Module-4 Software Testing

Verification and Validation, Software Testing, Planning;Software Inspections, Automated Static Analysis, Verification and Formal methods, Software Testing, System Testing; Component testing, Test case design, test automation

Chapter 22 and 23 in Eighth Edition

Testing is intended to show that a program does what it is **intended to do** and to **discover program defects** before it is put into use. When you test software, you execute a program using artificial data **(test cases)**. You check the results of the test run for errors, anomalies or information about the programs non-functional attributes.

Verification	Validation
Are we building the product right, The software should **conform to its specification?**	Are we building the right product". The software **should do** what the user really requires
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	**Validation** is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
Following activities are involved in **Verification**: Reviews, Meetings and Inspections.	Following activities are involved in **Validation**: Testing like black box testing, white box testing, gray box testing etc
Execution of code is not comes under **Verification**.	Execution of code comes under **Validation**.
Cost of errors caught in **Verification** is less than errors found in Validation	Cost of errors caught in **Validation** is more than errors found in Verification.

V & V – Has TWO principal objectives

- The discovery of defects in a system;

• The assessment of whether or not the system is useful and useable in an operational situation

• Goal is to establish confidence that the software is fit for purpose. (This does NOT mean completely free of defects. Rather, it must be good enough for its intended use)

• The type of use will determine the degree of confidence that is needed.

V & V Confidence Depends on following factors:

• Software function

• The level of confidence depends on how critical the software is to an organisation.

• User expectations

• Users may have low expectations of certain kind of software.

• Marketing environment

• Getting a product to market early may be more important than finding defects in the program.

V & V Process Is a whole life-cycle process applied at each stage in the software process. Verification & validation (V & V).Starts with requirement review and continues through design reviews, code inspection and product testing. It is very expensive, complex, monotonous BUT very important and critical activity that is manpower intensive and it can never be totally automated

TWO approaches for V & V

Software inspections

• Concerned with analysis of the static system representation to discover problems (static verification)

• May be supplement by tool-based document & code analysis

Software testing

• Concerned with exercising and observing product behaviour (dynamic verification)

• The system is executed with test data and its operational behaviour is observed

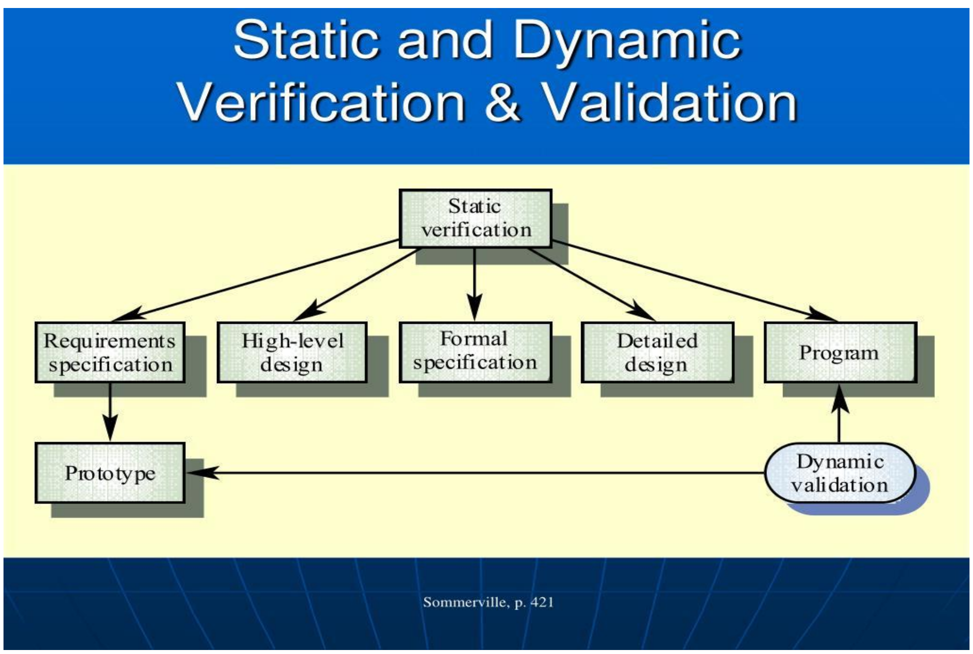
Verification & Validation techniques

• Inspection(Static Analysis)

Inspection involves program inspections, automated source code analysis, formal verification of programs. It can only check the correspondence between a program & its specifications (verification). It Cannot demonstrate the operational usefulness of the software being developed, Nor can it check emergent properties of the software like performance, reliability etc..

• Testing(Dynamic Analysis)

Testing is the most widely used and main technique for V & V that involves exercising the program with data. It can reveal the presence of errors NOT their absence. It is the only validation technique for non-functional requirements as the software has to be executed to see how it behaves. It should be used in conjunction with Inspection (static verification) to provide full V&V coverage



Testing & Debugging are Two distinct processes. While Testing is concerned with establishing the existence of defects in a program, Debugging is concerned with locating & repairing these errors.

Planning V & V is Very important since more than half the budget is spent on this process in many cases. Careful planning is required to get the most out of testing and inspection processes. It Should start early in the development process. The plan should identify the balance between static verification and testing. Test planning is about defining standards for the testing process rather than describing product tests

Test Plan is a link between development & testing, Planning document must contain following Details.

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.

Software Inspections Involve people examining the source representation with the aim of discovering anomalies and defects. It may be used before implementation as there is no need for system execution. It may be applied to any representation of the system (requirements, design, configuration data, test data, etc.). It is shown to be an effective technique for discovering program errors. Success depends on the quality of people performing inspection

Software Testing involves execution of code on the machine using test cases and test data prepared for these cases in advance. It is the only validation technique for non- functional requirements as the software has to be executed to see how it behaves. It should be used in conjunction with static verification to provide full V&V coverage

Inspection V/s Testing

Three advantages over testing:

- In testing, one defect may mask another so several executions are required; Many different defects may be discovered in a single inspection

• Inspection can be performed on incomplete versions too; But testing needs fully developed product

• Inspection can also consider broader quality attributes like standards compliance, programming style etc., but testing can only look for specific functionality / bug

BUT disadvantages of inspections are :

• Inspection is highly dependent on the expertise and interest of the people performing inspections

• Inspections can check conformance with a specification but not conformance with the customer's real requirements.

• Inspections cannot check non-functional characteristics such as: performance, usability, etc

The program inspection Process.

Program inspections are reviews, done to detect defects like logical errors, code anomalies, non-compliance of standards etc.. in the program. It was first developed by IBM in 70"s and is now a fairly widely used method of verification. It is carried out formally by a team of at least 4 people. Team consists of members who represent different viewpoints (Like for example of Author of the code, testing team representative, System expert and moderator). Inspection does not require execution of a system; Hence done before implementation. It may be used on any system element(doc) like Reqt. Design, configuration data, test data,Code…. . it is shown to be an effective technique for discovering program errors. Pre-conditions for inspection are :

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.

• Management must accept that inspection will increase costs early in the software process.

• Management should not use inspections for staff appraisal i.e. finding out who makes mistakes

Inspection Process (Inspection procedure)

• System overview presented to inspection team.

• Code and associated documents like checklists are distributed to inspection team in advance.

• Inspection takes place and discovered errors are noted.

• Modifications are made to repair discovered errors.

• Re-inspection may or may not be required.

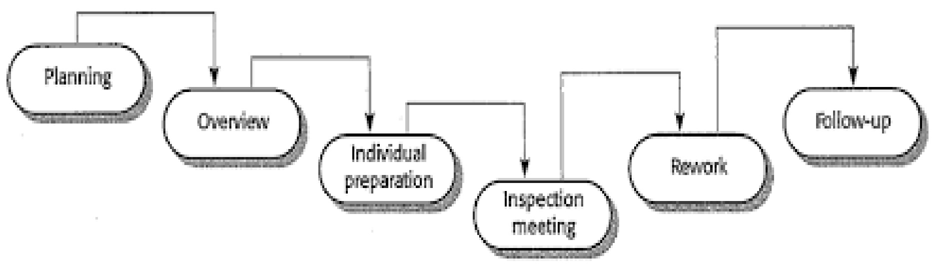


Fig 4.3: Inspection Process

Roles in the Inspection Process

1. **Author or owner**. The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
2. **Inspector**. Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues with the code being inspected such as lack of portability.
3. **Reader.** Presents the code or document at an inspection meeting.
4. **Chairman or moderator**. Manages the process and facilitates the inspection. Reports process results to the chief moderator.
5. **Scribe**. Records the results of the inspection meeting.
6. **Chief moderator.** Responsible for inspection process improvements, checklist updating, standards development, etc. Not necessarily involved in all inspections.

Inspection Checklists are commonly used tool in Inspection. It Contains a list of common errors that should be used to drive the inspection. It is Dependent on programming language and reflects the characteristic errors that are likely to arise in the language. In general, the 'weaker' the type checking, the larger the checklist. Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

Fault class	Inspection check
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Automated Static Analysis – is a process of examining a program to discover errors WITHOUT executing it (The essence of inspection) . ItCan be automated for some standard common errors resulting in STATIC ANALYZERS - Software tools that scan & parse the text of a program and detect possible faults and anomalies. It Can detect Whether the statement has been well formed. It Can do control flow analysis & even compute the data set needed to test the program. It Complements the error detection facilities of a compiler and serves as an effective V & V aid supplementing inspection. Automate static analysis includes Control Flow analysis, Data use Analysis, Interface analysis, Information flow analysis and Path analysis.

Static analysis check

Data faults –Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables

Control faults - Unreachable code Unconditional branches into loops

Input/output faults - Variables output twice with no intervening assignment

Interface faults - Parameter-type mismatches, Parameter number mismatches Non-usage of the results of functions, Uncalled functions and procedures

Storage management faults - Unassigned pointers Pointer arithmetic Memory leaks

Objectives of testing can be stated as :

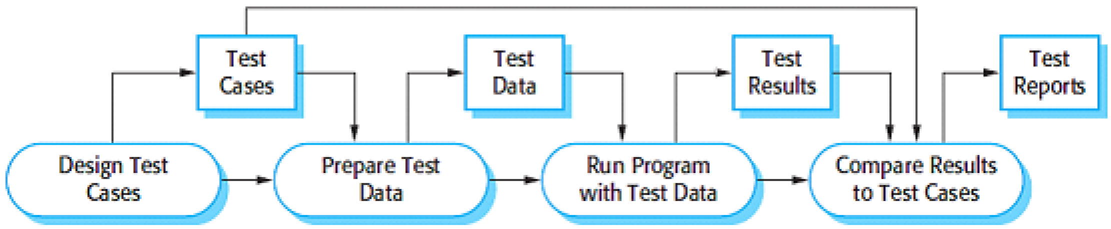
- To demonstrate to the developer and the customer that the software meets its requirements (Leads to validation Testing)
- To discover faults or defects in software where the behavior is incorrect, undesirable or does not conform to its specification (leads to verification testing)

Testing cannot demonstrate that the software is free of defects or it will behave as specified in every situation.

"Testing can only show the presence of errors, not their absence". Hence the goal of testing should be to convince developers & customers that the software is ready for use

\

A General Model of the Software Testing Process



- **Defect Testing**

- To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

- **An input-output model of program testing**

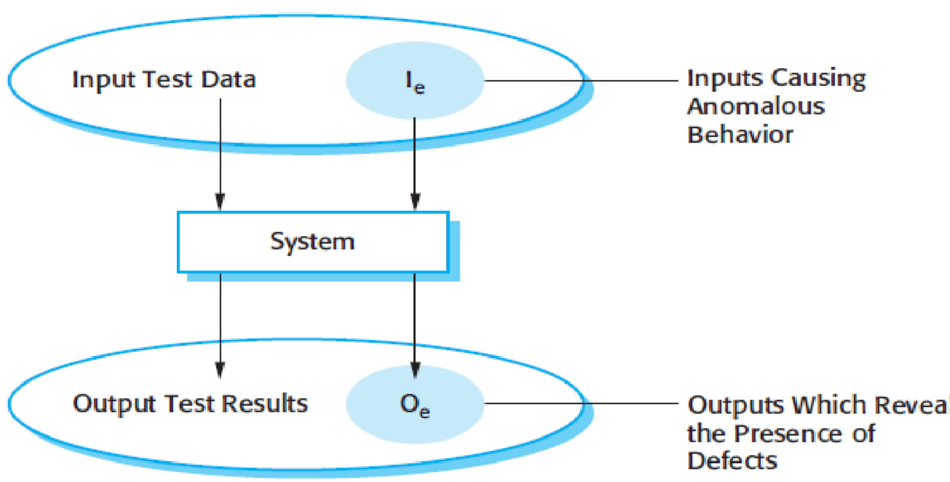


Fig 1 : Input-Output Model Of Program Testing

- **Inspections and testing**

  - Software inspections Concerned with **analysis** of the static system representation to discover problems (static verification)

    - May be supplement by tool-based document and code analysis.

- Software testing Concerned with **exercising and observing product behaviour** (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

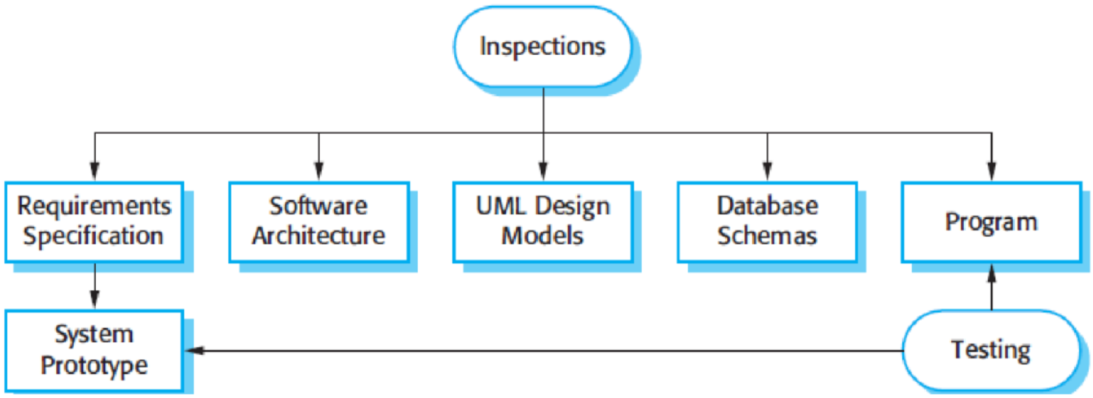


Fig 2: Inspection Method

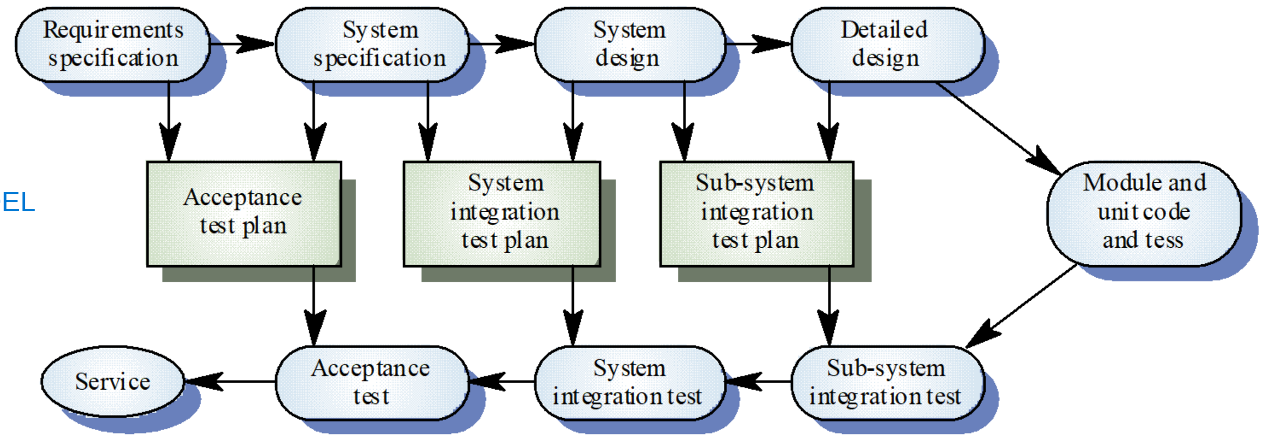
- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

- **Advantages of inspections**

- Incomplete versions of a system can be inspected **without additional costs**. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

The V-Model

V-MODEL



1.1 Stages of Testing Process (Activities)

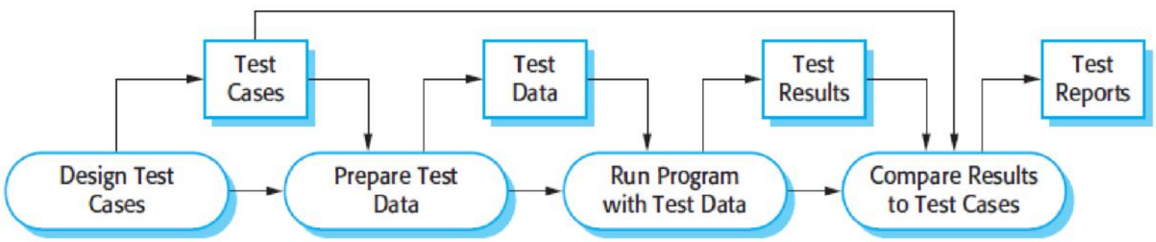


Fig 3: Stages Of Software Process (Activities)

1.2 Testing Types

1. **Development testing**, where the system is tested during development to discover bugs and defects.
2. **Release testing**, where a separate testing team tests a complete version of the system before it is released to users.
3. **User testing,** where users or potential users of a system test the system in their own environment.

1.3 Development testing

✧ Development testing includes all testing activities that are carried out by the team developing that particular software or system. **Which includes 3 stages,**

1. **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.

2. **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
3. **System testing,** where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

1.3.1 Unit testing

- Unit testing is the process of **testing individual components** in isolation.
- It is a defect testing process.
- Units may include,
  - **Individual functions** or methods within an object
  - **Object classes** with several attributes and methods
  - **Composite components** with defined interfaces used to access their functionality.
- Complete test coverage of a class involves
  - **Testing all operations** associated with an object
  - **Setting and interrogating** all object attributes
  - **Exercising the object** in all possible states.
- **Problem in unit testing** : **Inheritance** makes it more difficult to design object class tests as the information to be tested is not localised
- **Example: The weather station object interface**

Weather station is class here, in that identifier is method with attributes and associated test cases are,

Reportweather->reportstatus->powersave->remotecontrol->reconfigure->restart->shutdown

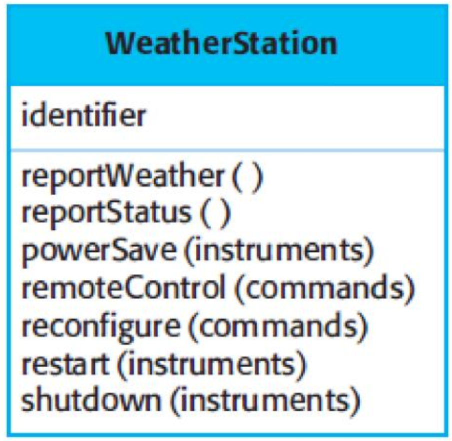


Fig 3: weather station

- **Unit test cases:**

  Effective unit test case means,

- The **defect in the components** must be revealed by the test cases.
- The component under test must show the results as expected in various test cases.

- **Testing Strategies**

1. **Partition testing:** where you identify groups of inputs that have common characteristics and should be processed in the same way.
   - You should choose tests from within each of these groups.
2. **Guideline-based testing:** where you use testing guidelines to choose test cases.
   - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

- **Partition testing(Equivalance Class Paretiotioning)**

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

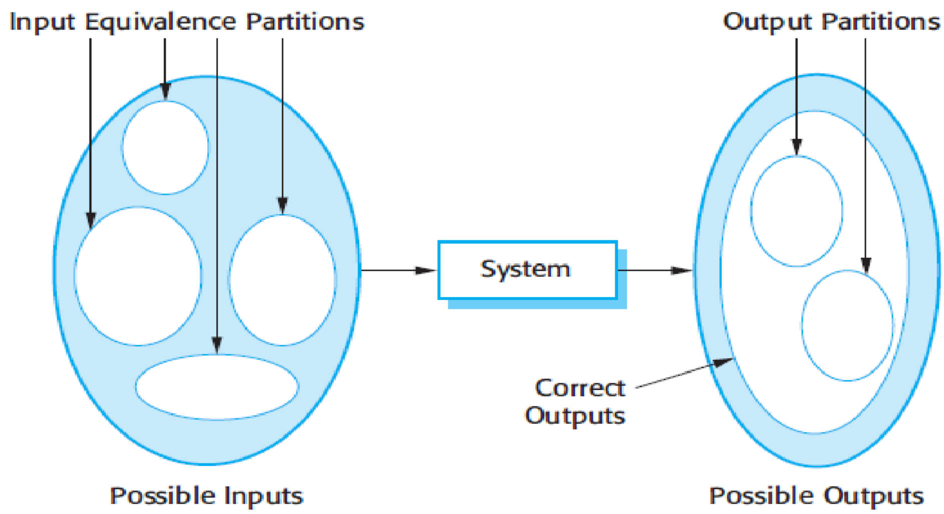


Fig 4: Equivalence partitioning

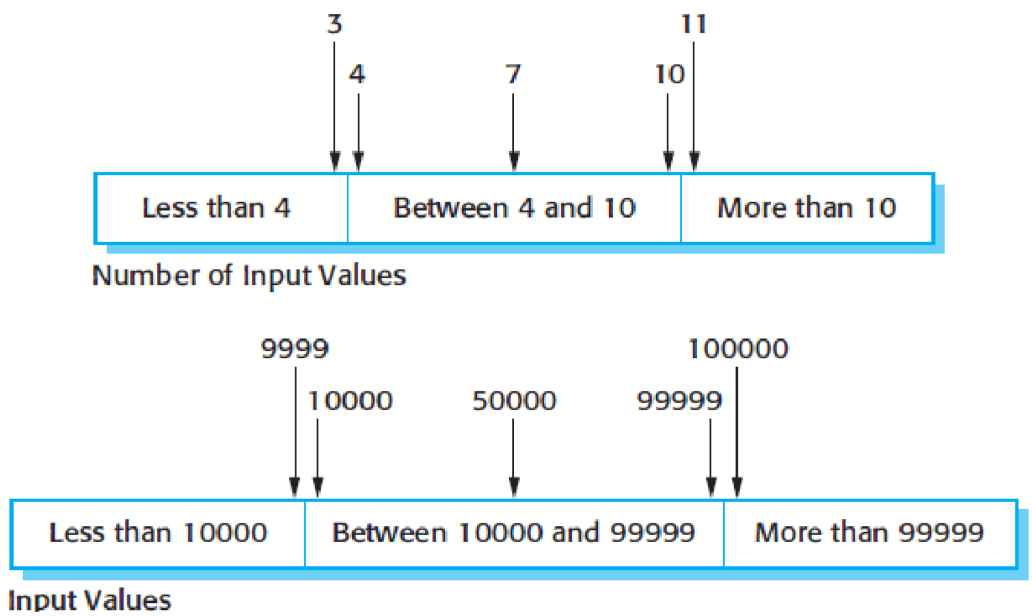


Fig 5: Equivalence Partitioning

➢ **General testing guidelines**

1. Choose inputs that force the system to **generate all error messages**
2. Design inputs that cause **input buffers** to overflow
3. **Repeat** the same input or series of inputs numerous times
4. Force **invalid outputs** to be generated
5. Force **computation results** to be too large or too small.

1.3.2 Component Testing

- Software components are often composite components that are made up of **several interacting objects.**
    - **For example**, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.

  You can assume that unit tests on the individual objects within the component have been completed.

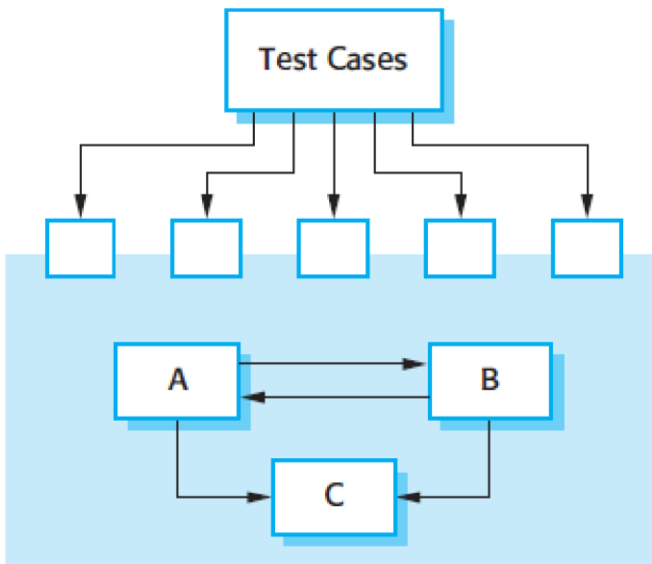


Fig 6: Interface Testing

- **Interface testing**
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- **Interface Types**

1. **Parameter interfaces** Data passed from one method or procedure to another.
2. **Shared memory** interfaces Block of memory is shared between procedures or functions.

3. **Procedural interfaces** Sub-system encapsulates a set of procedures to be called by other sub-systems.
4. **Message passing** interfaces Sub-systems request services from other sub-systems

- **Interface Errors**

  1. Interface misuse

A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

  2. Interface misunderstanding

A calling component embeds assumptions about the behaviour of the called component which are incorrect.

  3. Timing errors

  - The called and the calling component operate at different speeds and out-of-date information is accessed.

- **Interface testing guidelines**

1. Design tests so that **parameters** to a called procedure are at the extreme ends of their ranges.
2. Always test **pointer parameters** with null pointers.
3. Design tests which cause the **component to fail**.
4. Use **stress testing** in message passing systems.
5. In shared memory systems, **vary the order** in which components are activated.

1.3.3 System Testing

- System testing during development involves integrating components to create a version of the system and then **testing the integrated system.**
- The focus in system testing is testing the **interactions between components.**
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. System testing tests the emergent

behavior of a system. The main goal of the system testing is to test the interaction between the components.

During the system testing:

a. Checks the **components are compatible**
b. Checks that all components **interaction** correctly
c. Checking the **Transfer of correct data** at specified time across the interfaces of units by keeping a sequence diagram to know the sequence of activities.

Use-case testing

- ✧ The use-cases developed to identify system interactions can be used as a basis for system testing.
- ✧ Each use case usually involves several system components so testing the use case forces these interactions to occur.

The sequence diagrams associated with the use case documents the components and interactions that are being tested

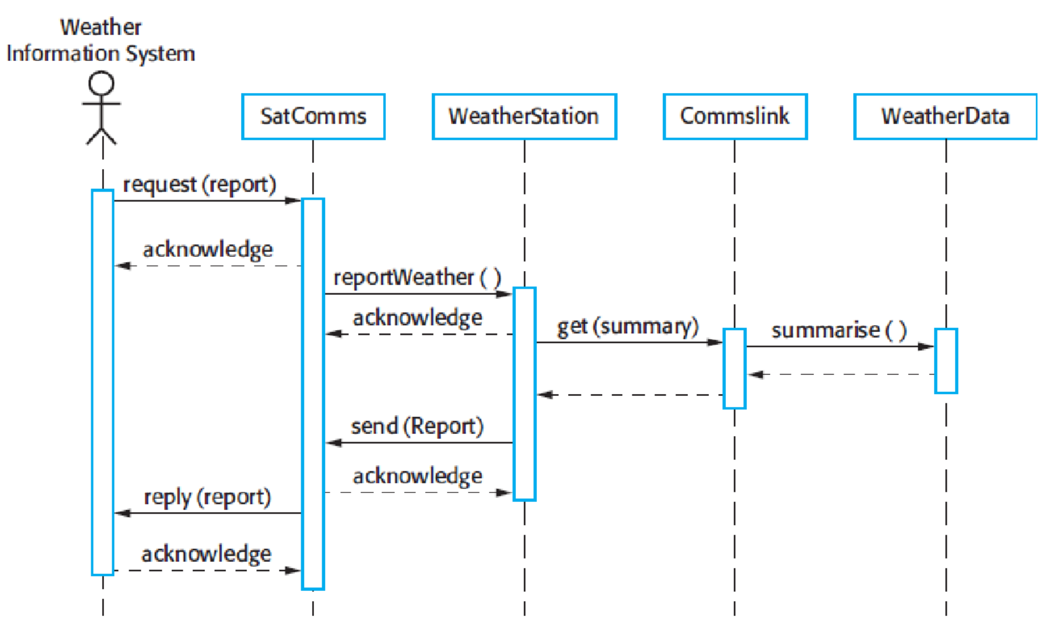


Fig 7: Collect weather data sequence chart

- ➢ **Testing policies**

- ✧ Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.

- **Examples of testing policies:**
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - Where user input is provided, all functions must be tested with both correct and incorrect input.

1.4 Release testing

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release Testing	System Testing
A separate team that has not been involved in the system development, should be responsible for release testing.	System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

- **Requirements based testing**

- Requirements-based testing involves examining each requirement and developing a test or tests for it.

Features tested by scenario

- It is a kind of release testing in which a typical scenario is specified and using this scenario the test cases are designed.
- **Scenario is nothing but a story** that describes the working of the system.

Example: scenario for ATM

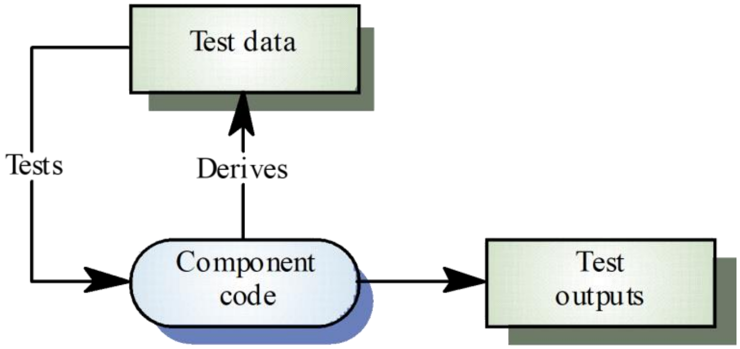
1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information. The system for call prompting

➢ **Performance testing**

- Part of release testing may involve testing the emergent properties of a system, such as **performance and reliability**.
- Tests should reflect the **profile of use of the system**.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

1.4 Structural Testing

Structural testing is an approach to test case design where the tests are derived from knowledge of the software's structure and implementation. This approach is sometimes called 'White-Box', Glass-box or Clear-box testing.



Consider Binary Search Program in Java

```
class BinSearch {

// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types by
// reference to a function and so return two values
// the key is -1 if the element is not found

    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} //BinSearch
```

Pre-Conditions for Binary Search are

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array

- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values
- Input array has an odd number of values

Binary Search Test Cases

Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Path Testing

The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once. The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control Statements with conditions are therefore nodes in the flow graph.

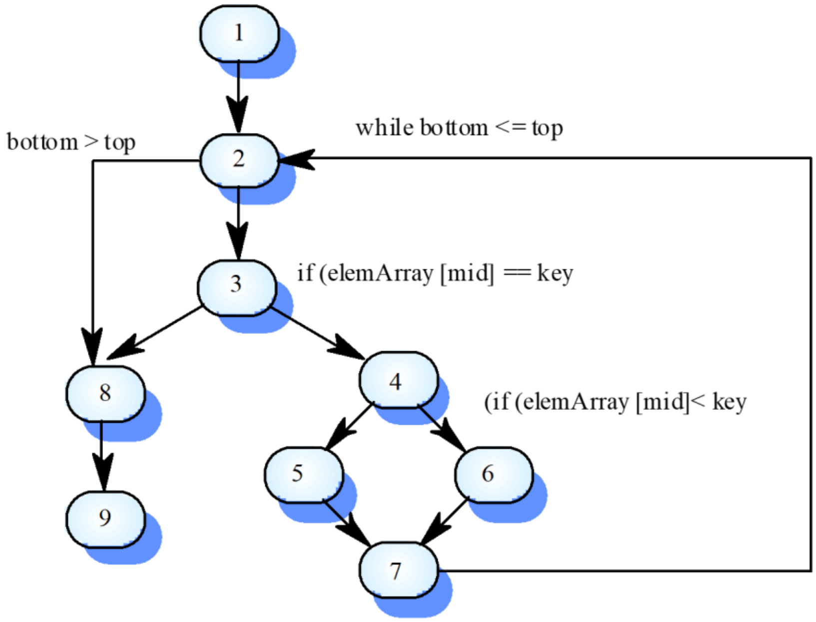
Program flow graphs

It describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node and Used as a basis for computing the cyclomatic complexity.

Cyclomatic complexity = Number of edges - Number of nodes +2

Cyclomatic Complexity

The number of tests to test all control statements equals the cyclomatic complexity. Cyclomatic complexity equals the number of conditions in a program and is useful if used with care. It Does not imply adequacy of testing. Although all paths are executed, all combinations of paths are not executed.



Independent Paths

1, 2, 3, 8, 9

1, 2, 3, 4, 6, 7, 2

1, 2, 3, 4, 5, 7, 2

1, 2, 3, 4, 6, 7, 2, 8, 9

Test cases should be derived so that all of these paths are executed. A dynamic program analyser may be used to check that paths have been executed.

1.5 Test Automation

Testing is an expensive process phase. Hence automation is necessary to reduce time and cost. Testing workbenches provide a range of tools to reduce the time required and total testing costs. Systems such as Junit support the automatic execution of tests. Most testing workbenches are open systems because testing needs are organisation-specific. They are sometimes difficult to integrate with closed design and analysis workbenches.

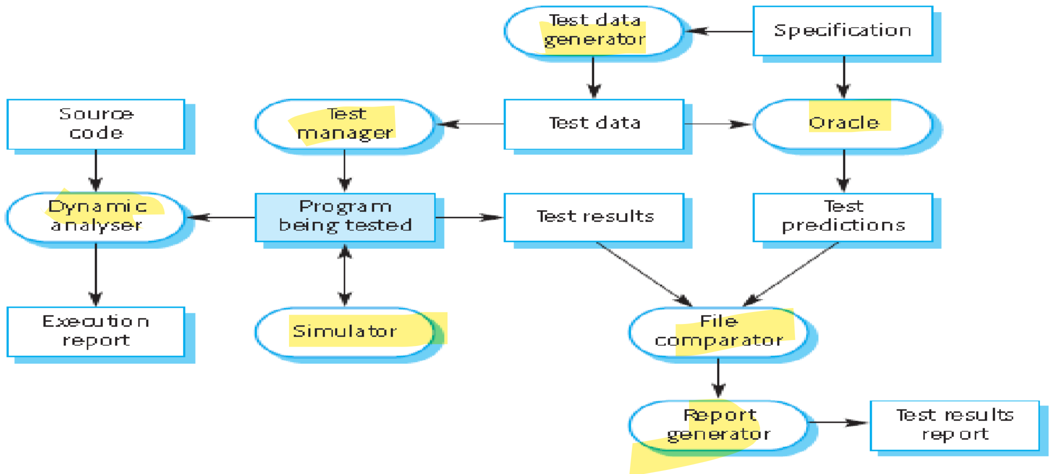


Fig 9:Test Automation

- Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. These tools now offer a range of facilities and their use can significantly reduce the costs of testing.
- A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.
- Fig 9 shows some of the tools that might be included in such a testing workbench.

1. ***Test manager*** - Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested.
2. ***Test data generator*** - Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
3. ***Oracle*** - Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems.
4. ***File comparator* -** Compares the results of program tests with previous test results and reports differences between them.
5. ***Report generator* -** Provides report definition and generation facilities for test results.

6. ***Dynamic analyser*** - Adds code to a program to count the number of times each statement has been executed.

7 Simulator Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute.