the application.
- Verification promises to verify that a program is free from errors but a closer look reveals that it has its own weakness. The person who verified a program might have made a mistake in the verification process there might be an **incorrect assumption on the input conditions, incorrect assumptions on the components that interface** with the program and so on. Thus **neither verification nor testing** is a **perfect** technique for proving the correctness of programs.

➢ Static Testing

- Static testing is carried out **without executing** the application under test. It is useful in discovery of **faults in the application, ambiguities and errors in the requirements** at a relatively low cost.
- This is carried out by an individual who **did not write the code or by a team** of individuals. A sample process of static testing is illustrated in **below figure**. 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.

Figure: Elements of Static testing



✓ Walkthroughs

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

✓ 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.
- Code inspection is carried out by a team. The team works according to **Inspection plan** that consists of the following elements,
  - **i.** Statement of purpose
  - **ii.** Work product to be inspected, this includes code and associated documents needed for inspection.
  - **iii.** Team formation, roles, and tasks to be performed.
  - **iv.** Rate at which the inspection task is to be completed
  - **v.** 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** are assigned roles of
  - **Moderator:** in charge of the process and leads the review.
  - **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.
  - **Recorder:** records any errors discovered or issues to be looked into.
  - **Author:** actual developer of the code.

✓ Use of static code analysis tools in static testing

- **Static code** analysis tools can provide **control flow** and **data flow** information. The control flow information presented in terms of a CFG (Control Flow Graph), is helpful

to the inspection team which allows the determination of the flow of control under different conditions. A CFG can be annotated with data flow information to make a data flow graph. This information is **valuable to the inspection team** in understanding the code as well as pointing out possible defects.

- Two commercially available **static code analysis tools** are **Purif**y from IBM Rationale and **Klockwork** from Klockwork. **LAPSE** (Light weight Analysis for Program Security in Eclipse) is an **open source tool** for analysis of Java programs.

✓ Software complexity and Static testing

- Several parameters are considered in making decision about which of the several modules should be inspected first. One such parameter is **module complexity**. A more complex module should be given with higher priority during inspection because they may contain more errors than less complex modules.
- Static analysis tools compute complexity metrics using one or more metrics discussed already.

Examples

➢ Generalized pseudo code

- Pseudo code provides a **"language neutral"** way to express logic.
- Program **source code units** can be interpreted either as **traditional components** (procedures and functions) or as **object oriented components** (classes and objects.)
- Terms such as **expression, variable list, and field description** are used with no formal definition

➢ The Triangle Problem

The triangle program is the most widely used example in software testing literature.

✓ Problem Statement

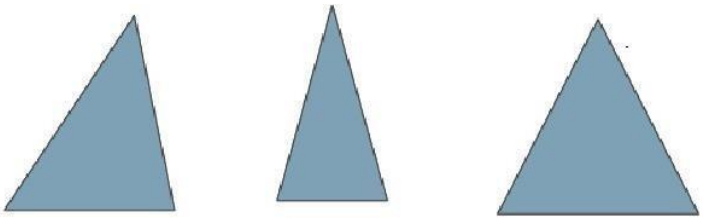
- **Simple version**: The triangle program accepts three **integers, a, b, and c,** as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: **Equilateral, Isosceles, Scalene, or Not a Triangle**.
- **Improved version**: The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

  c1. 1 <= a <= 200 **c4**. a + b > c

  c2. 1 <= b <= 200 **c5**. a + c > b

  c3. 1 <= c <= 200 **c6.** b + c > a

- The output of the program is the type of triangle determined by the three sides: **Scalene, Isosceles, Equilateral** or **Not A Triangle**. If an input value fails any of conditions **c1, c2 or c3,** then program outputs message as, for example, **"Value of b is not in the range of permitted values".** If values of a, b, and c satisfy conditions c1, c2, and c3, then one of four mutually exclusive outputs is given:
  1. If **no pair of sides** is **equal,** the program output is **Scalene.**
  2. If **exactly one pair** of sides is **equal,** the program output is **Isosceles.**
  3. If all **three sides are equal**, the program output is **Equilateral.**
  4. If any of **conditions c4, c5 and c6** is not met, the program output is **Not A Triangle**.



✓ Discussion

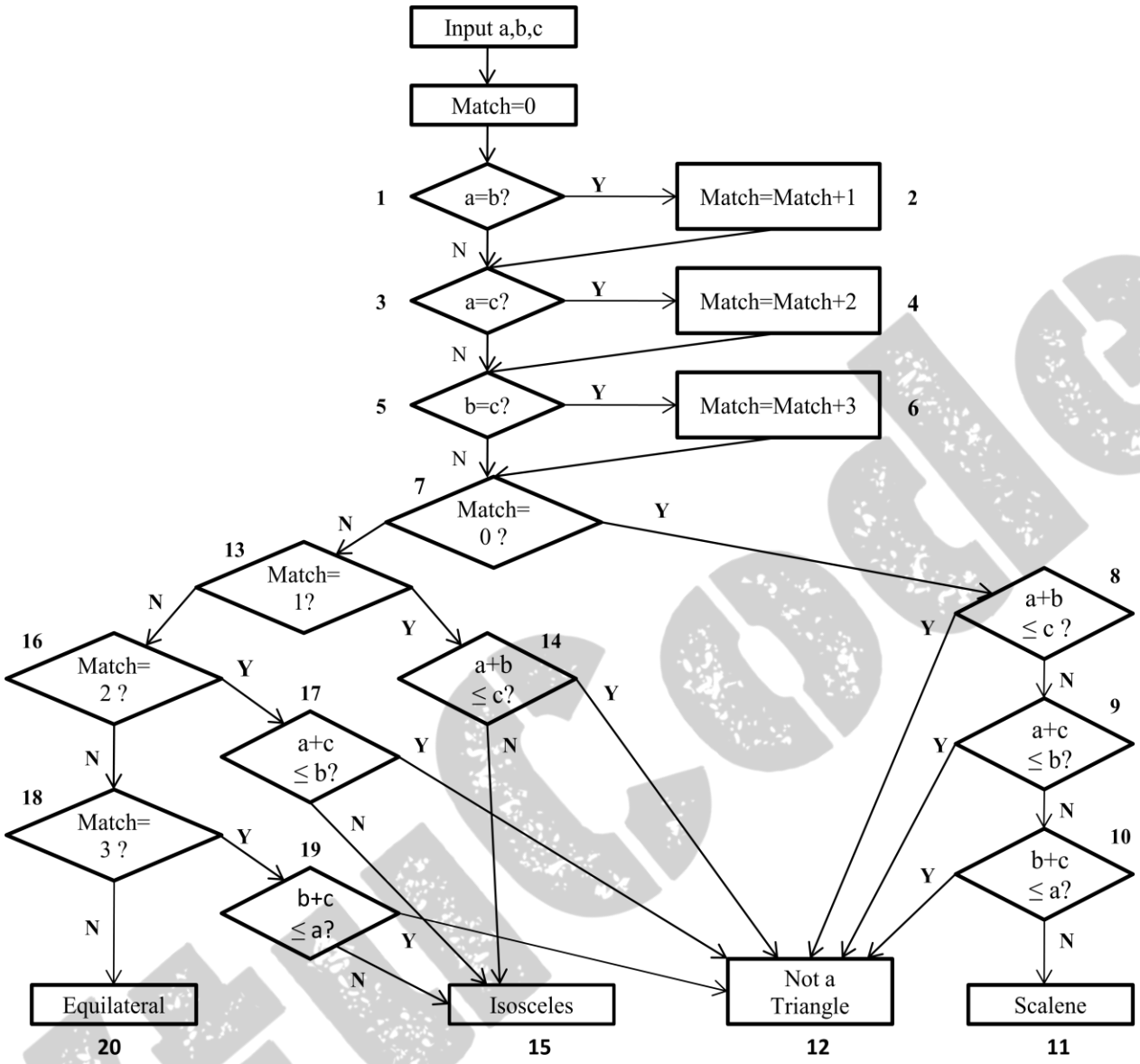
Triangle program contains **clear** but **complex logic**. The specification insists developers to know some details about triangles, its **inequality** i.e., **sum of two sides must be greater than the third side (preconditions).**

✓ Traditional Implementation

```
ProgramTriangle1    '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 ("Sides A, B, C are", a, b, 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 ("Not A Triangle")
          Else If (a + c) <=b
                Then Output ("Not A Triangle")
                  Else If (b + c) <=a
                        Then Output ("Not A Triangle")
                           Else Output ("Scalene")
                                EndIf
                        EndIf
                EndIf

Else If Match=1
        Then    If (a + b) <=c
                Then Output ("Not A Triangle")
                Else Output ("Isosceles")
        EndIf
```

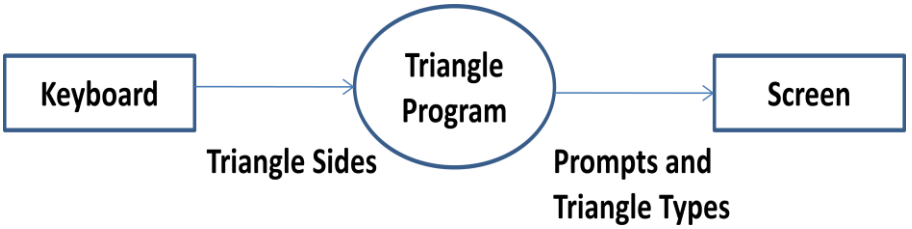
```
        Else If  Match=2
                Then  If (a + c) <=b
                        Then Output (“Not A Triangle”)
                                Else Output (“Isosceles”)
                EndIf

    Else If Match=3
                Then If (b + c) <=a
                        Then Output (“Not A Triangle”)
                                Else Ouput (“Isosceles”)
                EndIf
                         Else    Output (“Equilateral”)
                EndIf
            EndIf
        EndIf
    EndIf
End Triangle1
```

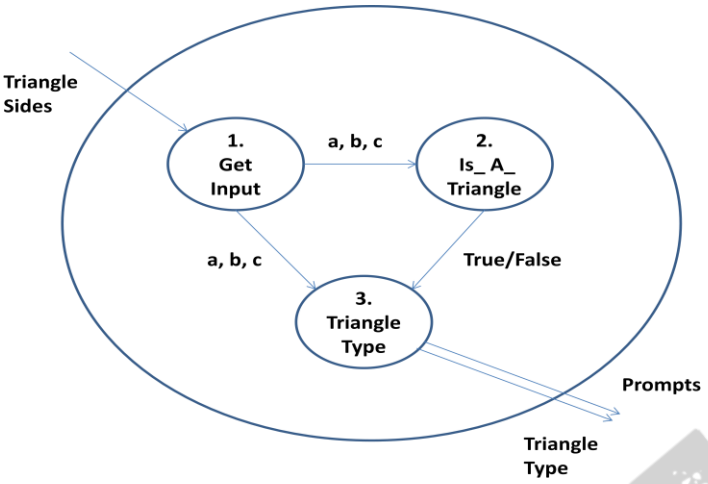
Flowchart for the Traditional Triangle Problem Implementation



✓ Structured Implementation

Figure: Dataflow diagram for a structured triangle program Implementation





Program Triangle2 ‘programming version of simpler specification

```
‘
Dim a ,b, c As INTEGER
Dim IsATriangle As Boolean
‘
‘ Step1 : Get Input
Output (“Enter 3 integers which are sides of a triangle”)
Input (a, b, c)
Output (“Sides A, B, C are”, a, b, 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 (“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 ("Sides A, B, C are", a, b, 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
```

➢ NextDate Function

✓ **Problem Statement**

- NextDate is a function of three variables: **month, date,** and **year**. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions

  c1. $1 \leq month \leq 12$
  c2. $1 \leq day \leq 31$
  c3. $1812 \leq year \leq 2019$

- If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value. **For example, "Value of month not in the range 1...12."** Because numerous invalid day, month, year combinations exist, NextDate collapses these into one message: **"Invalid Input Date."**

✓ **Discussion**

- Two sources of complexity in the NextDate function are the **complexity** of the **input domain** and the rule that determines when a **year is a leap year**. A year is **365.2422** days long; therefore leap years are used for the **"extra day"** problem. If we declare a leap year every fourth year, a slight error would occur.
- **Century years** are leap years only if they are **multiples of 400.** The years 1992, 1996 and 2000 are leap years, where 2000 is a century leap year and the century year **1900** is **not** a **leap year**.

✓ **NextDate Function 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 = 2019
                Then Output ("2019 is boundary year, so cannot increment")
                Else tomorrowyear = 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 in 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 <=2019)
        If NOT (c1)
                Then Output ("Value of day not in the range 1...31")
         EndIf
         If NOT (c2)
                Then Output ("Value of month not in the range 1..12")
         EndIf
         If NOT (c3)
                Then    Output ("Value of year not in the range 1812...2019")
         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 = 2019
              Then Output ("Year has reached upper bound value")
              Else tomorrowyear = 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  If (year is a leap year)
                Then tomorrowDay = 1
                     tomorrowMonth = 3
              Else Output ("Invalid Input Date")
           EndIf
Else
  Output ("February cannot have this day")

   EndIf
  EndIf
EndIf
EndCase
Output ("Tomorrow's date is ", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate2
```

➢ The Commission Problem

✓ Problem statement

- A **rifle salesperson** in the former Arizona Territory sold **rifle locks, stocks and barrels** made by a **gunsmith** in Missouri, **Locks cost $45, stocks cost $30, and barrels cost $25.**
- The salesperson has to sell at **least one complete rifle per month**, and production limits were such that the most the salesperson could **sell in 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 a month, the salesperson sent a very short **telegram** showing **–1 lock** sold. The **gunsmith then knew that the sales for the month were complete** and **computed** the salespersons commission as follows **"10% on sales up to (and including) $1000, 15% on the next $800, and 20% on any sales in excess of $1800".**
- The commission program produced a **monthly sales report** that gave the **total number** of **locks, stocks, and barrels** sold by the salespersons, **total dollar sales**, and finally, the **commission.**

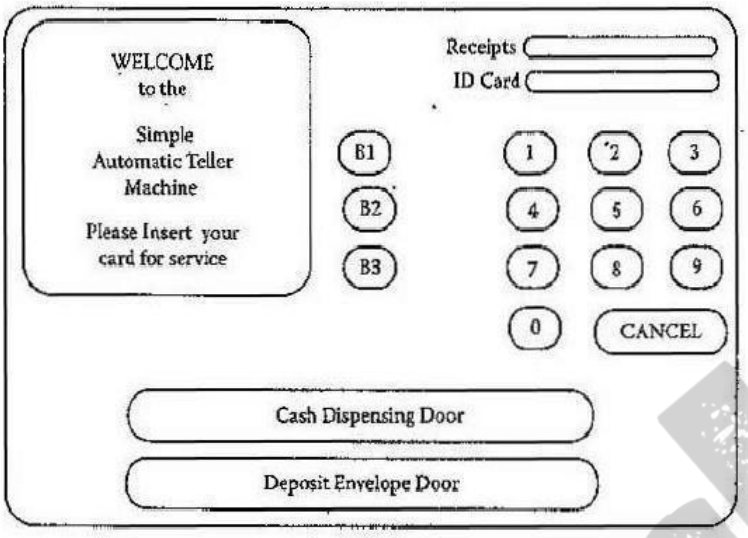
```
'Commission Problem
 Program Commission (INPUT, OUTPUT)
    Dim locks, stocks, barrels As Integer
    Dim lockprice, stockprice, barrelprice As Real
    Dim totalLocks, totalStocks,totalBarrels as Integer
    Dim lockSales, stockSales, barrelSales As Real
    Dim sales, commission As REAL
    '
    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 entry
          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 (Simple Automatic Teller Machine) System

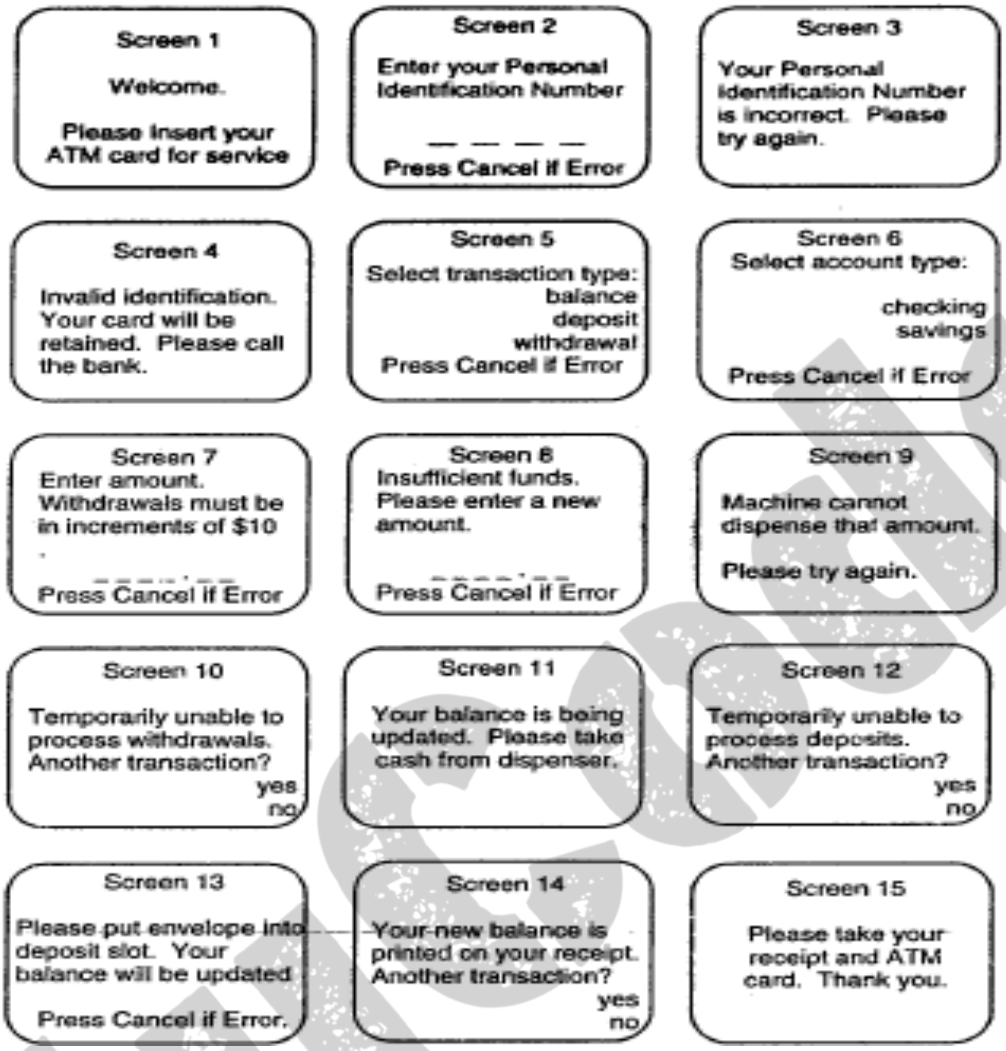
- The SATM system communicates with bank customers via the 15 screens using a terminal with features.
- SATM customers can select any of **three** transaction types: **deposits, withdrawals** and **balance inquiries**. This transaction can be done on two types of accounts: **checking and savings**.
- 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. 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 **Personal Identification Number (PIN).** If the PIN is correct, 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.

Figures: The SATM terminal



- 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 transactions 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 and then **screen 14** is 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 is entered, **screen 13** is displayed and deposit envelope is accepted 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 are processed by the master ATM (centralized) system once a day. 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, **screen10** is displayed; otherwise, **screen 7** is displayed, so the customer can enter the withdrawal amount. Now, the system checks the terminal status file to see if it has enough money to dispense. If it does not, **screen 9** is displayed; otherwise, the withdrawal is processed.
- The system checks the **customer balance**, if the funds are insufficient, **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 is 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 **screen14.**

Figure: SATM screens

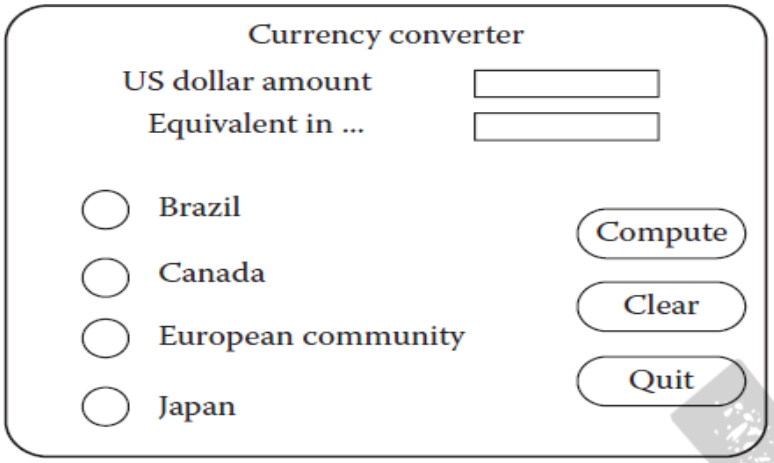


- When the **"No"** button is pressed in **screen 10, 12, or 14,** the system presents **screen 15** and returns the customer's ATM card. Once the card is removed from the slot, screen 1 is displayed. When the **"yes"** button is pressed in **screen 10, 12, or 14,** the system presents **screen 5** so the customer can select additional transactions.

➢ The Currency Converter

- The application converts U.S. dollars to any of the four currencies: **Brazilian reals, Canadian dollars, European Union Euros** and **Japanese yen**.
- Currency selection is governed by the **radio 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.
- This example illustrate a description with **UML (Unified Modeling Language)** and an **object-oriented** implementation

Figure: The Currency Converter



➢ Saturn Windshield wiper controller

- The windshield wiper on some Saturn automobiles is controlled by a **lever with a dial**.
- The lever have four positions – OFF, INT (for intermittent), LOW and HIGH.
- The dial has three positions 1, 2 and 3.
- The dial position indicates three intermittent speeds, and the dial position is relevant only when the lever is at INT position.
- The **decision table** below shows the windshield wiper speeds (wipes per minute) for lever and dial position.

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

Module 1

1. Define the terms
   Error, Fault, Failure, Incident, Test, Test cases
2. Write diagram and explain life cycle Model of testing.
3. Explain test cases based on typical test case information.
4. Explain Specified and Implemented program behaviors with Venn diagram. Extend the same with test cases.
5. Explain how Functional testing is done with diagram of functional test case identification methods.
6. Explain how structural testing is done with diagram of functional test case identification methods.
7. Differentiate Functional testing with Structural testing.
8. How error and faults can be classified? List them and give some example for each.
9. Write and explain Levels of testing with waterfall model.
10. What do you understand by a pseudo code? List some generalized pseudo code. Explain structured implementation and traditional implementation of triangle problem with conditions of all specifications.

11. Explain simplified and improved version of NextDate function with all specification.
12. Explain commission problem with all specifications/conditions with implementation.
13. Explain SATM (Simple Automated Teller Machine) with diagrams/figures.
14. Explain currency converter and Saturn wind shield wiper controller.
15. What is software quality?
16. What is meant by static and dynamic quality attributes?
17. Explain different dynamic quality attributes in detail.
18. Define reliability.
19. Briefly discuss about incompleteness and ambiguity of the requirements with example.
20. Write a note on valid and invalid inputs.
21. Define input domain and correctness of the program.
22. Discuss about testing and debugging with a neat diagram.
23. Give a test plan to test a sort program.
24. Briefly explain the different types of test metrics.
25. Explain in detail about test harness.
26. With an example explain about behavior of a program.
27. What is an oracle? Discuss about the relationship between the program under test and oracle.
28. Write a note on various types of test metrics.
29. Explain in detail about testing and verification.
30. With a neat diagram explain test generation strategies.
31. Explain in detail about Static testing.

VTU QUESTIONS (Repeatedly Asked)

1. What is software testing? Why it is so important in SDLC?
2. Explain the triangle problem statement along with flow chart for traditional implementation.
3. Explain the IEEE error and fault taxonomy and IEEE standard anomaly process.
4. Explain error and fault taxonomies.
5. Explain in detail various levels of software testing with embedded device like STAM (simple automatic teller machine) as an example.
6. Define the following
   i) Error ii) fault iii) failure iv) incident v)test vi) test case
7. Differentiate between functional testing and structural testing.
8. With a neat diagram, explain the SATM (simple automated teller machine) system.
9. What is software testing? Why it is so important in software development life cycle?
10. Explain with a neat diagram the currency converter and Saturn wind shield wiper controller.
11. Explain the structured implementation of a triangle problem with a neat dataflow diagram.
12. Explain testing and debugging with a neat diagram.
13. Define the following
    i) Reliability ii) Usability iii) Correctness iv) Performance
14. Explain the portrays of software testing life cycle.
15. Describe GUI application currency converter and embedded device Saturn wind shield wiper with diagram.