JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self-Belief | Self Discipline | Self Respect"

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

IT8076 – SOFTWARE TESTING

(Regulation 2017)

Year/Semester: III/VI CSE

2020 – 2021

Prepared by

Ms. R. Revathi

Assistant Professor/CSE

UNIT V TEST AUTOMATION

Software test automation – skills needed for automation – scope of automation – design and architecture for automation – requirements for a test tool – challenges in automation – Test metrics and measurements – project, progress and productivity metrics.

Software Test Automation

WHAT IS TEST AUTOMATION?

Developing software to test the software is called test automation.

Test automation can help address several problems.

- **Automation save time as software can execute test cases faster than human do** . The time thus saved can be used effectively for test engineers to

1. develop additional test cases to achieve better coverage;
2. perform some esoteric or specialized tests like ad hoc testing; or
3. 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.

- **Test automation can free the test engineers from mundane tasks and make them focus on more creative tasks. -**E.g- Ad hoc 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.
  Automating the more mundane tasks gives some time to the test engineers for creativity and challenging tasks.
- **Automated tests can be more reliable** -when an engineer executes a particular test case many times manually, there is a chance for human error. As with all machine-oriented activities, automation can be expected to produce more reliable results every time, and eliminates the factors of boredom and fatigue.
- **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. 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.
- **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 words, in different time zones, to monitor and control the tests, thus providing round the- clock coverage.

- **Certain types of testing cannot be executed without automation-**For example, if we want to study the behavior of a system with thousands of users logged in, there is now way one can perform these tests without using automated tolls.
- **Automation means end-to-end, not test execution alone -**Automation should consider all activities such as picking up the right product build, choosing the right configuration, performing installation, running the tests, generating the right test data, analyzing the results, and filling the defects in the defect repository. When talking about automation, this large picture should always be kept in mind.

TERMS USED IN AUTOMATION

A test case is a set of sequential steps to execute a test operating on a set of predefined inputs to produce certain expected outputs. There are two types of test cases – automated and manual. **A manual test case** is executed manually while an **automated test** case is executed using automation.

As we have seen earlier , testing involves several phases and several types of testing. Some test cases are repeated several times during a product release, because the product is built several times. Table describes some test cases for the log in example, on how the login can be tested for different types of testing.

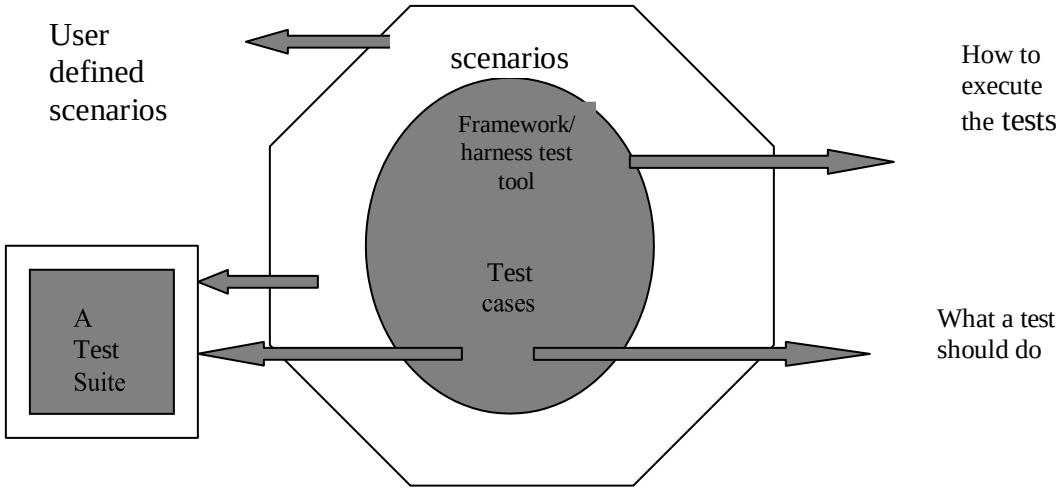
S.No	Test Cases for Testing	Belongs to What type of testing
1.	Check whether login works	Functionality
2.	Repeat Login operation in a loop for 48 hours	Reliability
3.	Perform Login from 10000 clients	Load /Stress Testing
4.	Measure time taken for Login operations in different conditions	Performance
5.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

From the above table , it is observed that there are 2 important dimensions

1) What operations have to be tested
2) How the operations have to be tested → scenarios

The how portion is called → scenarios.
What an operation has to do → product specific feature
How they are to be run → framework specific requirement
They are the generic requirements for all products that are being tested in an organization.

When a set of test cases is combined and associated with a set of scenarios, they are called "**test suite**". A test suite is nothing but a set of test cases that are automated and scenarios that are associated with the test cases.



Framework for test automation

SKILLS NEEDED FOR AUTOMATION

There are different "Generations of Automation". The skills required for automation depends on what generation automation the company is in or desires to be in the near future.

The automation of testing is broadly classified into three generations.

a. First generation – Record and playback

- Record and playback avoids the repetitive nature of executing tests. Almost all the test tools available in the market have the record and playback feature.
- 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. 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.
- When the application changes, all the scripts have to be re-recorded, thereby increasing the test maintenance costs.

b. 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.

c. 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 and everything else that is needed for those actions are automatically generated.
- Hence, automation in the third generation involves two major aspects- "test case automation" and "framework design".

Classification of skills for automation

Automation-first generation	**Automation- second generation**	**Automation-third generation**	
Skills for test case automation	**Skills for test case automation**	**Skills for test case automation**	**Skills for framework**
Scripting languages	Scripting languages	Scripting languages	Programming languages
Record- playback tools usage	Programming languages	Programming languages	Design and architecture skills for framework creation
	Knowledge of data generation techniques	Design and architecture of the product under test	Generic test requirements for multiple products
	Usage of the product under test	Usage of the framework	

SCOPE OF AUTOMATION

1. Identifying the Types of Testing Amenable to Automation

Certain types of tests automatically lend themselves to automation

a. **Stress, reliability, scalability, and performance testing** these types of testing require the test cases to be run form a large number of different machines for an extended period of time, such as 24 hours, 48 hours, and so on. Test cases belonging to these testing types become the first candidates for automation.

b. **Regression tests** Regression tests are repetitive in nature. These test cases are executed multiple times during the product development phases.

c. **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.

2. Automating Areas Less Prone To Change

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.

3. Automate Tests That Pertain to Standards

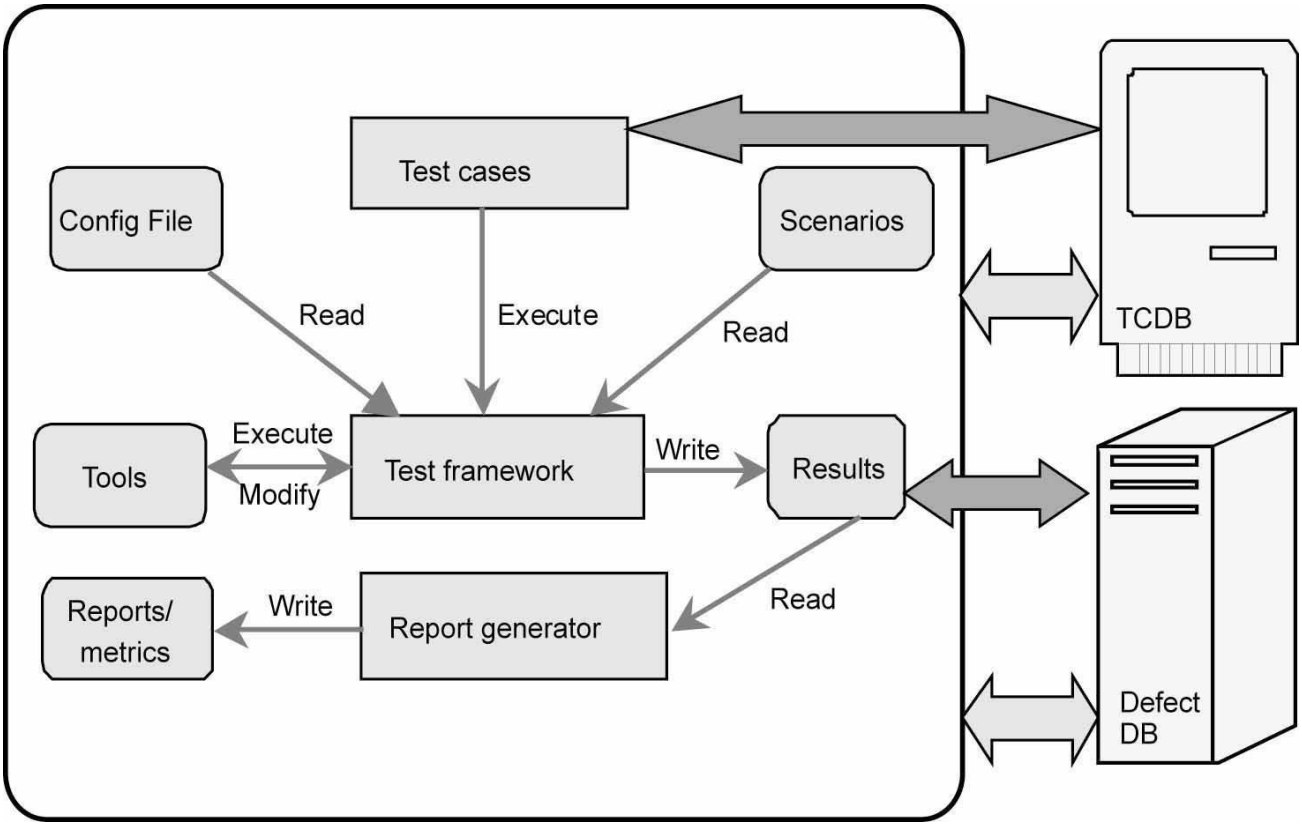
One of the tests that product may have to undergo is compliance to standards. For example, a product providing a JDBC interface should satisfy the standard JDBC tests. Automating for standards provides a dual advantage. Test suites developed for standards are not only used for product testing but can also be sold as test tools for the market.

4. Management Aspects in Automation

Prior to starting automation, adequate effort has to be spent to obtain management commitment. It involves significant effort to develop and maintain automated tools; obtaining management commitment is an important activity. Return on investment is another aspect to be considered seriously.

DESIGN AND ARCHITECTURE FOR 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 and in the interactions between modules.



Integration Testing, both internal interfaces and external interfaces have to be captured by design and architecture. In this figure the thin arrows represent the internal interfaces and the direction of flow and thick arrows show the external interfaces. All the

modules, their purpose, and interactions between them are described in the subsequent sections.

Architecture for test automation involves two major heads: a test infrastructure that covers a test case database and a defect database or defect repository. Using this infrastructure, the test framework provides a backbone that ties the selection and execution of test cases.

1. 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 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.
- 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 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.

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. 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.

3. Test Cases and Test Framework Modules

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 test framework is considered the core of automation design. It subjects the test cases to different scenarios. 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.

4. Tools and Result 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 eg , IP Packet Simulators. The result that comes 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.

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 test run; and detailed or debug reports which are generated for developers to debug the failed test cases and the product.
- 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.

GENERIC REQUIREMENTS FOR TEST TOOL/FRAMEWORK

Requirement 1: No hard coding in the test suite

- The variables for the test suite are called configuration 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.
- Changing test tool variables, without knowing their purpose, may impact the results of the tests.
- Providing inline comment for each of the variables will make the test suite more usable and may avoid improper usage of variables.

Ex: well documented config file

```
#Test Framework Configuration Parameter
TOOL_PATH =/tools
COMMONLIB_PATH =/tools/crm/lib
SUITE_PATH =/tools/crm

#parameter common to all the test cases in the test
VERBOSE_LEVEL =3
MAX_ERRORS=200
USER_PASSWD =hello123

#Test Case1 Parameter
TC1_USR_CREATE =0 # 1=yes 0=no
TC1_USR_PASSWD=hello123
TC1_MAX_USRS =200
```

Requirement 2: Test case/ suite expandability

Points to considered during expansion are

- ❖ 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 3: Reuse of code for different types of testing, test cases

Points to be considered during Reuse of codes are:

1) The test suite should only do what a test is expected to do. The test framework needs to take care of "how" and

2) The test programs need to be modular to encourage reuse of code.

Requirement 4: Automatic setup and cleanup

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.

A setup for one test case may work negatively for another test case. Hence, it is important not only to create.

Requirement 5: Independent test cases

Each test case should be executed alone; there should be no dependency between test cases such as test case-2 to be executed after test case-1 and so on. This requirement enables the test engineer to select and execute any test case at random without worrying about other dependencies.

Requirement 6: Test case dependency

Making a test case dependent on another makes it necessary for a particular test case to be executed before or after a dependent test case is selected for execution

Requirement 7: Insulating test cases during execution

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.

Requirement 8: Coding standards and directory structure

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.

Requirement 9: Selective execution of test cases

A Test Framework contains → many Test Suite
A Test Suite contains → many Test Program
A Test Program contains → many Test Cases

The selection of test cases need not be in any order and any combination should be allowed. Allowing test engineers to select test cases reduces the time. These selections are normally done as part of the scenario file. The selection of test cases can be done dynamically just before running the test cases, by editing the **scenario file**.

Example scenario file	**Meaning**
test-pgm-name 2,4,1,7-10	The test cases 2,4,1,7-10 are selected for execution
Tests-pgm-name	Executes all test cases

Requirement 10: Random execution of test cases

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.

Ex: **scenario file.**

Random test-pgm-name 2,1,5	test tool select one out of test cases 2,1,5 for execution
Random test-pgm-name1 (2,1,5) test-pgm-name2 test-pgm-name3	Test engineer wants one out of test program 1,2,3 to be randomly executed and if pgm-name1 is selected , then one out of test cases 2,1,5 to be randomly executed, if test program 2,3 are selected , then all TC in those 2 program are executed.

Requirement 11: parallel execution of test cases

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.

Ex: **scenario file.**

Instance, 5 test-pgm-name1 (3)	5 instances of test case 3 in test-pgm-name1 are executed
Instance, 5 test-pgm-name1 (2,1,5) test-pgm-name2 test-pgm-name3	5 instances of test programs are created , within each of the five instances that are created the test program 1,2,3, are executed in sequence .

Requirement 12: Looping the test cases

Reliability testing requires the test cases to be executed in a loop. There are two types of loops that are available.

1) iteration loop - gives the number of iterations of a particular test case to be executed.
2) timed loop - which keeps executing the test cases in a loop till the specified time duration is reached.

Ex: **scenario file.**

Repeat_loop, 50 test-pgm-name1 (3)	test case 3 in test-pgm-name1 is repeated 50 times.
Time_loop, 5 Hours test-pgm-name1 (2,1,5) test-pgm-name2 test-pgm-name3	TC 2,1,5 from test-pgm-name1 and all test cases from the test program2 and 3 are executed in order, in a loop for 5 hours

Requirement 13: Grouping of test scenarios

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.

Ex: **scenario file.**

Group_scenario1 Parallel, 2 AND repeat,10@scen1 Scen1 test-pgm1 (2,1,5) test-pgm2 test-pgm3	Group scenario was created to execute 2 instances of the individual scenario "scen1" in a loop 10 times

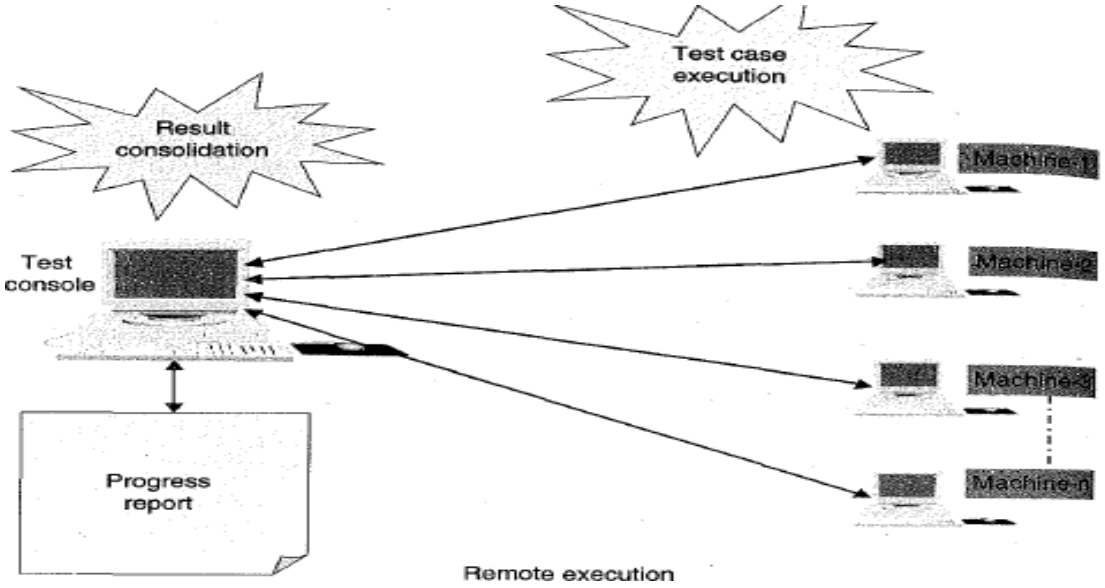
Requirement 14: Test case execution based on previous results

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

1. Rerun all test cases which were executed previously;
2. Resume the test cases from where they were stopped the previous time;
3. Rerun only failed/not run test cases; and
4. Execute all test cases that were executed previously.

Requirement 15: Remote execution of test cases

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.



Role of test console and multiple execution machine.

Requirement 16: Automatic archival of test data

The test cases have to be repeated the same way as before, with the same scenarios, same configuration variables and values, and so on. This requires that all the related information for the test cases have to be archived. It includes

1) What configuration variables were used
2) What scenario was used
3) What program were executed and from what path

Requirement 17: Reporting scheme

Every test suite needs to have a reporting scheme from where meaningful reports can be extracted. As we have seen in the design and architecture of framework, the report generator should have the capability to look at the results file and generate various reports. Audit logs are very important to analyze the behavior of a test suite and a product. A reporting scheme should include

1. When the framework, scenario, test suite, test program, and each test case were started/ completed;
2. Result of each test case;
3. Log messages;
4. Category of events and log of events; and
5. Audit reports

Requirement 18: Independent of languages

A framework or test tool should provide a choice of languages and scripts that are popular in the software development area.

- A framework should be independent of programming languages and scripts.
- A framework should provide choice of programming languages, scripts, and their combinations.
- A framework or test suite should not force a language/script.
- A framework or test suite should work with different test programs written using different languages, and scripts.
- The internal scripts and options used by the framework should allow the developers of a test suite to migrate to better framework.

Requirement 19: portability to different platforms

With the advent of platform-independent languages and technologies, there are many products in the market that are supported in multiple OS and language platforms.

- The framework and its interfaces should be supported on various platforms.
- Portability to different platforms is a basic requirement for test tool/ test suite.
- The language/script used in the test suite should be selected carefully so that it runs on different platforms.
- The language/ script written for the test suite should not contain platform-specific calls.

CHALLENGES IN AUTOMATION

- 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.
- 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 endeavors 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.

TEST METRICS AND MEASUREMENTS

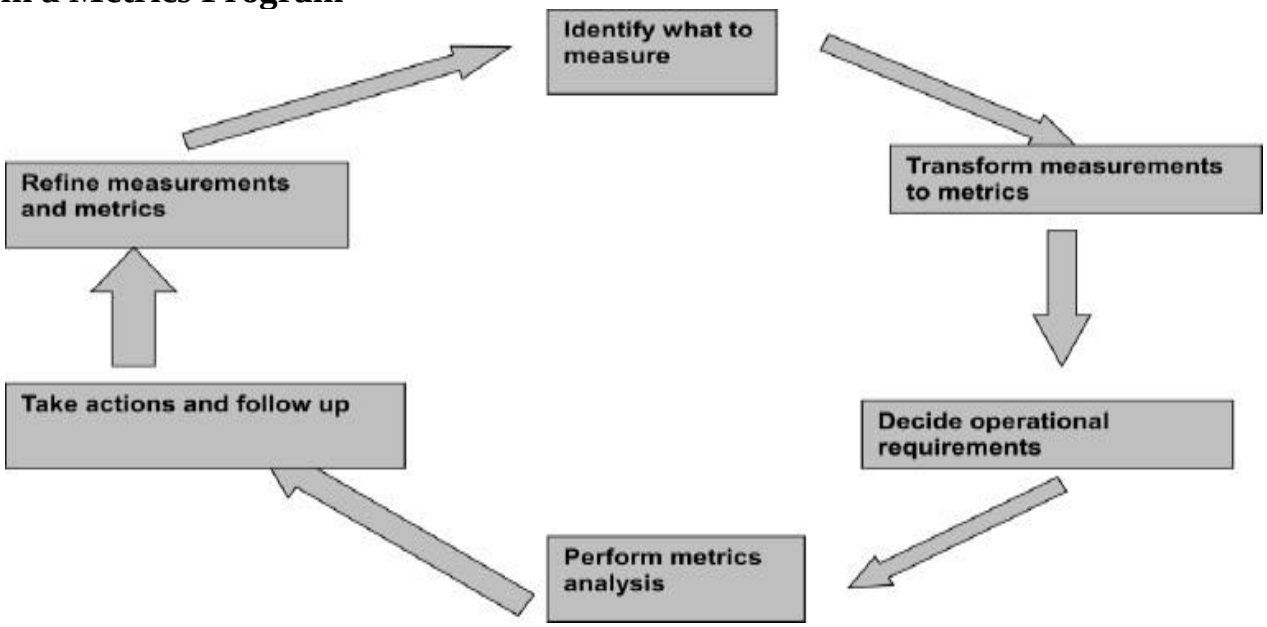
Definition

- Metrics are the source of measurement.
- Metrics derive information from raw data with a view to help in decision making.

Ex: No of defects , No of test cases , effort , schedule
Metrics are needed to know test case execution productivity and to estimate test completion date.

Effort : actual time that is spent on a particular activity or a phase Schedule : Elapsed days for a complete set of activities

Steps in a Metrics Program



Step1: Metrics program is to decide what measurements are important and collect data accordingly. Ex for Measurements: effort spent on testing , no of defects , no of test cases.

Step2: It deals with defining how to combine data points or measurement to provide meaningful metrics. A particular metric can use one or more measurements

Step3: It involves with operational requirement for measurements. It contains
Who should collect measurements?
Who should receive the analysis etc.
This step helps to decide on the appropriate periodicity for the measurements as well as assign operational responsibility for collecting, recording and reporting the measurements.
Daily basis measurements → no of testcases executed, no of defects found, defects fixed..
Weekly measurements → how may testcases produced 40 defects,

Step4: This step analyzes the metrics to identify both positive area and improvement areas on product quality.

Step5: The final step is to take necessary action and follow up on the action.

Step6: To continue with next iteration of metrics programs, measuring a different set of measurements, leading to more refined metrics addressing different issues.

WHY METRICS IN TESTING?

Knowing only how much testing got completed does not answer the question on when the testing will get completed and when the product is ready for release. To Answer these questions , one need to estimate the following

$$\text{Days needed to complete testing} = \frac{\text{Total test cases yet to be executed}}{\text{Total test case execution productivity}}$$

test case execution productivity → testcases executed per person day

$$\text{Total Days needed for defect fixes} = \frac{(\text{outstanding defects yet to fixed} + \text{Defects that can be found in future test cycles})}{\text{Defect fixing capability}}$$

$$\text{Days needed for Release} = \text{Max}(\text{Days needed for testing}, \text{days needed for defect fixes})$$

More accurate estimate with regression testing

$$\text{Days needed for Release} = \text{Max}(\text{Days needed for testing}, (\text{days needed for defect fixes} + \text{Days needed for regressing outstanding defect fixes}))$$

Metrics are needed to know test case execution productivity and to estimate test completion date.

Metrics in testing help in identifying

- When to make the release.
- What to release
- Whether the product is being released with known quality.

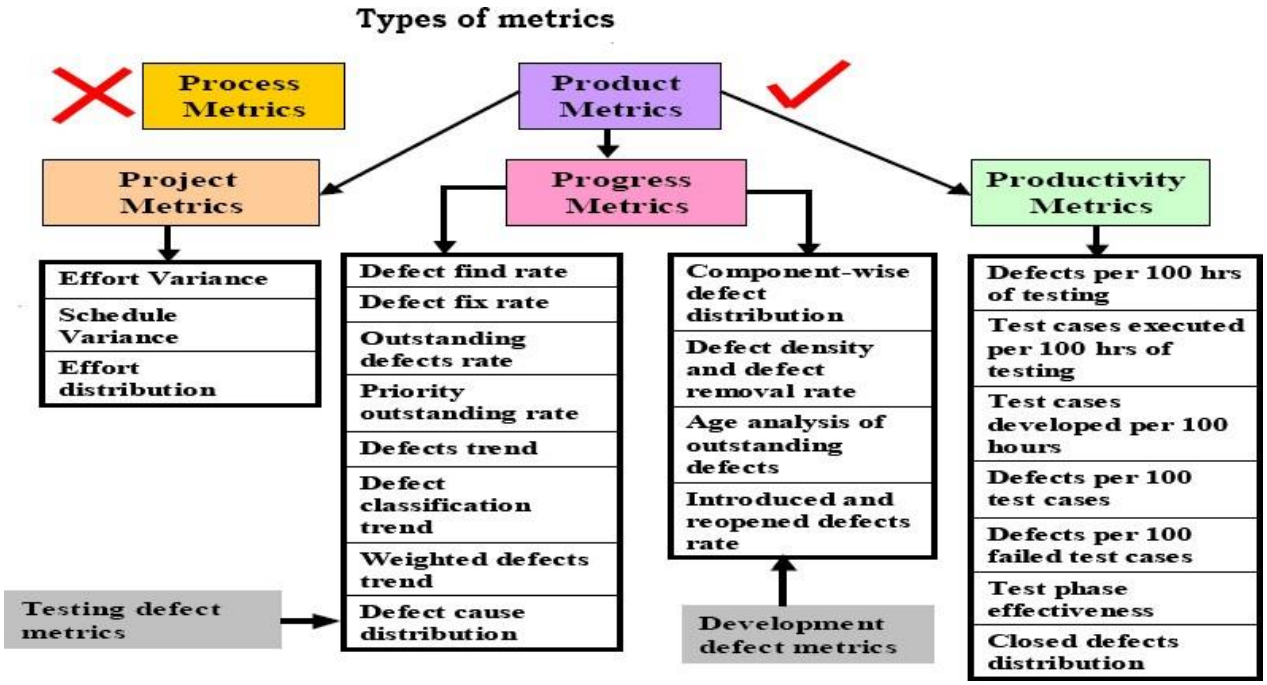
TYPES OF METRICS

Metrics can be classified into different types based on what they measure and what area they focus on. At a very high level, metrics can be classified as product metrics and process metrics.

Product metrics can be further classified as:

1. **Project metrics** A set of metrics that indicates how the project is planned and executed.
2. **Progress metrics** A set of metrics that tracks how the different activities of the project are progressing. The activities include both development activities and testing activities. Progress metrics is monitored during testing phases. Progress metrics helps in finding out the status of test activities and they are also good indicators of product quality. Progress metrics, for convenience, is further classified into
    1) test defect metrics and
    2) development defect metrics.

3. **Productivity metrics** A set of metrics that takes into account various productivity numbers that can be collected and used for planning and tracking testing activities. These metrics help in planning and tracking testing activities. These metrics help in planning and estimating of testing activities.



I) PROJECT METRICS

A typical project starts with requirements gathering and ends with product release. All the phases that fall in between these points need to be planned and tracked. In the planning cycle, the scope of the project is finalized. The project scope gets translated to size estimates, which specify the quantum of work to be done. This size estimate gets translated to effort estimate for each of the phases and activities by using the available productivity data available.

base lined effort →The initial effort

revised effort. → As the project progresses and if the scope of the project changes, then the effort estimates are re-evaluated again and this re-evaluated effort estimate is called revised effort.

Two factors **Effort :** actual time that is spent on a particular activity or a phase **Schedule :** Elapsed days for a complete set of activities

- If the effort is tracked closely & met then schedule can be met.
- If planned effort is equal to actual effort and schedule not met then project is not considered as successful one.
-

The basic measurements are

1. initial baselined effort and schedule
2. The actual effort
3. The revised estimate of effort and schedule

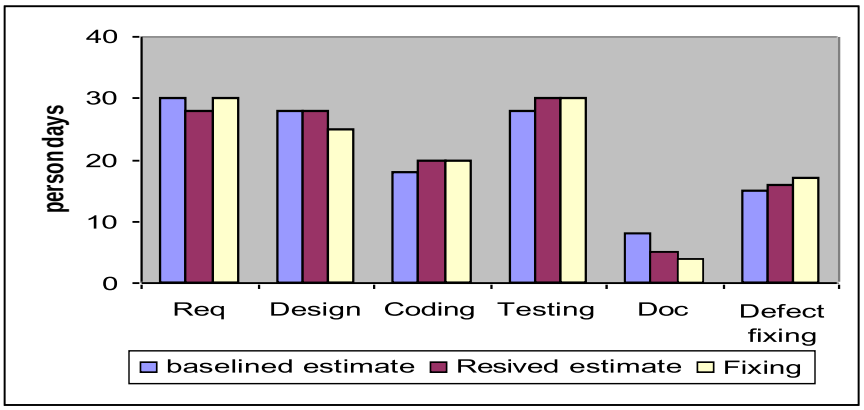
1. Effort Variance (Planned Vs Actual)

When the baselined effort estimates, revised effort estimates, and actual effort are plotted together for all the phases of SDLC, it provides many insights about the estimation process. As different set of people may get involved in different phases, it is a good idea to plot these effort numbers phase-wise. A sample data for each of the phase is plotted in the chart.

If there is a substantial difference between the baselined and revised effort, it points to incorrect initial estimation. Calculating effort variance for each of the phases provides a quantitative measure of the relative difference between the revised and actual efforts.

Calculating effort variance for each of the phases provides a quantitative measure of the relative difference between the revised and actual efforts.

$$\text{Effort variance \%} = \frac{\text{actual effort} - \text{revised estimate}}{\text{Revised estimate}} \times 100$$



Phase wise effort variation

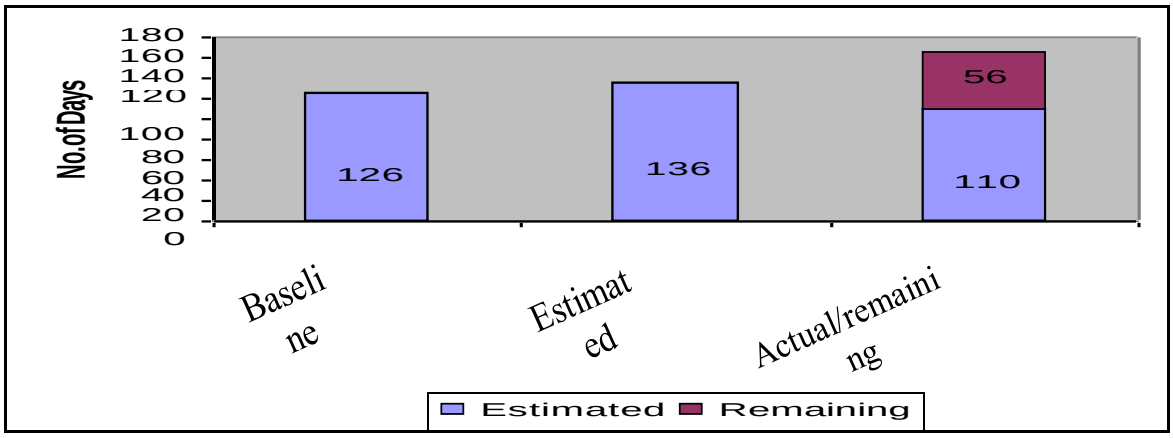
Sample variance percentage by phase.

Effort	Req	Design	Coding	Testing	Doc	Defect Fixing
Variance %	7.1	8.7	5	0	40	15

- All the baseline estimates, revised estimates, and actual effort are plotted together for each of the phases. The variance can be consolidated into as shown in the above table.
- A variance of more than 5% in any of the SDLC phase indicates the scope for improvement in the estimation. The variance is acceptable only for the coding and testing phases.
- The variance can be negative also. A negative variance is an indication of an over estimate.
- The variance is acceptable only for the coding and testing phases.

2. Schedule Variance (Planned vs Actual)

Schedule variance is calculated at the end of every milestone to find out how well the project is doing with respect to the schedule.



To get a real picture on schedule in the middle of project execution, it is important to calculate “remaining days yet to be spent” on the project and plot it along with the “actual

schedule spent" as in the above chart. "Remaining days yet to be spent" can be calculated by adding up all remaining activities. If the remaining days yet to be spent on project is not calculated and plotted, it does not give any value to the chart in the middle of the project, because the deviation cannot be inferred visually from the chare. The remaining days in the schedule becomes zero when the release is met.

Effort and schedule variance have to be analyzed in totality, not in isolation. This is because while effort is a major driver of the cost, schedule determines how best a product can exploit market opportunities, variance can be classified into negative variance, zero variance, acceptable variance, and unacceptable variance.

Interpretation of range of effort and schedule variation

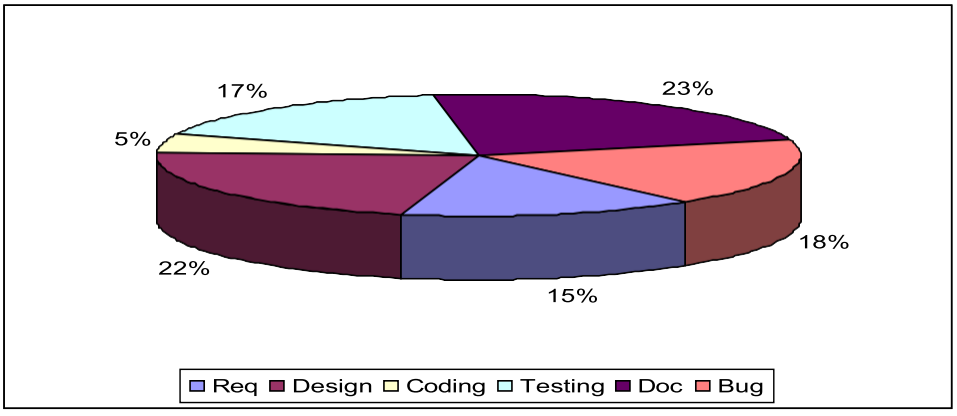
Effort Variance	Schedule Variance	Probable Causes /Result
Zero (or) Acceptable variance (0-5)	Zero variance	A Well executed Project
	Acceptable variance	Need slight improvement
Unacceptable variance (>5)	Zero (or) Acceptable variance	Under estimation , need further analysis
	Unacceptable variance	Under estimation of both effort and schedule
Negative variance	Zero (or) Acceptable variance	Over estimation, need improvement
	Negative variance	Over estimation, need improvement

3. Effort Distribution Across Phases

Adequate and appropriate effort needs to be spent in each of the SDLC phase for a quality product release.

The distribution percentage across the different phases can be estimated at the time of planning and these can be compared with the actual at the time of release for getting a comfort feeling on the release and estimation methods. A sample distribution of effort across phases is given in figure.

Actual Effort Distribution



Effort distribution :

Req > Testing > design > bug fixing > coding > doc

Mature organizations spend at least 10-15 % of the total effort in requirements and approximately the same effort in the design phase. The effort percentage for testing depends on the type of release and amount of change to the existing code base and functionality. Typically, organizations spend about 20 -50 % of their total effort in testing.

II) PROGRESS METRICS

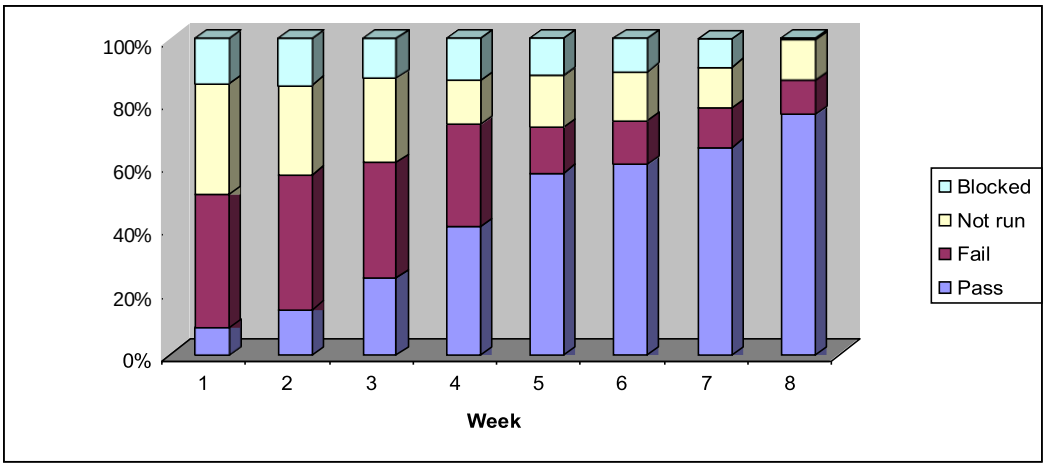
One of the main objectives of testing is to find as many defects as possible before any customer finds them. The number of defects that are found in the product is one of the main indicators of quality.

Defects get detected by the testing team and get fixed by the development team. Defect metrics are further classified in to

1. test defect metrics
2. development defect metrics

The progress chart gives

- pass rate
- fail rate of executed test cases
- pending test cases
- test cases that are waiting for defects to be fixed.



A scenario represented by such a progress chart shows that not only is testing progressing well, but also that the product quality is improving. The chart had shown a trend that as the weeks progress, the “not run” cases are not reducing in number, or “blocked” cases are increasing in number, or “pass” cases are not increasing, then it would clearly point to quality problems in the product that prevent the product from being ready for release.

1. TEST DEFECT METRICS

The next set of metrics helps us understand how the defects that are found can be used to improve testing and product quality.

Some organizations classify effects by assigning a defect priority (for example P1, P2, P3, and so on)Some organizations use defect severity levels (for example, S1, S2, S3, and so

on).The priority of a defect can change dynamically once assigned. Severity is absolute and does not change often as they reflect the state and quality of the product.

Table -Defect priority and defect severity – sample interpretation.

Defect priority is based on defect fixing and defect severity is based on functionality level.

Priority	**What it means**
1	Fix the defect on highest priority; fix it before the next build
2.	Fix the defect on high priority before next test cycle
3	Fix the defect on moderate priority when time permits, before the release
4	Postpone this defect for next release or live with this defect
Severity	**What it means**
1	The basic product functionality failing or product crashes
2	Unexpected error condition or a functionality not working
3	A minor functionality is failing or behaves differently than expected
4	Cosmetic issue and no impact on the users

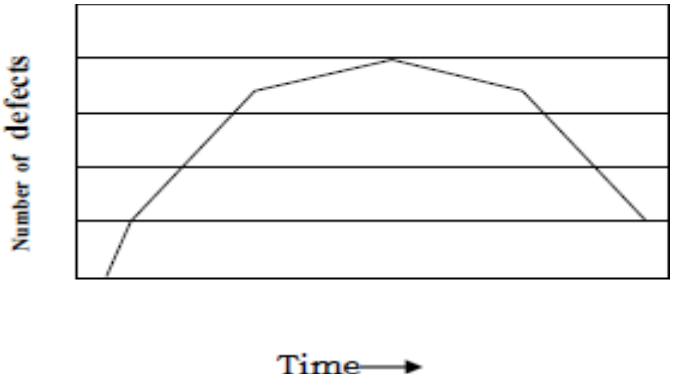
This defect classification is based on priority and severity.

Defect Classification	**What it Means**
Extreme	• Product crashes or unusable • Need to be fixed immediately
Critical	• Basic functionality of the product not working • Needs to be fixed before next test cycle starts
Important	• Extended functionality of the product not working • Does not affect the progress of testing • Fix it before the release
Minor	• Product Behaves differently • No impact on test team or customer • Fix it when time permits
Cosmetic	• Minor Irritant • Need not be fixed for this release

a) Defect Find Rate

The purpose of testing is to find defects early in the test cycle. The idea of testing is to find as many defects as possible early in the cycle. However, this may not be possible for two reasons. First, not all features of a product may become available early; because of scheduling of resources, the features of a product arrive in a particular sequence. Some of the test cases may be blocked because of some show stopper defects.

Once a majority of the modules become available and the defects that are blocking the tests are fixed, the defect arrival rate increases. After a certain period of defect fixing and testing, the arrival of defects tends to slow down and a continuation of that enables product release. This results in a “bell curve” as shown in figure.



b) Defect fix rate

The purpose of development is to fix defects as soon as they arrive. If the goal of testing is to find defects as early as possible, it is natural to expect that the goal of development should be to fix defects as soon as they arrive. There is a reason why defect fixing rate should be same as defect arrival rate. If more defects are fixed later in the cycle, they may not get tested properly for all possible side-effects.

c) Outstanding defects rate

In a well executed project, the number of outstanding defects is very close to zero all the time during the test cycle. The number of defects outstanding in the product is calculated by subtracting the total defects fixed from the total defects found in the product.

The number of defects outstanding = total defects found in the product - total defects fixed

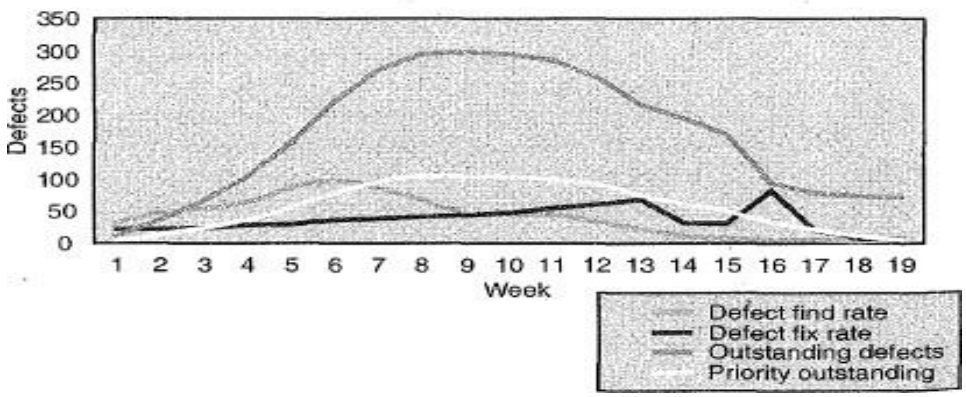
d) Priority outstanding rate

The modification to the outstanding defects rate curve by plotting only the high- priority defects and filtering out the low- priority defects is called priority outstanding defects. This is an important method because closer to product release, the product team would not want to fix the low – priority defects.

Normally only high-priority defects are tracked during the period closer to release. Some high-priority defects may require a change in design or architecture & fixed immediately

e) Defect trend

The effectiveness analysis increases when several perspectives of find rate, fix rate, outstanding, and priority outstanding defects are combined.



Defect trend

The following observations can be made.

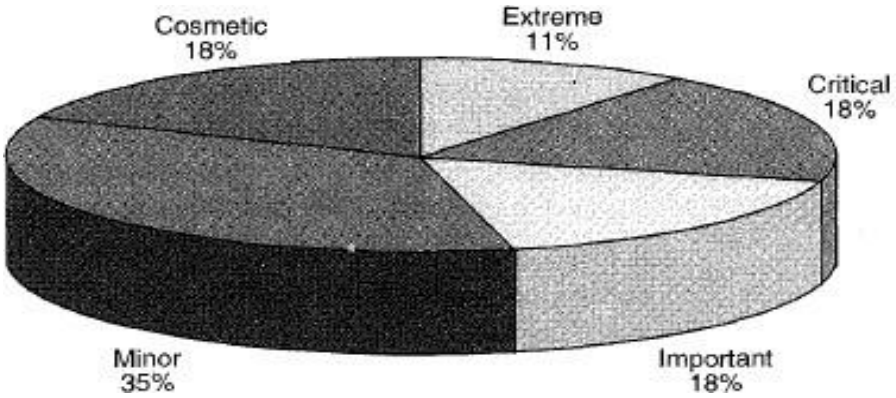
1. The find rate, fix rate, outstanding defects, and priority outstanding follow a bell curve pattern, indicating readiness for release at the end of the 19th week.
2. a sudden downward movement as well as upward spike in defect fixes rate needs analysis (13th to 17th week in the chart above)
3. By looking at the priority outstanding which shows close to zero defects in the 19th week, it can be concluded that all outstanding defects belong to low priority.
4. A smooth priority outstanding rate suggests that priority defects were closely tracked and fixed.

f) Defect Classification trend

Providing the perspective of defect classification in the chart helps in finding out release readiness of the product. When talking about the total number of outstanding defects, some of the questions that can be asked are

- How many of them are extreme defects?
- How many are critical?
- How many are important?

These questions require the charts to be plotted separately based on defect classification. The sum of extreme, critical, important, minor, and cosmetic defects is equal to the total number of defects.



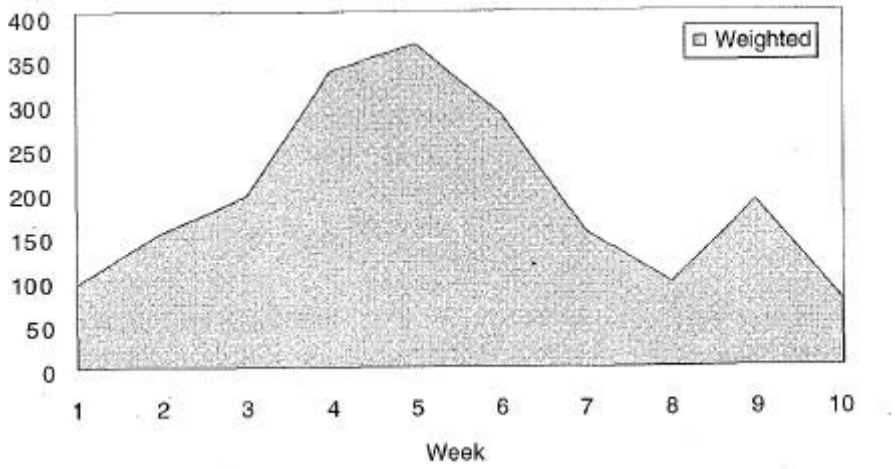
Pie chart of defect distribution

g) Weighted defects trend

Weighted defect helps in quick analysis of defect, instead of worrying about the classification of defects.

Weighted defects= (Extreme * 5 + Critical * 4 + important * 3 + Minor * 2 + Cosmetic)

Both “large defects” and “large number of small defects” affect product release.



Weighted defects trend.

From Figure it can be noted that

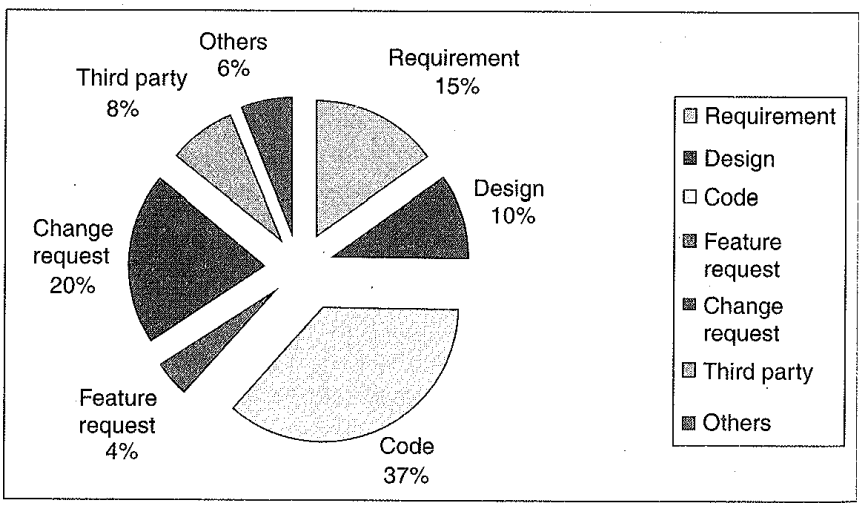
1. The ninth week has more weighted defects, which means existence of "large number of small defects" or "significant numbers of large defects" or a combination of the two. This is consistent with our interpretation of the same data using the stacked area chart.
2. The tenth week has a significant (more than 50) number of weighted defects indicating the product is not ready for release.

h) Defect cause distribution

Logical questions that would arise are:

1. Why are those defects occurring and what are the root causes?
2. What areas must be focused for getting more defects out of testing?

Finding the root causes of the defects help in identifying more defects and sometimes help in even preventing the defects.



Defect cause distribution chart

2. Development Defect Metrics

To map the defects to different components of the product , the parameter is LOC. It has

a) Component wise Defect Distribution
b) Defect Density & defect removal rate
c) Age Analysis of outstanding defect
d) Introduced and reopened defects trend

a) Component wise Defect Distribution

When module wise defect distribution is done , modules like install ,reports , client and database has > 20 defects indicating that more focus and resources are needed for these components.
So knowing the components producing more defects helps in defect fix plan and in deciding what to release.

b) Defect Density & defect removal rate

Defect density maps the defects in the product with the volume of code that is produced for the product.

Defects per KLOC = Total defects found in the product / total Executable line of code in KLOC

Variants to this metrics is to calculate AMD (add , modify , delete code) to find how a release affects product quality .

Defects per KLOC = Total defects found in the product / total Executable AMD line of code in KLOC

Defect removal rate = (defect found by verification activities + defects found in Unit test) / defect found by testing team) * 100

c) Age Analysis of outstanding defect

The time needed to fix a defect may be proportional to its age. It helps in finding out whether the defects are fixed as soon as they arrive and to ensure that long pending defects are given adequate priority.

d) Introduced and reopened defects trend

Introduced defect(ID): when adding new code or modifying the code to provide a defect fix , something that was working earlier may stop working , this is called ID.

reopened defects : fix that is provided in the code may not have fixed the problem completely or some other modification may have reproduced a defect that was fixed earlier. This is called as reopened defects.

Testing is not meant to find the same defects again ; release readiness should consider the quality of defect fixes.

III) PRODUCTIVITY METRICS

Productivity metrics combine several measurements and parameters with effort spend on the product. They help in finding out the capability of the team as well as for other purpose, such as

1. Estimating for the new release.
2. Finding out how well the team is progressing, understanding the reasons for (both positive and negative) variations in results.
3. Estimating the number of defects that can be found
4. Estimating release data and quality
5. Estimating the cost involved in the release.

a) Defects per 100 Hours of Testing

Defects per 100 hours of testing= (Total defects found in the product for a period / Total hours spent to get those defects) * 100

Test Cases Executed per 100 Hours of Testing

Test cases executed per 100 hours of testing = (Total test cases executed for a period / Total hours spent in test execution) * 100

b) Test cases Developed per 100 Hours of Testing

Test cases developed per 100 hours of testing= (Total test cases developed for a period / Total hours spent in test case development) * 100

c) Defects per 100 Test Cases

Defects per 100 test cases = (Total defects found for a period / Total test cases executed for the same period) * 100

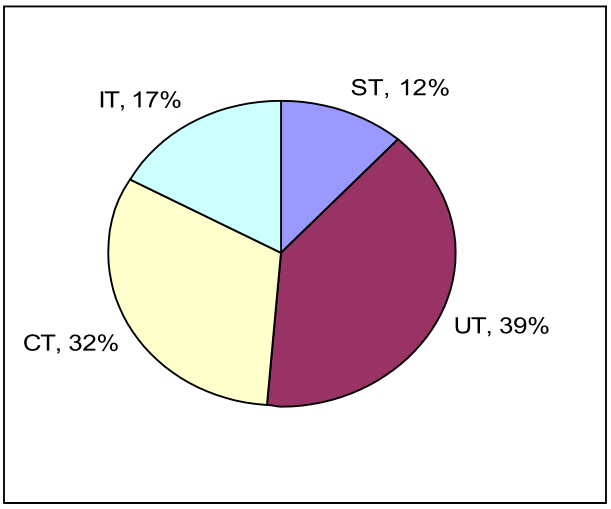
d) Defects per 100 Failed Test Cases

Defects per 100 failed test cases = (Total defects found for a period/Total test cases failed due to those defects) * 100

e) Test Phase Effectiveness

The following few observations can be made

1. A good proportion of defects were found in the early phases of testing (UT and CT).
2. Product quality improved from phase to phase (shown by less percent of defects found in the later test phases – IT and ST)



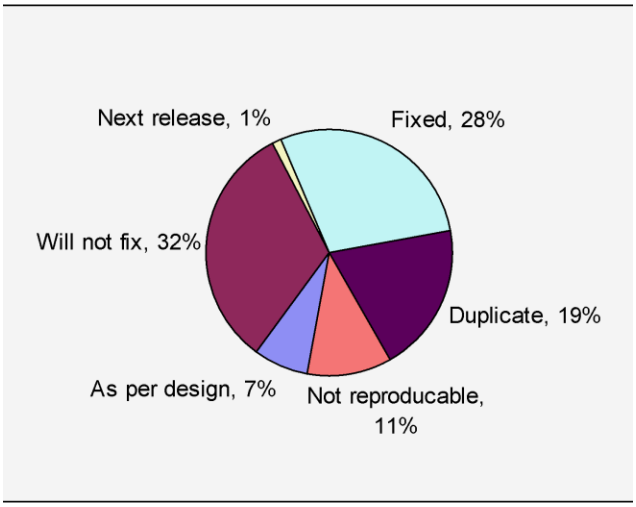
Test phase effectiveness

f) Closed Defect Distribution

The closed defect distribution helps in this analysis as shown in the figure below. From the chart, the following observations can be made.

1. Only 28% of the defects found by test team were fixed in the product. This suggests that product quality needs improvement before release.
2. Of the defects filled 19% were duplicates. It suggests that the test team needs to update itself on existing defects before new defects are filed.
3. Non-reproducible defects amounted to 11%. This means that the product has some random defects or the defects are not provided with reproducible test cases. This area needs further analysis.

4. Close to 40% of defects were not fixed for reasons “as per design,” “will not fix,” and “next release.” These defects may impact the customers. They need further discussion and defect fixes need to be provided to improve release quality.



Closed defect distribution