Module 3: Managing the Test Process

-by
Asst Prof. Rohini M Sawant

Managing the Test Process

- Test management is concerned with both test resource and test environment management.
- It is the role of test management to ensure that new or modified service products meet the business requirements for which they have been developed or enhanced.
- Since testing is viewed as a process, it must have an organization such that a testing group works for better testing and high quality software.
- To manage a testing process, thus, the first step is to get a test organization in place. Next, we must have a master test plan which guides us when and how to perform various testing activities.

TEST ORGANIZATION

The testing group is responsible for the following activities:

- Maintenance and application of test policies
- Development and application of testing standards
- Participation in requirement, design, and code reviews
- Test planning & execution
- Test measurement
- Test monitoring
- Defect tracking
- Acquisition of testing tools
- Test reporting

STRUCTURE OF TESTING GROUP

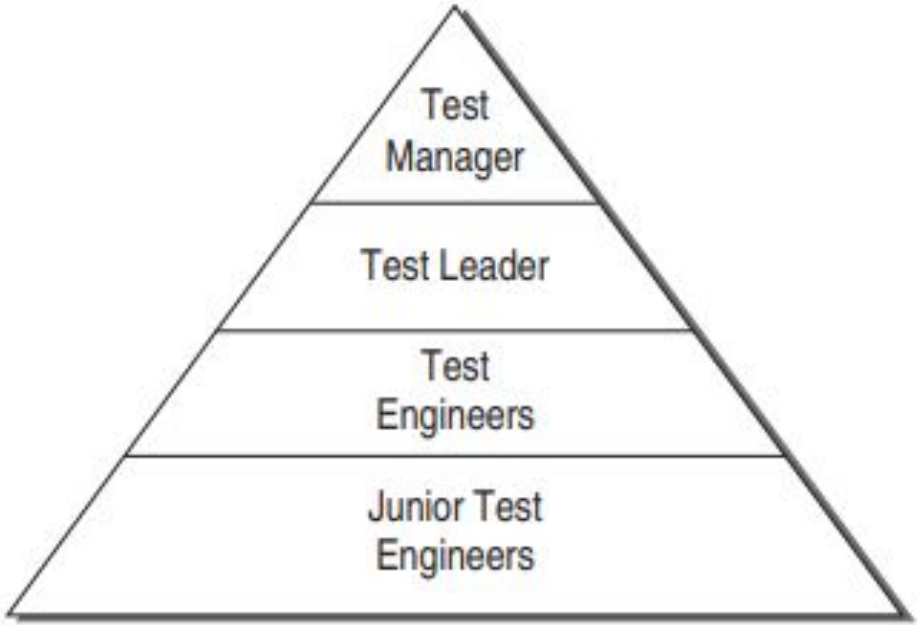


Figure 9.1 Testing Group Hierarchy

STRUCTURE OF TESTING GROUP

Test Manager: A test manager occupies the top level in the hierarchy. He has the following responsibilities:

(i) He is the key person in the testing group who will interact with project management, quality assurance, and marketing staff.

(ii) Takes responsibility for making test strategies with detailed master planning and schedule.

(iii) Interacts with customers regarding quality issues.

(iv) Acquires all the testing resources including tools.

(v) Monitors the progress of testing and controls the events. (vi) Participates in all static verification meetings.

(vii) Hires, fires, and evaluates the test team members.

STRUCTURE OF TESTING GROUP

Test Leader: The next tester in the hierarchy is the test leader who assists the test manager in meeting testing and quality goals. The prime responsibility of a test leader is to lead a team of test engineers who are working at the leaf-level of the hierarchy. The following are his responsibilities:

(i) Planning the testing tasks given by the test manager.

(ii) Assigning testing tasks to test engineers who are working under him.

(iii) Supervising test engineers.

(iv) Helping the test engineers in test case design, execution, and reporting.

(v) Providing tool training, if required.

(vi) Interacting with customers.

STRUCTURE OF TESTING GROUP

Test Engineers: Test engineers are highly experienced testers. They work under the leadership of the test leader. They are responsible for the following tasks:

(i) Designing test cases.

(ii) Developing test harness.

(iii) Set-up test laboratories and environment.

(iv) Maintain the test and defect repositories.

Junior Test Engineers: Junior test engineers are newly hired testers. They usually are trained about the test strategy, test process, and testing tools. They participate in test design and execution with experienced test engineers.

TEST PLANNING

- According to the test process as discussed in STLC, testing also needs planning as is needed in SDLC. Since software projects become uncontrolled if not planned properly, the testing process is also not effective if not planned earlier.
- Moreover, if testing is not effective in a software project, it also affects the final software product. Therefore, for a quality software, testing activities must be planned as soon as the project planning starts.
- **A test plan** is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities. Test plan is driven with the business goals of the product. In order to meet a set of goals, the test plan identifies the following:

  „ Test items

  „ Features to be tested

  „ Testing tasks

  „ Tools selection

  „ Time and effort estimate

  „ Who will do each task

  „ Milestones

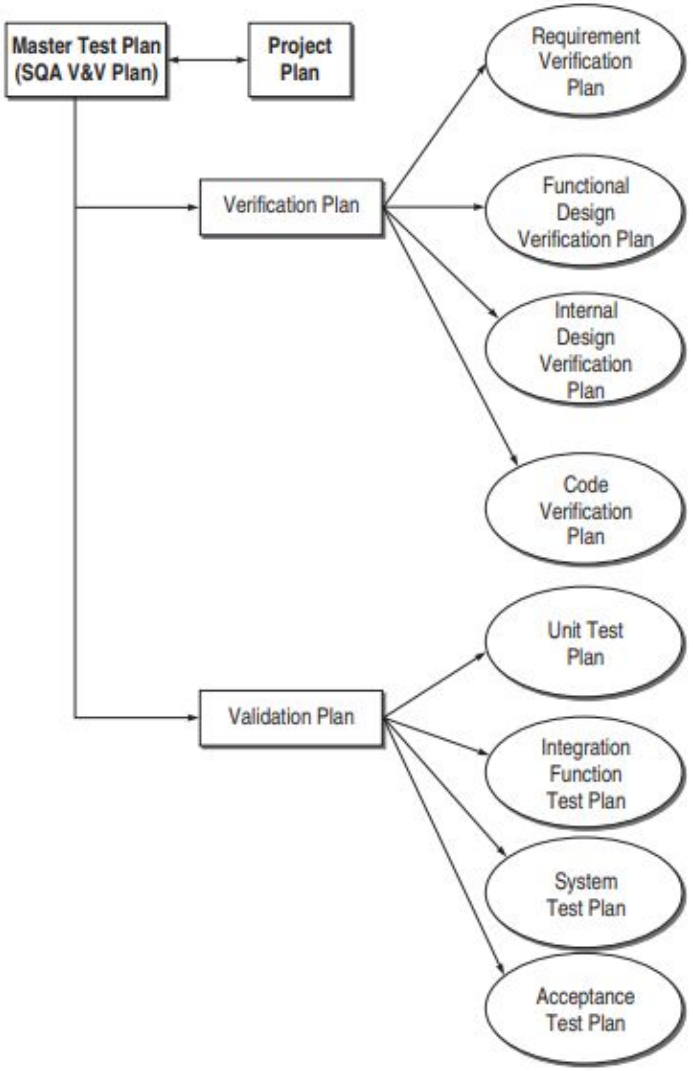
  „ Any risks

TEST PLAN COMPONENTS

Figure 9.2 Test plan components

TEST PLAN HIERARCHY

Test plans can be organized in several ways depending on the organizational policy. There is often a hierarchy of plans that includes several levels of quality assurance and test plans. At the top of the plan hierarchy is a master plan which gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, and configuration control. Below the master test plan, there is individual planning for each activity in the figure.



TEST PLANNING

VERIFICATION TEST PLAN:Verification test planning includes the following tasks:

- The item on which verification is to be performed.
- The method to be used for verification: review, inspection, walkthrough.
- The specific areas of the work product that will be verified.
- The specific areas of the work product that will not be verified.
- Risks associated.
- Prioritizing the areas of work product to be verified.
- Resources, schedule, facilities, tools, and responsibilities.

VALIDATION TEST PLAN: Validation test planning includes the following tasks:

- Testing techniques
- Testing tools
- Support software and documents
- Configuration management
- Risks associated, such as budget, resources, schedule, and training

TEST PLANNING

Unit Test Plan: It generally includes:

- Module overview
- Inputs & Outputs to the module
- Logic flow diagram
- Test approach
- Features to be tested & Features not to be tested
- Test constraints
- Test tools
- Updates
- Milestones
- Budget

TEST PLANNING

Integration Test Plan: It generally includes:

- Documents to be used
- Modules for integration
- Availability
- Subsystems
- Strategy for integration
- List of Tasks to be tested & List of Tasks not to be tested

Function Test Plan: It generally includes:

- List the objectives of function testing
- Partitioning/Functional decomposition
- Traceability matrix formation
- List the functions to be tested

TEST PLANNING

Integration Test Plan: It generally includes:

- Partition the requirements
- System description
- Features to be tested
- Strategy and reporting format
- Develop a requirement coverage matrix
- Smoke test set
- Entry/Exit criteria
- Suspension criteria
- Resource requirements
- Participating team & organizations
- Schedule estimation

DETAILED TEST DESIGN AND TEST SPECIFICATIONS

TEST DESIGN SPECIFICATION:

A test design specification should have the following components according to IEEE recommendation:

- Identifier A unique identifier is assigned to each test design specification with a reference to its associated test plan.
- Features to be tested: The features or requirements to be tested are listed with reference to the items mentioned in SRS/SDD.
- Approach refinements: In the test plan, an approach to overall testing was discussed. Here, further details are added to the approach. For example, special testing techniques to be used for generating test cases are explained.
- Test case identification The test cases to be executed for a particular feature/ function are identified here
- Feature pass/fail criteria The criteria for determining whether the test for a feature has passed or failed, is described.

TEST CASE SPECIFICATIONS

- Purpose: The purpose of designing and executing the test case should be mentioned here. It refers to the functionality you want to check with this test case.
- Test items needed: List the references to related documents that describe the items and features, e.g. SRS, SDD, and user manual.
- Special environmental needs: In this component, any special requirement in the form of hardware or software is recognized. Any requirement of tool may also be specified.
- Special procedural requirements: Describe any special condition or constraint to run the test case, if any.
- Inter-case dependencies There may be a situation that some test cases are dependent on each other. Therefore, previous test cases which should be run prior to the current test case must be specified.
- Input specifications This component specifies the actual inputs to be given while executing a test case.
- Test procedure: The step-wise procedure for executing the test case is described here.
- Output specifications: Whether a test case is successful or not is decided after comparing the output specifications with the actual outputs achieved.

There is a system for railway reservation system. There are many functionalities in the system, as given below:

S. No.	Functionality	Function ID in SRS	Test cases
1	Login the system	F3.4	T1
2	View reservation status	F3.5	T2
3	View train schedule	F3.6	T3
4	Reserve seat	F3.7	T4
5	Cancel seat	F3.8	T5
6	Exit the system	F3.9	T6

Suppose we want to check the functionality corresponding to 'view reservation status'. Its test specification is given in Fig. 9.4.

- **Test case Specification Identifier**
  T2
- **Purpose**
  To check the functionality of 'View Reservation Status'
- **Test Items Needed**
  Refer function F3.5 in SRS of the system.
- **Special Environmental Requirements**
  Internet should be in working condition. Database software through which the data will be retrieved should be in running condition.
- **Special Procedural Requirements**
  The function 'Login' must be executed prior to the current test case.
- **Inter-case Dependencies**
  T1 test case must be executed prior to the current test case execution.
- **Input Specifications**
  Enter PNR number in 10 digits between 0–9 as given below:
  4102645876
  21A2345672
  234
  asdgggggggg
- **Test Procedure**
  Press 'View Reservation status' button.
  Enter PNR number and press ENTER.
- **Output Specifications**
  The reservation status against the entered PNR number is displayed as S12 or RAC13 or WL12 as applicable.

TEST PROCEDURE SPECIFICATIONS/TEST RESULT SPECIFICATIONS

- A test procedure is a sequence of steps required to carry out a test case or a specific task. This can be a separate document or merged with a test case specification.

Test Log

Test log is a record of the testing events that take place during the test. Test log is helpful for bug repair or regression testing. The developer gets valuable clues from this test log, as it provides snapshots of failures. The format for preparing a test log according to IEEE [56] is given below:

- **Test log identifier**
- **Description** Mention the items to be tested, their version number, and the environment in which testing is being performed.
- **Activity and event entries** Mention the following:
  - (i) Date
  - (ii) Author of the test
  - (iii) Test case ID
  - (iv) Name the personnel involved in testing

- **Test Log Identifier**

  TL2
- **Description**

  Function 3.5 in SRS v 2.1. The function tested in Online environment with Internet.
- **Activity and Event Entries**

  Mention the following:

  (i) Date: 22/03/2009

  (ii) Author of test: XYZ

  (iii) Test case ID: T2

  (iv) Name of the personnel involved in testing: ABC, XYZ

  (v) For each execution, record the results and mention pass/fail status

  The function was tested with the following inputs:

Inputs	Results	Status
4102645876	S12	Pass
21A2345672	S14	Fail
234	Enter correct 10 digit PNR number	Pass
asdgggggggg	RAC12	Fail

  (vi) Report any anomalous unexpected event before or after the execution.

  Nil

Test Incident Report

This is a form of bug report. It is the report about any unexpected event during testing which needs further investigation to resolve the bug. Therefore, this report completely describes the execution of the event. It not only reports the problem that has occurred but also compares the expected output with the actual results. Listed below are the components of a test incident report [56]:

- **Test incident report identifier**
- **Summary** Identify the test items involved, test cases/procedures, and the test log associated with this test.
- **Incident description** It describes the following:
  (i) Date and time
  (ii) Testing personnel names
  (iii) Environment
  (iv) Testing inputs
  (v) Expected outputs
  (vi) Actual outputs
  (vii) Anomalies detected during the test
  (viii) Attempts to repeat the same test
- **Impact** The originator of this report will assign a severity value/rank to this incident so that the developer may know the impact of this problem and debug the critical problem first.

- **Test Incident Report Identifier**
  TI2
- **Summary**
  Function 3.5 in SRS v 2.1. Test Case T2 and Test Log TL2.
- **Incident Description**
  It describes the following:
  (i) Date and time: 23/03/2009, 2.00pm
  (ii) Testing personnel names: ABC, XYZ
  (iii) Environment: Online environment with X database
  (iv) Testing inputs
  (v) Expected outputs
  (vi) Actual outputs
  (vii) Anomalies detected during the test
  (viii) Attempts to repeat the same test

Testing Inputs	Expected outputs	Actual outputs	Anomalies detected	Attempts to repeat the same test
4102645876	S12	S12	nil	–
21A2345672	Enter correct 10 digit PNR number	S14	Alphabet is being accepted in the input.	3
234	Enter correct 10 digit PNR number	Enter correct 10 digit PNR number	nil	–
asdgggggggg	Enter correct 10 digit PNR number	RAC12	Alphabet is being accepted in the input.	5

- **Impact**
  The severity value is 1(Highest).

Test Summary Report

It is basically an evaluation report prepared when the testing is over. It is the summary of all the tests executed for a specific test design specification. It can provide the measurement of how much testing efforts have been applied for the test. It also becomes a historical database for future projects, as it provides information about the particular type of bugs observed.

- **Test Summary Report Identifier**
  TS2
- **Description**
  SRS v 2.1

S. No.	Functionality	Function ID in SRS	Test cases
1	Login the system	F3.4	T1
2	View reservation status	F3.5	T2
3	View train schedule	F3.6	T3
4	Reserve seat	F3.7	T4
5	Cancel seat	F3.8	T5
6	Exit the system	F3.9	T6

- **Variances**
  Nil
- **Comprehensive Statement**
  All the test cases were tested except F3.7, F3.8, F3.9 according to the test plan.
- **Summary of Results**

S. No.	Functionality	Function ID in SRS	Test cases	Status
1	Login the system	F3.4	T1	Pass
2	View reservation status	F3.5	T2	Bug Found. Could not be resolved.
3	View train schedule	F3.6	T3	Pass

- **Evaluation**
  The functions F3.4, F3.6 have been tested successfully. Function F3.5 couldn't be tested as the bug has been found. The bug is that the PNR number entry has also accepted alphabetical entries as wrong

Software Metrics

- Today, every technical process demands measurement. We cannot describe a product by simply saying that it should be of high quality or that it should be reliable.
- Software metrics are used by the software industry to quantify the development, operation, and maintenance of software.
- Metrics give us information regarding the status of an attribute of the software and help us to evaluate it in an objective way.
- Measurement of these attributes helps to make the characteristics and relationships between the attributes clearer. This in turn supports informed decision-making.
- Software metrics can be defined as 'the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.'

Need for Software Metrics

- Measurements are a key element for controlling software engineering processes.
- By controlling, it is meant that one can assess the status of the process, observe the trends to predict what is likely to happen, and take corrective action for modifying our practices.
- Measurements also play their part in increasing our understanding of the process by making visible relationships among process activities and entities involved.

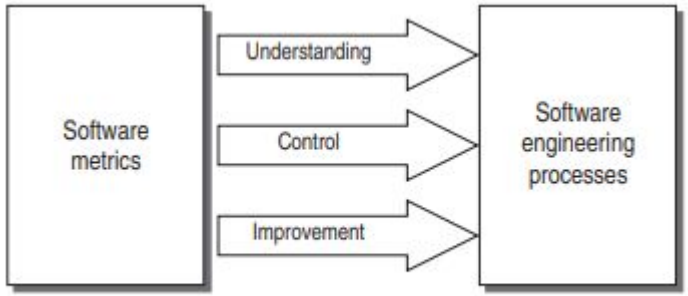


Figure 10.1 Need for software metrics

Need for Software Metrics

On the basis of this discussion, software measurement is needed for the following activities:

- **Understanding:** Metrics help in making the aspects of a process more visible, thereby giving a better understanding of the relationships among the activities and entities they affect.
- **Control:** Using baselines, goals, and an understanding of the relationships, we can predict what is likely to happen and correspondingly, make appropriate changes in the process to help meet the goals.
- **Improvement:** By taking corrective actions and making appropriate changes, we can improve a product. Similarly, based on the analysis of a project, a process can also be improved.

Entities to be Measured

The entities considered in software measurement are:

- ***Processes*** Any activity related to software development.
- ***Product*** Any artifact produced during software development.
- ***Resource*** People, hardware, or software needed for a process.

The attributes of an entity can be internal or external.

- ***Internal attributes*** of any entity can be measured only based on the entity and therefore, measured directly. For example, size is an internal attribute of any software measurement.
- ***External attributes*** of any entity can be measured only with respect to how the entity is related with the environment and therefore, can only be measured indirectly. For example, reliability, an external attribute of a program, does not depend only on the program itself but also on the compiler, machine, and user. Productivity, an external attribute of a person, clearly depends on many factors such as the kind of process and the quality of the software delivered.

CLASSIFICATION OF SOFTWARE METRICS

PRODUCT VS. PROCESS METRICS

- Software metrics may be broadly classified as either product metrics or process metrics.
- Product metrics are measures of the software product at any stage of its development, from requirements to installed system.
- Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation produced.
- Example: LOC, Error Discovery Rate, Defect Removal Efficiency.
- Process metrics, on the other hand, are measures of the software development process, such as the overall development time, type of methodology used, or the average level of experience of the programming staff.
- Example: Test Cases Coverage, Test Cases Passed, Test Cases Blocked etc.

CLASSIFICATION OF SOFTWARE METRICS

OBJECTIVE VS. SUBJECTIVE METRICS

- Objective measures should always result in identical values for a given metric, as measured by two or more qualified observers.
- For subjective measures, even qualified observers may measure different values for a given metric.
- For example, for product metrics, the size of product measured in line of code (LOC) is an objective measure.
- In process metrics, the development time is an example of objective measure, while the level of a programmer's experience is likely to be a subjective measure

CLASSIFICATION OF SOFTWARE METRICS

PRIMITIVE VS. COMPUTED METRICS

- Primitive metrics are those metrics that can be directly observed, such as the program size in LOC, the number of defects observed in unit testing, or the total development time for the project.
- Computed metrics are those that cannot be directly observed but are computed in some way from other metrics.
- For example, productivity metrics like LOC produced per person-month or product quality like defects per thousand lines of code.

CLASSIFICATION OF SOFTWARE METRICS

PRIVATE VS. PUBLIC METRICS

- This classification is based on the use of different types to process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to individuals and serve as an indicator for individuals only.
- Examples of private metrics include defect rates (by individual and by module) and errors found during development.
- Public metrics assimilate information that originally was private to individuals and teams.
- Project-level defect rates, effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

SIZE METRICS

The software size is an important metric to be used for various purposes. At the same time, it is difficult to measure because, unlike other physical products, a software cannot be measured directly with conventional units.

Various approaches used for its measurement are given below.

- LINE OF CODE (LOC): This metric is based on the number of lines of code present in the program. The lines of code are counted to measure the size of a program.
- The comments and blank lines are ignored during this measurement.
- The LOC metric is often presented on thousands of lines of code (KLOC).
- It is often used during the testing and maintenance phases, not only to specify the size of the software product, but also it is used in conjunction with other metrics to analyse other aspects of its quality and cost.

SIZE METRICS

TOKEN COUNT (HALSTEAD PRODUCT METRICS) :The problem with LOC is that it is not consistent, because all lines of code are not at the same level. Some lines are more difficult to code than others. Another metric set has been given by Halstead.

He stated that any software program could be measured by counting the number of operators and operands. Some Halstead metrics are given below:

Program Vocabulary

It is the number of unique operators plus the number of unique operands as given below:

$$n = n1 + n2$$

where n = program vocabulary
$n1$ = number of unique operators
$n2$ = number of unique operands

Program Length

It is the total usage of all the operators and operands appearing in the implementation. It is given as,

$$N = N1 + N2$$

where N = program length

$N1$ = all operators appearing in the implementation
$N2$ = all operands appearing in the implementation

SIZE METRICS

Program Volume

The volume refers to the size of the program and it is defined as the program length times the logarithmic base 2 of the program vocabulary. It is given as,

$$V = N \log_2 n$$

where V = program volume

N = program length

n = program vocabulary

SIZE METRICS

FUNCTION POINT ANALYSIS (FPA)

- It is based on the idea that the software size should be measured according to the functionalities specified by the user.
- Therefore, FPA is a standardized methodology for measuring various functions of a software from the user's point of view.
- The size of an application is measured in function points.

Process to Calculate Function Points

The process used to calculate the function points is given below [122]:

1. Determine the type of project for which the function point count is to be calculated. For example, development project (a new project) or enhancement project.
2. Identify the counting scope and the application boundary.
3. Identify data functions (internal logical functions and external interface files) and their complexity.
4. Identify transactional functions (external inputs, external outputs, and external queries) and their complexity.
5. Determine the unadjusted function point count (UFP).
6. Determine the value adjustment factor, which is based on 14 general system characteristics (GSCs).
7. Calculate the adjusted function point count (AFP).

ATTRIBUTES AND CORRESPONDING METRICS IN SOFTWARE TESTING

An organization needs to have a reference set of measurable attributes and corresponding metrics that can be applied at various stages during the course of execution of an organization-wide testing strategy.

Therefore, we discuss measurable attributes for software testing and the corresponding metrics based on the attribute categorization done by Wasif Afzal and Richard Torkar, as shown in Table

Category	Attributes to be Measured
Progress	• Scope of testing • Test progress • Defect backlog • Staff productivity • Suspension criteria • Exit criteria
Cost	• Testing cost estimation • Duration of testing • Resource requirements • Training needs of testing group and tool requirement • Cost-effectiveness of automated tool
Quality	• Effectiveness of test cases • Effectiveness of smoke tests • Quality of test plan • Test completeness
Size	• Estimation of test cases • Number of regression tests • Tests to automate

PROGRESS

Scope of testing: It helps in estimating the overall amount of work involved, by documenting which parts of the software are to be tested, thereby estimating the overall testing effort required. It also helps in identifying the testing types that are to be used for covering the features under test.

Tracking test progress: The progress of testing should also be measured to keep in line with the schedule, budget, and resources. If we monitor the progress with these metrics by comparing the actual work done with the planning, then the corrective measures can be taken in advance, thereby controlling the project. The test progress S curve shows the following set of information on one graph:

„ Planned number of test cases to be completed successfully by a week.

„ Number of test cases attempted in a week.

„ Number of test cases completed successfully by a week.

Defect backlog: It is the number of defects that are outstanding and unresolved at one point of time with respect to the number of defects occurring. If the backlog increases, the bugs should be prioritized according to their criticality

PROGRESS

Staff productivity: Measurement of testing staff productivity helps to improve the contributions made by the testing professionals towards a quality testing process. The basic and useful measures of a tester's productivity are:

„ Time spent in test planning.

„ Time spent in test case design.

„ Number of test cases developed.

„ Number of test cases developed per unit time.

Suspension criteria: These are the metrics that establish conditions to suspend testing. It describes the circumstances under which testing would stop temporarily. The suspension criteria of testing describes in advance that the occurrence of certain events/conditions will cause the testing to stop temporarily.

Exit criteria: The exit criteria are established for all the levels of a test plan. It indicates the conditions that move the testing activities forward from one level to the next. If we are clear when to exit from one level, the next level can start.

COST

Testing cost estimation: The metrics supporting the budget estimation of testing need to be established early. It also includes the cost of test planning itself.

Duration of testing: There is also a need to estimate the testing schedule during test planning. As a part of the testing schedule, the time required to develop a test plan is also estimated. A testing schedule contains the timelines of all testing milestones.

Resource requirements

Training needs of testing groups and tool requirements

Cost-effectiveness of automated tools: When a tool is selected to be used for testing, it is beneficial to evaluate its cost-effectiveness. The cost-effectiveness is measured by taking into account the cost of tool evaluation, tool training, tool acquisition, and tool update and maintenance.

QUALITY

Effectiveness of test cases: The test cases produced should be effective so that they uncover more and more bugs. The measurement of their effectiveness starts from the test case specifications. There are several measures based on faults to check the effectiveness of test cases. Some of them are discussed here:

1. Number of faults

2. Number of failures

3. Defect-removal efficiency is another powerful metric for test-effectiveness, which is defined as the ratio of the number of faults actually found in testing and the number of faults that could have been found in testing.

4. Defect age is another metric that can be used to measure the test effectiveness, which assigns a numerical value to the fault, depending on the phase in which it is discovered. Defect age is used in another metric called defect spoilage to measure the effectiveness of defect-removal activities.

$$\text{Spoilage} = \frac{\text{Sum of (Number of defects} \times \text{Defect age)}}{\text{Total number of defects}}$$

Consider a project with the following distribution of data and calculate its defect spoilage.

SDLC phase	No. of defects	Defect age
Requirement Specs.	34	2
HLD	25	4
LLD	17	5
Coding	10	6

Solution

$$\text{Spoilage} = (34 \times 2 + 25 \times 4 + 17 \times 5 + 10 \times 6)/86 = 3.64$$

QUALITY

Effectiveness of smoke tests: Smoke tests are required to ensure that the application is stable enough for testing, thereby assuring that the system has all the functionality and is working under normal conditions. This testing does not identify faults, but establishes confidence over the stability of a system. The tests that are included in smoke testing cover the basic operations that are most frequently used, e.g. logging in, addition, and deletion of records.

Quality of test plan: The quality of the test plan produced is also a candidate attribute to be measured, as it may be useful in comparing different plans for different products, noting down the changes observed, and improving the test plans for the future. There are ten dimensions that contribute to the philosophy of test planning. These ten dimensions are:

1. Theory of objective
2. Theory of scope
3. Theory of coverage
4. Theory of risk
5. Theory of data
6. Theory of originality
7. Theory of communicatio
8. Theory of usefulness
9. Theory of completeness
10. Theory of insightfulness

QUALITY

Measuring test completeness: This attribute refers to how much of code and requirements are covered by the test set. The advantages of measuring test coverage are that it provides the ability to design new test cases and improve existing ones. There are two issues: (i) whether the test cases cover all possible output states and (ii) the adequate number of test cases to achieve test coverage. The relationship between code coverage and the number of test cases is described by the following expression

$$C(x) = 1 - e^{-(p/N) * x}$$

SIZE

Estimation of test cases To fully exercise a system and to estimate its resources, an initial estimate of the number of test cases is required. Therefore, metrics estimating the required number of test cases should be developed.

Number of regression tests Regression testing is performed on a modified program that establishes confidence that the changes and fixes against reported faults are correct and have not affected the unchanged portions of the program [51]. However, the number of test cases in regression testing becomes too large to test. Therefore, careful measures are required to select the test cases effectively.

Some of the measurements to monitor regression testing are [60]:

- Number of test cases re-used
- Number of test cases added to the tool repository or test database
- Number of test cases rerun when changes are made to the software
- Number of planned regression tests executed
- Number of planned regression tests executed and passed

Tests to automate Tasks that are repetitive in nature and tedious to perform manually are prime candidates for an automated tool. The categories of tests that come under repetitive tasks are:

- Regression tests
- Smoke tests
- Load tests
- Performance tests

If automation tools have a direct impact on the project, it must be measured. There must be benefits attached with automation including speed, efficiency, accuracy and precision, resource reduction, and repetitiveness. All these factors should be measured.

ESTIMATION MODELS FOR ESTIMATING TESTING EFFORTS

11.4.1 Halstead Metrics

The metrics derived from Halstead measures described in Chapter 10 can be used to estimate testing efforts, as given by Pressman [116]. Halstead developed expressions for program volume V and program level PL which can be used to estimate testing efforts. The program volume describes the number of volume of information in bits required to specify a program. The program level is a measure of software complexity. Using these definitions, Halstead effort e can be computed as [124]:

$$PL = 1/[(n1/2) \times (N2/n2)]$$

$$e = V/PL$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\textbf{Percentage of testing effort } (k) = e(k)/\sum e(i)$$

where $e(k)$ is the effort required for module k and $\sum e(i)$ is the sum of Halstead effort across all modules of the system.

In a module implementation of a project, there are 17 unique operators, 12 unique operands, 45 total operators, and 32 total operands appearing in the module implementation. Calculate its Halstead effort.

Solution

$$n = n1 + n2 = 17 + 12 = 92$$

$$N = N1 + N2 = 45 + 32 = 77$$

$$V = N \log_2 n = 77 \times \log_2 92 = 502.3175$$

$$\text{PL} = 1/[(n1/2) \times (N2/n2)] = 3/68 = 0.044$$

$$e = V/PL = 11416.30$$

ESTIMATION MODELS FOR ESTIMATING TESTING EFFORTS

DEVELOPMENT RATIO METHOD: This model is based on the estimation of development efforts. The number of testing personnel required is estimated on the basis of the developer-tester ratio.

The results of applying this method is dependent on numerous factors including the type and complexity of the software being developed, testing level, scope of testing, test-effectiveness during testing, error tolerance level for testing, and available budget.

The method of estimating tester-to-developer ratios, based on heuristics, is proposed by K. Iberle and S. Bartlett. This method selects a baseline project(s), gathers testers-to-developers ratios, and collects data on various effects like developer-efficiency at removing defects before testing, developer-efficiency at inserting defects, defects found per person, and the value of defects found. After that, an initial estimate is made to calculate the number of testers based upon the ratio of the baseline projecT

ESTIMATION MODELS FOR ESTIMATING TESTING EFFORTS

PROJECT-STAFF RATIO METHOD

Project-staff ratio method makes use of historical metrics by calculating the percentage of testing personnel from the overall allocated resources planned for the project [12]. The percentage of a test team size may differ according to the type of project. The template can be seen in Table 11.3.

Table 11.3 Project-staff ratio

Project type	Total number of project staff	Test team size %	Number of testers
Embedded system	100	23	23
Application development	100	8	8

ESTIMATION MODELS FOR ESTIMATING TESTING EFFORTS

TEST PROCEDURE METHOD: This model is based on the number of test procedures planned. The number of test procedures decides the number of testers required and the testing time . Thus, the baseline of estimation here is the quantity of test procedures.

It includes developing a historical record of projects including the data related to size (e.g. number of function points, number of test procedures used) and test effort measured in terms of personnel hours.

	Number of test procedures (NTP)	Number of person-hours consumed for testing (PH)	Number of hours per test procedure = PH/ NTP	Total period in which testing is to be done (TP)	Number of testers = PH/TP
Historical Average Record	840	6000	7.14	10 months (1600 hrs)	3.7
New Project Estimate	1000	7140	7.14	1856 hrs	3.8

ARCHITECTURAL DESIGN METRIC USED FOR TESTING

Card and Glass [57] introduced three types of software design complexity that can also be used in testing. These are discussed below.

Structural Complexity

It is defined as

$$S(m) = f_{out}^2(m)$$

where S is the structural complexity and $f_{out}(m)$ is the fan-out of module m.

This metric gives us the number of stubs required for unit testing of the module m. Thus, it can be used in unit testing.

Data Complexity

This metric measures the complexity in the internal interface for a module m and is defined as

$$D(m) = v(m) / [f_{out}(m) + 1]$$

where $v(m)$ is the number of input and output variables that are passed to and from module m.

This metric indicates the probability of errors in module m. As the data complexity increases, the probability of errors in module m also increases. For example, module X has 20 input parameters, 30 internal data items, and 20 output parameters. Similarly, module Y has 10 input parameters, 20 internal data items, and 5 output parameters. Then, the data complexity of module X is more as compared to Y, therefore X is more prone to errors. Therefore, testers should be careful while testing module X.

System Complexity

It is defined as the sum of structural and data complexity:

$$SC(m) = S(m) + D(m)$$

Since the testing effort of a module is directly proportional to its system complexity, it will be difficult to unit test a module with higher system complexity. Similarly, the overall architectural complexity of the system (which is the sum total of system complexities of all the modules) increases with the increase in each module's complexity. Consequently, the efforts required for integration testing increase with the architectural complexity of the system.

11.6 Information Flow Metrics used for Testing

Researchers have used information flow metrics between modules. For understanding the measurement, let us understand the way the data moves through a system:

- **Local direct flow** exists if
  - (i) a module invokes a second module and passes information to it.
  - (ii) the invoked module returns a result to the caller.
- **Local indirect flow** exists if the invoked module returns information that is subsequently passed to a second invoked module.
- **Global flow** exists if information flows from one module to another via a global data structure.

The two particular attributes of the information flow can be described as follows:

(i) **Fan-in** of a module m is the number of local flows that terminates at m, plus the number of data structures from which information is retrieved by m.

(ii) **Fan-out** of a module m is the number of local flows that emanate from m, plus the number of data structures that are updated by m.

11.6.1 Henry and Kafura Design Metric

Henry and Kafura's information flow metric is a well-known approach for measuring the total level of information flow between individual modules and the rest of the system. They measure the information flow complexity as

$$IFC(m) = length(m) \times ((fan\text{-}in(m) \times fan\text{-}out(m))^2$$

Higher the IF complexity of m, greater is the effort in integration and integration testing, thereby increasing the probability of errors in the module.

11.8 Function Point Metrics for Testing

The function point (FP) metric is used effectively for measuring the size of a software system. Function-based metrics can be used as a predictor for the overall testing effort. Various project-level characteristics (e.g. testing effort and time, errors uncovered, number of test cases produced) of past projects can be collected and correlated with the number of FP produced by a project team. The team can then project the expected values of these characteristics for the current project.

Listed below are a few FP measures:

1. Number of hours required for testing per FP.
2. Number of FPs tested per person-month.
3. Total cost of testing per FP.
4. Defect density measures the number of defects identified across one or more phases of the development project lifecycle and compares that value with the total size of the system. It can be used to compare the density levels across different lifecycle phases or across different development efforts. It is calculated as

   Number of defects (by phase or in total) / Total number of FPs
5. Test case coverage measures the number of test cases that are necessary to adequately support thorough testing of a development project. This measure does not indicate the effectiveness of test cases, nor does it guarantee that all conditions have been tested. However, it can be an effective comparative measure to forecast anticipated requirements for testing that may be required on a development system of a particular size. This measure is calculated as

   Number of test cases / Total number of FPs

Capers Jones [146] estimates that the number of test cases in a system can be determined by the function points estimate for the corresponding effort. The formula is

$$\textit{Number of test cases} = (\textit{function points})^{1.2}$$

Function points can also be used to measure the acceptance test cases. The formula is

$$\textit{Number of test cases} = (\textit{function points}) \times 1.2$$

The above relationships show that test cases grow at a faster rate than function points. This is intuitive because, as an application grows, the number of inter-relationships within the applications becomes more complex. For example, if a development application has 1,000 function points, there should be approximately 4,000 total test cases and 1,200 acceptance test cases.

Example 11.3

In a project, the estimated function points are 760. Calculate the total number of test cases in the system and the number of test cases in acceptance testing. Also, calculate the defect density (number of total defects is 456) and test case coverage.

Solution

Total number of test cases $= (760)^{1.2} = 2864$

Number of test cases for acceptance testing $= (760) \times 1.2 = 912$

Defect density $= 456/760 = 0.6$

Test case coverage $= 2864/760 = 3.768$

TEST POINT ANALYSIS (TPA)

- Test point analysis is a technique to measure the black-box test effort estimation, proposed by Drs Eric P W M Van Veenendaal CISA and Ton Dekkers [111].
- The estimation is for system and acceptance testing.
- The technique uses function point analysis in the estimation. TPA calculates the test effort estimation in test points for highly important functions, according to the user and also for the whole system.
- As the test points of the functions are measured, the tester can test the important functionalities first, thereby predicting the risk in testing.
- This technique also considers the quality characteristics, such as functionality, security, usability, efficiency, etc. with proper weightings.

11.9.1 Procedure for Calculating TPA

For TPA calculation, first the dynamic and static test points are calculated (see Fig. 11.1). *Dynamic test points* are the number of test points which are based on dynamic measurable quality characteristics of functions in the system. Dynamic test points are calculated for every function. To calculate a dynamic function point, we need the following:

- Function points (FPs) assigned to the function.
- Function dependent factors (FDC), such as complexity, interfacing, function-importance, etc.
- Quality characteristics (QC).

The dynamic test points for individual functions are added and the resulting number is the dynamic test point for the system.

Similarly, *static test points* are the number of test points which are based on static quality characteristics of the system. It is calculated based on the following:

- Function points (FPs) assigned to the system.
- Quality requirements or test strategy for static quality characteristics (QC).

After this, the dynamic test point is added to the static test point to get the total test points (TTP) for the system. This total test point is used for calculating *primary test hours* (PTH). PTH is the effort estimation for primary testing activities, such as preparation, specification, and execution. PTH is calculated based on the environmental factors and productivity factors. Environmental factors include development environment, testing environment, testing tools, etc. Productivity factor is the measure of experience, knowledge, and skills o the testing team.

Secondary testing activities include management activities like controllin the testing activities. *Total test hours* (TTH) is calculated by adding some allow ances to secondary activities and PTH. Thus, TTH is the final effort estima tion for the testing activities.

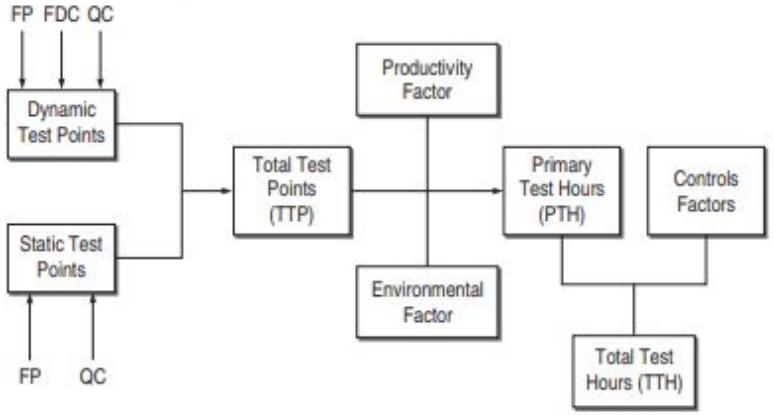


Figure 11.1 Procedure for test point analysis

11.9.2 Calculating Dynamic Test Points

The number of dynamic test points for each function in the system is calculated as

$$DTP = FP \times FDC_w \times QC_{dw}$$

where DTP = number of dynamic test points

FP = function point assigned to function.

FDC_w = weight-assigned function-dependent factors

QC_{dw} = quality characteristic factor, wherein weights are assigned to dynamic quality characteristics

FDC_w is calculated as

$$FDC_w = ((FI_w + UIN_w + I + C) \div 20) \times U$$

where FI_w = function importance rated by users

UIN_w = weights given to usage intensity of the function, i.e. how frequently the function is being used

I = weights given to the function for interfacing with other functions, i.e. if there is a change in function, how many functions in the system will be affected

C = weights given to the complexity of function, i.e. how many conditions are in the algorithm of function

U = uniformity factor

The ratings done for providing weights to each factor of FDC_w can be seen in Table 11.8.

Table 11.8 Ratings for FDC_w factors

Factor/ Rating	Function Importance (FI)	Function usage Intensity (UIN)	Interfacing (I)	Complexity (C)	Uniformity Factor
Low	3	2	2	3	0.6 for the function, wherein test specifications are largely re-used such as in clone function or dummy function. Otherwise, it is 1.
Normal	6	4	4	6	
High	12	12	8	12	

QC_{dw} is calculated based on four dynamic quality characteristics, namely suitability, security, usability, and efficiency. First, the rating to every quality characteristic is given and then, a weight to every QC is provided. Based on the rating and weights, QC_{dw} is calculated.

$$QC_{dw} = \Sigma\ (rating\ of\ QC / 4) \times weight\ factor\ of\ QC$$

Rating and weights are provided based on the following characteristics, as shown in Table 11.9.

Table 11.9 Ratings and weights for QC_{dw}

Characteristic/ Rating	Not Important (0)	Relatively Unimportant (3)	Medium Importance (4)	Very Important (5)	Extremely Important (6)
Suitability	0.75	0.75	0.75	0.75	0.75
Security	0.05	0.05	0.05	0.05	0.05
Usability	0.10	0.10	0.10	0.10	0.10
Efficiency	0.10	0.10	0.10	0.10	0.10

11.9.3 Calculating Static Test Points

The number of static test points for the system is calculated as

$$STP = FP \times \sum QC_{sw} / 500$$

where STP = static test point
FP = total function point assigned to the system
QC_{sw} = quality characteristic factor, wherein weights are assigned to static quality characteristics

Static quality characteristic refers to what can be tested with a checklist. QC_{sw} is assigned the value 16 for each quality characteristic which can be tested statically using the checklist.

$$Total\ test\ points\ (TTP) = DTP + STP$$

11.9.4 Calculating Primary Test Hours

The total test points calculated above can be used to derive the total testing time in hours for performing testing activities. This testing time is called primary test hours (PTH), as it is an estimate for primary testing activities like preparation, specification, execution, etc. This is calculated as

$$PTH = TTP \times productivity\ factor \times environmental\ factor$$

Productivity factor It is an indication of the number of test hours for one test point. It is measured with factors like experience, knowledge, and skill set of the testing team. Its value ranges between 0.7 to 2.0. But explicit weightage for testing team experience, knowledge, and skills have not been considered in this metric.

Environmental factor Primary test hours also depend on many environmental factors of the project. Depending on these factors, it is calculated as

$$Environmental\ factor = weights\ of\ (test\ tools + development\ testing + test\ basis + development\ environment + testing\ environment + testware)/21$$

These factors and their weights are discussed below.

Test tools It indicates the use of automated test tools in the system. Its rating is given in Table 11.10.

Table 11.10 Test tool ratings

1	Highly automated test tools are used.
2	Normal automated test tools are used.
4	No test tools are used.

Test tools It indicates the use of automated test tools in the system. Its rating is given in Table 11.10.

Table 11.10 Test tool ratings

1	Highly automated test tools are used.
2	Normal automated test tools are used.
4	No test tools are used.

Development testing It indicates the earlier efforts made on development testing before system testing or acceptance testing for which the estimate is being done. If development test has been done thoroughly, then there will be less effort and time needed for system and acceptance testing, otherwise, it will increase. Its rating is given in Table 11.11.

Table 11.11 Development testing ratings

2	Development test plan is available and test team is aware about the test cases and their results.
4	Development test plan is available.
8	No development test plan is available.

Test basis It indicates the quality of test documentation being used in the system. Its rating is given in Table 11.12.

Table 11.12 Test basis rating

3	Verification as well as validation documentation are available.
6	Validation documentation is available.
12	Documentation is not developed according to standards.

Development environment It indicates the development platforms, such as operating systems, languages, etc. for the system. Its rating is given in Table 11.13.

Table 11.13 Development environment rating

2	Development using recent platform.
4	Development using recent and old platform.
8	Development using old platform.

Test environment It indicates whether the test platform is a new one or has been used many times on the systems. Its rating is given in Table 11.14.

Table 11.14 Test environment rating

1	Test platform has been used many times.
2	Test platform is new but similar to others already in use.
4	Test platform is new.

Testware It indicates how much testware is available in the system. Its rating is given in Table 11.15.

Table 11.15 Testware rating

1	Testware is available along with detailed test cases.
2	Testware is available without test cases.
4	No testware is available.

11.9.5 Calculating Total Test Hours

Primary test hours is the estimation of primary testing activities. If we also include test planning and control activities, then we can calculate the total test hours.

Total test hours = PTH + Planning and control allowance

Planning and Control allowance is calculated based on the following factors:

Team size

3	≤ 4 team members
6	5–10 team members
12	>10 team members

Planning and control tools

2	Both planning and controlling tools are available.
4	Planning tools are available.
8	No management tools are available.

Planning and control allowance (%)
= weights of (team size + planning and control tools)

Planning and control allowance (hours)
= planning and control allowance (%) × PTH

Thus, the total test hours is the total time taken for the whole system. TTH can also be distributed for various test phases, as given in Table 11.16.

Table 11.16 TTH distribution

Testing phase	% of TTH
Plan	10%
Specification	40%
Execution	45%
Completion	5%

Example 11.4

Calculate the total test points for a module whose specifications are: functions points = 414, ratings for all FDC_w factors are normal, uniformity factor =1, rating for all QC_{dw} are 'very important' and for QC_{sw}, three static qualities are considered.

Solution

$$FDC_w = ((FI_w + UIN_w + I + C)/20) \times U = ((6+4+4+6)/20) \times 1 = 1$$

$$QC_{dw} = \Sigma \text{ (rating of QC / 4)} \times \text{weight factor of QC}$$
$$= (5/4 \times 0.75) + (5/4 \times 0.05) + (5/4 \times 0.10) + (5/4 \times 0.10)$$
$$= 0.9375 + 0.0625 + 0.125 + 0.125 = 1.25$$

$$DTP = FP \times FDC_w \times QC_{dw}$$
$$= 414 \times 1 \times 1.25 = 517.5$$

$$STP = FP \times \Sigma QC_{sw} / 500$$
$$= 414 \times ((16 \times 3)/500) = 39.744$$

Total test points (TTP) = DTP + STP = 517.5 + 39.744 = 557.244

Efficient Test Suite Management

- Software testing is a continuous process that takes place throughout the life cycle of a project.
- Test cases in an existing test suite can often be used to test a modified program.
- However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite.
- Thus, the size of a test suite grows as the software evolves.
- Due to resource constraints, it is important to prioritize the execution of test cases so as to increase the chances of early detection of faults.
- A reduction in the size of the test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software.

12.2 Minimizing the Test Suite and its Benefits

A test suite can sometimes grow to an extent that it is nearly impossible to execute. In this case, it becomes necessary to minimize the test cases such that they are executed for maximum coverage of the software. Following are the reasons why minimization is important:

- Release date of the product is near.
- Limited staff to execute all the test cases.
- Limited test equipments or unavailability of testing tools.

When test suites are run repeatedly for every change in the program, it is of enormous advantage to have as small a set of test cases as possible. Minimizing a test suite has the following benefits:

- Sometimes, as the test suite grows, it can become prohibitively expensive to execute on new versions of the program. These test suites will often contain test cases that are no longer needed to satisfy the coverage criteria, because they are now obsolete or redundant. Through minimization, these redundant test cases will be eliminated.
- The sizes of test sets have a direct bearing on the cost of the project. Test suite minimization techniques lower costs by reducing a test suite to a minimal subset.
- Reducing the size of a test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software, thereby reducing the cost of regression testing.

Thus, it is of great practical advantage to reduce the size of test cases.

TEST SUITE PRIORITIZATION

The reduction process can be best understood if the cases in a test suite are prioritized in some order. The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests. For example, the following priority categories can be determined for the test cases:

- Priority 1 The test cases must be executed, otherwise there may be worse consequences after the release of the product. For example, if the test cases for this category are not executed, then critical bugs may appear.
- Priority 2 The test cases may be executed, if time permits.
- Priority 3 The test case is not important prior to the current release. It may be tested shortly after the release of the current version of the software.
- Priority 4 The test case is never important, as its impact is nearly negligible.

TEST SUITE PRIORITIZATION

Some of them are discussed here: „

- Testers or customers may want to get some critical features tested and presented in the first version of the software. Thus, the important features become the criteria for prioritizing the test cases. But the consequences of not testing some low-priority features must be checked. Therefore, risk factor should be analysed for every feature in consideration.
- Prioritization can be on the basis of the functionality advertised in the market. It becomes important to test those functionalities on a priority basis, which the company has promised to its customers.
- The rate of fault detection of a test suite can reveal the likelihood of faults earlier. „ Increase the coverage of coverable code in the system under test at a faster rate, allowing a code coverage criterion to be met earlier in the test process.
- Increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
- Increase the likelihood of revealing faults related to specific code changes, earlier in the regression testing process.

PRIORITIZATION TECHNIQUES

Prioritization can be done at two levels, as discussed below.

- **Prioritization for regression test suite**: This category prioritizes the test suite of regression testing. Since regression testing is performed whenever there is a change in the software, we need to identify the test cases corresponding to modified and affected modules.
- **Prioritization for system test suite:** This category prioritizes the test suite of system testing. Here, the consideration is not the change in the modules. The test cases of system testing are prioritized based on several criteria: risk analysis, user feedback, fault-detection rate, etc.

COVERAGE-BASED TEST CASE PRIORITIZATION

This type of prioritization [58] is based on the coverage of codes, such as statement coverage, branch coverage, etc. and the fault exposing capability of the test cases. Test cases are ordered based on their coverage. For example, count the number of statements covered by the test cases. The test case that covers the highest number of statements will be executed first. Some of the techniques are discussed below:

- **Total Statement Coverage Prioritization:** This prioritization orders the test cases based on the total number of statements covered. It counts the number of statements covered by the test case and orders them in a descending order. If multiple test cases cover the same number of statements, then a random order may be used.
- For example, if T1 covers 5 statements, T2 covers 3, and T3 covers 12 statements; then according to this prioritization, the order will be T3, T1, T2.

Additional Statement Coverage Prioritization

Total statement coverage prioritization schedules the test cases based on the total statements covered. However, it will be useful if it can execute those statements as well that have not been covered yet. Additional statement coverage prioritization iteratively selects a test case T_1, that yields the greatest statement coverage, then selects a test case which covers a statement uncovered by T_1. Repeat this process until all statements covered by at least one test case have been covered.

For example, if we consider Table 12.1, according to total statement coverage criteria, the order is 2,1,3. But additional statement coverage selects test case 2 first and next, it selects test case 3, as it covers statement 4 which has not been covered by test case 2. Thus, the order according to addition coverage criteria is 2,3,1.

Table 12.1 Statement coverage

Statement	Statement Coverage		
	Test case 1	Test case 2	Test case 3
1	X	X	X
2	X	X	X
3		X	X
4			X
5			
6		X	
7	X	X	
8	X	X	
9	X	X	

Total Branch Coverage Prioritization

In this prioritization, the criterion to order is to consider condition branches in a program instead of statements. Thus, it is the coverage of each possible outcome of a condition in a predicate. The test case which will cover maximum branch outcomes will be ordered first. For example, see Table 12.2. Here, the order will be 1, 2, 3.

Table 12.2 Branch coverage

Branch Statements	Branch Coverage		
	Test case 1	Test case 2	Test case 3
Entry to while	X	X	X
2-true	X	X	X
2-false	X		
3-true		X	
3-false	X		

Additional Branch Coverage Prioritization

Here, the idea is the same as in additional statement coverage of first selecting the test case with the maximum coverage of branch outcomes and then, selecting the test case which covers the branch outcome not covered by the previous one.

12.6.2 Risk-based Prioritization

Risk-based prioritization [22] is a well-defined process that prioritizes modules for testing. It uses risk analysis to highlight potential problem areas, whose failures have adverse consequences. The testers use this risk analysis to select the most crucial tests. Thus, risk-based technique is to prioritize the test cases based on some potential problems which may occur during the project.

- **Probability of occurrence/fault likelihood** It indicates the probability of occurrence of a problem.
- **Severity of impact/failure impact** If the problem has occurred, how much impact does it have on the software.

  Risk analysis uses these two components by first listing the potential problems and then, assigning a probability and severity value for each identified problem, as shown in Table 12.4. By ranking the results in this table in the form of risk exposure, testers can identify the potential problems against which the software needs to be tested and executed first. For example, the problems in the given table can be prioritized in the order of P_5, P_4, P_2, P_3, P_1.

A risk analysis table consists of the following columns:

- **Problem ID** A unique identifier to facilitate referring to a risk factor.
- **Potential problem** Brief description of the problem.
- **Uncertainty factor** It is the probability of occurrence of the problem. Probability values are on a scale of 1 (low) to 10 (high).
- **Severity of impact** Severity values on a scale of 1 (low) to 10 (high).
- **Risk exposure** Product of probability of occurrence and severity of impact.

Table 12.4 Risk analysis table

Problem ID	Potential Problem	Uncertainty Factor	Risk Impact	Risk Exposure
P_1	Specification ambiguity	2	3	6
P_2	Interface problems	5	6	30
P_3	File corruption	6	4	24
P_4	Databases not synchronized	8	7	56
P_5	Unavailability of modules for integration	9	10	90

12.6.3 Prioritization Based on Operational Profiles

This is not a prioritization in true sense, but the system is developed in such a way that only useful test cases are designed. So there is no question of prioritization. In this approach, the test planning is done based on the operation profiles [128, 129] of the important functions which are of use to the customer. An operational profile is a set of tasks performed by the system and their probabilities of occurrence. After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.

PRIORITIZATION USING RELEVANT SLICES

- During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original program, except where a change is expected.
- But re-running the test suite for every change in the software makes regression testing a time-consuming process.
- If we can find the portion of the software which has been affected with the change in software, then we can prioritize the test cases based on this information.
- This is called the slicing technique. The various definitions related to this technique have been defined:
- Execution Slice: The set of statements executed under a test case is called the execution slice of the program.
- Dynamic Slice The set of statements executed under a test case having an effect on the program output is called the dynamic slice of the program with respect to the output variables.
- Relevant Slice The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case, is known as the relevant slice of the program.

PRIORITIZATION BASED ON REQUIREMENTS

- This technique is used for prioritizing the system test cases. The system test cases also become too large in number, as this testing is performed on many grounds.
- Since system test cases are largely dependent on the requirements, the requirements can be analysed to prioritize the test cases.
- This technique does not consider all the requirements on the same level. Some requirements are more important as compared to others.
- Thus, the test cases corresponding to important and critical requirements are given more weight as compared to others, and these test cases having more weight are executed earlier

Customer-assigned priority of requirements Based on priority, the customer assigns a weight (on a scale of 1 to 10) to each requirement. Higher the number, higher is the priority of the requirement.

Requirement volatility This is a rating based on the frequency of change of a requirement. The requirement with a higher change frequency is assigned a higher weight as compared to the stable requirements.

Developer-perceived implementation complexity All the requirements are not equal on a implementation level. The developer gives more weight to a requirement which he thinks is more difficult to implement.

Fault proneness of requirements This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for a new software.