Software Testing & Audit

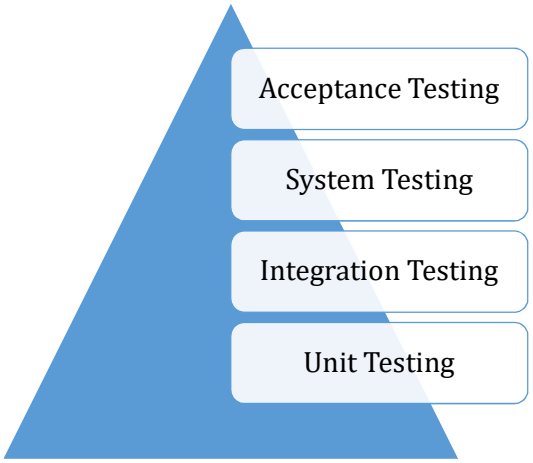
Unit-4

Jayash Kumar Sharma
Department of Computer Science & Engineering
Anand Engineering College
Mob: +91-9639325975
Email: jayash.sharma@gmail.com

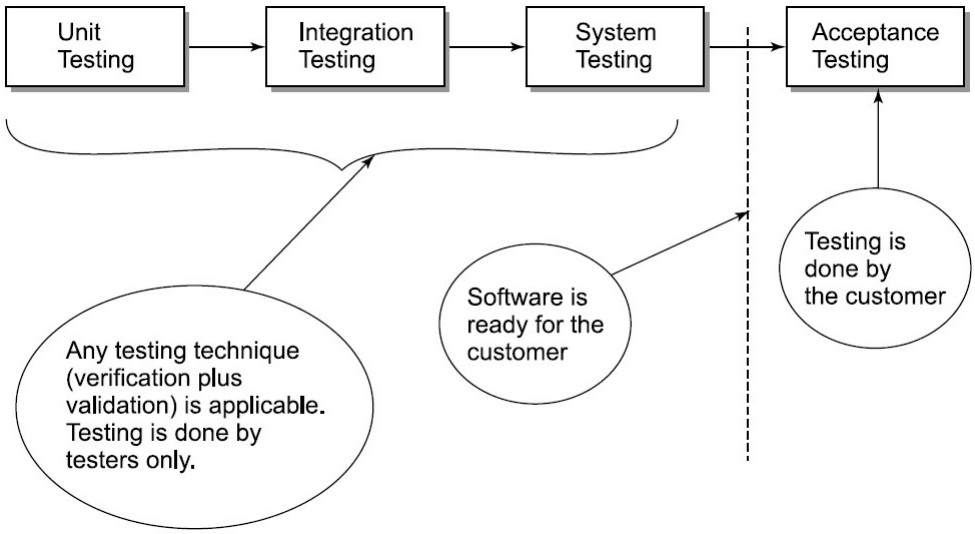
Software Testing Activities

- Testing start with the First phase of SDLC.
- Test cases are generated from SRS and SDD.
- Development and testing activities run in parallel.
- Testing is carried out at many levels (Sometimes with the help of testing tools).

Levels of Testing



Levels of Testing



Unit Testing

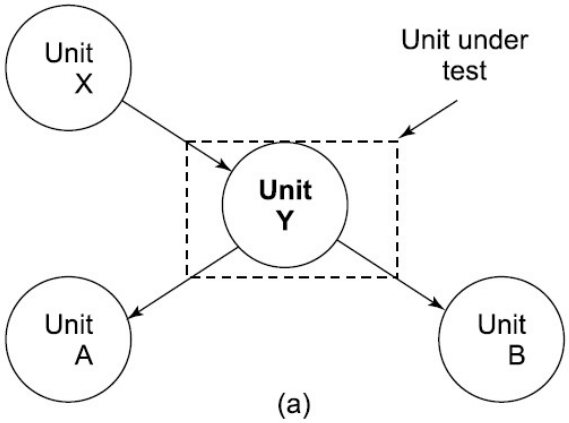
- We develop software in parts / units.
- Every unit is expected to have a defined functionality.
- This unit can be called a component, module, procedure, function, etc., which will have a purpose and may be developed independently and simultaneously.
- Testing such units is referred as Unit Testing.

Unit Testing

How can we run a unit independently?

- A unit may not be completely independent.
- It may be calling a few units and also be called by one or more units.
- We may have to write additional source code to execute a unit.

Unit Testing

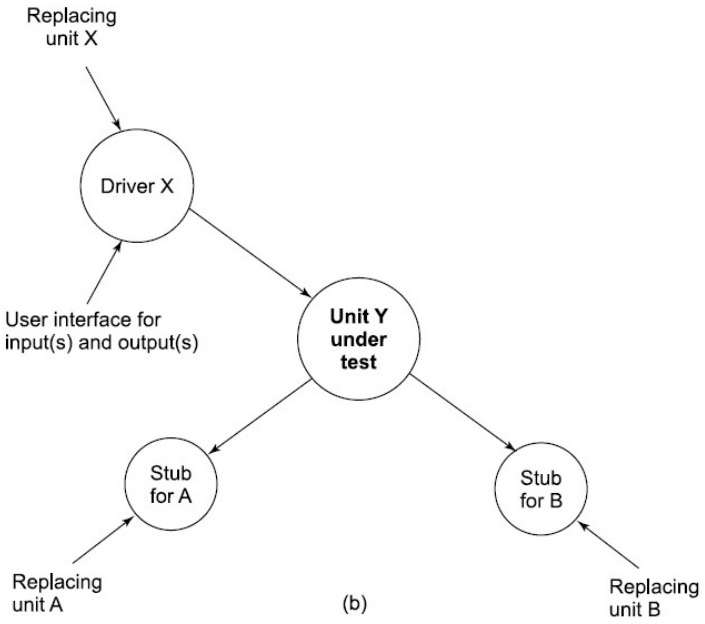


- Unit X is calling Unit Y
- Unit Y calling unit A and B

Unit Testing

- To execute unit Y independently, we may have to write additional source code in a unit Y which may handle the activities of a unit X and the activities of a unit A and a unit B.
- The additional source code to handle the activities of a unit X is called '*driver*' and the additional source code to handle the activities of a unit A and a unit B is called '*stub*'.
- The complete additional source code which is written for the design of stub and driver is called *scaffolding*.

Unit Testing



Unit Testing

- The scaffolding should be removed after the completion of unit testing.
- Many white box testing techniques may be effectively applicable at unit level.
- Stubs and drivers should be simple and small in size to reduce the cost of testing.
- Generally, it is difficult to avoid the requirement of stubs and drivers
- We may only minimize the requirement of scaffolding depending upon the functionality and its division in various units.

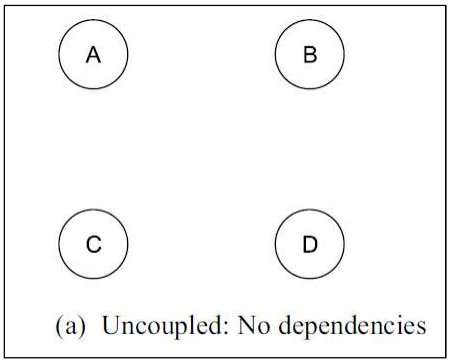
Integration Testing

- A software program may have many units.
- When we combine two units, we may like to test the interfaces amongst these units.
- We combine two or more units because they share some relationship.
- This relationship is represented by an interface and is known as *coupling*.
- The coupling is the measure of the degree of interdependence between units.

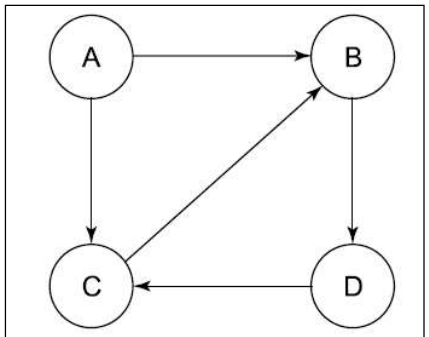
Integration Testing

- Two units with high coupling are strongly connected and thus, dependent on each other.
- Two units with low coupling are weakly connected and thus have low dependency on each other.
- Hence, highly coupled units are heavily dependent on other units.
- Loosely coupled units are comparatively less dependent on other units.

Integration Testing

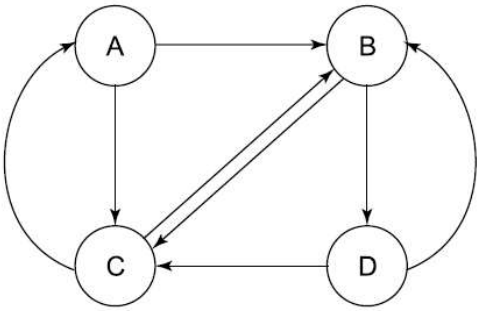


(a) Uncoupled: No dependencies



(b) Loosely coupled units: Few dependencies

Integration Testing



(c) Highly coupled units: Many dependencies

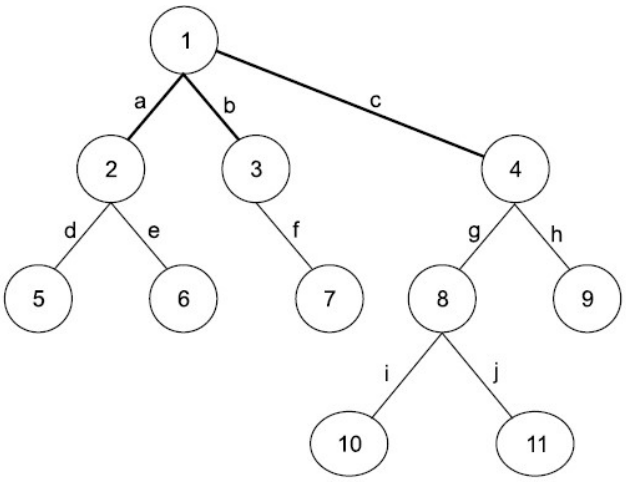
Integration Testing

- Coupling increases as the number of calls amongst units increases or the amount of shared data increases.
- A design with high coupling may have more errors & need more test cases.
- Loose coupling minimizes the interdependence, errors & need less test cases.
- A good design should have low coupling and thus interfaces become very important.

Integration Testing

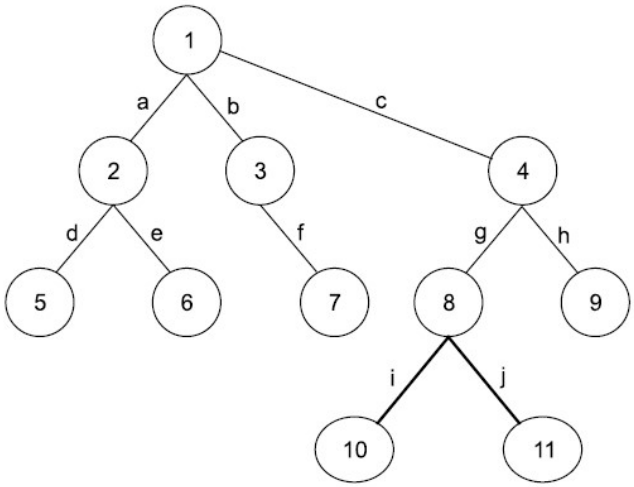
- Some of the steps to minimize coupling are :
  - Pass only data, not the control information.
  - Avoid passing undesired data.
  - Minimize parent/child relationship between calling and called units.
  - Minimize the number of parameters to be passed between two units.
  - Avoid passing complete data structure.
  - Do not declare global variables.
  - Minimize the scope of variables.

Integration Testing – Top Down Approach



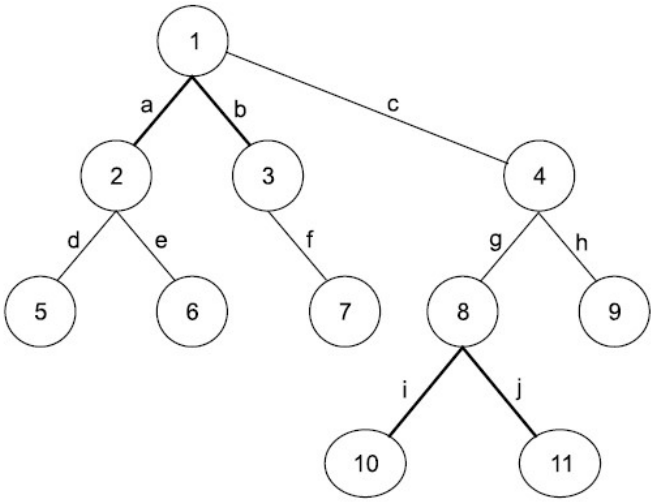
- Starts from the main unit and keeps on adding all called units of the next level.
- After completion of integration testing at this level, add the next level of units and so on till we reach the lowest level units (leaf units).
- No drivers, only stubs will be designed.

Integration Testing – Bottom Up Approach



- Start from the bottom (leaf units)
- Keep on adding upper level units till we reach the top (root node).
- No stubs, only Drivers

Integration Testing – Sandwich Approach



- Runs from top and bottom concurrently, depending upon the availability of units.
- Meet somewhere in the middle.

Integration Testing

- Each approach has its own advantages and disadvantages.
- In practice, sandwich integration approach is more popular.
- This can be started as and when two related units are available.
- Any functional or structural testing techniques can be used to design test cases.

System Testing

- System testing is performed after the completion of unit and integration testing.
- Complete software is tested along with its expected environment.
- Generally use functional testing techniques, although a few structural testing techniques.

System Testing

- System - Combination of the software, hardware and other associated parts that together provide product features and solutions.
- System testing ensures that each system function works as expected and it also tests for non-functional requirements like performance, security, reliability, stress, load, etc.
- Only phase of testing which tests both functional and non-functional requirements of the system.
- A team of the testing people does the system testing under the supervision of a test team leader.

System Testing

- All associated documents and manuals of the software are also reviewed.
- Utmost care should be taken for the defects found during the system testing phase.
- A proper impact analysis should be done before fixing the defect.
- Sometimes, if the system permits, instead of fixing the defects, they are just documented and mentioned as the known limitations. (when fixing is very time consuming or technically it is not possible in the present design)

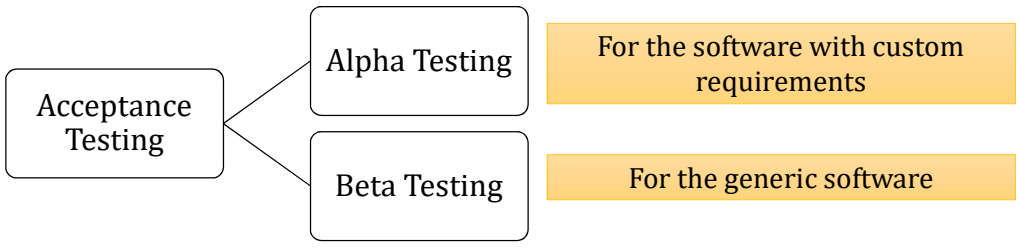
System Testing

- First phase in which the complete product is tested with a specific focus on the customer's expectations.
- After the completion of this phase, customers are invited to test the software.

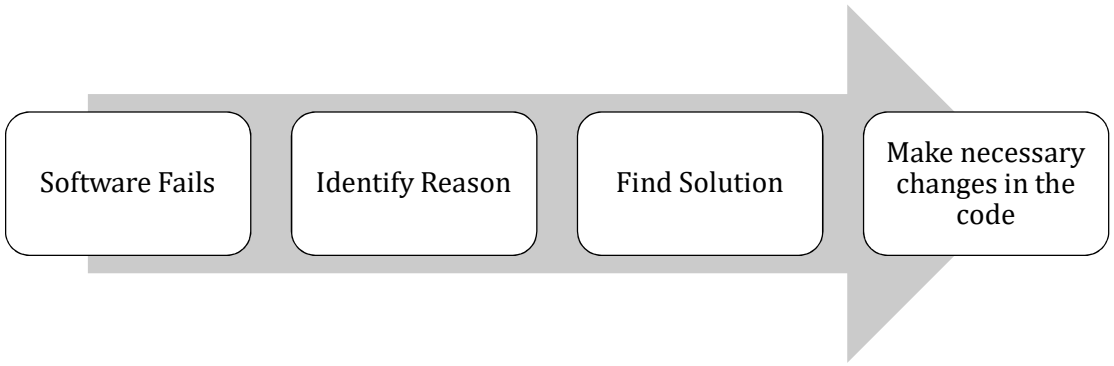
Acceptance Testing

- Extension of system testing.
- This testing may range from ad-hoc usage to systematic well-planned usage of the product.
- The testing is done for the purpose of accepting a product is known as acceptance testing.

Acceptance Testing



Debugging



Debugging : The process of identifying and correcting a software error

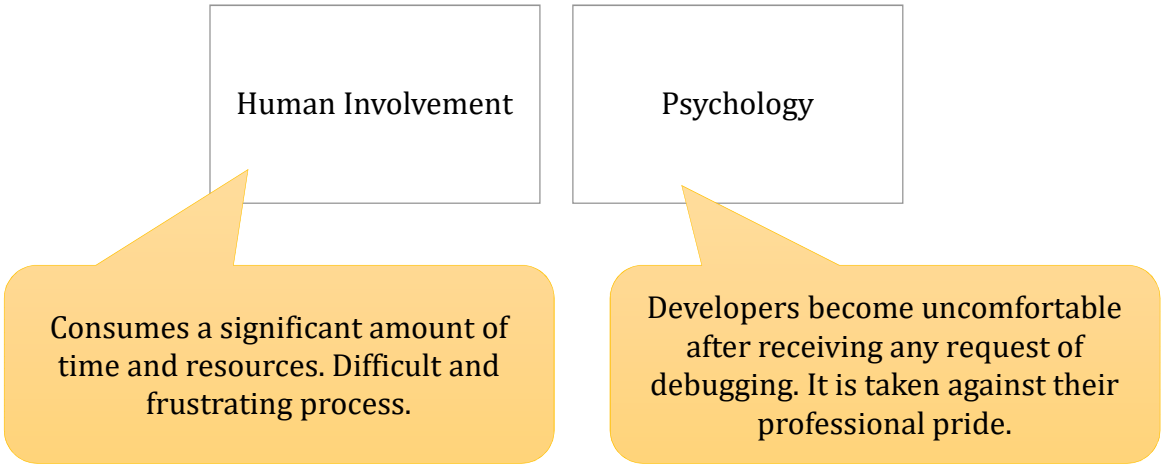
Debugging

- Debugging starts after receiving a failure report and completes after ensuring that all corrections have been rightly placed and the software does not fail with the same set of input(s).
- The debugging is quite a difficult phase and may become one of the reasons for the software delays.
- Every bug detection process is different and it is difficult to know how long it will take to detect and fix a bug.

Debugging

- In order to remove bugs, developers should understand that a problem prevails and then he/she should do the classification of the bug.
- The next step is to identify the location of the bug in the source code
- Finally take the corrective action to remove the bug.

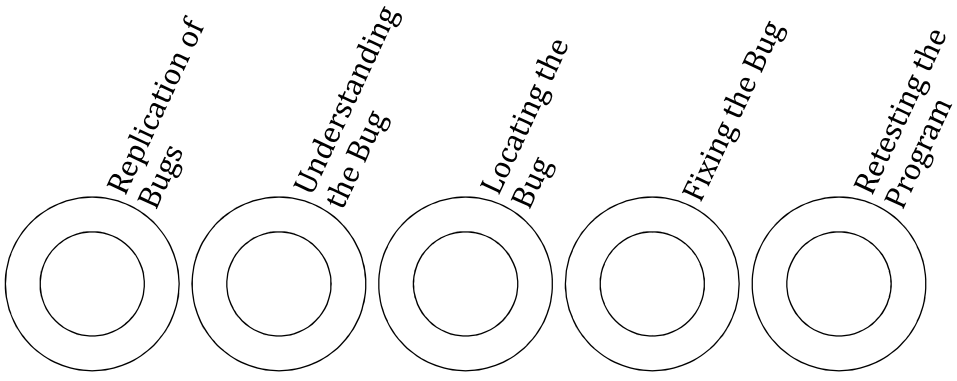
Why Debugging is so Difficult?



Debugging Process

- Debugging: detecting and removing bugs from the programs.
- Whenever a program generates an unexpected behavior, it is known as a failure of the program.
- This failure may be *mild, annoying, disturbing, serious, extreme, catastrophic or infectious*.
- Depending on the type of failure, actions are required to be taken.

Debugging Process - Steps



Debugging Process – Replication of Bug

- Recreate the undesired behavior under controlled conditions.
- The same set of input(s) should be given under similar conditions to the program and the program after execution, should produce a similar unexpected behavior.
- If this happens, we are able to replicate a bug. In many cases, this is simple and straight forward.
- In other cases, replication may be very difficult.

Debugging Process – Replication of Bug

- It may require many steps or in an interactive program such as a game, it may require precise timing.
- In worst cases, replication may be nearly impossible.
- If we do not replicate the bug, how will we verify the fix?
- Hence, failure to replicate a bug is a real problem.

Debugging Process – Replication of Bug

- Some of the reasons for non-replication of a bug are:
  - The user incorrectly reported the problem.
  - The program has failed due to hardware problems like memory overflow, poor network connectivity, network congestion, non-availability of system buses, deadlock conditions, etc.
  - The program has failed due to system software problems like operating system, compilers, device drivers, etc.

Debugging Process – Understanding the Bug

- After replicating the bug, we may like to understand the bug (find the reason(s) for this failure).
- There may be one or more reasons.
- Generally the most time consuming activity as we need to understand the program very clearly for understanding a bug.
- For designers and source code writers, there may not be any problem for understanding the bug. Otherwise it is (In case programmers / designers not available).

Debugging Process – Understanding the Bug

- For other, it depends on the readability of the program.
- Good Readability of Source code and associated documents: Easy to understand the bug
- Poor readability (which happens in many situations) and associated documents > Difficult to understand the bug.

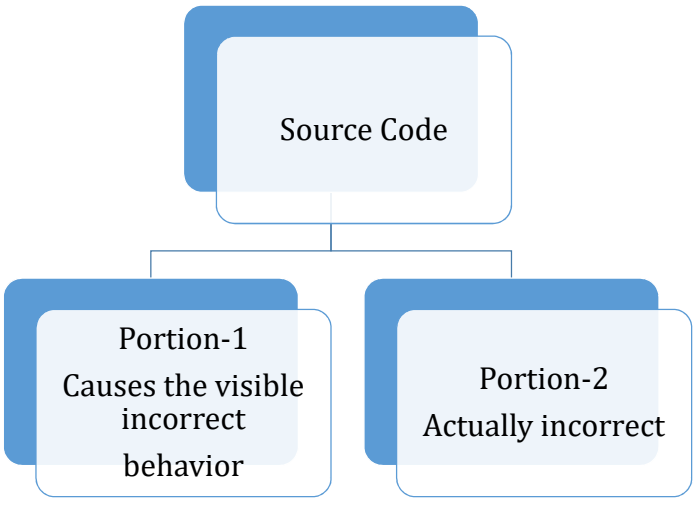
Debugging Process – Understanding the Bug

- The worst cases are large programs written by many persons over many years.
- These programs may not have consistency and may become poorly readable over time due to various maintenance activities.
- A debugger may also be helpful for understanding the program.
- A debugger inspects a program statement-wise and may be able to show the dynamic behavior of the program using a breakpoint.

Debugging Process – Understanding the Bug

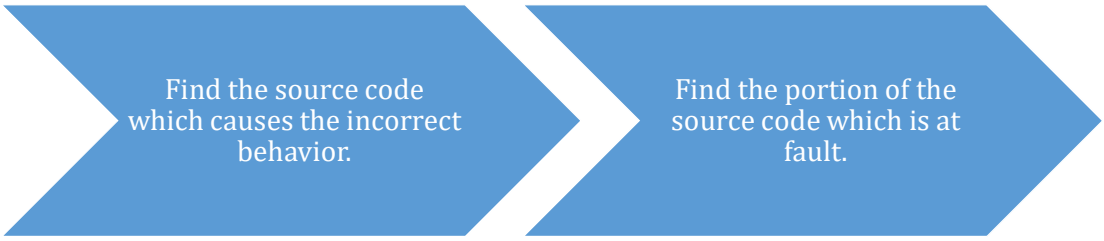
- The breakpoints are used to pause the program at any time needed.
- At every breakpoint, we may look at values of variables, contents of relevant memory locations, registers, etc.
- In order to understand a bug, program understanding is essential.

Debugging Process – Locate the Bug



Debugging Process – Locate the Bug

- Both portions may overlap.
- Sometimes, both portions may be in different parts of the program.



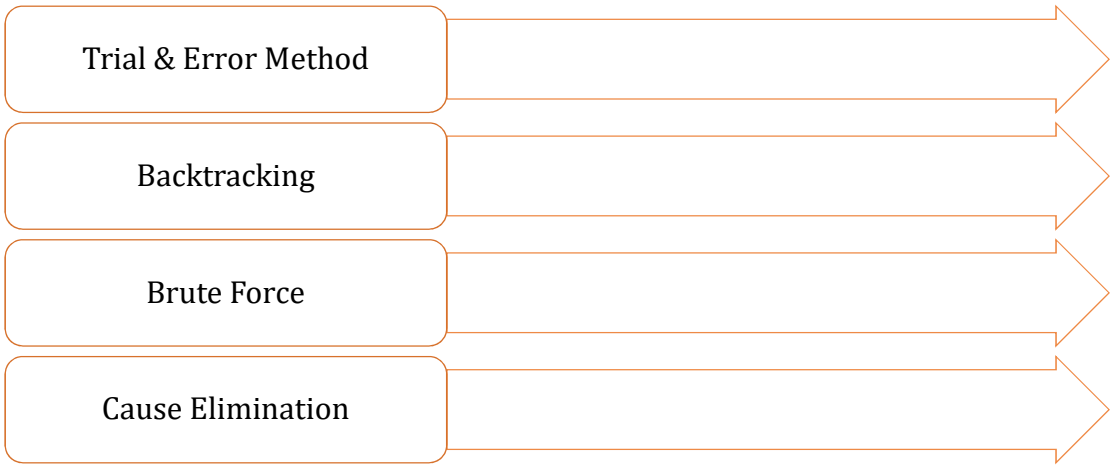
Debugging Process – Locate the Bug

- Problematic source code may be identified by:
  - Manual inspection
  - help of a debugger
  - Breakpoints
  - Print Statements
  - Inspection of Source Code – Most useful & powerful way

Debugging Process – Fix the Bug & Re-test Program

- The fixing of a bug is a programming exercise rather than a debugging activity.
- After making necessary changes in the source code, we may have to re-test the source code in order to ensure that the corrections have been rightly done at right place.
- This re-testing activity is called regression testing which is a very important activity of any debugging process.

Debugging Approaches



Debugging Approaches - Trial & Error Method

- This approach is dependent on the ability and experience of the debugging persons.
- After getting a failure report, it is analyzed and the program is inspected.
- Based on experience and intelligence, and also using the 'hit and trial' technique, the bug is located and a solution is found.
- Slow approach & impractical in large programs.

Debugging Approaches - Backtracking

- Useful in small programs.
- We start at the point where the program gives an incorrect result and trace backward the source code manually until a cause of the failure is found.
- The backtracked source code is analyzed properly.

Debugging Approaches - Backtracking

- A variation of backtracking is Forward Tracking.
- Print statements or other means are used to examine a succession of intermediate results to determine at what point the result first became wrong.
- As the program size increases, it becomes difficult to manage these approaches.

Debugging Approaches – Brute Force

- Probably the most common and efficient approach to identify the cause of a software failure.
- In this approach, memory dumps are taken, run time traces are invoked and the program is loaded with print statements.
- we try to find a clue by the information produced which leads to identification of cause of a bug.

Debugging Approaches – Cause Elimination

- Cause elimination introduces the concept of binary partitioning.
- Data related to error occurrence are organized to isolate potential causes.
- A list of all possible causes is developed and tests are conducted to eliminate each until a single one remains for validation.
- The cause is identified, properly fixed and re-tested accordingly.

Exploratory Testing

- Exploratory testing is about exploring, finding out about the software.
  - What it does
  - what it doesn't do
  - what works and what doesn't work
- The tester is constantly making decisions about what to test next and where to spend the (limited) time.
- This is an approach that is most useful when there are no or poor specifications and when time is severely limited.

Exploratory Testing

- Testers are involved in minimum planning and maximum test execution.
- The planning involves:
  1. The creation of a test charter
  2. a short declaration of the scope of a short (1 to 2 hour) time-boxed test effort
  3. The objectives
  4. Possible approaches to be used

Exploratory Testing

- The test design and test execution activities are performed in parallel typically without formally documenting the test conditions, test cases or test scripts.
- This does not mean that other, more formal testing techniques will not be used.
- Example: the tester may decide to us boundary value analysis and test the most important boundary values without necessarily writing them down.

Exploratory Testing

- Some notes are written during the exploratory-testing session, for report preparation.
- Test logging is undertaken as test execution is performed, documenting the key aspects of what is tested, any defects found and any thoughts about possible further testing.
- Exploratory testing can be used as a check on the formal test process by helping to ensure that the most serious defects have been found.

Software Testing Tools

- Most effort-consuming task in software testing > To design the test cases.
- The execution of these test cases may not require much time and resources.
- Both, designing and execution of test cases are normally handled manually.
- Do we really need a tool? If yes, where and when can we use it (designing of test cases or execution of test cases or both)?

Software Testing Tools

- Software testing tools may be used to reduce the time of testing and to make testing as easy as possible.
- Automated testing may be carried out without human involvement.
- This may help us in the areas where a similar dataset is to be given as input to the program again and again.
- A tool may undertake repeated testing, unattended (and without human intervention), during nights or on weekends.

Software Testing Tools

- Many non-functional requirements may be tested with the help of a tool.

Example:

To test the performance of a software under load, many computers, manpower and other resources may be required. A tool may simulate multiple users on one computer and also a situation when many users are accessing a database simultaneously.

Categories of Software Testing Tools

1. Static tools
2. Dynamic tools
3. Process Management tools.

Static Software Testing Tools

- Static software testing tools perform analysis of the programs without executing them at all.
- They may also find the source code which will be hard to test and maintain.
- Static testing is about prevention.

Static Software Testing Tools

Type of Static Software Tool	Details
Complexity Analysis Tool	• These tools may take the program as an input, process it and produce a complexity value as output. • This value may be an indicator of the quality of design and implementation. Example : Cyclomatic complexity, Halstead software size measures

Static Software Testing Tools

Type of Static Software Tool	Details
Syntax and Semantic Analysis tools	• These tools find syntax and semantic errors. • Early detection of such errors (before compilation) may help to minimize other associated errors. • Compilers are helpless in finding semantic errors. Example of Semantic Errors: Non-declaration of a variable, double declaration of a variable, 'divide by zero' issue, unspecified inputs and non-initialization of a variable • Many tools are language dependent and may parse the source code, maintain a list of errors and provide implementation information.

Static Software Testing Tools

Type of Static Software Tool	Details
Flow Graph Generator Tool	• These tools are language dependent. • Take the program as an input and convert it to its flow graph. • These tools assist us to understand the risky and poorly designed areas of the source code.

Static Software Testing Tools

Type of Static Software Tool	Details
Code comprehension tools	• These tools may help us to understand unfamiliar source code. • They may also identify dead source code, duplicate source code and areas that may require special attention and should be reviewed seriously.

Static Software Testing Tools

Type of Static Software Tool	Details
Code inspectors	• Enforce standards in a uniform way for many programs. • They inspect the programs and force us to implement the guidelines of good programming practices. • Language dependent but most of the guidelines of good programming practices are similar in many languages. • They may also suggest possible changes in the source code for improvement.

Dynamic Software Testing Tools

- Dynamic software testing tools select test cases and execute the program to get the results.
- Analyze the results and find reasons for failures (if any) of the program.
- They will be used after the implementation of the program and may also test non-functional requirements like efficiency, performance, reliability, etc.

Dynamic Software Testing Tools

Type of Dynamic Software Tool	Details
Coverage Analysis Tools • Automated QA's time • Insure++ • Logicscope.	• Used to find the level of coverage of the program (statement, branch, condition, multiple condition, path etc) after executing the selected test cases. • Give us an idea about the effectiveness of the selected test cases. • Highlight the unexecuted portion of the source code and force us to design special test cases for that portion of the source code. • Some tools also generate a number of commented lines, non-commented lines, local variables, global variables, duplicate declaration of variables, etc.

Dynamic Software Testing Tools

Type of Dynamic Software Tool	Details
Performance Testing Tools • Load Runner • J Meter • Silk Performer • QALOAD • AutoController • IBM Rational's Performance Tester	• Tests the performance of the software under stress / load. • Example: for a result management software, we may observe the performance with 10 users 100 users entering the data simultaneously. • Example: A website with 10 users, 100 users, 1000 users, etc. working simultaneously. • A tool may help us to simulate such situations and test these situations in various stress conditions.,

Dynamic Software Testing Tools

Type of Dynamic Software Tool	Details
Functional / Regression Testing Tools • IBM Rational's Robot • Win Runner • QA Centre • Silktest.	• Test the software on the basis of its functionality without considering the implementation details. • They may also generate test cases automatically and execute them without human intervention. • Many combinations of inputs may be considered for generating test cases automatically and these test cases may be executed, thus, relieving us from repeated testing activities.

Process Management Tools

- Help us to manage and improve the software testing process.
- We may create a test plan, allocate resources and prepare a schedule for unattended testing, for tracking the status of a bug using such tools.
- They improve many aspects of testing and make it a disciplined process.
- Selection of any tool is dependent upon the application, expectations, quality requirements and available trained manpower in the organization.

Process Management Tools

- IBM Rational Test Manager
- Test Director
- Silk Plan Pro
- QA Director.

Configuration management tools : for bug tracking and its management

- IBM Rational Software's Clear DDTs
- Bugzilla
- Jitterbug.

Software Test Plan

- A document to specify the systematic approach to plan the testing activities of the software.
- The test plan document may force us to maintain a certain level of standards and disciplined approach to testing.
- This document addresses the scope, schedule, milestones and purpose of various testing activities.
- It also specifies the items and features to be tested and features which are not to be tested.

Software Test Plan

- It includes Pass/fail criteria, roles and responsibilities of persons involved, associated risks and constraints
- A test plan document is prepared after the completion of the SRS document and may be modified along with the progress of the project.

Automated Test Data Generation

- Generating test data requires proper understanding of the SRS document, SDD document and source code of the software.
- Test data generation is not an easy and straightforward process.
- Test cases are written manually on the basis of selected techniques and execute them to see the correctness of the software.
- Is it possible to generate test data automatically?

Test Adequacy Criteria

- We may generate a large pool of test data randomly or using any other technique.
- This data is used as input(s) for testing the software.
- We may keep testing the software if we do not know when to stop testing.
- How would we come to know that enough testing is performed?

Test Adequacy Criteria

- Defining Test Adequacy Criteria is the solution.
- Once we define this, our goal is to generate a test suite that may help us to achieve defined test adequacy criteria.

Ways to Define Test Adequacy Criteria

- Every statement of the source code should be executed at least once (statement coverage).
- Every branch of the source code should be executed at least once (branch coverage).
- Every condition should be tested at least once (condition coverage).
- Every path of the source code should be executed at least once (path coverage).

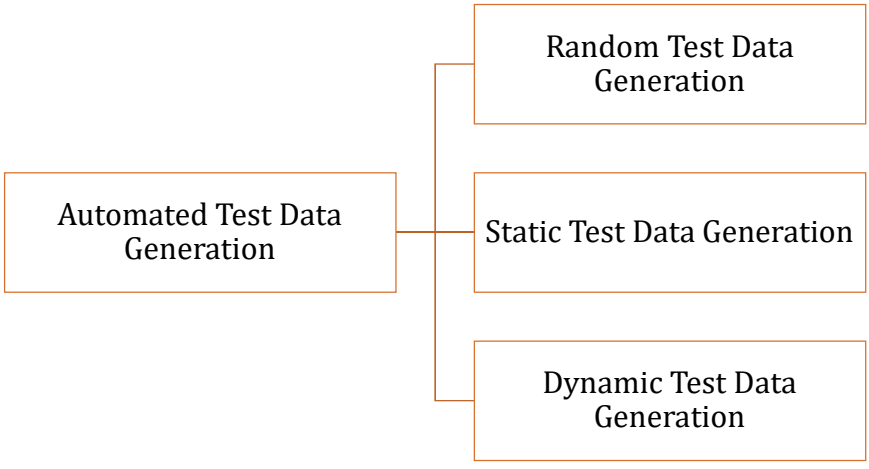
Ways to Define Test Adequacy Criteria

- Every independent path of the source code should be executed at least once (independent path coverage).
- Every stated requirement should be tested at least once.
- Every possible output of the program should be verified at least once.
- Every definition use path and definition clear path should be executed at least once.

Test Adequacy Criteria

- The definition of test adequacy criteria is very important and significant to ensure the correctness of the software.
- When our test suite fails to meet the defined criteria, we generate another test suite that does satisfy the criteria.
- Many times, it may be difficult to generate a large number of test data manually to achieve the criteria.
- Automatic test data generation process may be used to satisfy the defined criteria.

Approaches to Test Data Generation



Random Test Data Generation

- The simplest way is to generate test data randomly without considering any internal structure and / or functionality of the software.
- Random testing generates test data arbitrarily and executes the software using that data as inputs.
- The output of the software is compared with the expected output based on the inputs generated using random testing.

Random Test Data Generation - Advantage

- Random testing is not expensive.
- Fast Technique.
- Only needs a random number generator along with the software to make it functional.

Random Test Data Generation - Disadvantage

- It may not even generate test data that executes every statement of the source code.
- For any reasonably sized software, it may be difficult to attain '100% statement coverage' which is one of the easiest test adequacy criteria.
- Low chances of finding semantically small bugs (a bug that is only revealed by a small percentage of the program inputs).
- Large software or more complex test adequacy criteria may further increase the problems of random test data generators.

Static Test Data Generation

- Test data can be generated by statically evaluating the program
- The techniques based on static evaluation are called static test data generation techniques.
- Static test data generation techniques do not require the execution of the program.
- Use Symbolic Execution to identify constraints on input variables for the particular test adequacy criterion.

Static Test Data Generation

- The program is examined thoroughly and its paths are traversed without executing the program.
- Static test data generation techniques may not be useful for programs containing a large number of paths.

Static Test Data Generation – Symbolic Execution

- In symbolic execution, symbolic values are assigned to variables instead of actual values to generate the test data.
- We may define a constraint system with the help of input variables which determines the conditions that are necessary for the traversal of a given path.
- We have to find a path and then to identify constraints which will force us to traverse that particular path.
- Example: Consider a given program for determination of the nature of roots of a quadratic equation.

Static Test Data Generation – Symbolic Execution

```
1.  void main()
2.  {
3.  int a,b,c,valid=0,d;
4.  clrscr();
5.  printf("Enter values of a, b and c:\n");
6.  scanf("%d\n %d\n %d",&a,&b,&c);
7.  if((a>=0)&&(a<=100)&&(b>=0)&&(b<=100)&&(c>=0)&&(c<=100)){
8.          valid=1;
9.          if(a==0){
10.         valid=-1;
11.         }
12. }
13. if(valid==1){
14.         d=b*b-4*a*c;
15.         if(d==0){
```

Static Test Data Generation – Symbolic Execution

```
16.         printf("Equal roots");
17.         }
18.         else if(d>0){
19.         printf("Real roots");
20.         }
21.         else{
22.         printf("Imaginary roots");
23.         }
24. }
25. else if(valid==-1){
26.         printf("Not quadratic");
27.         }
28. else {
29.         printf("The inputs are out of range");
30. }
31. getch();
32. }
```

Static Test Data Generation – Symbolic Execution

- We select the path (1-7, 13, 25, 28-32) for the purpose of symbolic execution.
- The input variables a, b and c are assigned the constant variables x, y and z respectively.

a = x
b = y
c = z

Static Test Data Generation – Symbolic Execution

- At statement number 7, we have to select a false branch to transfer control to statement number 13.
- Hence, the first constraint of the constraint system for this path is:

(x <= 0 or x > 100 or
y <= 0 or y > 100 or
z <= 0 or z > 100) and
(valid = 0)

Static Test Data Generation – Symbolic Execution

- The path needs statement number 13 to become false so that control is transferred to statement number 25.
- The second constraint of the constraint system for this path is:

  valid != 1

Static Test Data Generation – Symbolic Execution

- Finally, the path also needs statement number 25 to become false so that control is transferred to statement number 28.
- Hence, the third constraint of the constraint system for this path is:

  valid != -1

Static Test Data Generation – Symbolic Execution

- To execute the selected path, all of the above mentioned constraints should be satisfied.
- One of the test cases that traverse the path (1-7, 13, 25, 28-32) may include the following inputs:

(x = -1, y = 101, z = 101)

Static Test Data Generation – Symbolic Execution

Sr. No.	X	Y	Z	Expected Output	Path	Constraint	Feasible
1.	101	50	50	Input values not in range	1-7, 13, 25, 28-32	• (x <= 0 or x > 100 or y <= 0 or y > 100 or z <= 0 or z > 100) and (valid = 0) • valid != 1 • valid != -1	Yes
2.	-	-	-	-	1-7, 13, 25-27, 31, 32	• (x <= 0 or x > 100 y <= 0 or y > 100 z <= 0 or z > 100) and (valid = 0) • valid != 1 • valid = -1	No

Static Test Data Generation – Symbolic Execution

- Therefore, in symbolic execution, constraints are identified for every predicate node of the selected path.
- Test data can be generated for selected paths automatically using identified constraints.

Static Test Data Generation – Symbolic Execution

Problems:

- Path selection and execution may be time taking activity
- There are other constructs which can not be evaluated easily (tree, pointer, array, graph etc)

Above problems limit the applicability of symbolic execution.

Dynamic Test Data Generation

- Requires actual execution of the program with some selected inputs.
- The values of variables are known during execution of the program.
- We also determine the program flow with such selected input(s).
- If the desired program flow / path is not executed, source code is carefully examines and identify problem area is identified.