Unit IV

□ TEST AUTOMATION

- Selecting and Installing Software Testing Tools
- Software Test Automation
- Skills needed for Automation
- Scope of Automation
- Design and Architecture for Automation
- Requirements for a Test Tool
- Challenges in Automation
- Tracking the Bug
- Debugging
- Case study using Bug Tracking Tool

Introduction to Test Automation

- □ *Developing software to test the software is called test automation.* Test automation can help address several problems
- □ **Automation saves time as software can execute test cases faster than human do**
- □ This can help in running the tests overnight or unattended. The time thus saved can be used effectively for test engineers to
- □ Develop additional test cases to achieve better coverage;
- □ Perform some specialized tests like ad hoc testing;
- □ Perform some extra manual testing
- □ The time saved in automation can also be utilized to develop additional test cases, thereby improving the coverage of testing

- □ Moreover, the automated tests can be run overnight, saving the elapsed time for testing, thereby enabling the product to be released frequently.
- □ **Test automation can free the test engineers from mundane tasks and make them focus on more creative tasks**
  - We read about ad hoc testing . This testing requires intuition and creativity to test the product for those perspectives that may have been missed out by planned test cases
  - If there are too many planned test cases that need to be run manually and adequate automation does not exist, then the test team may spend most of its time in test execution
  - This creates a situation where there is no scope for intuition and creativity in the test team
  - This also creates fatigue and boredom in the test team. Automating the more mundane tasks gives some time to the test engineers for creativity and

- □ **Automated tests can be more reliable**
  - When an engineer executes a particular test case many times manually, there is a chance for human error or a bias because of which some of the defects may get missed out.
- □ **Automation helps in immediate testing**
  - Automation reduces the time gap between development and testing as scripts can be executed as soon as the product build is ready
  - Automation can be designed in such a way that the tests can be kicked off automatically, after a successful build is over. Automated testing need not wait for the availability of test engineers

- □ **Automation can protect an organization against attrition of test engineers**
  - Automation can also be used as a knowledge transfer tool to train test engineers on the product as it has a repository of different tests for the product
  - On the other hand, automating tests makes the test execution less person dependent.
- □ **Test automation opens up opportunities for better utilization of global resources**
  - Manual testing requires the presence of test engineers, but automated tests can be run round the clock, twenty-four hours a day and seven days a week
  - This will also enable teams in different parts of the world, in different time zones, to monitor and control the tests, thus providing round the-clock coverage.

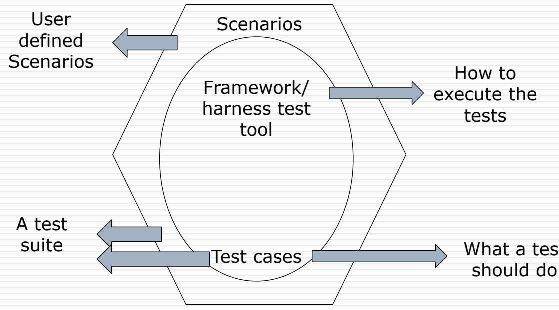
Terms used in Automation

- □ There are two types of test cases – automated and manual
- □ Manual is executed manually
- □ Automated test case is executed using automation
- □ Test case can be represented in many forms. It can be documented as a set of simple steps, or it could be an assertion statement or a set of assertion
- □ An example of an assertion is "Opening a file, which is already opened should fail".
- □ An assertion statement includes the expected result in the definition itself and makes it easy for the automation engineer to write the code
- □ Some times some tests we repeatedly perform like "log in to the system" are generally performed in a large no. of test test cases for a product. This presents an opportunity for the automation code to be resued for different purposes and scenarios

- □ Below table described some test cases for the log in example

S.No	Test cases for testing	Belongs to what type of testing
1	Check whether log in works	Functionality
2	Repeat log in operation in a loop for 48 hours	Reliability
3	Perform log in from 10000 clients	Load/Stress testing
4	Measure time taken for log in operations in different conditions	Performance
5	Run log in operation from a machine running Japanese language	Internationalization

- □ Framework for test automation



- □ **SKILLS NEEDED FOR AUTOMATION**
- □ There are different *"Generations of Automation"*
- □ The skills required for automation depends on what generation of automation the company is in or desires to be in the near future.
- □ The automation of testing is broadly classified into three generations
- □ **First generation—Record and Playback**
  - Record and playback avoids the repetitive nature of executing tests
  - A test engineer *records* the sequence of actions by keyboard characters or mouse clicks and those recorded scripts are *played back* later, in the same order as they were recorded
  - Since a recorded script can be played back multiple times, it reduces the tedium of the testing function
  - Besides avoiding repetitive work, it is also simple to record and save the script.

- But this generation of tool has several disadvantages.
  - The scripts may contain hard-coded values, thereby making it difficult to perform general types of tests
  - For example, when a report has to use the current date and time, it becomes difficult to use a recorded script
  - The handling error condition is left to the testers and thus, the played back scripts may require a lot of manual intervention to detect and correct error conditions.
  - When the application changes, all the scripts have to be rerecorded, thereby increasing the test maintenance costs
  - Thus, when there is frequent change or when there is not much of opportunity to reuse or re-run the tests, the record and playback generation of test automation tools may not be very effective.

- **Second generation—Data-driven**
- This method helps in developing test scripts that generates the set of input conditions and corresponding expected output
- This enables the tests to be repeated for different input and output conditions
- The approach takes as much time and effort as the product.
- However, changes to application does not require the automated test cases to be changed as long as the input conditions and expected output are still valid
- This generation of automation focuses on input and output conditions using the black box testing approach.

- **Third generation—Action-driven**
- This technique enables a layman to create automated tests
- There are no input and expected output conditions required for running the tests
- All actions that appear on the application are automatically tested, based on a generic set of controls defined for automation.
- The set of actions are represented as objects and those objects are reused.
- The user needs to specify only the operations (such as log in, download, and so on) and everything else that is needed for those actions are automatically generated.
- The input and output conditions are automatically generated and used
- The scenarios for test execution can be dynamically changed using the test framework that is available in this approach of automation

- Hence, automation in the third generation involves two major aspects—"test *case automation*” and “*framework design*

SCOPE OF AUTOMATION

- The first phase involved in product development is requirements gathering
- it is no different for test automation as the output of automation can also be considered as a product (the automated tests).
- The automation requirements define what needs to be automated looking into various aspects.
- The specific requirements can vary from product to product, from situation to situation, from time to time. We present below some generic tips for identifying the scope for automation

- *4.1 Identifying the Types of Testing Amenable to Automation*
- Certain types of tests automatically lend themselves to automation.
- **Stress, reliability, scalability, and performance testing**
- These types of testing require the test cases to be run from a large number of different machines for an extended period of time, such as 24 hours, 48 hours, and so on.
- It is just not possible to have hundreds of users trying out the product day in and day out—they may neither be willing to perform the repetitive tasks, nor will it be possible to find that many people with the required skill sets
- Test cases belonging to these testing types become the first candidates for automation.

- **Regression tests**
- Regression tests are repetitive in nature. These test cases are executed multiple times during the product development phases
- Given the repetitive nature of the test cases, automation will save significant time and effort in the long run
- **Functional tests**
- These kinds of tests may require a complex set up and thus require specialized skill, which may not be available on an ongoing basis

- Automating these once, using the expert skill sets, can enable using less-skilled people to run these tests on an ongoing basis.
- In the product development scenario, a lot of testing is repetitive as a good product can have a long lifetime if the periodic enhancements and maintenance releases are taken into account
- This provides an opportunity to automate test cases and execute them multiple times during release cycles
- As a thumb rule, if test cases need to be executed at least ten times in the near future, say, one year, and if the effort for automation does not exceed ten times of executing those test cases, then they become candidates for automation
- Of course, this is just a thumb rule and the exact choice of what to automate will be determined by various factors such as availability of skill sets, availability of time for designing automated test scripts vis-à-vis release pressures, cost of the tool, availability of support, and so on.

- The summary of arriving at the scope of what to automate is simply that we should choose to automate those functions (based on the above guidelines) that can amortize the investments in automation with minimum time delay
- *.4.2 Automating Areas Less Prone to Change*
- In a product scenario, the changes in requirements are quite common. In such a situation, what to automate is easy to answer
- Automation should consider those areas where requirements go through lesser or no changes
- Normally change in requirements cause scenarios and new features to be impacted, not the basic functionality of the product

- As an example, in the car manufacturing, the basic components of the car such as steering, brake, and accelerator have not changed over the years
- While automating, such basic functionality of the product has to be considered first, so that they can be used for *"regression test bed"* and *"daily builds and smoke test."*

- User interfaces normally go through significant changes during a project
- To avoid rework on automated test cases, proper analysis has to be done to find out the areas of changes to user interfaces, and automate only those areas that will go through relatively less change
- The non-user interface portions of the product can be automated first. While automating functions involving user interfaces-and non-user interface-oriented ("backend") elements, clear demarcation and "pluggability" have to be provided so that they can be executed together as well as executed independently.

- This enables the non-GUI portions of the automation to be reused even when GUI goes through changes.
- 4.3 Automate Tests that Pertain to Standards
- One of the tests that products may have to undergo is compliance to standards
- . For example, a product providing a JDBC interface should satisfy the standard JDBC tests. These tests undergo relatively less change. Even if they do change, they provide backward compatibility by which automated scripts will continue to run.
- Automating for standards provides a dual advantage. Test suites developed for standards are not only used for product testing but can also be sold as test tools for the market

- . A large number of tools available in the commercial market were internally developed for in-house usage. Hence, automating for standards creates new opportunities for them to be sold as commercial tools
- In case there are tools already available in the market for checking such standards, then there is no point in reinventing the wheel and rebuilding these tests. Rather, focus should be towards other areas for which tools are not available and in providing interfaces to other tools
- Testing for standards have certain legal and organization requirements. To certify the software or hardware, a test suite is developed and handed over to different companies

- The certification suites are executed every time by the supporting organization before the release of software and hardware.
- This is called *"certification testing"* and requires perfectly compliant results every time the tests are executed
- The companies that do certification testing may not know much about the product and standards but do the majority of this testing. Hence, automation in this area will go a long way.
- This is definitely an area of focus for automation. For example, some companies develop test suites for their software product and hardware manufacturers execute them before releasing a new hardware platform

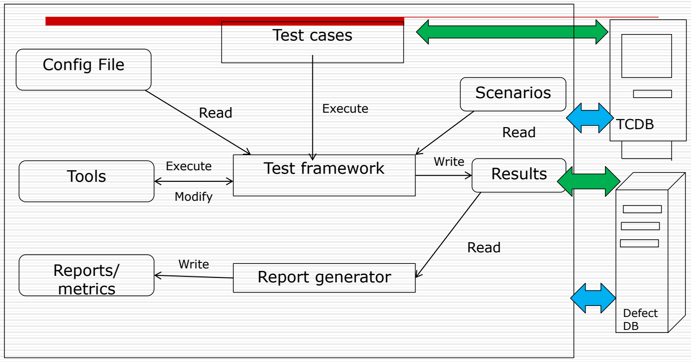
- This enables the customers to ascertain that the new hardware that is being released is compatible with software products that are popular in the market.
- *.4.4 Management Aspects in Automation*
- Prior to starting automation, adequate effort has to be spent to obtain *management commitment.* Automation generally is a phase involving a large amount of effort and is not necessarily a one-time activity
- The automated test cases need to be maintained till the product reaches obsolesence.
- Since it involves significant effort to develop and maintain automated tools, obtaining management commitment is an important activity

- Since automation involves effort over an extended period of time, management permissions are only given in phases and part by part.
- Hence, automation effort should focus on those areas for which management commitment exists already
- What to automate takes into account the technical and management aspects, as well as the long-term vision
- *Return on investment* is another aspect to be considered seriously. Effort estimates for automation should give a clear indication to the management on the expected return on investment
- While starting automation, the effort should focus on areas where good permutations and combinations exist. This enables automation to cover more test cases with less code.

- Secondly, test cases which are easy to automate in less time should be considered first for automation. Some of the test cases do not have pre-associated expected results and such test cases take long to automate
- Such test cases should be considered for the later phases of automation. This satisfies those amongst the management who look for quick returns from automation.
- In line with Stephen Covey's principle of "First Things First," [COVE-89] it is important to automate the critical and basic functionalities of a product first
- To achieve this, all test cases need to be prioritized as high, medium, and low, based on customer expectations, and automation should start from high priority and then cover medium and low-priority requirements.

- DESIGN AND ARCHITECTURE OF AUTOMATION
- Design and architecture is an important aspect of automation
- As in product development, the design has to represent all requirements in modules and in the interactions between modules
- In Figure, the thin arrows represent the internal interfaces and the direction of flow and thick arrows show the external interfaces
- Architecture for test automation involves two major heads
- a test infrastructure that covers a test case database and a defect database or defect repository.
- These are shown as external modules in Figure.

- Using this infrastructure, the test framework provides a backbone that ties the selection and execution of test cases.



External Modules

- There are two modules that are external modules to automation—TCDB and defect DB
- All the test cases, the steps to execute them, and the history of their execution (such as when a particular test case was run and whether it passed/failed) are stored in the TCDB
- The test cases in TCDB can be manual or automated
- The interface shown by thick arrows represents the interaction between TCDB and the automation framework only for automated test cases
- Please note that manual test cases do not need any interaction between the framework and TCDB
- Defect DB or *defect database* or *defect repository* contains details of all the defects that are found in various products that are tested in a particular organization

- It contains defects and all the related information (when the defect was found, to whom it is assigned, what is the current status, the type of defect, its impact, and so on).
- Test engineers submit the defects for manual test cases. For automated test cases, the framework can automatically submit the defects to the defect DB during execution.
- These external modules can be accessed by any module in automation framework, not just one or two modules
- In Figure the "*green*" thick arrows show specific interactions and "*blue"* thick arrows show multiple interactions.

5.2 Scenario and Configuration File Modules

- *scenarios* are nothing but information on "how to execute a particular test case."
- A *configuration file* contains a set of variables that are used in automation
- The variables could be for the test framework or for other modules in automation such as tools and metrics or for the test suite or for a set of test cases or for a particular test case
- A configuration file is important for running the test cases for various execution conditions and for running the tests for various input and output conditions and states
- The values of variables in this configuration file can be changed dynamically to achieve different execution, input, output, and state conditions

5.3 Test Cases and Test Framework Modules

- A *test case* in Figure means the automated test cases that are taken from TCDB and executed by the framework. Test case is an object for execution for other modules in the architecture and does not represent any interaction by itself.
- A *test framework* is a module that combines "what to execute" and "how they have to be executed."
- It picks up the specific test cases that are automated from TCDB and picks up the scenarios and executes them
- The variables and their defined values are picked up by the test framework and the test cases are executed for those values
- The test framework is considered the core of automation design

- It subjects the test cases to different scenarios.
- For example, if there is a scenario that requests a particular test case be executed for 48 hours in a loop, then the test framework executes those test cases in the loop and times out when the duration is met.
- The framework monitors the results of every iteration and the results are stored.
- The test framework contains the main logic for interacting, initiating, and controlling all modules.

- A test framework can be developed by the organization internally or can be bought from the vendor.

- Test framework and test tool are the two terms that are used interchangeably
- To differentiate between the usage of these two terms (wherever needed), in this chapter "framework" is used to mean an internal tool developed by the organization and "test tool" is used to mean a tool obtained from a tool vendor

5.4 Tools and Results Modules

- When a test framework performs its operations, there are a set of tools that may be required. For example, when test cases are stored as source code files in TCDB, they need to be extracted and compiled by build tools

- In order to run the compiled code, certain runtime tools and utilities may be required.
- For example, IP Packet Simulators or User Login Simulators or Machine Simulators may be needed. In this case, the test framework invokes all these different tools and utilities.
- When a test framework executes a set of test cases with a set of scenarios for the different values provided by the configuration file, the results for each of the test case along with scenarios and variable values have to be stored for future analysis and action
- The results that come out of the tests run by the test framework should not overwrite the results from the previous test runs
- The history of all the previous tests run should be recorded and kept as archives

- The archive of results help in executing test cases based on previous test results.
- For example, a test engineer can request the test framework to "execute all test cases that are failed in previous test run
- " The audit of all tests that are run and the related information are stored in the module of automation.

5.5 Report Generator and Reports/Metrics Modules

- Once the results of a test run are available, the next step is to prepare the test reports and metrics. Preparing reports is a complex and time-consuming effort and hence it should be part of the automation design
- There should be customized reports such as an executive report, which gives very high level status; technical reports, which give a moderate level of detail of the tests run; and detailed or debug reports which are generated for developers to debug the failed test cases and the product.
- The periodicity of the reports is different, such as daily, weekly, monthly, and milestone reports.

- Having reports of different levels of detail and different periodicities can address the needs of multiple constituents and thus provide significant returns.
- The module that takes the necessary inputs and prepares a formatted report is called a *report generator*. Once the results are available, the report generator can generate *metrics*
- All the reports and metrics that are generated are stored in the reports/metrics module of automation for future use and analysis

GENERIC REQUIREMENTS FOR TEST TOOL/FRAMEWORK

- we described a generic framework for test automation. We will now present certain detailed criteria that such a framework and its usage should satisfy.

- Requirement 1: No hard coding in the test suite
- One of the important requirements for a test suite is to keep all variables separately in a file
- By following this practice, the source code for the test suite need not be modified every time it is required to run the tests for different values of the variables

- The file in which all variable names and their associated values are kept is called *configuration file*
- The variables belonging to the test tool and the test suite need to be separated so that the user of the test suite need not worry about test tool variables.
- Providing inline comment for each of the variables will make the test suite more usable and may avoid improper usage of variables
- To summarize
- Adding a test case should not affect other test cases
- Adding a test case should not result in retesting the complete test suite
- Adding a new test suite to the framework should not affect existing test suites

- Requirement 2: Test case/suite expandability
- As we have seen in the "log in" example, the functionality of the product when subjected to different scenarios becomes test cases for different types of testing
- This encourages the reuse of code in automation
- The reuse of code is not only applicable to various types of testing; it is also applicable for modules within automation.
- All those functions that are needed by more than one test case can be separated and included in libraries
- When writing code for automation, adequate care has to be taken to make them modular by providing functions, libraries and including files

- To summarize
- The test suite should only do what a test is expected to do. The test framework needs to take care of "how,"
- The test programs need to be modular to encourage reuse of code.

- Requirement 3: Reuse of code for different types of testing, test cases
- The test cases may expect some objects to be created or certain portions of the product to be configured in a particular way.
- If this portion is not met by automation, then it introduces some manual intervention before running the test cases
- When test cases expect a particular setup to run the tests, it will be very difficult to remember each one of them and do the setup accordingly in the manual method.
- Hence, each test program should have a "setup" program that will create the necessary setup before executing the test cases
- The test framework should have the intelligence to find out what test cases are executed and call the appropriate setup program.

- Requirement 4: Automatic setup and cleanup
- A setup for one test case may work negatively for another test case. Hence, it is important not only to create the setup but also "undo" the setup soon after the test execution for the test case.
- Hence, a "cleanup" program becomes important and the test framework should have facilities to invoke this program after test execution for a test case is over
- Requirement 5: Independent test cases
- Making test cases independent enables any one case to be selected at random and executed.
- Making a test case dependent on an other makes it necessary for a particular test case to be executed before or after a dependent test case is selected for execution.

- A test tool or a framework should provide both features. The framework should help to specify the dynamic dependencies between test cases.
- Requirement 6: Test case dependency
- Insulating test cases from the environment is an important requirement for the framework or test tool.
- At the time of test case execution, there could be some events or interrupts or signals in the system that may affect the execution
- Consider the example of automatic pop-up screens on web browsers. When such pop-up screens happen during execution, they affect test case execution as the test suite may be expecting some other screen based on an earlier step in the test case.

- Requirement 7: Insulating test cases during execution
- Hence, to avoid test cases failing due to some unforeseen events, the framework should provide an option for users to block some of the events.
- There has to be an option in the framework to specify what events can affect the test suite and what should not.
- Coding standards and proper directory structures for a test suite may help the new engineers in understanding the test suite fast and help in maintaining the test suite. Incorporating the coding standards improves portability of the code
- The test framework should provide an option or force the directory structure to enable multiple programmers to develop test suites/test cases in parallel, without duplicating the parts of the test case and by reusing the portion of the code.

- Requirement 8: Coding standards and directory structure
- A framework may have multiple test suites; a test suite may have multiple test programs; and a test program may have multiple test cases
- The test tool or a framework should have a facility for the test engineer to select a particular test case or a set of test cases and execute them. The selection of test cases need not be in any order and any combination should be allowed.

- Requirement 9: Selective execution of test cases
- ***Example:***
- test-program-name 2, 4, 1, 7-10
- In the above scenario line, the test cases 2, 4, 1, 7, 8, 9, 10 are selected for execution in the same order mentioned
- The hyphen (-) is used to mention the test cases in the chosen range—(7-10) have all to be executed
- If the test case numbers are not mentioned in the above example, then the test tool should have the facility to execute all the test cases

- Requirement 10: Random execution of test cases
- While it is a requirement for a test engineer to select test cases from the available test cases as discussed in requirement 8 above, the same test engineer may sometimes need to select a test case randomly from a list of test cases.
- Giving a set of test cases and expecting the test tool to select the test case is called random execution of test cases. A test engineer selects a set of test cases from a test suite; selecting a random test case from the given list is done by the test tool.

Example 1:

```
random
   test-program-name 2, 1, 5
```

Example 2:

```
random
   test-programl (2, 1, 5 )
   test-program2
   test-program3
```

- In the first example, the test engineer wants the test tool to select one out of test cases 2, 1, 5 and executed
- In the second example, the test engineer wants one out of test programs 1, 2, 3 to be randomly executed and if program 1 is selected, then one out of test cases 2, 1, 5 to be randomly executed.
- In this example if test programs 2 or 3 are selected, then all test cases in those programs are executed

- Requirement 11: Parallel execution of test **cases**
- There are certain defects which can be unearthed if some of the test cases are run at the same time.

- In a multi-tasking and multi processing operating systems it is possible to make several instances of the tests and make them run in parallel. Parallel execution simulates the behavior of several machines running the same test and hence is very useful for performance and load testing.

Example 1:

```
instances,5
   Test-program1(3)
```

Example 2:

```
Time_loop, 5 hours
   test-program1(2, 1, 5)
test-program2
   test-program3
```

- In the first example above, 5 instances of test case 3 from test program l, are created; in the second example, 5 instances of 3 test programs are created. Within each of the five instances that are created, the test programs 1, 2, 3 are executed in sequence

- Requirement 12: Looping the test cases
- As discussed earlier, reliability testing requires the test cases to be executed in a loop. There are two types of loops that are available
- One is the *iteration* loop which gives the number of iterations of a particular test case to be executed. The other is the *timed* loop, which keeps executing the test cases in a loop till the specified time duration is reached. These tests bring out reliability issues in the product.

Example 1:

```
Repeat_loop, 50
   Test-program1(3)
```

Example 2:

```
Time_loop, 5 hours
   test-program1(2, 1, 5)
test-program2
   test-program3
```

- In the first example, test case 3 from test program 1 is repeated 50 times and in the second example, test cases 2, 1, 5 from test program1 and all test cases from test programs 2 and 3 are executed in order, in a loop for five hours
- Requirement 13: Grouping of test scenarios
- We have seen many requirements and scenarios for test execution. Now let us discuss how we can combine those individual scenarios into a group so that they can run for a long time with a good mix of test cases
- The group scenarios allow the selected test cases to be executed in order, random, in a loop all at the same time. The grouping of scenarios allows several tests to be executed in a predetermined combination of scenarios

- The following is an example of a group scenario.
- **Example:**
- group_scenario1
- parallel, 2 AND repeat, 10 @ scenl
- scen1
- test_program1 (2, 1, 5)
- test_program2
- test_program3
- In the above example, the group scenario was created to execute two instances of the individual scenario "scen1" in a loop for 10 times. The individual scenario is defined to execute test program1 (test cases 2, 1 and 5), test program2 and test program3. Hence, in the combined scenario, all test programs are executed by two instances simultaneously in an iteration loop for 10 times.

Requirement 14: Test case execution based on previous results

As we have seen , regression test methodology requires that test cases be selected based on previous result and history

Hence, automation may not be of much help if the previous results of test execution are not considered for the choice of tests. Not only for regression testing, it is a requirement for various other types of testing also. One of the effective practices is to select the test cases that are not executed and test cases that failed in the past and focus more on them. Some of the common scenarios that require test cases to be executed based on the earlier results are

□ Rerun all test cases which were executed previously;

□ Resume the test cases from where they were stopped the previous time;

□ Rerun only failed/not run test cases; and

□ Execute all test cases that were executed on "Jan 26, 2005."

□ With automation, this task becomes very easy if the test tool or the framework can help make such choices.

Requirement 15: Remote execution of test cases

□ Most product testing requires more than one machine to be used. Hence there is a facility needed to start the testing on multiple machines at the same time from a central place

□ . The central machine that allocates tests to multiple machines and co-ordinates the execution and result is called *test console or test monitor.*

□ In the absence of a test console, not only does executing the results from multiple machines become difficult, collecting the results from all those machines also becomes difficult. In the absence of a test console, the results of tests need to be collected manually and consolidated

□ As it is against the objective of automation to introduce a manual step, this requirement is important for the framework to have

□ To summarize

□ It should be possible to execute/stop the test suite on any machine/set of machines from the test console.

□ The test results and logs can be collected from the test console.

□ The progress of testing can be found from the test console.

❑ CHALLENGES IN AUTOMATION

❑ As can be inferred from the above sections, test automation presents some very unique challenges. The most important of these challenges is management commitment.

❑ Automation should not be viewed as a panacea for all problems nor should it be perceived as a quick-fix solution for all the quality problems in a product.

❑ Automation takes time and effort and pays off in the long run

❑ However, automation requires significant initial outlay of money as well as a steep learning curve for the test engineers before it can start paying off.

❑ Management should have patience and persist with automation

❑ The main challenge here is because of the heavy front-loading of costs of test automation, management starts to look for an early payback

❑ Successful test automation endeavours are characterized by unflinching management commitment, a clear vision of the goals, and the ability to set realistic short-term goals that track progress with respect to the long-term vision

❑ When any of these attributes are found lacking in the management team, it is likely that the automation initiative will fail

□ SELECTING A TEST TOOL

□ Having identified the requirements of what to automate, a related question is the choice of an appropriate tool for automation

□ Selecting the test tool is an important aspect of test automation for several reasons as given below

1. Free tools are *not well supported and get phased out* soon. It will be extremely dangerous to see a release stalled because of a problem in a test tool.

2. Developing in-house tools *takes time.* Even though in-house tools can be less expensive and can meet needs better, they are often developed by the personal interest shown by a few engineers.

□ They tend to have poor documentation and thus, once the person who developed the tools leaves the organization, the tools become unusable. Furthermore, such tool development takes a back seat if the pressure of actual product testing and delivery dates comes into play. Hence, this cannot be a sustained effort

3. Test tools sold by vendors are *expensive*

4. Test tools require strong *training*

□ Test automation cannot be successful unless the people using the tools are properly trained

5. Test tools generally *do not meet all the requirements* for automation

□ Since tools are meant to be generic, they may not fully satisfy the needs of a particular customer

6. Not all test tools run on all platforms

□To amortize the costs of automation, the tools and the automated tests should be reusable on all the platforms on which the product under test runs. Portability of the tool and the scripts to multiple platforms is therefore a key factor in deciding the test automation tool.

Criteria for Selecting Test Tools

□we looked at some reasons for evaluating the test tools and how requirements gathering will help.

□ These change according to context and are different for different companies and products. We will now look into the broad categories for classifying the criteria.

□ The categories are

□ Meeting requirements;

□ Technology expectations;

□ Training/skills; and

□ Management aspects.

❑ Debugging

❑ Overview

❑ In brief, debugging is what you do after you have executed a successful test case

❑ Remember that a successful test case is one that shows that a program does not do what it was designed to do.

❑ Debugging is a two-step process that begins when you find an error as a result of a successful test case

❑ Step 1 is the determination of the exact nature and location of the suspected error within the program.

❑ Step 2 consists of fixing the error

- These seem to be the main reasons:
- Your ego may get in the way
- Like it or not, debugging confirms that programmers are not perfect, committing errors in either the design or the coding of the program
- You may run out of steam
- Debugging is the most mentally taxing activity. Moreover, debugging usually is performed under a tremendous amount of organizational or self-induced pressure to fix the problem as quickly as possible.
- You may lose your way
- Compared to other software development activities, comparatively little research, literature, and formal instruction exist on the process of debugging

- Debugging by Brute Force
- The most common scheme for debugging a program is the "brute force" method.
- It is popular because it requires little thought and is the least mentally taxing of the methods, but it is inefficient and generally unsuccessful.
- Brute force methods can be partitioned into at least three categories
- 1. Debugging with a storage dump.
- 2. Debugging according to the common suggestion to "scatter print statements throughout your program."
- 3. Debugging with automated debugging tools

- The first, debugging with a storage dump (usually a crude display of all storage locations in hexadecimal or octal format) is the most inefficient of the brute force methods. Here's why:
- It is difficult to establish a correspondence between memory locations and the variables in a source program
- With any program of reasonable complexity, such a memory dump will produce a massive amount of data, most of which is irrelevant
- A memory dump is a static picture of the program, showing the state of the program at only one instant in time; to find errors, you have to study the dynamics of a program (state changes over time).

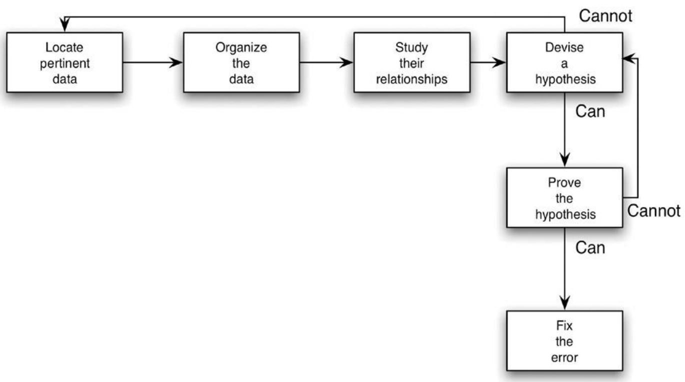
- Automated debugging tools work similarly to inserting print statements within the program, but rather than making changes to the program, you analyze the dynamics of the program with the debugging features of the programming language or special interactive debugging tools.
- Typical language features that might be used are facilities that produce printed traces of statement executions, subroutine calls, and/or alterations of specified variables.
- A common function of debugging tools is the ability to set breakpoints that cause the program to be suspended when a particular statement is executed or when a particular variable is altered, and then the programmer can examine the current state of the program.

- Again, this method is largely hit or miss and often results in an excessive amount of irrelevant data.
- The general problem with these brute force methods is that they ignore the process of thinking.
- we could recommend brute force methods only

  (1) when all other methods fail or

  (2) as a supplement to, not a substitute for, the thought processes we'll describe next.

- Debugging by Induction
- It should be obvious that careful thought will find most errors without the debugger even going near the computer
- One particular thought process is induction, where you move from the particulars of a situation to the whole
- That is, start with the clues (the symptoms of the error, possibly the results of one or more test cases) and look for relationships among the clues. The induction process is illustrated in Figure



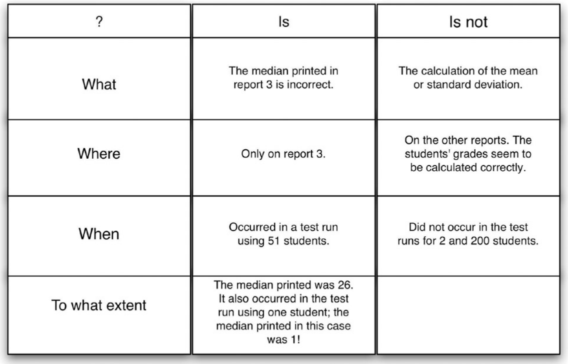
- *Locate the pertinent data* : A major mistake debuggers make is failing to take account of all available data or symptoms about the problem
- The first step is the enumeration of all you know about what the program did correctly and what it did incorrectly
- *Organize the data* : the second step is to structure the pertinent data to let you observe the patterns.
- Of particular importance is the search for contradictions, events such as that the error occurs only when the customer has no outstanding balance in his or her margin account.
- You can use a form such as the one shown in Figure to structure the available data

?	Is	Is not
What		
Where		
When		
To what extent		

- The "what" boxes list the general symptoms, the "where" boxes describe where the symptoms were observed, the "when" boxes list anything that you know about the times that the symptoms occur, and the "to what extent" boxes describe the scope and magnitude of the symptoms.

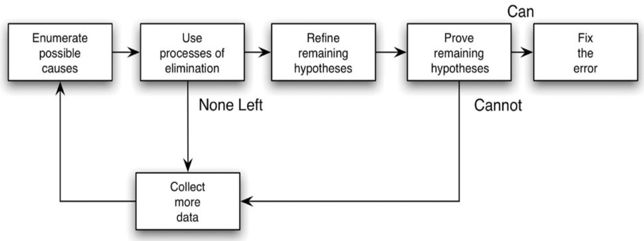
- *Devise a hypothesis* : study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error.
- it is vital to prove the reasonableness of the hypothesis before you proceed. If you skip this step, you'll probably succeed in correcting only the problem symptom, not the problem itself
- Prove the hypothesis by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues.

If it does not, either the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present



?	Is	Is not
What	The median printed in report 3 is incorrect.	The calculation of the mean or standard deviation.
Where	Only on report 3.	On the other reports. The students' grades seem to be calculated correctly.
When	Occurred in a test run using 51 students.	Did not occur in the test runs for 2 and 200 students.
To what extent	The median printed was 26. It also occurred in the test run using one student; the median printed in this case was 1!	

- **Debugging by Deduction**
- The process of deduction proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion


Enumerate possible causes
Use processes of elimination
Refine remaining hypotheses
Prove remaining hypotheses
Can
Fix the error
None Left
Cannot
Collect more data

- you induce a suspect from the clues, you start with a set of suspects and, by the process of elimination and refinement ,decide who must have done it. The steps are as follows:
- Enumerate the possible causes or hypotheses
  - The first step is to develop a list of all conceivable causes of the error. They don't have to be complete explanations; they are merely theories to help you structure and analyze the available data
- Use the data to eliminate possible causes
  - Carefully examine all of the data, particularly by looking for contradictions (Figure could be used here), and try to eliminate all but one of the possible causes.

  - If all are eliminated, you need more data through additional test cases to devise new theories.
  - If more than one possible cause remains, select the most probable cause—the prime hypothesis—first
- Refine the remaining hypothesis
  - The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error.
  - Hence, the next step is to use the available clues to refine the theory.
  - For example, you might start with the idea that "there is an error in handling the last transaction in the file" and refine it to "the last transaction in the buffer is overlaid with the end-of-file indicator."

- Prove the remaining hypothesis
  - vital step is identical to step 4 in the induction method.
- Debugging by Backtracking
  - An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went astray
  - In other words, start at the point where the program gives the incorrect result—such as where incorrect data were printed
  - At this point you deduce from the observed output what the values of the program's variables must have been.

- By performing a mental reverse execution of the program from this point and repeatedly using the process of "if this was the state of the program at this point, then this must have been the state of the program up here you can quickly pinpoint the error.
- With this process you're looking for the location in the program between the point where the state of the program was what was expected and the first point where the state of the program was what was not expected.

- Debugging by Testing
- The last "thinking type" debugging method is the use of test cases
- This probably sounds a bit peculiar since the beginning of this chapter distinguishes debugging from testing.
- However, consider two types of test cases:
- test cases for testing, where the purpose of the test cases is to expose a previously undetected error, and test cases for debugging, where the purpose is to provide information useful in locating a suspected error.

- The difference between the two is that test cases for testing tend to be "fat" because you are trying to cover many conditions in a small number of test cases.
- Test cases for debugging, on the other hand, are "slim" since you want to cover only a single condition or a few conditions in each test case.
- In other words, after a symptom of a suspected error is discovered, you write variants of the original test case to attempt to pinpoint the error
- Actually, this method is not an entirely separate method; it often is used in conjunction with the induction method to obtain information needed to generate a hypothesis and/or to prove a hypothesis

- □ It also is used with the deduction method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis.
- □ Debugging Principles
- □ Error-Locating Principles
  - ■ Think : The most effective method of debugging is a mental analysis of the information associated with the error's symptoms. An efficient program debugger should be able to pinpoint most errors without going near a computer.

- □ If You Reach an Impasse, Sleep on It
  - ■ The human subconscious is a potent problem solver
  - ■ What we often refer to as inspiration is simply the subconscious mind working on a problem when the conscious mind is working on something else such as eating, walking, or watching a movie
  - ■ If you cannot locate an error in a reasonable amount of time (perhaps 30 minutes for a small program, several hours for a larger one), drop it and work on something else, since your thinking efficiency is about to collapse anyway.
  - ■ After forgetting about the problem for a while, your subconscious mind will have solved the problem, or your conscious mind will be clear for a fresh examination of the symptoms
- □ If You Reach an Impasse, Describe the Problem to Someone Else

- ❑ Talking about the problem with someone else may help you discover something new. In fact, often simply by describing the problem to a good listener, you will suddenly see the solution without any assistance from the listener
- ❑ Use Debugging Tools Only as a Second Resort
  - ❑ Use debugging tools after you've tried other methods, and then only as an adjunct to, not a substitute for, thinking debugging tools, such as dumps and traces, represent a haphazard approach to debugging
  - ❑ Experiments show that people who shun such tools, even when they are debugging programs that are unfamiliar to them, are more successful than people who use the tools.

- □ Avoid Experimentation—Use It Only as a Last Resort
  - ■ The most common mistake novice debuggers make is trying to solve a problem by making experimental changes to the program
  - ■ You might say, "I know what is wrong, so I'll change this DO statement and see what happens."
  - ■ This totally haphazard approach cannot even be considered debugging; it represents an act of blind hope

- ❑ **Error-Repairing Techniques**
- ❑ Where There Is One Bug, There Is Likely to Be Another
  - ❑ when you find an error in a section of a program, the probability of the existence of another error in that same section is higher than if you hadn't already found one error
  - ❑ In other words, errors tend to cluster. When repairing an error, examine its immediate vicinity for anything else that looks suspicious
- ❑ Fix the Error, Not Just a Symptom of It
  - ❑ Another common failing is repairing the symptoms of the error, or just one instance of the error, rather than the error itself. If the proposed correction does not match all the clues about the error, you may be fixing only a part of the error.

- □ The Probability of the Fix Being Correct Is Not 100 Percent
  - ■ You can never assume that code added to a program to fix an error is correct. Statement for statement, corrections are much more error prone than the original code in the program.
  - ■ A solid regression testing plan can help ensure that correcting an error does not induce another error somewhere else in the application.
- □ The Probability of the Fix Being Correct Drops as the Size of the Program Increases
  - ■ Stating it differently, in our experience the ratio of errors due to incorrect fixes versus original errors increases in large programs. In one widely used large program, one of every six new errors discovered is an error in a prior correction to the program

- □ Beware of the Possibility That an Error Correction Creates a New Error
  - ■ Not only do you have to worry about incorrect corrections, but also you have to worry about a seemingly valid correction having an undesirable side effect, thus introducing a new error
  - ■ One implication is that not only does the error situation have to be tested after the correction is made, but you must also perform regression testing to determine whether a new error has been introduced.

- ❑ The Process of Error Repair Should Put You Temporarily Back into the Design Phase
  - ❑ Given the error-prone nature of corrections, common sense says that whatever procedures, methodologies, and formalism were used in the design process should also apply to the error-correction process
- ❑ Change the Source Code, Not the Object Code
  - ❑ When debugging large systems, particularly a system written in an assembly language, occasionally there is the tendency to correct an error by making an immediate change to the object code with the intention of changing the source program later. Two problems associated with this approach are

  - ❑ (1) it usually is a sign that "debugging by experimentation" is being practiced, and
  - ❑ (2) the object code and source program are now out of synchronization, meaning that the error could easily surface again when the program is recompiled or reassembled