Master Thesis
Software Engineering
Thesis no: MSE-2007:02
January 2007

Metrics in Software Test Planning and Test Design Processes

Wasif Afzal

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Wasif Afzal
Address:
Folkparksvägen 16: 14, 37240,
Ronneby, Sweden.
E-mail:
mwasif_afzal@yahoo.com

University advisor(s):
Dr. Richard Torkar
Assistant Professor
Department of Systems and Software Engineering
Blekinge Institute of Technology
School of Engineering
PO Box 520, SE – 372 25 Ronneby
Sweden

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Software metrics plays an important role in measuring attributes that are critical to the success of a software project. Measurement of these attributes helps to make the characteristics and relationships between the attributes clearer. This in turn supports informed decision making.

The field of software engineering is affected by infrequent, incomplete and inconsistent measurements. Software testing is an integral part of software development, providing opportunities for measurement of process attributes. The measurement of software testing process attributes enables the management to have better insight in to the software testing process.

The aim of this thesis is to investigate the metric support for software test planning and test design processes. The study comprises of an extensive literature study and follows a methodical approach. This approach consists of two steps. The first step comprises of analyzing key phases in software testing life cycle, inputs required for starting the software test planning and design processes and metrics indicating the end of software test planning and test design processes. After establishing a basic understanding of the related concepts, the second step identifies the attributes of software test planning and test design processes including metric support for each of the identified attributes.

The results of the literature survey showed that there are a number of different measurable attributes for software test planning and test design processes. The study partitioned these attributes in multiple categories for software test planning and test design processes. For each of these attributes, different existing measurements are studied. A consolidation of these measurements is presented in this thesis which is intended to provide an opportunity for management to consider improvement in these processes.

Keywords: Metrics, Attributes, Software Test Planning and Test Design.

CONTENTS

1 INTRODUCTION

This chapter provides the background for this thesis, as well as the purpose, aims and objectives of the thesis. The reader will find the research questions along with the research methodology.

1.1 Background

There is a need to establish a software testing process that is cost effective and efficient to meet the market pressures of delivering low cost and quality software. Measurement is a key element of an effective and efficient software testing process as it evaluates the quality and effectiveness of the process. Moreover, it assesses the productivity of the personnel involved in testing activities and helps improving the software testing procedures, methods, tools and activities [21]. Gathering of software testing related measurement data and proper analysis provides an opportunity for the organizations to learn from their past history and grow in software testing process maturity.

Measurement is a tool through which the management identifies important events and trends, thus enabling them to make informed decisions. Moreover, measurements help in predicting outcomes and evaluation of risks, which in turn decreases the probability of unanticipated surprises in different processes. In order to face the challenges posed by the rapidly changing and innovative software industry, the organizations that can control their software testing processes are able to predict costs and schedules and increase the effectiveness, efficiency and profitability of their business [22].Therefore, knowing and measuring what is being done is important for an effective testing effort [9].

1.2 Purpose

The purpose of this thesis is to offer visibility in to the software testing process using measurements. The focus of this thesis is at the possible measurements in the Test Planning and Test Design process of the system testing level.

1.3 Aims and Objectives

The overall goal of the research is to investigate metrics support provided for the test planning and test design activities for Concurrent Development/Validation Testing Model, thereby, enabling the management to have better insight in to the software testing process.

The scope of the research is restricted to the investigation of the metrics used in Test Planning and Test Design activities of the software testing process. Following objectives are set to meet the goal:

- Analyzing the key phases in the Software Testing Life Cycle
- Understanding of the key artifacts generated during the software test planning and test design activities, as well as the inputs required for kicking-off these processes
- Understanding of the role of measurements in improving Software Testing process
- Investigation into the attributes of Test Planning and Test Design processes those are measurable
- Understanding of the current metric support available to measure the identified attributes of the Test Planning and Test Design processes
- Analysis of when to collect those metrics and what decisions are supported by using those metrics

1.4 Research Questions

The main research question for the thesis work is:

- How do the measurable attributes of software test planning and test design processes contribute towards informed decision making?

As a pre-requisite of meeting the requirements of above research question, there is a need to establish some basic understanding of the related concepts. This will be achieved by a literature study concerning the phases of a software testing life cycle, inputs required for starting the software test planning and test design processes and metrics indicating the end of software test planning and test design processes. After having done that, the main research question will be answered by conducting a literature study about the attributes of software test planning and test design processes that are measurable, currently available metrics to support the measurement of identified attributes and when to collect these attributes so as to contribute to informed decision making.

1.4.1 Relationship between Research Questions and Objectives

Figure 1 depicts the relationship between research questions and objectives.

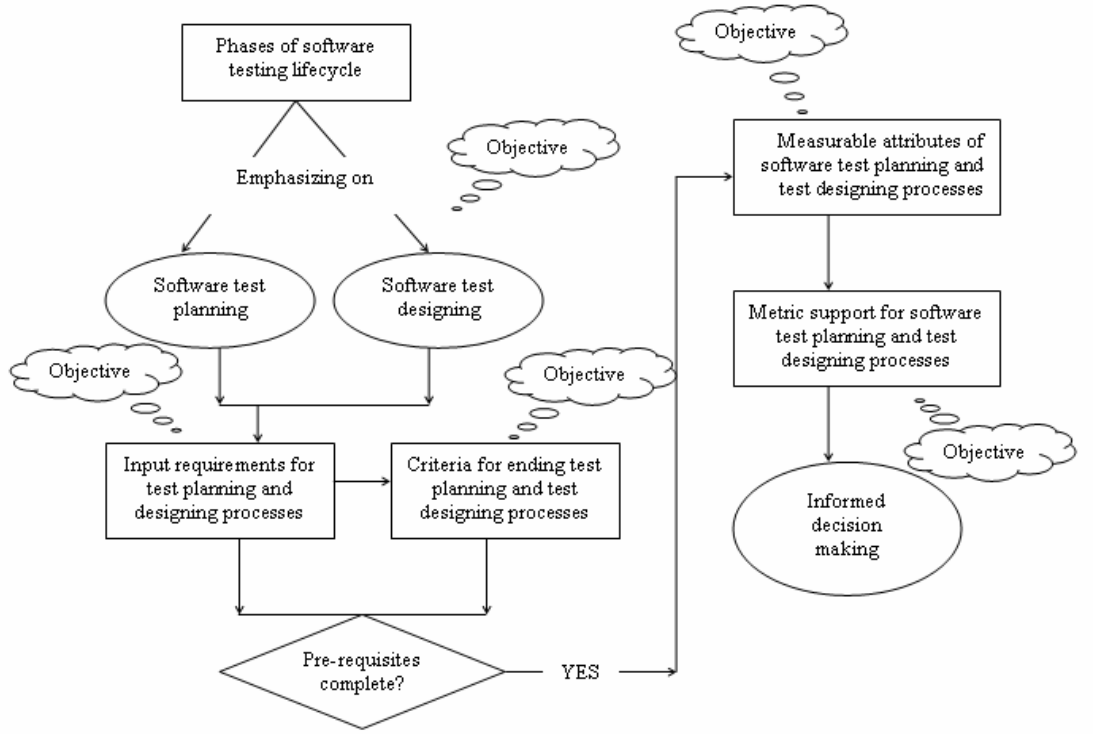

Figure 1. Relationship between research questions and objectives.

1.5 Research Methodology

A detailed and comprehensive literature study will be carried out to gather material related to software testing lifecycle in general and for software test planning, test design and metrics in particular. The literature study will encompass the material as written down in articles, books and web references.

The literature study is conducted with the purpose of finding key areas of interest related to software testing and metrics support for test planning and test design processes. The literature study is divided into two steps. The first step creates a relevant context by

analyzing key phases in the software testing lifecycle, inputs required for starting the software test planning and test design processes and metrics indicating the end of these two processes. The second step identifies the attributes of software test planning and test design processes along with metric support for each of the identified attributes.

While performing the literature study, it is important to define a search strategy. The main search engines used in the study were IEEE Xplore and ACM digital library. The search was performed with combination of various key terms that we were interested in. By using of the above mentioned search engines and using combination of terms for searching, we expect to have minimized the chances of overlooking papers and literature. In addition, manual search was performed on the relevant conference proceedings. The most notable of these is the International Software Metrics Symposium proceedings. In addition to IEEE Xplore and ACM digital library, web search engines Google and Yahoo were used occasionally with the purpose of having a diverse set of literature sources.

For each paper of interest, relevant material was extracted and compared against the other relevant literature. Specifically; the abstract, introduction and the conclusion of each of the articles were studied to assess the degree of relevance of each article. The references at the end of each research paper were also quickly scanned to find more relevant articles. Then a summary of the different views of authors was written to give reader different perspectives and commonalities on the topic.

It is also worth mentioning the use of books written by well-known authors in the field of software metrics and software testing. It is to complement the summarizations done from the research articles found from above mentioned search strategy. A total of twenty five books were referred. The reference lists given at the end of relevant chapters in these books also proved a valuable source of information.

1.5.1 Threats to Validity

Internal validity examines whether correct inferences are derived from the gathered data [101]. The threat to internal validity is reduced to an extent by referring to multiple perspectives on a relevant topic. Moreover, these perspectives are presented in their original form to the extent possible to further minimize the internal validity threats. Also an effort is made to provide a thick [101] discussion on topics.

Another type of validity is called conclusion validity which questions whether the relationships reached in the conclusions are reasonable or not. A search criterion was used to mitigate the conclusion validity threats. This criterion used searching available in established databases, books and web references.

Another validity type as mentioned in [101] is the construct validity which evaluates the use of correct definitions and measures of variables. By searching the whole of databases like IEEE Xplore and ACM digital library for a combination of key terms, complemented by reference to books and use of prior experience and knowledge in the domain, an effort is made to reduce the threats associated with construct validity.

External validity addresses generalizing the results to different settings [101]. An effort is made here to come up with a complete set of attributes that can then be tailored in different settings. External validity is not addressed in this thesis and will be taken up as a future work.

1.6 Thesis Outline

The thesis is divided into following chapters. Chapter 2 introduces the reader to some relevant definitions. Chapter 3 provides an insight into the software testing lifecycle used in the thesis as a baseline and also outlines the starting/ending criteria and input requirements for software test planning and test design processes. Chapter 4 introduces the reader to basic concepts related to software measurements. Chapters 5 and 6 discuss the relevant attributes for software test planning and test design processes respectively. Chapter 7 and 8 cover the

metrics relevant for the respective attributes of the two processes. The recommendations, conclusions and further work are presented in Chapter 9.

2 SOFTWARE TESTING

This chapter explains the concept of software testing along with a discussion on different level of testing.

2.1 The Notion of Software Testing

Software testing is an evaluation process to determine the presence of errors in computer software. Software testing cannot completely test software because exhaustive testing is rarely possible due to time and resource constraints. Testing is fundamentally a comparison activity in which the results are monitored for specific inputs. The software is subjected to different probing inputs and its behavior is evaluated against expected outcomes. Testing is the *dynamic analysis of the product* [18]; meaning that the testing activity probes software for faults and failures while it is actually executed. It is apart from static code analysis, in which analysis is performed without actually executing the program. As [1] points out *if you don't execute the code to uncover possible damage, you are not a tester*. The following are some of the established software testing definitions:

- *Testing is the process of executing programs with the intention of finding errors* [2].
- *A successful test is one that uncovers an as-yet-undiscovered error* [2].
- *Testing can show the presence of bugs but never their absence* [3].
- *The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large scale and small scale systems* [4].
- *Testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item* [5].
- *Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e. testing artifacts) in order to measure and improve the quality of the software being tested* [6].

Software testing is one element of the broader topic called verification and validation (V&V). Software verification and validation uses reviews, analysis and testing techniques to determine whether a software system and its intermediate products fulfill the expected fundamental capabilities and quality attributes [7].

There are some pre-established principles about testing software. Firstly, testing is a process that confirms the existence of quality, not establishing quality. Quality is the overall responsibility of the project team members and is established through right combinations of methods and tools, effective management, reviews and measurements. [8] quotes Brian Marick, a software testing consultant, as saying t*he first mistake that people make is thinking that the testing team is responsible for assuring quality*. Secondly, the prime objective of testing is to discover faults that are preventing the software in meeting customer requirements. Moreover, testing requires planning and designing of test cases and the testing effort should focus on areas that are most error prone. The testing process progresses from component level to system level in an incremental way, and exhaustive testing is rarely possible due to the combinatorial nature of software [8].

2.2 Test Levels

During the lifecycle of software development, testing is performed at several stages as the software is evolved component by component. The accomplishment of reaching a stage in the development of software calls for testing the developed capabilities. Test driven

development takes a different approach in which tests are driven first and functionality is developed around those tests. The testing at defined stages is termed as test levels and these levels progresses from individual units to combining or integrating the units into larger components. Simple projects may consist of only one or two levels of testing while complex projects may have more levels [6]. Figure 2 depicts the traditional waterfall model with added testing levels.

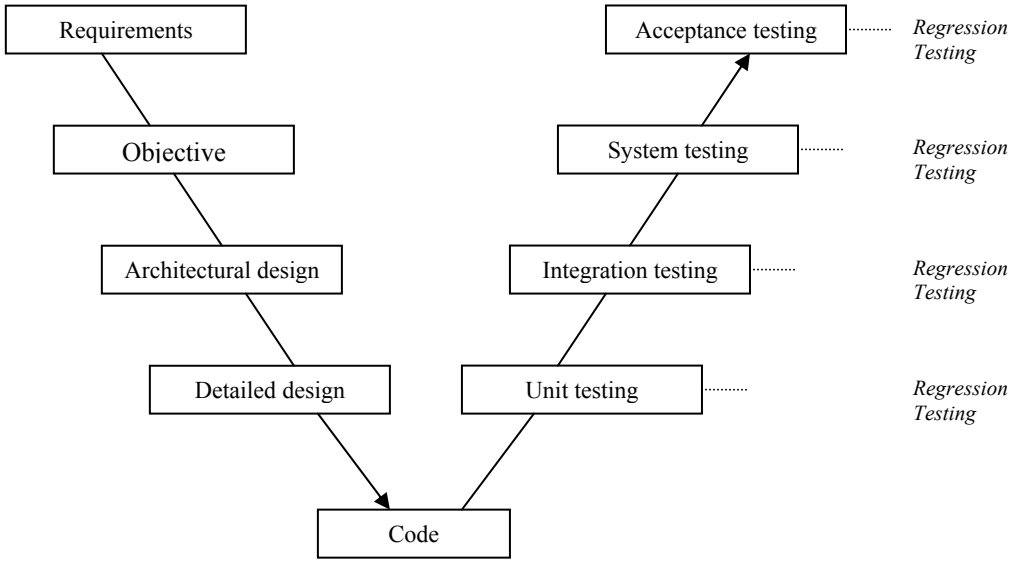

Figure 2. V- model of testing.

The identifiable levels of testing in the V-model are unit testing, integration testing, system testing and acceptance testing. V-model of testing is criticized as being reliant on the timely availability of complete and accurate development documentation, derivation of tests from a single document and execution of all the tests together. In spite of its criticism, it is the most familiar model [1]. It provides a basis for using a consistent testing terminology.

Unit testing finds bugs in internal processing logic and data structures in individual modules by testing them in isolated environment [9]. Unit testing uses the component-level design description as a guide [8]. Unit testing a module requires creation of stubs and drivers as shown in Figure 3 below.

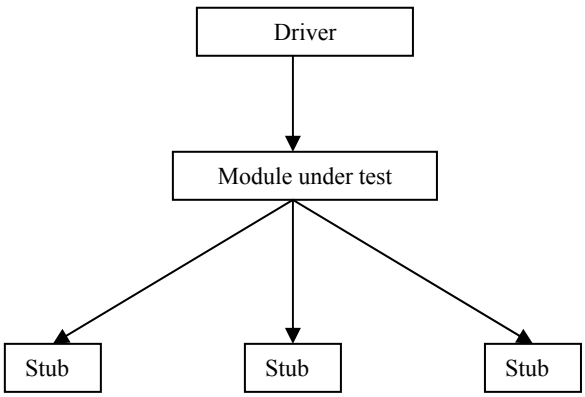

Figure 3. Drivers and stubs in unit testing [9].

According to IEEE standard 1008-1987, unit testing activities consists of planning the general approach, resources and schedule, determining features to be tested, refining the general plan, designing the set of tests, implementing the refined plan and designing,

executing the test procedure and checking for termination and evaluating the test effort and unit [10].

As the individual modules are integrated together, there are chances of finding bugs related to the interfaces between modules. It is because integrating modules might not provide the desired function, data can be lost across interfaces, imprecision in calculations may be magnified, and interfacing faults might not be detected by unit testing [9]. These faults are identified by integration testing. The different approaches used for integration testing includes incremental integration (top-down and bottom-up integration) and big-bang. Figure 4 and Figure 5 below shows the bottom-up and top-down integration strategies respectively [19].

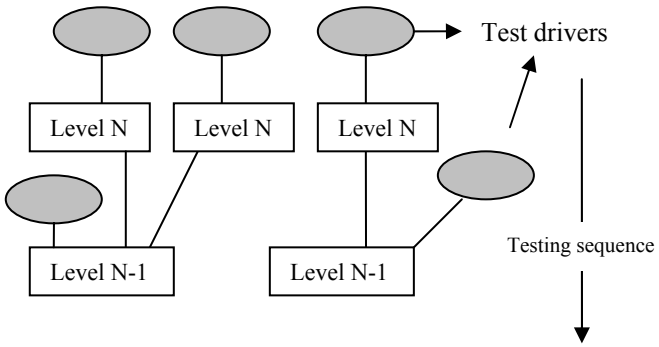

Figure 4. Bottom-up.

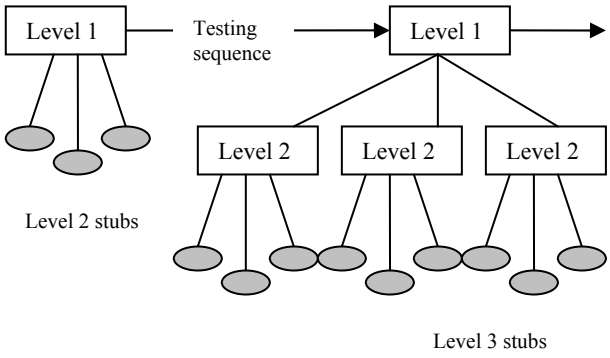

Figure 5. Top- down.

The objective of system testing is to determine if the software meets all of its requirements as mentioned in the Software Requirements Specifications (SRS) document. The focus of system testing is at the requirements level.

As part of system and integration testing, regression testing is performed to determine if the software still meets the requirements if changes are made to the software [9].

Acceptance testing is normally done by the end customers or while customers are partially involved. Normally, it involves selecting tests from the system testing phase or the ones defined by the users [9].

3 SOFTWARE TESTING LIFECYCLE

The software testing process is composed of different activities. Various authors have used different ways to group these activities. This chapter describes the different software testing activities as part of software testing lifecycle by analyzing relevant literature. After comparing the similarities between different testing activities as proposed by various authors, a description of the key phases of software testing lifecycle has been described.

3.1 The Need for a Software Testing Lifecycle

There are different test case design methods in practice today. These test case design methods need to be part of a well-defined series of steps to ensure successful and effective software testing. This systematic way of conducting testing saves time, effort and increases the probability of more faults being caught [8]. These steps highlights when different testing activities are to be planned i.e. effort, time and resource requirements, criteria for ending testing, means to report errors and evaluation of collected data.

There are some common characteristics inherent to the testing process which must be kept in mind. These characteristics are generic and irrespective of the test case design methodology chosen. These characteristics recommends that prior to commencement of testing, formal technical reviews are to be carried out to eliminate many faults earlier in the project lifecycle. Secondly, testing progresses from smaller scope at the component level to a much broader scope at the system level. While moving from component level to the complete system level, different testing techniques are applicable at specific points in time. Also the testing personnel can be either software developers or part of an independent testing group. The components or units of the system are tested by the software developer to ensure it behaves the way it is expected to. Developers might also perform the integration testing [8] that leads to the construction of complete software architecture. The independent testing group is involved after this stage at the validation/system testing level. One last characteristic that is important to bear in mind is that testing and debugging are two different activities. Testing is the process which confirms the presence of faults, while debugging is the process which locates and corrects those faults. In other words, debugging is the fixing of faults as discovered by testing. The following Figure 6 shows the testing and debugging cycles side by side [20].

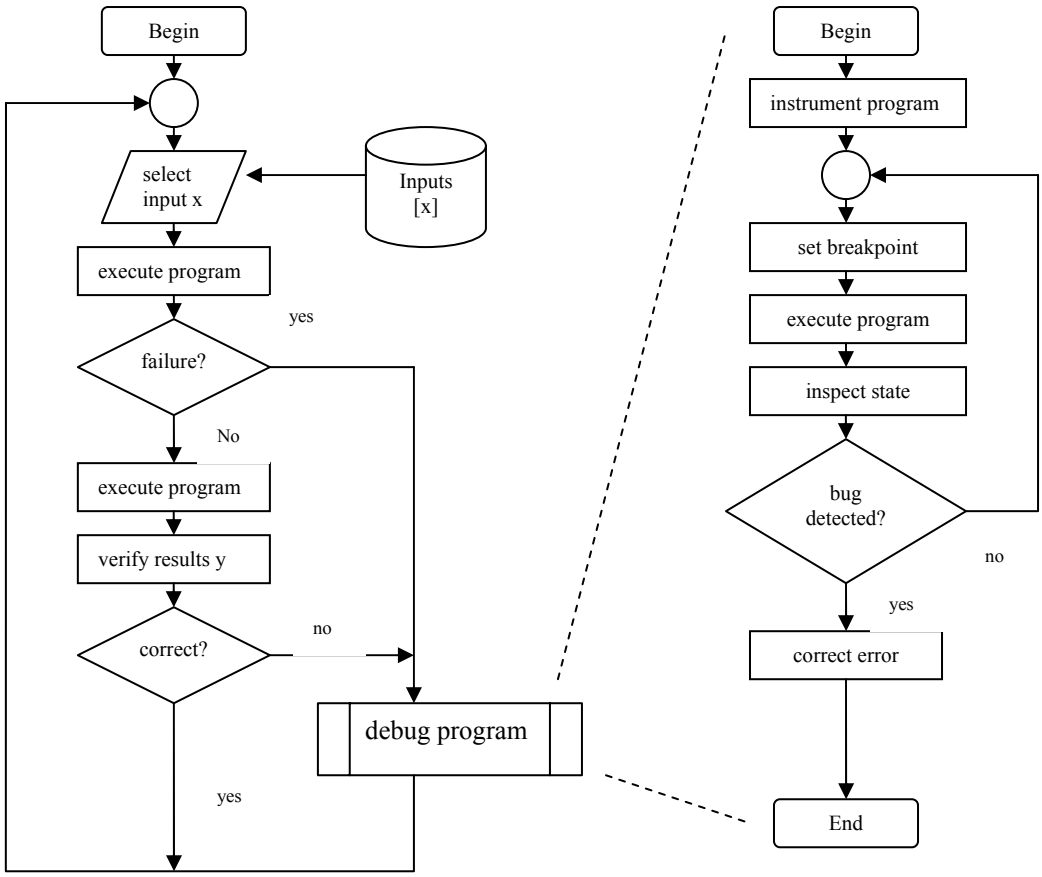

Figure 6. Testing and debugging cycle.

A successful software testing lifecycle should be able to fulfill the objectives expected at each level of testing. The following section intends to describe the expectations of a software testing lifecycle.

3.2 Expectations of a Software Testing Lifecycle

A successful software testing lifecycle accommodates both low level and high level tests. The low level tests verify the software at the component level, while the high level tests validate the software at the system level against the requirements specification. The software testing lifecycle acts as a guide to its followers and management so that the progress is measurable in the form of achieving specific milestones. Previously, the traditional approach to testing was seen as only execution of tests [6]. But modern testing process is a lifecycle approach including different phases. There have been lots of recommendations for involving the software testing professionals early into the project life cycle. These recommendations are also understandable because early involvement of testing professionals helps to prevent faults earlier before they propagate to later development phases. The early involvement of testers in the development lifecycle helps to *recognize omissions, discrepancies, ambiguities, and other problems that may affect the project requirement's testability, correctness and other qualities* [11].

3.3 Software Testing Lifecycle Phases

There has been different proposed software testing lifecycle phases in literature. The intention here is to present an overview of different opinions that exist regarding the

software testing lifecycle phases and to consolidate them. A detailed description of each phase in the lifecycle will follow after consolidating the different views.

According to [8], a software testing lifecycle involving any of the testing strategy must incorporate the following phases:

- Test planning
- Test case design
- Test execution
- Resultant data collection and evaluation

According to [17], the elements of a core test process involve the following activities:

- Test planning
- Test designing
- Test execution
- Test review

According to the [7], the test activities in the life cycle phases include:

- Test plan generation
- Test design generation
- Test case generation
- Test procedure generation
- Test execution

According to [6], the activities performed in a testing lifecycle are as following:

- Plan strategy (planning)
- Acquire testware (analysis, design and implementation)
- Measure the behavior (execution and maintenance)

[12] recommends a testing strategy that is a plan-driven test effort. This strategy is composed of the following eight steps:

- State your assumptions
- Build your test inventory
- Perform analysis
- Estimate the test effort
- Negotiate for the resources to conduct the test effort
- Build the test scripts
- Conduct testing and track test progress
- Measure the test performance

The first four steps are made part of planning; the fifth one is for settlement to an agreement and last three are devoted to testing.

[13] takes a much simpler approach to software testing lifecycle and gives the following three phases:

- Planning
- Execution
- Evaluation

[14] shows the test process as being composed of following phases:

- Planning
- Specification
- Execution
- Recording
- Completion

These activities are shown to be functioning under the umbrella of test management, test asset management and test environment management.

[15] divides the software testing lifecycle into following steps:

- Test Planning and preparation
- Test Execution
- Analysis and follow-up

3.4 Consolidated View of Software Testing Lifecycle

After analyzing the above mentioned software testing lifecycle stages, it can be argued that authors categorize the activities of software testing in to following broad categories:

- Test planning
- Test designing
- Test execution
- Test review

These categories are shown in Figure 7 below.

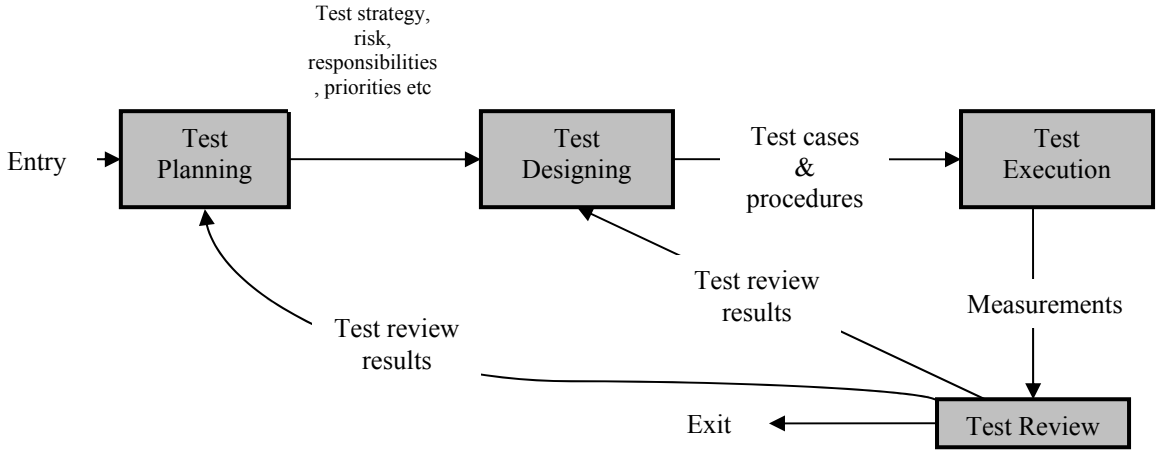

Figure 7. Software testing lifecycle.

3.5 Test Planning

Test planning is one of the keys to successful software testing [6]. The goal of test planning is to take into account the important issues of testing strategy, resource utilization, responsibilities, risks and priorities. Test planning issues are reflected in the overall project planning. The test planning activity marks the transition from one level of software development to the other, estimates the number of test cases and their duration, defines the test completion criteria, identifies areas of risks and allocates resources. Also identification of methodologies, techniques and tools is part of test planning which is dependent on the type of software to be tested, the test budget, the risk assessment, the skill level of available

staff and the time available [7]. The output of the test planning is the test plan document. Test plans are developed for each level of testing.

The test plan at each level of testing corresponds to the software product developed at that phase. According to [7], the deliverable of requirements phase is the software requirements specification. The corresponding test plans are the user acceptance and the system/validation test plans. Similarly, the design phase produces the system design document, which acts as an input for creating component and integration test plans, see Table 1.

Table 1. Test plan generation activities [7].

Lifecycle Activities	Requirements	Design	Implementation	Test
Test Plan Generation	- System - Acceptance	- Component - Integration		

The creation of test plans at different levels depends on the scope of project. Smaller projects may need only one test plan. Since a test plan is developed for each level of testing, [6] recommends developing a *Master Test Plan* that directs testing at all levels. General project information is used to develop the master test plan. Figure 8 presents the test plans generated at each level [6]:

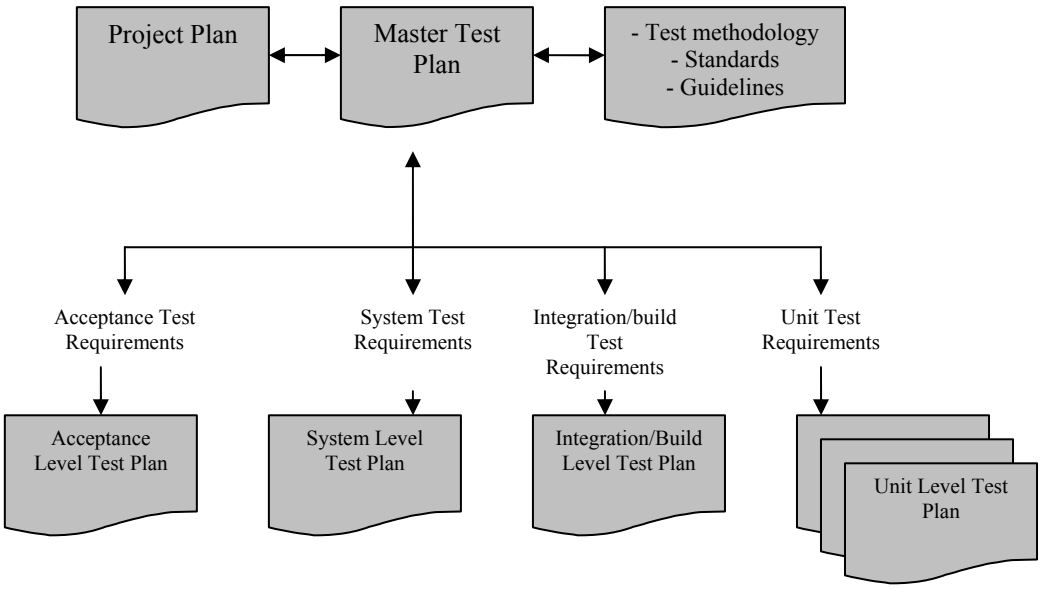

Figure 8. Master vs detailed test plans.

Test planning should be started as early as possible. The early commencement of test planning is important as it highlights the unknown portions of the test plan so that planners can focus their efforts. Figure 9 presents the approximate start times of various test plans [6].

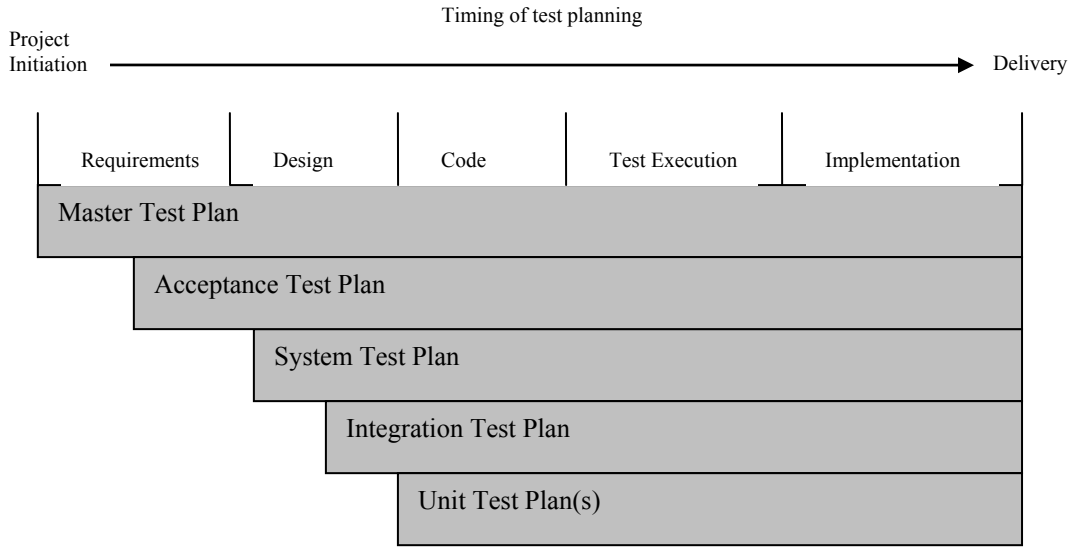

Figure 9. Timing of test planning.

In order to discuss the main elements of a test plan, the system testing plan is taken here as an example. As discussed in the prior section, system testing involves conformance to customer requirements and this is the phase when functional as well as non-functional tests are performed. Moreover, regression tests are performed on bulk of system tests. System testing is normally performed by an independent testing group therefore a manager of testing group or senior test analyst prepares the system test plan. The system test plan is started as soon as the first draft of requirements is complete. The system test plan also takes input from the design documentation. The main elements of a system test plan that are discussed below are planning for delivery of builds, exit/entrance criterion, smoke tests, test items, software risk issues and criterion, features to be tested, approach, items pass/fail criteria, suspension criteria and schedule.

- The system plan mentions the process of delivery of builds to test. For this, the role of software configuration management takes importance because the test manager has to plan delivery of builds to the testing group when changes and bug fixes are implemented. The delivery of the builds should be in such a way that *changes are gathered together and re-implemented into the test environment in such a way as to reduce the required regression testing without causing a halt or slowdown to the test due to a blocking bug* [6].
- The exit/entrance criterion for system testing has to be setup. The entrance criteria will contain some exit criteria from the previous level as well as establishment of test environment, installation of tools, gathering of data and recruitment of testers if necessary. The exit criteria also needs to be established e.g. all test cases have been executed and no identified major bugs are open.
- The system test plan also includes a group of test cases that establish that the system is stable and all major functionality is working. This group of test cases is referred to as smoke tests or testability assessment criteria.
- The test items section describes what is to be tested within the scope of the system test plan e.g. version 1.0 of some software. The IEEE standard recommends referencing supporting documents like software requirements specification and installation guide [6].
- The software risk issues and assumptions section helps the testing group concentrate its effort on areas that are likely to fail. One example of a testing risk is that delay in bug fixing may cause overall delays. One example assumption is that out of box features of any third party tool will not be tested.

- The features to be tested section include the functionalities of the test items to be tested e.g. testing of a website may include verification of content and flow of application.
- The approach section describes the strategy to be followed in the system testing. This section describes the type of tests to be performed, standards for documentation, mechanism for reporting and special considerations for the type of project e.g. a test strategy might say that black box testing techniques will be used for functional testing.
- Items pass/fail criteria establishes the pass/fail criteria for test items identified earlier. Some examples of pass/fail criteria include % of test cases passed and test case coverage [6].
- Suspension criteria identify the conditions that suspend testing e.g. critical bugs preventing further testing.
- The schedule for accomplishing testing milestones is also included, which matches the time allocation in the project plan for testing. *It's important that the schedule section reflect how the estimates for the milestones were determined* [6].

Along with the above sections, the test plan also includes test deliverables, testing tasks, environmental needs, responsibilities, staffing and training needs. The elements of test plan are presented in Figure 10.

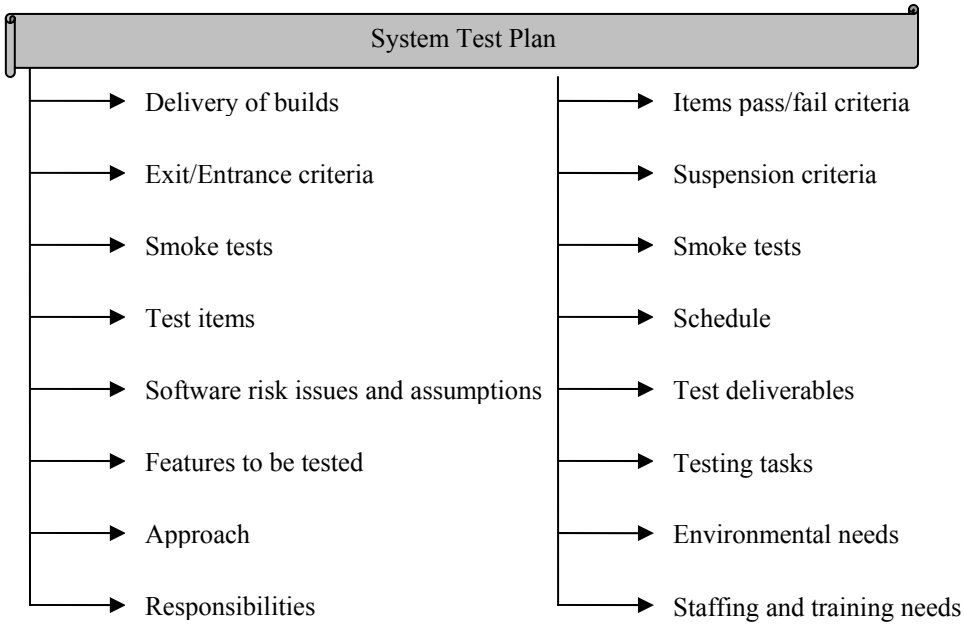

Figure 10. Elements of a system test plan.

3.6 Test Design

The Test design process is very broad and includes critical activities like determining the test objectives (i.e. broad categories of things to test), selection of test case design techniques, preparing test data, developing test procedures, setting up the test environment and supporting tools. Brian Marick points out the importance of test case design: *Paying more attention to running tests than to designing them is a classic mistake* [8].

Determination of test objectives is a fundamental activity which leads to the creation of a testing matrix reflecting the fundamental elements that needs to be tested to satisfy an objective. This requires the gathering reference materials like software requirements specification and design documentation. Then on the basis of reference materials, a team of experts (e.g. test analyst and business analyst) meet in a brainstorming session to compile a list of test objectives. For example, while doing system testing, the test objectives that can be

compiled may include functional, navigational flow, input data fields validation, GUI, data exchange to and from database and rule validation. The design process can take advantage from some generic test objectives applicable to different projects. After the list of test objectives have been compiled, it should be prioritized depending upon scope and risk [6]. The objectives are now ready to be transformed into lists of items that are to be tested under an objective e.g. while testing GUI, the list might contain testing GUI presentation, framework, windows and dialogs. After compiling the list of items, a mapping can be created between the list of items and any existing test cases. This helps in re-using the test cases for satisfying the objectives. This mapping can be in the form of a matrix. The advantages offered by using the matrix are that it helps in identifying the majority of test scenarios and in reducing redundant test cases. This mapping also identifies the absence of a test case for a particular objective in the list; therefore, the testing team needs to create those test cases.

After this, each item in the list is evaluated to assess for adequacy of coverage. It is done by using tester's experience and judgment. More test cases should be developed if an item is not adequately covered. The mapping in the form of a matrix should be maintained throughout the system development.

While designing test cases, there are two broad categories, namely black box testing and white box testing. Black box test case design techniques generate test cases without knowing the internal working of the system. White box test case design techniques examine the structure of code to examine how the system works. Due to time and cost constraints, the challenge designing test cases is that *what subset of all possible test cases has the highest probability of detecting the most errors* [2]. Rather than focusing on one technique, test cases that are designed using multiple techniques is recommended. [11] recommends a combination of functional analysis, equivalence partitioning, path analysis, boundary value analysis and orthogonal array testing. The tendency is that structural (white box) test case design techniques are applied at lower level of abstraction and functional (black box) test case design techniques are likely to be used at higher level of abstraction [15]. According to Tsuneo Yamaura, *there is only one rule in designing test cases: cover all features, but do not make too many test cases* [8].

The testing objectives identified earlier are used to create test cases. Normally one test case is prepared per objective, this helps in maintaining test cases when a change occurs. The test cases created becomes part of a document called test design specification [5]. The purpose of the test design specification is to group similar test cases together. There might be a single test design specification for each feature or a single test design specification for all the features.

The test design specification documents the input specifications, output specifications, environmental needs and other procedural requirements for the test case. The hierarchy of documentation is shown in Figure 11 by taking an example from system testing [6].

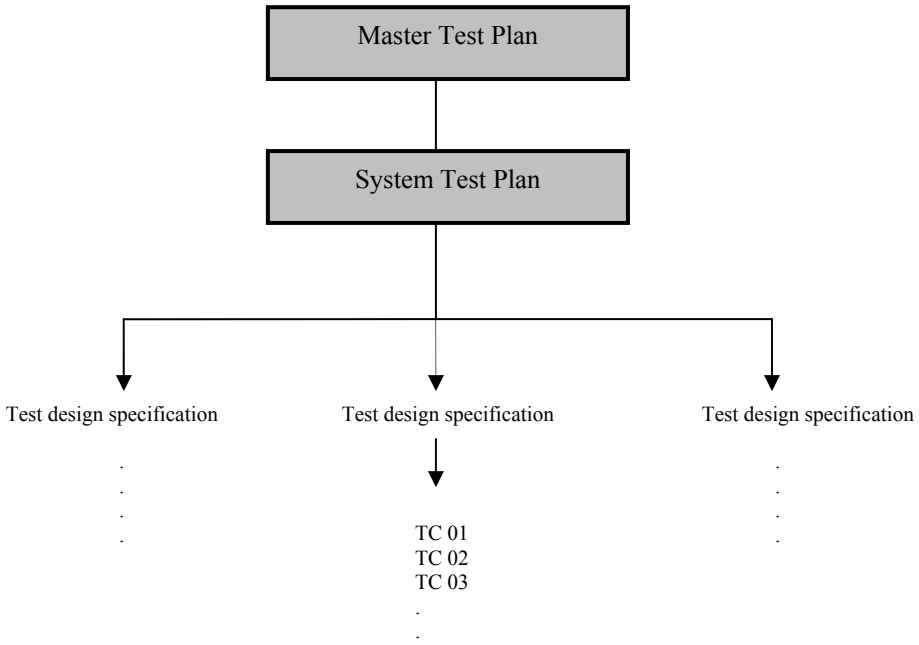

Figure 11. Test design specification.

After the creation of test case specification, the next artifact is called Test Procedure Specification [5]. It is a description of how the tests will be run. Test procedure describes *sequencing of individual test cases and the switch over from one test run to another* [15]. Figure 12 shows the test design process applicable at the system level [6].

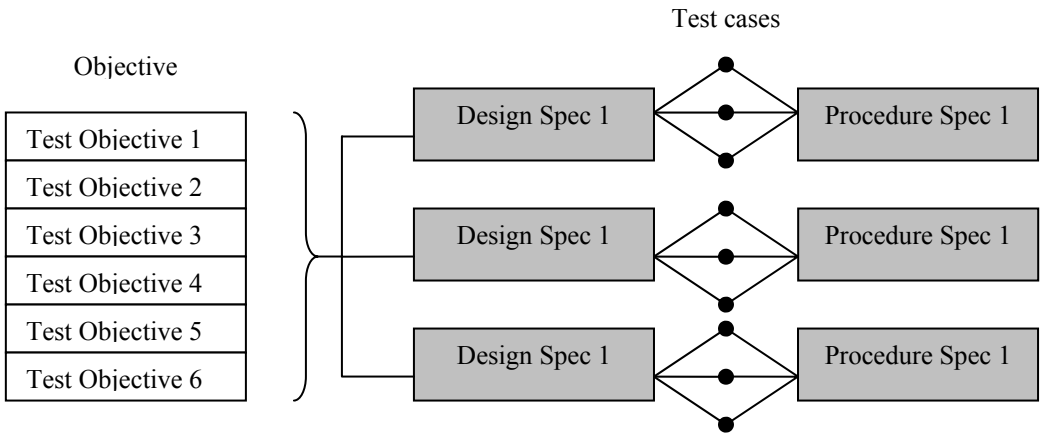

Figure 12. System level test design.

Preparation of test environment also comes under test design. Test environment includes e.g. test data, hardware configurations, testers, interfaces, operating systems and manuals. The test environment more closely matches the user environment as one moves higher up in the testing levels.

Preparation of test data is an essential activity that will result in identification of faults. Data can take the form of e.g. messages, transactions and records. According to [11], the test data should be reviewed for several data concerns:

Depth: The test team must consider the quantity and size of database records needed to support tests.
Breadth: The test data should have variation in data values.
Scope: The test data needs to be accurate, relevant and complete.

Data integrity during testing: The test data for one person should not adversely affect data required for others.
Conditions: Test data should match specific conditions in the domain of application.

It is important that an organization follows some test design standards, so that everyone conforms to the design guidelines and required information is produced [16]. The test procedure and test design specification documents should be treated as living documents and they should be updated as changes are made to the requirements and design. These updated test cases and procedures become useful for reusing them in later projects. If during the course of test execution, a new scenario evolves then it must be made part of the test design documents. After test design, the next activity is test execution as described next.

3.7 Test Execution

As the name suggests, test execution is the process of running all or selected test cases and observing the results. Regarding system testing, it occurs later in software development lifecycle when code development activities are almost completed. The outputs of test execution are test incident reports, test logs, testing status and test summary reports [6]. See Figure 13.

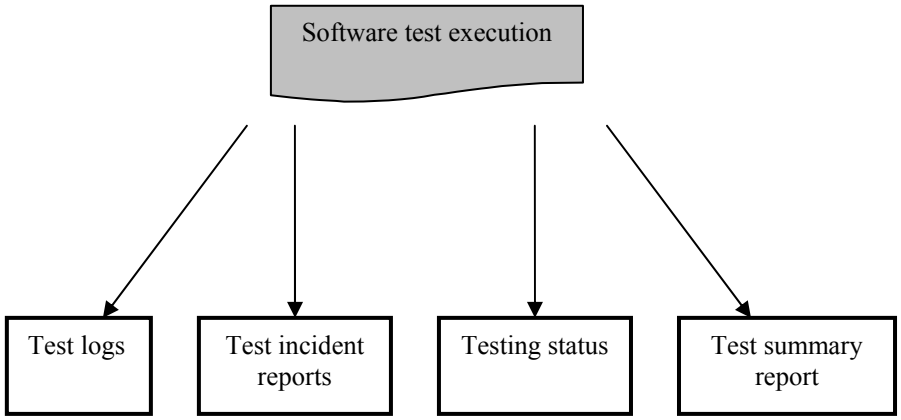

Figure 13. Artifacts for software test execution.

Different roles are involved in executing tests at different levels. At unit test level, normally developers execute tests; at integration test levels, both testers and developers may be involved; at system test level, normally testers are involved but developers and end users may also participate and at acceptance testing level, the majority of tests are performed by end-users, although the testers might be involved as well.

When starting the execution of test cases, the recommended approach is to first test for the presence of major functionality in the software. The testability assessment criteria or smoke tests defined in the system test plan provides a set of test cases that ensures the software is in a stable condition for the testing to continue forward. The smoke tests should be modified to include test cases for modules that have been modified because they are likely to be risky.

While executing test cases, there are always new scenarios discovered that were not initially designed. It is important that these scenarios are documented to build a comprehensive test case repository which can be referred to in future.

As the test cases are executed, a test log records chronologically the details about the execution. *A test log is used by the test team to record what occurred during test execution* [5]. A test log helps in replicating the actions to reproduce a situation occurred during testing.

Once a test case is successful i.e. it is able to find a failure, a test incident report is generated for problem diagnosis and fixing the failure. In most of the cases, the test incident report is automated using a defect tracking tool. Out of the different sections in a test incident report as outlined by [5], the most important ones are:

- Incident summary
- Inputs
- Expected results
- Actual results
- Procedure step
- Environment
- Impact

The *incident summary* relates the incident to a test case. *Inputs* describe the test data used to produce the defect, *expected results* describe the result as expected by correct implementation of functionality, *actual results* are results produced after executing test case, *procedure step* describes the sequence of steps leading to an incident, *environment* describes environment used for testing e.g. system test environment and *impact* assigns severity levels to a bug which forms the basis for prioritizing it for fixation. Different severity levels are used e.g. minor, major and critical. The description of these severity levels is included in the test plan document. [6] highlights the attributes of a good incident report as being based on factual data, able to recreate the situation and write out judgment and emotional language.

In order to track the progress of testing, testing status is documented. Testing status reports normally include a description of module/functionality tested, total test cases, number of executed test cases, weights of test cases based on either time it takes to execute or functionality it covers, percentage of executed test cases with respect to module and percentage of executed test cases with respect to application.

The testing application produces a *test summary report* at end of testing to summarize the results of designated testing activities.

In short, the test execution process involves allocating test time and resources, running tests, collecting execution information and measurements and observing results to identify system failures [6]. After test execution, the next activity is test review as described next.

3.8 Test Review

The purpose of the test review process is to analyze the data collected during testing to provide feedback to the test planning, test design and test execution activities. When a fault is detected as a result of a successful test case, the follow up activities are performed by the developers. These activities involve developing an understanding of the problem by going through the test incident report. The next step is the recreation of the problem so that the steps for producing the failure are re-visited to confirm the existence of a problem.

Problem diagnosis follows next which examines the nature of problem and its cause(s). Problem diagnosis helps in locating the exact fault, after which the activity of fixing the fault begins [6]. The fixing of defects by developers triggers an adjustment to test execution activity. For example a common practice is that the testing team does not execute the test cases related to one that produced the defect. They rather continue with others and as soon as the fix arrives from the development end, the failing test case is re-run along with other related test cases so as to be sure that the bug fix has not adversely affected the related functionality. This practice helps in saving time and effort when executing test cases with higher probability of finding more failures.

The review process uses the testing results of the prior release of software. The direct results of the previous testing activity may help deciding in which areas to focus the test activities. Moreover, it helps in schedule adjustment, resource allocation(s) and adjustment(s), planning for post-release product support and planning for future products [6].

Regarding review of overall testing process, different assessments can be performed e.g. reliability analysis, coverage analysis and overall defect analysis. *Reliability analysis* can be performed to establish whether the software meets the pre-defined reliability goals or not. If they are met, then the product can be released and a decision on a release date can be reached. If not, then time and resources required to reach the reliability goal are hopefully predictable. *Coverage analysis* can be used as an alternative criterion for stopping testing. *Overall defect analysis* can lead to identify areas that are highly risky and helps focusing efforts on these areas for focused quality improvement.

3.9 Starting/Ending Criteria and Input Requirements for Software Test Planning and Test Design Processes

After describing the software testing lifecycle activities, this section continues to describe the starting and ending criteria and input requirements for software test planning and test design activities. The motivation for discussing them is to be able to better define the scope of study that is to follow in subsequent chapters.

It is a recommended approach that test planning should be started as soon as possible and proceeded in parallel to software development. If the test team decides to form a master test plan, it is done using general project information while detailed level test plans require specific software information. It is depicted in Figure 14 [6].

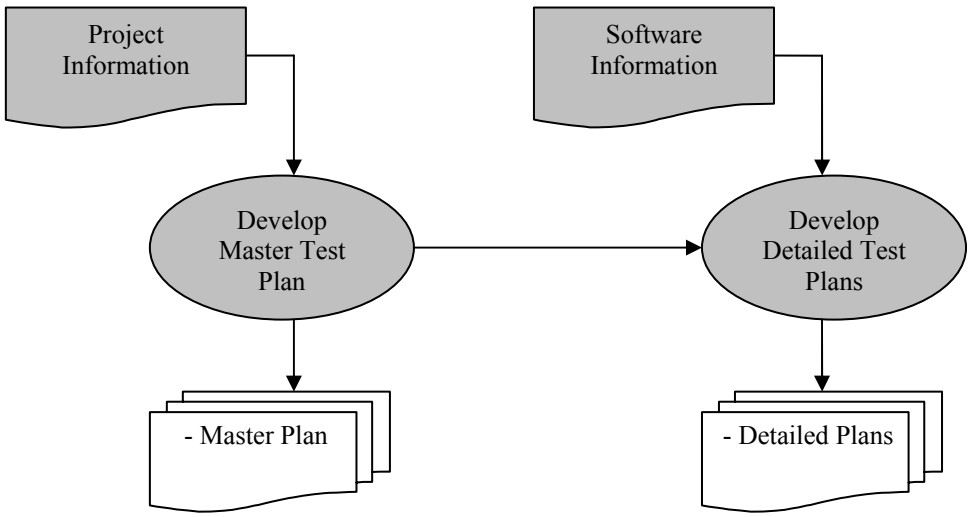

Figure 14. Broad information needs for test planning.

The test plan document generated as part of test planning is considered as a live document, i.e. it can be updated to reflect the latest changes in software documentation. When complete pre-requisites for starting the test planning are not available, the test plans are prepared with the help of as much information as available and then changes are incorporated later on as the documents and input deliverables are made available. The test plan at each level of testing must account for information that is specific to that level e.g. risks and assumptions, resource requirements, schedule, testing strategy and required skills. The test plans according to each level are written in an order such that the plan prepared first is executed in the last i.e. acceptance test plan is the fist plan to be formalized but is executed last. The reason for its early preparation is that the artifacts required for its completion are available first.

An example of a system test plan is taken here. The activity start time for the system test plan is normally when the requirements phase is complete and the team is working on the design documents [6]. According to [7], the input requirements for the system test plan generation are *concept documentation, SRS, interface requirements documentation and user*

documentation. If a master test plan exists, then the system test plan seeks to add content to the sections in the master test plan that is specific to the system test plan. While describing the *test items* section in [5], IEEE recommends the following documentation to be referenced:

- *Requirements specification*
- *Design specification*
- *User's guide*
- *Operations guide*

All important test planning issues are also important project planning issues, [6] therefore a tentative (if not complete) project plan is to be made available for test planning. A project plan provides useful perspective in planning the achievement of software testing milestones, features to be tested and not to be tested, environmental needs, responsibilities, risks and contingencies. Figure 14 can be modified to fit the input requirements for system testing as shown in Figure 15:

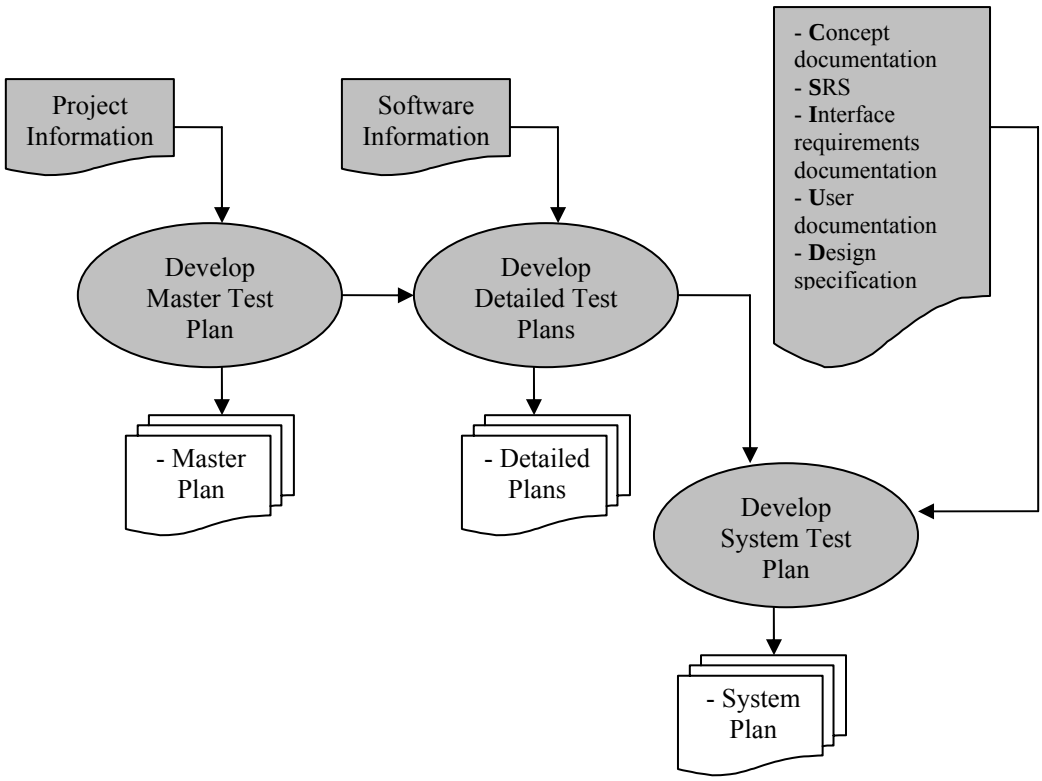

Figure 15. Information needs for system test plan.

Since test planning is an activity that is live (agile), therefore, there is no stringent criterion for its ending. However, the system test schedule mentions appropriate timelines regarding when a particular milestone will be achieved in system testing (in accordance with the overall time allocated for system testing in project schedule), therefore, the test team needs to shift efforts to test design according to the set schedule. Figure 16 depicts a system test schedule for an example project.

ID		Task Name	Duration	Start	Finish
1		**PSG-GP-0.1.00**	**9.5 days**	**Thu 7/14/05**	**Wed 7/27/05**
2		**Test Planning**	**3 days**	**Thu 7/14/05**	**Mon 7/18/05**
3		Test Plan and QA Schedule Preparation	1 day	Thu 7/14/05	Thu 7/14/05
4		Test Point Preparation	2 days	Fri 7/15/05	Mon 7/18/05
5		**Testing and Bug Reporting**	**5 days**	**Wed 7/20/05**	**Wed 7/27/05**
6		Testing Cycle X1	2.5 days	Wed 7/20/05	Fri 7/22/05
7		Web Site Publishing	0.5 days	Wed 7/27/05	Wed 7/27/05

Figure 16. An example system test schedule.

In some organizations, a checklist called *software quality assurance and testing checklist* is used to check whether important activities and deliverables are produced as part of software planning activity. A completion of these activities and deliverables marks the transition from test planning to test design (Table 2).

Table 2. Software quality assurance and testing checklist. This checklist is not exhaustive and is presented here as an example.

Sr. no.	Task	Executed (Y/N/N.A.)	Reason if task not executed	Execution date
1.	Have the QA resources, team responsibilities, and management activities been planed, developed and implemented?			
2.	Have the testing activities that will occur throughout the lifecycle been identified?			
3.	Has QA team lead (QA TL) prepared QA schedule and shared it with project manager (PM)?			
4.	Has QA TL prepared the Measurement Plan?			
5.	Has QA team received all reference documents well before end of test planning phase?			
6.	Has QA Team identified and sent testing risks and assumptions for the application under test (AUT) to PM, development team and QA section lead (QA SL)?			
7.	Has QA TL discussed the Testability Assessment Criteria (TAC) with PM, Development team lead (Dev TL) and QA SL?			
8.	Has QA Team sent a copy of test plan for review to QA TL, QA SL and PM before actual testing?			
9.	Has QA team received any feed back on test plan from PM?			
10.	Have the QA deliverables at each phase been specified and communicated to the PM?			

After software testing teams gain some confidence over the completion of the test planning activities, the test design activities start. Test design activity begins with the gathering of some reference material. These reference materials include [6]:

- *Requirements documentation*
- *Design documentation*
- *User's manual*
- *Product specifications*
- *Functional requirements*
- *Government regulations*
- *Training manuals*
- *Customer feedback*

Figure 17 depicts these needs for test design.

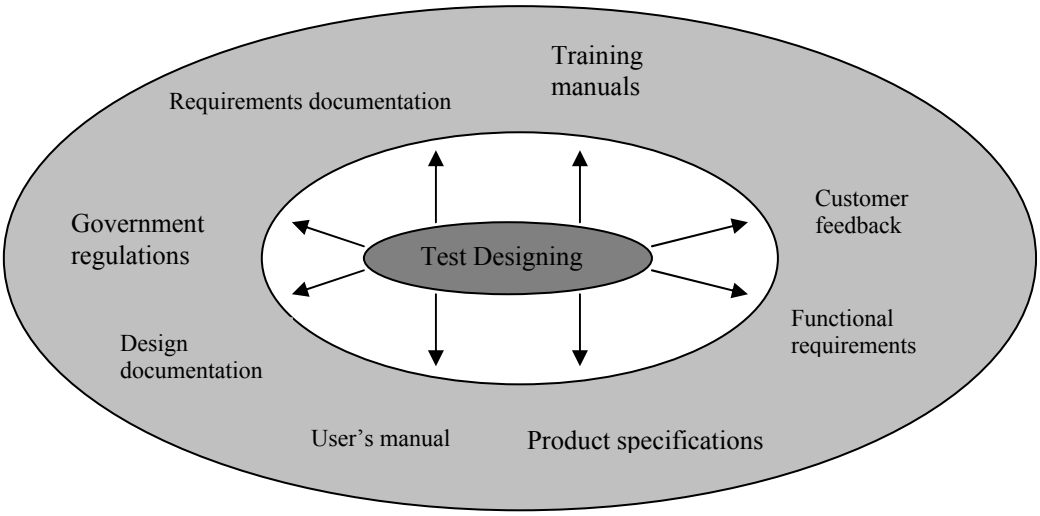

Figure 17. Test designing needs.

The following activities usually mark the end of the test design activity:

- Test cases and test procedures have been developed
- Test data has been acquired
- Test environment is ready
- Supporting tools for test execution are implemented

4 SOFTWARE MEASUREMENT

After defining the key activities in the software testing process and defining when the software test planning and test design activities starts and ends, we want to measure the attributes of the software test planning and test design activities. The identification of the attributes forms the basis for deriving useful metrics that contribute towards informed decision making.

4.1 Measurement in Software Engineering

Measurement analyzes and provides useful information about the attributes of entities. Software measurement is integral to improving software development [23]. [24] describes an entity as *an object or an event in the real world*. A software engineering example of entity being an object is the programming code written by a software engineer and the example of an event is the test planning phase of a software project. An attribute is the feature or property or characteristic of an entity which helps us to differentiate among different entities. Numbers or symbols are assigned to the attributes of entities so that one can reach some judgements about the entities by examining their attributes [24]. According to [26], *if the metrics are not tightly linked to the attributes they are intended to measure, measurement distortions and dysfunctional should be common place.* Moreover, there should be an agreement on the meaning of the software attributes so that only those metrics are proposed that are adequate for the software attributes they ought to measure [25].

Measurements are a key element for controlling the software engineering processes. By controlling, it is meant that one can assess the status of the process, observe trends to predict what is likely to happen and take corrective actions for modifying our practices. Measurements also play its part in increasing our understanding of the process by making visible relationships among process activities and entities involved. Lastly, measurements leads to improvement in our processes by modifying the activities based on different measures (Figure 18).

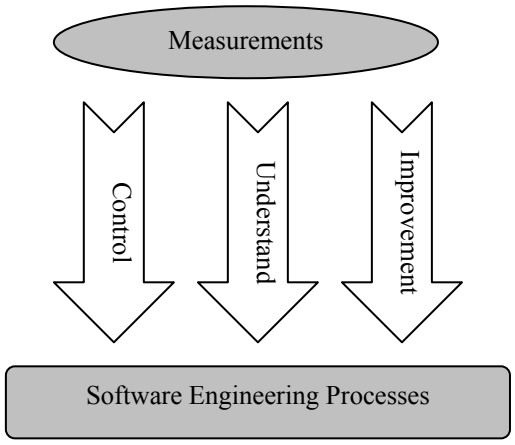

Figure 18. Contribution of measurements towards software engineering processes.

[27] claims that the reasons for carrying out measurements is to gain understanding, to evaluate, to predict and to improve the processes, products, resources and environments. [24] describes the key stages of formal measurement. The formal process of measurement begins with the identification of attributes of entities. It follows with the identification of empirical relations for identified attribute. The next step is the identification of numerical

relations corresponding to each empirical relation. The final two steps involve defining mapping from real world entities to numbers and checking that numerical relations preserve, and are preserved by empirical relations. Figure 19 shows the key stages of formal measurement [24].

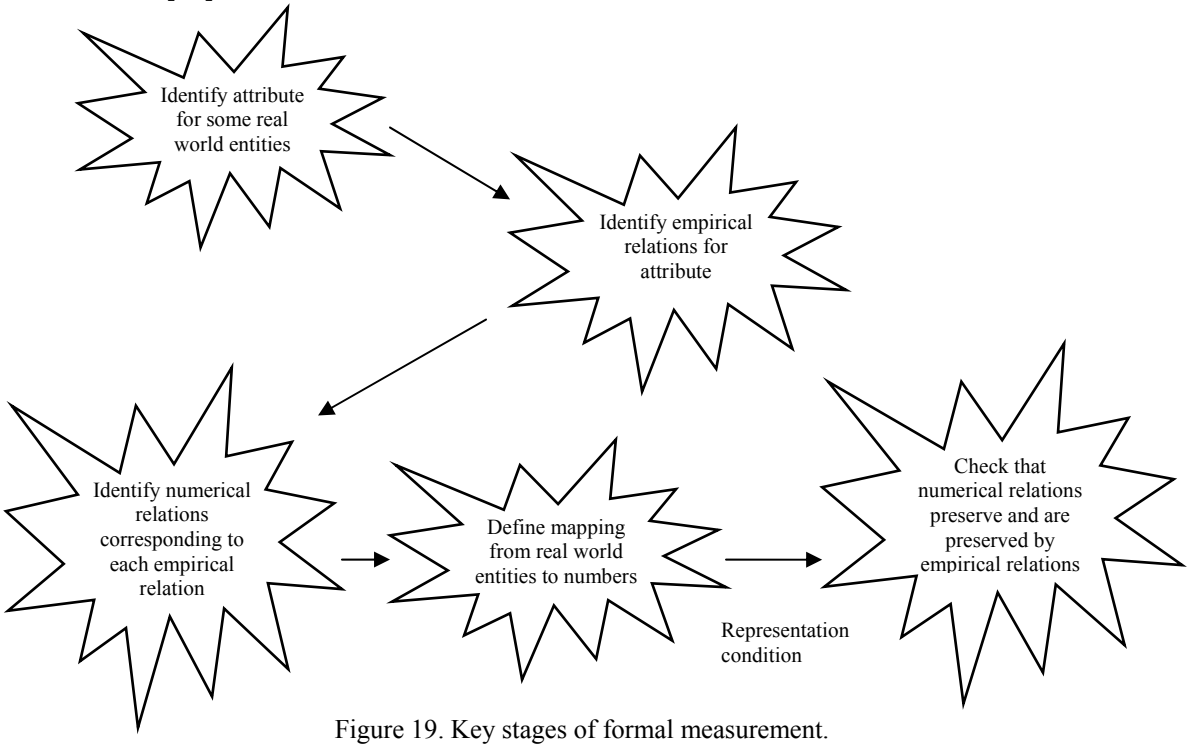

Figure 19. Key stages of formal measurement.

At a much higher level, the software measurement approach, in an organization that is geared towards process improvement, constitutes a closed-loop feedback mechanism consisting of steps as setting up business objectives, establishing quality improvement goals in support of business objectives, establishing metrics that measure progress of achieving these goals and identification and implementation of development process improvement goals [28] (Figure 20).

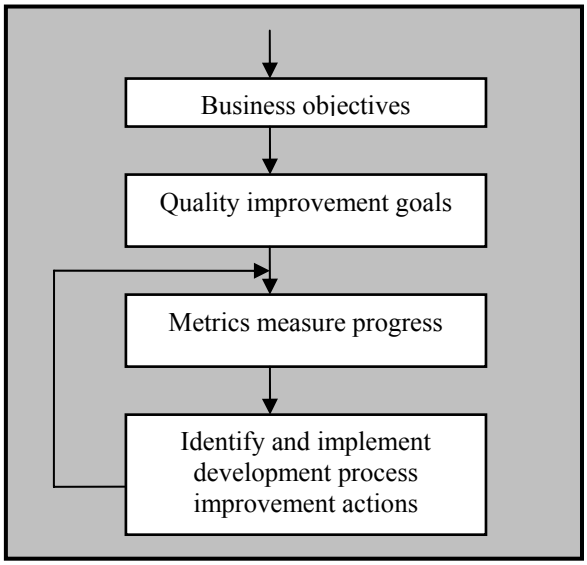

Figure 20. Software measurement approach [28].

[24] differentiates between three categories of measures depending upon the entity being measured. These measures are process, product and resource measures. [29] defines a process metric as *a metric used to measure characteristics of the methods, techniques and tools employed in developing, implementing and maintaining the software system.* In addition, [29] defines a product metric as *a metric used to measure the characteristics of any intermediate or final product of software development process.*

The measurement of an attribute can either be direct or indirect. *The direct measurement of an attribute involves no other attribute or entity* [24]. [29] describes a direct measure as the one that *does not depend upon a measure of any other attribute.* Example of a direct attribute is duration of testing process measured in elapsed time in hours [24]. [26] believes that attributes in the field of software engineering are too complex that measures of them cannot be direct, while [26] claims that the measures of supposedly direct attributes depend on many other system-affecting attributes. The indirect measurement of an attribute is dependent on other attributes. An example of an indirect attribute is test effectiveness ratio computed as a ratio of the number of items covered and the total number of items [24]. There needs to be a distinction made between internal attributes and external attributes of a product, process or resource. According to [24], an internal attribute is the one that can be measured *by examining the product, process or resource on its own, separate from its behaviour.* On the other hand, external attributes are measured only with respect to *how the product, process or resource relates to its environment* [24].

4.2 Benefits of Measurement in Software Testing

Measurements also have to offer benefits to software testing, some of which are listed below [30]:

- *Identification of testing strengths and weaknesses.*
- *Providing insights into the current state of the testing process.*
- *Evaluating testing risks.*
- *Benchmarking.*
- *Improving planning.*
- *Improving testing effectiveness.*
- *Evaluating and improving product quality.*
- *Measuring productivity.*
- *Determining level of customer involvement and satisfaction.*
- *Supporting controlling and monitoring of the testing process.*
- *Comparing processes and products with those both inside and outside the organization.*

4.3 Process Measures

There are many factors which affect organizational performance and software quality and according to [23], process is only one of them. Process connects three elements that have an impact on organizational performance and these are skills and motivation of *people*, complexity of *product*, and finally *technology*. Also the process exists in an environment where it interacts with customer characteristics, business conditions and development environment (Figure 21).

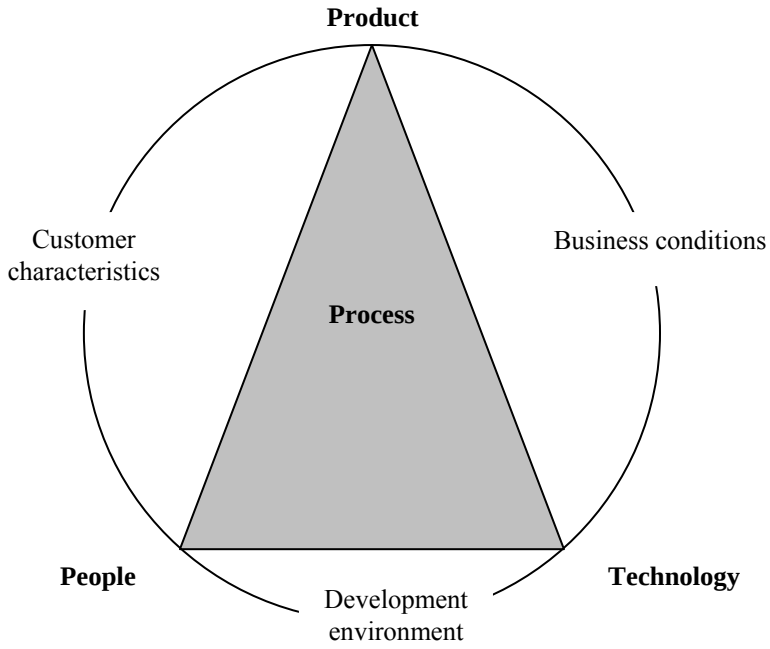

Figure 21. Factors in software quality and organizational performance [8].

The process of measuring process attributes involves examining the process of interest and deciding what kind of information would help us to understand, control or improve the process [24]. In order to assess the quality of a process, there are several questions that need to be answered, [24] e.g.:

1. *How long it takes for the process to complete?*
2. *How much it will cost?*
3. *Is it effective?*
4. *Is it efficient?*

The above questions mark the *tactical* application of software metrics. The tactical use of metrics is useful in project management where they are used for project planning, project estimation, project monitoring and evaluation of work products. The use of software metrics in process improvement marks a *strategic* application for the organization as it is used as a strategic advantage for the organization [31]. A common strategy to understand, control and improve a software process is to measure specific attributes of the process, derive a set of meaningful metrics for those attributes, and using the indicators provided by those metrics to enhance the efficacy of a process. According to [24], there are three internal process attributes that can be measured directly, namely:

1. *The duration of the process or one of its activities.*
2. *The effort associated with the process or one of its activities.*
3. *The number of incidents of a specified type arising during the process or one of its activities.*

These measures, when combined with others, offer visibility into the activities of a project. According to [31], the most common metrics measure size, effort, complexity and time. Focussing on calendar time, engineering effort and faults is used commonly by organizations to maximize customer satisfaction, minimize engineering effort and schedule and minimize faults [31]. There are external process attributes as well which are important

for managing a project. The most notable of these attributes are controllability, observability and stability [24]. These attributes are often described with subjective ratings but these ratings may form the basis for the derivation of empirical relations for objective measurement.

According to [32], the basic attributes required for planning, tracking and process improvement are size, progress, reuse, effort, cost, resource allocations, schedule, quality, readiness for delivery and improvement trends.

Moreover, according to [33], the minimal set of attributes to be measured by any organization consists of system size, project duration, effort, defects, and productivity.

The basic advantage of measuring process attributes is that it assists management in making process predictions. The information gathered during past processes or during early stages of a current process helps to predict what is likely to happen later on [24]. These predictions becomes the basis for important decision makings e.g. during the testing process, end of testing predictions and reliability/quality predictions can guide sound judgement. *Process metrics are collected across all projects and over long periods of time* [8]. It is important to bear in mind that predictions are estimates or a range rather than a single number. Formally, an estimate is a *median of unknown distribution* [24], e.g. if median is used as a prediction for project completion, there is a 50% chance that the project will take longer. Therefore, a good estimate is one which is given as a confidence interval with upper and lower bounds. Also estimates become more objective as more data is available, so estimates are not static.

4.4 A Generic Prediction Process

One way of describing the prediction process is in terms of models and theories [24]. The prediction process comprises the following steps:

1. Identification of key variables affecting attribute.
2. Formulation of theory showing relationships among variables and with attribute.
3. Building of a model reflecting the relationships.

In Figure 22 one can see the prediction process.

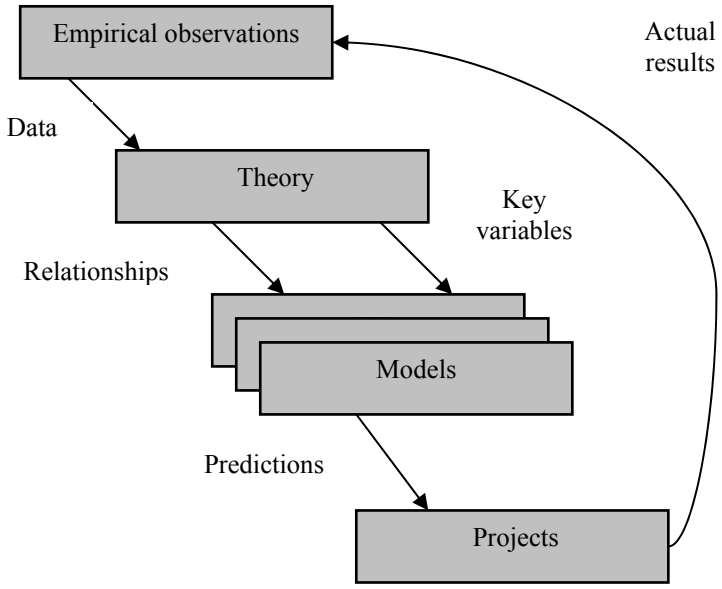

Figure 22. General prediction process [24].

5 ATTRIBUTES FOR SOFTWARE TEST PLANNING PROCESS

A process-oriented approach for software testing ensures that the work products are tested as they are complete at key process points. Attributes identification of software test planning process leads to generation of useful metrics which satisfies the four key aspects of managing a test planning effort i.e. establishing goals for test planning process, execution of the test planning process, measurement of the results of test planning process and changing the test planning process [31] (See Figure 23).

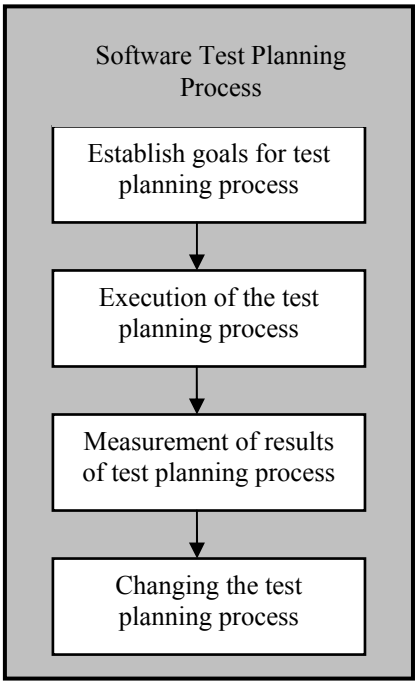

Figure 23. Managing software test planning process through metrics.

The system test planning process takes vital input from the development process in the form of knowledge of where to optimally utilize the testing resources. This knowledge comes from the results of inspection process during development which exposes potentially problematic areas. For example, the design and code inspections highlight error prone modules and functions which are overly complex. The test planning can obviously take advantage of focusing its effort on those error prone and overly complex modules [31].

The attributes to be measured is dependent on three factors [33]. Firstly, the attributes to measure is dependent on the time and phase in the software development lifecycle. Secondly, new attributes to measure emerges from new business needs. Thirdly, attributes to be measured are aligned to ultimate goal of the project.

The following text identifies the attributes that are useful for measurement during software test planning.

5.1 Progress

Following attributes are defined under the progress category of metrics:

- Suspension criteria for testing.
- The exit criteria.

- Scope of testing.
- Monitoring of testing status.
- Staff productivity.
- Tracking of planned and unplanned submittals.

5.1.1 The Suspension Criteria for Testing

The suspension criteria of testing are based on the measurement of certain attributes. Therefore, metrics that establish conditions for suspending testing are to be identified at the test planning phase.

5.1.2 The Exit Criteria

The exit criteria for testing needs to be established during test planning. The decision is to be based on metrics, so that an exit criterion is flagged after meeting a condition.

5.1.3 Scope of Testing

The test planning activity also determines the scope of testing. So metrics to answer how much of it is to be tested, is required [12].

5.1.4 Monitoring of Testing Status

On time and in budget completion of testing tasks is essential to meet schedule and cost goals. Therefore, the testing status needs to be monitored for adherence to schedule and cost [30].

5.1.5 Staff Productivity

Management is interested in knowing the staff productivity. Measures for tester's productivity should be established at the test planning phase to help a test manager learn how a software tester distributes his time over various testing activities [30]. In this metric, using measures for the size of the test plan (e.g. in terms of length) and time taken by the staff to complete it, the productivity of the staff can be gauged [24]. Productivity is related to the efficiency of the tester in making the test plan.

5.1.6 Tracking of Planned and Unplanned Submittals

The test planning progress is affected by the incomplete or incorrect source documentation as submittals [31]. The tracking of planned and unplanned submittals therefore becomes important so that a pattern of excessive or erratic submittals can be achieved.

5.2 Cost

Following attributes are defined under the cost category of metrics:

- Testing cost estimation.
- Duration of testing.
- Resource requirements.

- Training needs of testing group and tool requirement.

5.2.1 Testing Cost Estimation

The test planning phase also has to establish the estimates for budgets. Therefore, metrics supporting testing budget estimation need to be established early. This measurement helps answering the question of how much will it cost to test? Moreover the cost of test planning itself has to be included and measured.

5.2.2 Duration of Testing

Estimating a testing schedule is one of the core activities of test planning. Therefore, metrics that assist in the creation of a testing schedule is required. As part of the testing schedule, the time required to develop a test plan is also to be estimated, i.e. duration of test planning activities, so that test design activity can begin.

5.2.3 Resource Requirements

The test planning activity has to estimate the number of testers required for carrying out the system testing activity.

5.2.4 Training Needs of Testing Group and Tool Requirement

The test planning must identify the training needs for the testing group and tool requirement. Metrics indicating the need of training and tool requirement needs to be established.

5.3 Quality

Following attributes are defined under the quality category of metrics:

- Test coverage.
- Effectiveness of smoke tests.
- The quality of test plan.
- Fulfillment of process goals.

5.3.1 Test Coverage

During test planning, test coverage decisions are to be made. Test coverage helps to answer how much of the requirements, design, code and interfaces is tested? There are different forms of coverage namely, e.g. code coverage, requirements coverage, design coverage and interface coverage [6]. Therefore, test coverage is an attribute that a testing group wants to measure to assess the percentage of code, requirements, design or interface covered by a test set.

5.3.2 Effectiveness of Smoke Tests

Smoke tests establish that stability of application for carrying out testing. Metrics establishing the effectiveness of smoke tests is required to be certain that application is stable enough for testing.

5.3.3 The Quality of Test Plan

The quality of the test plan produced should also be assessed for attributes that are essential for an effective test plan. The outcome of estimating the quality of test plan produced can be used to compare different plans for different products, to predict the effects of change, to assess the effects of new practices and to set the targets for process improvement. Also the test plan is to be inspected for probable number of faults. It is important because using models of expected detection rates, this information can help us to decide whether testing has been effective or not.

5.3.4 Fulfillment of Process Goals

The test planning activity should be able to meet the process goals expected of it. For example, test planning activity may require sign-off from quality assurance section lead before sending it to upper management. Therefore, metrics for meeting process goals is also important.

5.4 Improvement Trends

Following attributes are defined under the improvements trends category of metrics:

- Count of faults prior to testing.
- Expected number of faults.
- Bug classification.

5.4.1 Count of Faults Prior to Testing

The count of faults captured prior to testing during the requirements analysis and design phases of software development identifies training needs for the resources and process improvement opportunities. For example, if a large number of faults are found in requirements phase, it can be concluded that organization needs to implement a requirements review process.

5.4.2 Expected Number of Faults

An estimate of an expected number of faults in the software under test helps gauging the quality of the software.

5.4.3 Bug Classification

The test planning activity also needs to classify bugs into types and severity levels.

The identified attributes and their description appear in Appendix 9. One can see the classification of attributes in to categories in Figure 24.

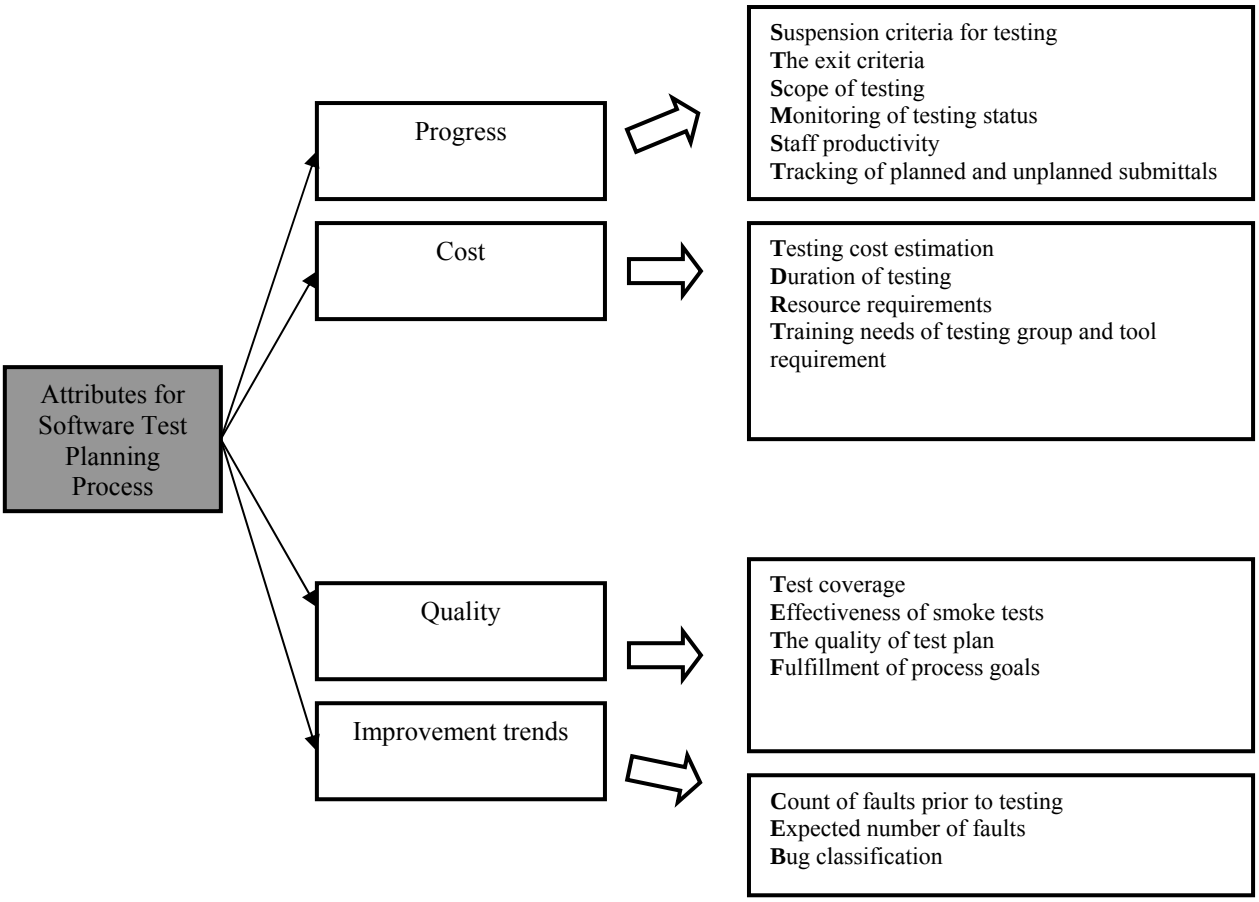

Figure 24. Classification of attributes into categories.

6 ATTRIBUTES FOR SOFTWARE TEST DESIGN PROCESS

After having defined the attributes of the software test planning process, we are interested in identifying the attributes of the software test design process. Identification of attributes defines the basic characteristics that are necessary for developing *cost models, planning aids, and general management principles* [35]. Identification of attributes leads to selecting the measures, which is an important step in establishing a measurement program [35].

The following text identifies the attributes that are useful for measurement during software test design process.

6.1 Progress

Following attributes are defined under the progress category of metrics:

- Tracking testing progress.
- Tracking testing defect backlog.
- Staff productivity.

6.1.1 Tracking Testing Progress

In order to manage the software test design process, it is important to track the progress of testing. Tracking (monitoring) the progress of testing gives early indication if the testing activity is behind schedule and to flag appropriate measures to deal with the situation. Tracking the progress of testing requires metrics that measure number of planned test cases, number of completed/available test cases and number of unplanned test cases [30]. Tracking the progress of testing on multiple releases of a product provides insight into reasons for schedule slippage for a particular release to indicate mitigation actions well in advance. Tracking the progress of testing can also take vital input from growth of the product size over time [34]. By analyzing the pattern of product size on weekly basis, the weekly test progress on those parts of product size can be better explained. The metrics such as number of test cases developed from requirements specification and the number of test cases developed from design helps track progress of test design. Counting the number of test cases is one of the simple counts that a testing analyst tracks [36].

6.1.2 Tracking Testing Defect Backlog

Software faults that are not resolved prior to starting test design activities are likely to affect the testing progress. Specifically, as the accumulated number of faults found during development increases, and are not resolved prior to test design, the test progress will be negatively impacted [34]. Therefore, it is useful to track the testing defect backlog over time. One way of tracking it is to use *defect removal percentage* [9] for prior release so that decisions regarding additional testing can be wisely taken in the current release.

6.1.3 Staff Productivity

Development of test cases marks a significant activity during the software test design phase. As the test cases are developed, it is interesting to measure the productivity of the staff in

developing these test cases. This estimation is useful for the managers to estimate the cost and duration of incorporating change [24].

6.2 Cost

Following attributes are defined under the cost category of metrics:

- Cost effectiveness of automated tool.

6.2.1 Cost Effectiveness of Automated Tool

When a tool is selected to be used for testing, it is beneficial to evaluate the cost effectiveness of tool. The evaluation of the cost effectiveness of the tool measures cost of tool evaluation, cost of tool training, cost of tool acquisition and cost of tool update and maintenance [30].

6.3 Size

Following attributes are defined under the size category of metrics:

- Estimation of test cases.
- Number of regression tests.
- Tests to automate.

6.3.1 Estimation of Test Cases

The activity of developing test cases can progress well if an initial estimate of number of test cases required to fully exercise the application is made. Therefore, metrics estimating the required number of test cases are useful for managing test case development.

6.3.2 Number of Regression Tests

For a new release of software, regression testing is to be performed as part of system testing to ensure that any changes and bug fixes have been correctly implemented and does not negatively affect other parts of the software. Therefore, the number of test cases to be made part of a regression testing suite is an important factor. The measure of the *defects reported in each baseline* and *defect removal percentage* [9] can help make this important decision.

6.3.3 Tests to Automate

As part of test design activity, it is also important to take the decision about which test cases to automate and which are to be executed manually. This decision is important to balance the cost of testing as some tests are more expensive to automate than if executed manually [11].

6.4 Strategy

Following attributes are defined under the strategy category of metrics:

- Sequence of test cases.
- Identification of areas for further testing.

- Combination of test techniques.
- Adequacy of test data.

6.4.1 Sequence of Test Cases

As the test cases are developed as part of the test design process, there must be a rationale for the sequence of test cases [34]. In other words, since not all the test cases are of equal importance, therefore, the test cases need to be prioritized or weighted.

6.4.2 Identification of Areas for Further Testing

Since exhaustive testing is rarely possible, the test design process needs to identify high-risk areas that require more testing [6]. Therefore, metrics identifying the risky areas are candidates of more thorough evaluation. One way to identify areas of additional testing is to use complexity measures [9]. Another probable way is to use *Inspection Summary Reports* [9] which identifies faults in logic, interface, data definition and documentation to identify modules requiring additional testing.

6.4.3 Combination of Test Techniques

Pertaining to the focus of current study on system testing, it is also important to know about what combination of test techniques to use to be effective in finding more faults. Since it is recommended to use a combination of test techniques to ensure high-quality software [11,31], it would be interesting to come up with evidence of a test technique that performs better than other. Development of test cases based on a technique is dependent on the suitability of test cases for specific parts of the system [11]. The selection of testing techniques should ideally help to narrow down a large set of possible input conditions.

6.4.4 Adequacy of Test Data

Test data is defined to provide input to the tests. Incorrect or simple test data increases the chances of missing faults and reducing test coverage [11]. Therefore, it is important to define adequate test data that has the potential of exposing as much faults as possible.

6.5 Quality

Following attributes are defined under the quality category of metrics:

- Effectiveness of test cases.
- Fulfillment of process goals.
- Test completeness.

6.5.1 Effectiveness of Test Cases

It is obvious that an effective test case is able to discover more faults. A test case consists of a set of conditions which tests whether a requirement is fulfilled by the application under test or not and is marked with known inputs and expected outputs. Test effectiveness is the fault-finding ability of the set of test cases [12]. Therefore, it needs to be determined how good a test case is being developed. Measures that establish the quality of test cases produced are therefore required. Related to the effectiveness of test cases, it is also useful to measure the

number of faults in the test cases so as to decide whether the test cases are effective or not [24]. For software in which stability is an important quality attribute, the release-release comparisons of system crashes and hangs can point out the effectiveness of the stress tests and provides indications of product delivery readiness.

6.5.2 Fulfillment of Process Goals

As part of test design process, there are process goals expected to be met. These goals are either prescribed by company standards or are required for certifications. The test design process needs to satisfy these goals so as to establish completion criteria of standards.

6.5.3 Test Completeness

As the test cases are developed, it is imperative that the requirements and code are covered properly i.e. as mentioned in the exit criteria of the test plan [9]. This is to ensure the completeness of tests [31]. Tracking the test coverage as test cases are developed therefore provides a useful measure of the extent to which test cases are covering specified requirements and code. It also helps in estimating the effectiveness of test cases.

The identified attributes and their description appear in Appendix 8. One can see the classification of attributes of software test design process in Figure 25.

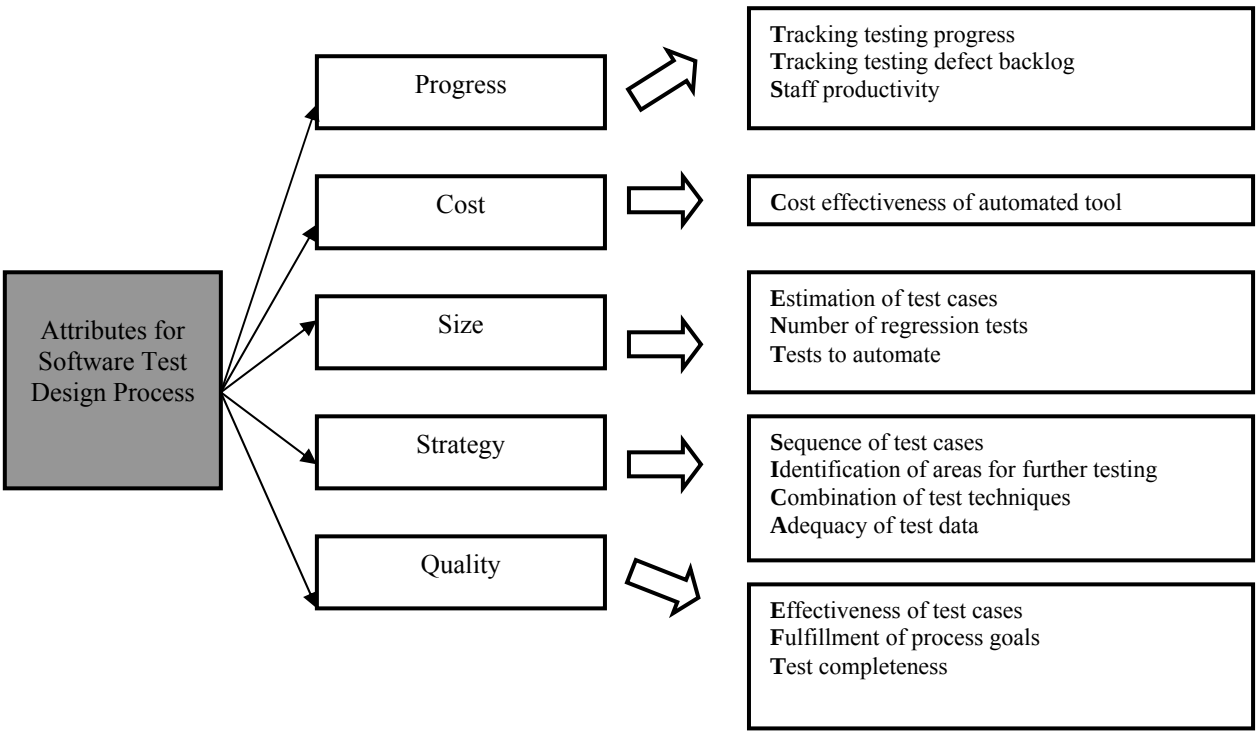

Figure 25. Classification of attributes into categories.

7 METRICS FOR SOFTWARE TEST PLANNING ATTRIBUTES

Earlier we categorized the attributes for software test planning process into four categories of progress, cost, quality and improvement trends. Now we proceed to address available metrics for each attribute within the individual categories.

7.1 Metrics Support for Progress

The attributes falling in the category of progress included (Figure 26):

- Suspension criteria for testing.
- The exit criteria.
- Scope of testing.
- Monitoring of testing status.
- Staff productivity.
- Tracking of planned and unplanned submittals.

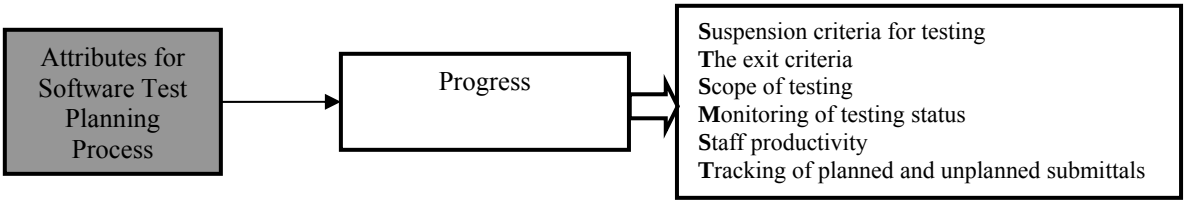

Figure 26. Attributes falling into progress category.

The following text describes the available metric support for each identified attribute within the progress category.

7.1.1 Measuring Suspension Criteria for Testing

The suspension criteria for testing mentions circumstances under which the testing would stop temporarily. The suspension criteria of testing describes in advance that the occurrence of certain events/conditions will cause the testing to suspend. According to [5], suspension criteria are used to *suspend all or a portion of the testing activity on the test items associated with test plan.*

One way to suspend testing is when there are a specific number of severe faults or total faults detected by testing that makes it impossible for the testing to move forward. In such a case, suspending testing acts as a safeguard for precious testing timescales.

There are often dependencies among the testing activities. For example, if there is an activity on a critical path, the subsequent activities needs to be halted until this task is completed [6].

There are certain environmental needs that are required for testing to move forward. These environmental needs include [6]:

- *Hardware (platforms, printers, scanners, modems simulators).*
- *Communications (gateways, connections, authorizations, protocols).*
- *Interfaces (internal, external).*
- *Documentation (requirements, design, user operations).*
- *Software (software under test, operating system, test data, test procedures).*

- *Supplies (forms, papers).*
- *Personal (users, developers, operators, testers).*
- *Facilities (location, space, security).*

Any incompleteness in the above mentioned aspects of test environment may prevent testing of the application. For example, testing progress will be affected if there are resource shortages.

A summary of the discussion on the suspension criteria for testing appears in Table 3.

Table 3. Summary of results for the suspension criteria of testing attribute.

Attribute studied	The suspension criteria for testing
Purpose	To determine the circumstances under which the testing would stop temporarily.
Use	• Saves precious testing time by raising the conditions that suspends testing. • Indicates short comings in the application and environmental setup earlier on. • Acts as a gauge to measure initial stability of the application. • Avoids additional testing work later on and reduces much frustration.
Types of conditions indicating a suspension of testing	• Incomplete tasks on the critical path. • Large volume of bugs. • Critical bugs. • Incomplete test environments (including resource shortages).

7.1.2 Measuring the Exit Criteria

The exit criteria section is an important part of the test plan which indicates the conditions when testing activities moves forward from one level to the next. This practice encourages careful management of the software development process [37]. The exit criteria are established for every level of a test plan. It is important in the sense that if the exit criterion for integration testing is not stringent, such that system testing is started prior to the completion of integration testing, then many of the bugs that should have been found during the integration testing would be found in the system testing where the cost of finding and fixing them is much greater [6]. There is a need to apply limits to the scope of testing because *testing resources are finite, the test budget and number of test engineers are limited and deadlines approach quickly* [11]. The exit criteria determine the termination of the testing effort and ambiguous specification of exit criteria creates confusion among the testing team as when to stop testing. It is recommended that the exit criteria are communicated to the development team prior to the approval of test plan and the exit criteria should be standardized as much as possible [11] that reflects proven criteria in prior projects.

If there are any open faults, then software development manager along with the project manager decides with the members of the change control board whether to fix the faults or defer them to the next release or take the risk of shipping it to the customer with the faults.

There are some metrics that can help clarify the exit criteria [11]:

1. Rate of fault discovery in regression tests: If the rate of fault discovery in regression tests is high, it indicates that fixes made by development team is breaking the previously working functions and introducing new faults. If the frequency of faults detected by

regression tests falls below an acceptable limit, the testing team can decide to halt the testing effort.
2. Frequency of failing fault fixes: If there is a substantial frequency of faults marked fixed by developers that are not actually fixed, then the testing team can re-evaluate the exit criteria.
3. Fault detection rate: If the fault detection rate decreases dramatically as testing proceeds then this declining trend establishes confidence that majority of faults have been discovered.

For system testing, the exit criterion establishes the promotion of testing effort to the acceptance testing level. As an example, the exit criteria for system testing might specify [38 – 40]:

1. All high severity faults identified in system testing are fixed and tested.
2. If there are any open faults that are of medium or low risks, appropriate sign-offs must be taken from project manager or a business analyst or any other authority.
3. The user acceptance tests are ready to be executed.
4. All the test documentation including the test plans and test procedures have been approved.
5. All test scripts and procedures have been executed and open faults have been deferred to next release with appropriate sign-offs.
6. 90% of all test cases must pass.
7. Code coverage must equal at least 80%.
8. All changes made as part of the fixes to faults have been tested.
9. The test documentation has been updated to reflect the changes.

Each and every statement above has its disadvantages. For instance, the exit criterion of 90% of all test cases must pass is dependent very much on the quality of the developed test cases. Similarly, taking appropriate sign-offs places a dependency on the personnel signing off to assess the completeness of testing. Therefore, the above mentioned exit criteria must be used in combination to mitigate the disadvantages.

The summary of discussion on exit criteria for testing appears in Table 4.

Table 4. Summary of results for exit criteria attribute.

Attribute studied	Exit criteria
Purpose	To determine the conditions that indicates completion of testing activity at a particular level.
Use	• Flags the conditions when testing has to stop. • Helps the testing team in prioritizing different testing activities. • Helps transferring control from one level to testing to the next level.
Metrics studied for exit criteria of testing	• Rate of fault discovery in regression tests. • Frequency of failing fault fixes. • Fault detection rate.
Limitations	• The exit criteria for testing are driven by situational circumstances.

7.1.3 Measuring Scope of Testing

The scope of testing is to be decided at the test planning phase and is an important part of the master test plan. The testing scope is an important part of the test plan template recommended by IEEE [5]. Basically, scope of testing helps to answer the question of what to test and what not to test [11]. Informally, the scope of testing helps estimating an overall amount of work involved by documenting which parts of the software are to be tested [12]. *Scope of the tests include what items, features, procedures, functions, objects, clusters and sub-systems will be tested* [21].

The scope of testing is directly related to the nature of the software products. As [21] points out, in case of mission/safety critical software, the breadth of testing requires extensive system tests for functionality, reliability, performance, configuration and stress. This approach will then ultimately affect:

- Number of tests and test procedures required.
- Quantity and complexity of testing tasks.
- Hardware and software needs for testing.

While documenting the scope of testing the following simple questions must be kept in mind:

1. What are main testing types that will be performed for this current release?
2. What are the objectives of each of the test type specified?
3. What basic documents are required for successful completion of testing activities?
4. What is the ultimate goal of the testing effort i.e. what is the expectation out of the testing effort?
5. What are the areas/functionalities/features that will be tested? [41]
6. What are the areas/functionalities/features that will not be tested?
7. When to start a particular type of testing? [42]
8. When to end a particular type of testing? [42]

The factors influencing the decision to include or exclude features in a testing scope is driven by situational circumstances including number of available testing resources, amount of available testing time, areas that are more prone to faults, areas that are changed in the current release and areas that are most frequently used by the customer. A summary of above discussion appears in Table 5.

Table 5. Summary of results for scope of testing attribute.

Attribute studied	Scope of testing
Purpose	To determine what parts of software are to be tested.
Use	• Helps answering what to test and what not to test? • Helps in estimating the overall testing effort required. • Helps in identifying the testing types that are to be used for covering the features under test.
Metrics for scope of testing	Following questions helps outlining testing scope: • What are the main testing types and objectives of each testing type? • What features are to be tested and what features are to be excluded? • When to start and end a particular type of testing?
Limitations	• The scope of testing is driven by situational circumstances.

- The scope of testing is driven by the nature of software under test.

7.1.4 Monitoring of Testing Status

Monitoring of testing status will be discussed in combination with an attribute of software test design process, namely *tracking testing progress*.

7.1.5 Staff Productivity

Staff productivity will be discussed in combination with the same attribute as being identified in the software test design process.

7.1.6 Tracking of Planned and Unplanned Submittals

Data collected during and prior to testing of an application play a vital role in identifying the areas that can be improved in the overall process. Since processes are used to test applications, therefore, indicating process improvements will help companies to remain competitive.

Tracking of planned and unplanned submittals is a simple metric that can identify potential problem areas with in the process. Submittals that are required for testing, falls into two broad categories of source code and relevant documentation. Planned submittals are those that are required to be submitted according to a schedule and unplanned submittals are those that are resubmitted due to the problems in the submitted submittals. If any of these submittals are incomplete or incorrect, it can lead to situations where these submittals are to be resubmitted again so that testing activities can progress.

An example of tracking number of planned and unplanned submittals for a project is given in [31]. In this example the number of planned and unplanned submittals is tracked on monthly basis. It is very easy to monitor the differences between the planned and unplanned submittals, which indicate a potential problem in the process leading towards testing. Tracking for planned and unplanned submittals enables the project managers in motivating reasons for lapses in the testing activity. Moreover, on the basis of excessive or erratic submittals, more time for testing can be sought from management to counter the negative impacts on the testing progress.

A summary of discussion on tracking of planned and unplanned submittals appear in Table 6.

Table 6. Summary of results for tracking of planned and unplanned submittals attribute.

Attribute studied	Tracking of planned and unplanned submittals
Purpose	To assess the impact of incomplete submittals on test planning process.
Use	• To identify potential problem areas within the process leading into testing. • To enable motivating reasons for lapses in the testing activity. • To motivate management for buying more testing time.
Study related to tracking of planned and unplanned submittals	• It is useful to track planned and unplanned submittals using a graph of planned and unplanned submittals over monthly basis [31].

7.2 Metric Support for Cost

The attributes falling into the category of cost included (Figure 27):

- Testing cost estimation.
- Duration of testing.
- Resource requirements i.e. number of testers required.
- Training needs of testing group and tool requirement.

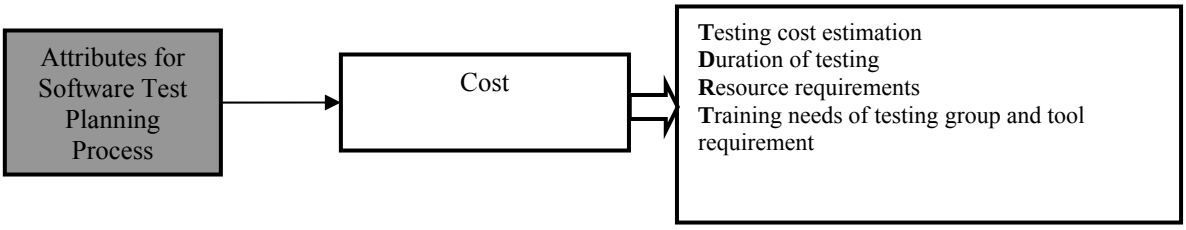

Figure 27. Attributes falling into cost category.

The following text describes the available metric support for each identified attribute within the cost category.

7.2.1 Measuring Testing Cost Estimation, Duration of Testing and Testing Resource Requirements

We will use the term cost estimation here as a common term that includes predictions about the likely amount of effort, time, budget and staffing levels required. Moreover, we make no distinctions between cost estimation and effort estimation as these terms are used interchangeably in the software engineering community [24].

The cost estimates has its use throughout the software development life cycle. They are used when bidding for a contract and determining the feasibility of the project in terms of cost benefit analysis. Cost predictions supports planning and resource allocations, feasibility of whether the project can be done at reasonable cost and whether the key resources will be available at the required time [24].

A testing schedule contains the timelines of all testing milestones. The testing milestones are major testing activities carried out according to the particular level of testing. A work break-down structure is used to break testing efforts into tasks and activities. The timelines of activities in a testing schedule matches the time allocated for testing in the overall project plan. Moreover, the testing schedule plays an important part in scoping the overall software development schedule [16]. The estimates of time and resources in the testing schedule should be accurate enough in order to establish confidence about the delivery date of the software. According to the 1995 Standish Group's Chaos report [47], 89% of projects suffer from over-runs. Moreover, [48] reports that 30-40% of the projects suffers from overruns.

In this regards, it is important to mention how the estimates about the testing milestones are established. These formal estimations are important especially when the testing schedule is bounded by strict time lines. It allows planning for risks and contingencies in time [6, 49]. Ideally, the testing schedule is to be supported by objective estimates. An objective estimate is one which is obtained via a formal process [50]. But the important question is, if it is possible to objectively estimate the testing time? According to [50], *overly optimistic estimates are harming the credibility of the software industry*. According to Ferens and Christensen [51], *in general, model validation showed that the accuracy of the models was*

no better than within 25 percent of actual development cost or schedule, about one half of the time, even after calibration.

Traditionally, more focus had been placed on estimating the schedule for software development rather than for software testing, but since a testing effort is influenced by so many factors it is not sufficient to loosely define software testing time estimates. Some of the factors influencing the testing time estimates are [11]:

- The testing maturity of the organization.
- The complexity of the software application under test.
- The scope of testing.
- The skill level and experience of testers.
- The organization of the testing team.

Estimation formulas that make use of complex equations are not only difficult to use but are also rarely precise [11]. The complexity of these formulas is due to the fact that the time estimation is dependent on variety of factors. Therefore, we concentrate our efforts here to identify simpler estimation models for estimating testing time and resources.

Development ratio method:

As discussed before, traditionally, the emphasis is on the estimation of software development effort for a project. The software development effort estimation can be used as a basis for estimating the resource requirements for the testing effort, thereby helping to outline an estimated testing schedule. The estimation of the number of testing personnel required, based upon the number of developers, gives an easy way to estimate the testing staff requirements. The results of applying this method is dependent on numerous factors including type of software being developed, complexity of software being developed, testing level, scope of testing, test effectiveness during testing, error tolerance level for testing and available budget. Table 7 shows the test team size calculations for functional and performance testing at the integration and the system test levels [11].

Table 7. Test team size calculations based on development ratio method.

Product Type	Number of Developers	Ratio of Developers to Testers	Number of Testers
Commercial Product (Large Market)	30	3:2	20
Commercial Product (Small Market)	30	3:1	10
Development & Heavy COTS Integration for Individual Client	30	4:1	7
Government (Internal) Application Development	30	5:1	6
Corporate (Internal) Application Development	30	4:1	7

Another method of estimating tester to developer ratios is proposed by [52]. The process uses heuristics. It begins with choosing a baseline project(s), collecting data on its ratios of testers to developers and collecting data on the various effects like developer efficiency at removing defects before test, developer efficiency at inserting defects, defects found per person and value of defects found. After that, an initial estimate is made about the number of testers based upon the ratio of the baseline project. The initial estimate is adjusted using professional experience to show the differences of the above mentioned effects between the current project and baseline project.

Project staff ratio method:

Project staff ratio method makes use of historical metrics by calculating the percentage of testing personnel from overall allocated resources planned for the project. Table 8 shows the test team size calculations using the project staff ratio method at integration and system test levels [11].

Table 8. Test team size calculations based on project staff ratio method.

Development Type	Project Staffing Level	Test Team Size Factor	Number of Testers
Commercial Product (Large Market)	50	27%	13
Commercial Product (Small Market)	50	16%	8
Application Development for Individual Client	50	10%	5
Development & Heavy COTS Integration for Individual Client	50	14%	7
Government (Internal) Application Development	50	11%	5
Corporate (Internal) Application Development	50	14%	7

Test procedure method:

The test procedure method uses the size measures of testing artifacts like the planned number of test procedures for estimating the number of person hours of testing and number of testers required. This method is comparatively less effective because it is solely dependent on the number of test procedures planned and executed. The application of the test procedure method requires much before-hand preparation. It includes development of a historical record of projects including data related to size (e.g. number of function points, number of test procedures used) and test effort measured in terms of personnel hours. Based on the historical development size estimates, the number of test procedures required for the new project is estimated. Again, the test team looks at the relationship between the number of test procedures and testing hours expended for the historically similar projects to come up with estimated number of testing hours for the current project. This method is dependent on various factors including that *the projects to be compared should be similar in terms of nature, technology, required expertise and problems solved* (Table 9 shows an example of using the test procedure method [11]). The test team reviews historical records of test efforts on two or more similar projects with 860 test procedures and 5,300 personal hours on average. The historical number of hours spent on planning, designing, executing and reporting the results per test procedure was approximately 6.16. For the new project, the estimates are shown in Table 9.

Table 9. Test team size calculations based on test procedure method.

	Number of Test Procedures	Factor	Number of Person Hours	Performance Period	Number of Testers
Historical Record (Average of Two or More Similar Projects)	860	6.16	5,300	9 months (1,560 hrs)	3.4
New Project Estimate	1,120	6.16	6,900	12 months (2,080 hrs)	3.3

Task planning method:

In this method, the historical records of number of personnel hours expended to perform testing tasks are used to estimate a software required level of effort. The historical records include time recording data related to the work break-down structure so that the records match the different testing tasks (Table 10 shows the use of the task planning method [11]). The historical record indicates that an average project involving 860 test procedures required 5,300 personal hours with a factor of 6.16. This factor is used to estimate the number of personnel hours required to perform the estimated 1,120 test procedures.

Table 10. Test team size calculations based on task planning method.

	Number of Test Procedures	Factor	Test personnel hours
Historical Record (Similar Project)	860	6.16	5,300
New Project Estimate	1,120	6.16	6,900

Table 11 shows how the test team estimates the hours required on various test tasks within the test phases of the new project. The *adjusted estimate* column shows how the revisions are made to the preliminary estimate as conditions change.

Table 11. Time expended in each testing task using task-planning method.

Phase	Historical Value	% of Project	Preliminary Estimate	Adjusted Estimate
Project Startup	140	2.6	179	179
Early Project Support (requirements analysis, etc.)	120	2.2	152	152
Decision to Automate Testing	90	1.7	117	-
Test Tool Selection and Evaluation	160	3	207	-
Test Tool Introduction	260	5	345	345
Test Planning	530	10	690	690
Test Design	540	10	690	690
Test Development	1,980	37	2,553	2,553
Test Execution	870	17	1,173	1,173
Test Management and Support	470	9	621	621
Test Process Improvement	140	2.5	173	-
PROJECT TOTAL	5,300	100%	6,900	6,403

Using the adjusted personnel hour estimate of 6,403 hours, the test team size is calculated to be 3.1 test engineers over the twelve-month project (Table 12) [11].

Table 12. Test team size estimate based on personnel-hour estimate.

	Number of Test Procedures	Personnel-Hour Estimate	Adjusted Estimate	Performance Period	Number of Testers
New Project Estimate	1,120	5.71	6,403	12 months (2,080 hrs)	3.1

[53] categorizes the estimation methodologies into six types:

- Expert opinion.
- Using benchmark data.
- Analogy.

- Proxy points.
- Custom models.
- Algorithmic models.

There are two common approaches to apply any of the estimation methodologies, namely bottom-up and top-down. In bottom-up estimation, the low-level estimates of products or tasks are combined into higher-level ones. On the other hand, top-down estimation begins with a full estimate of overall process or product and follows up with the calculation of component parts as relative portions of the larger whole [24].

Expert opinion:

Expert opinion involves true experts making the effort estimates if size estimates are available to be used at benchmark data. There are two approaches to expert opinion, one is work and activity decomposition and other is system decomposition [53]. In work and activity decomposition, subjective personal opinion rather than quantitative data and history is used. The estimator thinks about *the work to be done, the skill of the people who will do it to create an outline of the required activities, durations and number of people required to do those activities per time period* [53]. In system decomposition method, the system is decomposed into lower level modules or components and total effort is calculated by combining the effort associated with each module. In case there are conflicts among the estimations from experts, [53] recommends using the Delphi Methods, more specifically the Wideband Delphi Method, which relies upon frequent interaction to arrive at a consensus.

Using benchmark data:

This approach is based on the premise that productivity is a function of size and application domain. If an estimate of the size of the test procedures is available and delivery rate can be predicted, then effort estimate is simply the ratio of size and delivery rate. The size of test procedures and the delivery rate can be estimated using historical records of data.

Analogy:

Estimation by analogy makes use of analogs (completed projects that are similar) and use of experience and data from those to estimate the new project. This method can both be tool based or manual.

Proxy points:

Proxy point estimation methods use the external characteristics of the system to determine its size. The primary proxy point methods in use are function points, object points (used with COCOMO II) and use case points. Proxy point estimation models use a three step process for estimating effort. First the size is estimated using any proxy point metric, then the estimates are adjusted for complexity factors and finally estimates are adjusted by a productivity factor like function point per staff month.

Custom models:

The use of custom models involves changing or adjusting the standard estimation models to better match the current project environment.

Algorithmic models:

The algorithmic models are empirical models that are derived using regression analysis [53]. Most of the algorithmic models first estimates size, then uses size estimates to derive the

effort, which in turn is used to estimate cost and schedule. Algorithmic models form the basis for both manual models and tool based models.

Using Halstead metrics for estimating testing effort:

The metrics derived from Halstead measures can be used to estimate testing effort. The measures proposed by Halstead are [8]:

n1= the number of distinct operators that appear in the program.
n2= the number of distinct operands that appear in the program.
N1= the total number of operator occurrences.
N2= the total number of operand occurrences.

Halstead developed expressions for program volume V and program level PL which can be used for testing effort estimation. The program volume V describes the number of volume of information in bits required to specify a program. The program level PL is a measure of software complexity. *Using these definitions for program volume V and program level PL, Halstead effort e can be computed as* [8]

$PL = 1/ [(n_1/2) * (N_2/n_2)]$
$e = V/PL$

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

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 [8].

Budget estimates to follow a documented procedure:

The most significant cost in most projects is human resources. Once the duration of testing is established and it is known how much time each tester will spend on the project, then the budget of the testing resources can be determined by calculating the rate of the resources [43]. It is important to document the assumptions that are made while estimating the cost to improve risk assessment and improvement in estimates for future projects.

Limitations of the approaches used for estimation:

1. The projections from the past testing effort cannot be relied upon entirely. The testing professionals are required to use their experience to deal with unusual circumstances.
2. The testing maturity of the organization must be considered while estimating.
3. The scope of the testing requirements must be clear.
4. The introduction of an automated tool introduces complexity into the project and must be considered appropriately.
5. The domain knowledge of tester is an important attribute that affects the time lines.
6. The test team organizational styles must be taken into consideration.
7. The time at which test planning activity begins matters because early start of the test planning activity enables better understanding of requirements and early detection of errors.
8. The presence or absence of documented processes for testing makes up another important factor for scheduling testing time.
9. Highly risky software requires more detailed testing and must be taken into account while estimating testing schedule.

10. There is a likelihood of change in requirements and design during development [24].
11. The software should be delivered on time for testing [54].

A summary of measurements for the duration of testing and testing resource requirements appear in Table 13.

Table 13. Summary of results for testing cost estimation, duration of testing and testing resource requirements attribute.

Attribute studied	Testing cost estimation, duration of testing and testing resource requirements
Purpose	To estimate the cost of testing required for carrying out the testing activity.
Use	• Helps in finalizing the overall software development schedule. • Prediction of testing timelines helps in tracking the progress of testing activities. • Prediction of required testing resources and testing time helps identifying the required effort associated with the overall testing activity.
Techniques studied	• Development ratio method. • Project staff ratio method. • Test procedure method. • Task planning method. • Expert opinion. • Using benchmark data. • Analogy. • Proxy points. • Custom models. • Algorithmic models. • Using Halstead metrics.
Limitations	• The estimations are not to be relied upon entirely, must be complemented with experience and personal judgment. • The estimations are dependent on the maturity of the testing processes. • The scope of the test requirements must be clear for more realistic estimations. • Complexities are associated with the introduction of an automated testing tool. • The estimations are affected by the level of domain knowledge of the testers.

7.2.2 Measuring Training Needs of Testing Group and Tool Requirement

Training needs of testing group and tool requirement will be discussed in combination with the *tests to automate* attribute identified in the software test design process.

7.3 Metric Support for Quality

The attributes falling in the category of quality included (Figure 28):

- Test coverage.
- Effectiveness of smoke tests.
- The quality of test plan.
- Fulfillment of process goals.

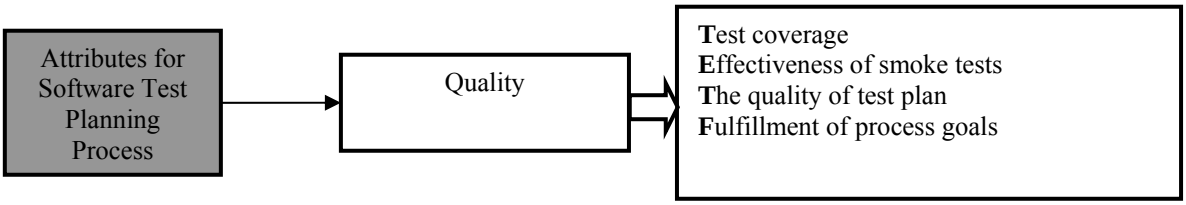

Figure 28. Attributes falling into quality category.

The following text describes the available metric support for each identified attribute within the quality category.

7.3.1 Measuring Test Coverage

Test coverage will be discussed in combination with the *test completeness* attribute as identified in the software test design process.

7.3.2 Measuring Effectiveness of Smoke Tests

Smoke tests form a group of test cases that establishes the stability of system i.e. assurance that the system has all the functionality and is working under normal conditions [6]. The basic purpose of smoke tests is not to identify faults, but to let the system testing team establish confidence over the stability of the system to start testing the system. The basic purpose of smoke testing is the automated testing of the critical-high level functionality of the application [11]. There are some established good practices for smoke tests:

- Smoke tests are created to be broad in scope as opposed to depth. Therefore, smoke tests targets the high level functionality of the application.
- Smoke tests should exercise the system from end to end, capable of exposing major problems [67].
- 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.
- Smoke tests need to be a subset of the regression testing suite, (it is time consuming and human-intensive to run the smoke tests manually each time a new build is received for testing). Automated smoke tests are executed against the software each time the build is received ensuring that the major functionality is working.
- The smoke tests increases in size and coverage as the system evolves [67].
- Smoke tests should be the very first tests that are to be automated because they will be running most of the times. If the smoke tests are automated, it can lead to an optimization of both the testing and the development environments [11]. Software is ready to be used after each build in a known working state after performing automated smoke tests.
- Smoke tests are to be run in the system test environment.

- Successful execution of smoke tests forms the entrance criteria for the system testing, after the software passes unit and integration testing.
- Other examples of smoke tests include [11]:
  - *Verifying that the database points to the correct environment.*
  - *The database is the correct version.*
  - *Sessions can be launched.*
  - *Data can be entered, selected and edited.*
- Smoke tests can also benefit from a historical database of fault-finding subset of test suites that have proved valuable over other projects.

Smoke tests are executed earlier on in the testing lifecycle and deciding which tests to execute first, depend on the quality of the software, resources available, existing test documentation and the results of risk analysis [6].

Smoke tests form an important part of a typical software build sequence. The typical software build sequence is as following [11]:

1. Software build compilation.
2. Unit tests.
3. Smoke tests.
4. Regression tests.

According to [68], there are four types of release models, namely tinderbox, nightly, weekly and milestone. In the tinderbox release model, developer builds, compiles and releases continuously for immediate feedback. In nightly release model, a release engineer compiles and builds the code which is good for validating blocking defects. Weekly release models are ideal for more comprehensive testing and milestone releases are made after code freeze. With the tinderbox release model, the smoke tests are run on developer machines on every change. In nightly releases, automated smoke tests are run on testing machines which reports status every morning. In weekly and milestone release models, smoke tests are performed by testers to certify reasonable quality.

A summary of discussion on effectiveness of smoke tests appear in Table 14.

Table 14. Summary of effectiveness of smoke tests attribute.

Attribute studied	Effectiveness of smoke tests
Purpose	To establish the stability of the application for carrying out testing.
Use	• Establishes the stability of the system and builds. • Establishes the entrance criteria for system testing.
Best practices	• Smoke tests are broad in scope. • Smoke tests exercise the system from end to end. • Smoke tests cover the basic operations used frequently. • Smoke tests are to be automated and made part of regression testing. • Smoke tests are to be developed in collaboration with developers. • Code reviews help identify critical areas for smoke testing. • A clean test environment should be used for smoke tests.

7.3.3 Measuring the Quality of Test Plan

There are number of formats and standards for documenting a test plan but they provide very few differences in terms of which format is better than the other. A test plan can be evaluated only in terms of fulfilling the expectations of it or *with regards to those functions that it intends to serve* [69]. The quality of test plan is also dependent on situational factors and judgments. According to the test plan evaluation model presented by [69], a test plan needs to possess the quality attributes as described in Table 15.

Table 15. Description of test plan quality attributes.

Test plan quality attribute	Description [69]
Usefulness	Will the test plan serve its intended functions?
Accuracy	Is it accurate with respect to any statements of fact?
Efficiency	Does it make efficient use of available resources?
Adaptability	Will it tolerate reasonable change and unpredictability in the project?
Clarity	Is the test plan self-consistent and sufficiently unambiguous?
Usability	Is the test plan document concise, maintainable, and helpfully organized?
Compliance	Does it meet externally imposed requirements?
Foundation	Is it the product of an effective test planning process?
Feasibility	Is it within the capability of the organization that must perform it?

In order to evaluate the quality attributes for test plan, [69] suggests heuristics i.e. accepted rules of thumb reflecting experience and study. These heuristics are given in Appendix 10.

Berger describes a multi-dimensional qualitative method of evaluating test plans using rubrics [70]. Rubrics take the form of a table, where each row represents a particular dimension, and each column represents a ranking of the dimension. According to [70], there are ten dimensions that contribute to the philosophy of test planning. These ten dimensions are as following:

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 Communication.
8. Theory of Usefulness.
9. Theory of Completeness.
10. Theory of Insightfulness.

Against each of these theories, there is a rating of excellent, average and poor depending upon certain criteria. Appendix 1 [70] contains the evaluation of the test plan against each of the theories.

Table 16 summarizes the discussion on quality of a test plan attribute.

Table 16. Summary of quality of test plan attribute.

Attribute studied	Quality of test plan
Purpose	To assess the extent to which a test plan meets its expectations.
Use	• Measurement of test plan's quality highlights the areas that need improvement to better plan the testing effort.
Studies related to quality of test plan	• Nine quality attributes that describe the quality of a test plan including usefulness, accuracy, efficiency, adaptability, clarity, usability, compliance, foundation and feasibility are discussed [69]. • A method of rubric to evaluate a test plan using ten dimensions of objective, scope, coverage, risk, data, originality, communication, usefulness, completeness and insightfulness is discussed [70].
Limitations	• A quantitative measurement for evaluating a test plan based on one dimension e.g. length is not useful. • Multiple characteristics must be studied to evaluate a test plan. • Instead of numeric counts, observation and experience are used to rate the attributes of a test plan [70].

7.3.4 Measuring Fulfillment of Process Goals

The software test planning activity is expected to fulfill goals that are expected of it at the start. Test planning makes a fundamental maturity goal at the Testing Maturity Model (TMM) level 2. The goals at TMM level 2 support testing as a managed process. A managed process is planned, monitored, directed, staffed and organized [21]. The planning component of the managed process is taken care of at TMM level 2. The setting of goals is important because they become the basis for decision making. The goals may be business-related, technical or political in nature. The goals that are expected of a process also differ in levels. At the top level, there are general organizational goals; at the intermediate level, there are functional unit level goals; and then there are project level and personal level goals. The test plan includes both general testing goals and lower level goal statements. An organization can use a checklist to evaluate the process of test planning. In order to check the fulfillment of test planning goals, some points are given in Appendix 11 while a checklist for the test planning process is given in Appendix 2 as a reference [71].

7.4 Metric Support for Improvement Trends

The attributes falling in the category of improvement trends included (Figure 29):

- Count of faults prior to testing.
- Expected number of faults.
- Bug classification.

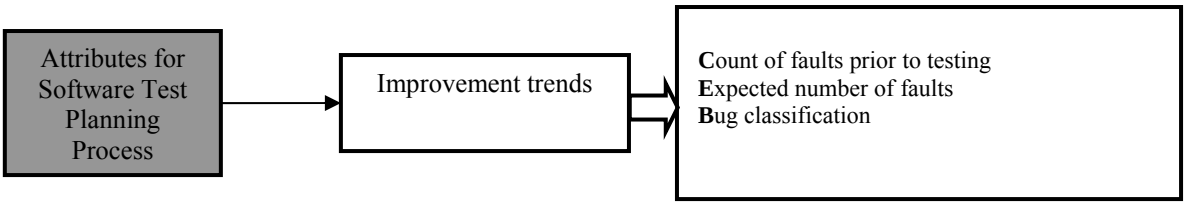

Figure 29. Attributes falling into improvement trends category.

The following text describes the available metric support for each identified attribute within the improvement trends category.

7.4.1 Count of Faults Prior to Testing and Expected Number of Faults

Count of faults prior to testing and expected number of faults will be discussed in combination with *identification of areas for further testing* attribute as being identified in the software test design process.

7.4.2 Bug Classification

There are different ways to classify the faults found in the application during the course of system testing. One such classification is given by [21]. According to this classification, the faults found in the application code can be classified as belonging to following categories as shown in Table 17.

Table 17. Classification of faults.

Classification of faults	Fault types [21]
Algorithmic and processing	1. *Unchecked overflow and underflow conditions.* 2. *Comparing inappropriate data types.* 3. *Converting one data type to another.* 4. *Incorrect ordering of arithmetic operators.* 5. *Misuse or omission of parenthesis.* 6. *Precision loss.* 7. *Incorrect use of signs.*
Control, logic and sequence	1. *Incorrect expression of case statements.* 2. *Incorrect iterations of loops.* 3. *Missing paths.*
Typographical	1. *Syntax errors i.e. incorrect spelling of a variable name.*
Initialization	1. *Incorrect initialization statements.*
Data flow	1. *Faults in the operational sequences of data flows.*
Data	1. *Incorrect implementation of data structures.* 2. *Incorrect setting of flags, indices and constants.*
Module interface	1. *Incorrect or inconsistent parameter types.* 2. *Incorrect number of parameters.* 3. *Improper ordering of parameters.* 4. *Incorrect sequence of calls.* 5. *Calls to non-existent modules.*
Code documentation	1. *Incomplete, unclear, incorrect and out-of-date code documentation.*
External hardware, software interfaces defects	1. *Problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling, data exchanging with hardware, protocols, formats, interfaces with build files, and timing sequences.*

According to [79], software faults can be classified as permanent design faults which are deterministic in nature and are identified easily. The other type of software faults is classified as being temporary internal faults that are transient and are difficult to be detected through testing. [80] extends the classification of software faults as proposed by [79] by including another classification of faults called aging related faults. According to this view [80], as the software is in operational use, potential fault conditions gradually accumulate over time leading to performance degradation and/or transient failures.

Apart from the classification of faults, there are severity levels associated with faults. A five level error classification method, as given by [81], is outlined in Table 18.

Table 18. Severity levels.

Severity level	Description
Catastrophic	Critical loss of data, critical loss of system availability, critical loss of security, critical loss of safety.
Severe	Defects causing serious consequences for system.
Major	A defect that needs to be fixed and there is a work around.
Minor	Defects having negligible consequences.
No effect	Trivial defects.

Some organizations follow the following severity levels:

Level 0

This level contains all bugs that come under:
- System Crash.
- Unrecoverable data loss.
- No work around.
- Loss of Main functionality.
- Loss of Data.

Level 1

This level contains all bugs that come under:
- Failure of Rule.
- Field validation.
- User friendliness.
- Interface appearance e.g., spelling mistakes and widget alignment.

Level 2

This level contains all bugs that come under:
- Interface appearance e.g. proper scrolling etc.
- Punctuation, grammatical mistakes.

Level 3

- Something, which is part of or not part of the functionality committed, but should or should not be incorporated in the system to enhance user friendliness or look and feel.

A summary of classification of faults and the severity levels appear in Table 19.

Table 19. Summary of results for bug classification attribute.

Attribute studied	Bug classification
Purpose	To categorize the faults found during system testing and to assign severity levels.
Use	• Categorizing the faults according to types and severity levels helps prioritizing the fixing of faults. • Indicates the urgency of fixing serious faults that are causing a major system function to fail. • Helps prioritizing the testing of scenarios for the testing team. • Helps estimating the quality of the application under test.
Bug classification and severity levels studies	• Classification of defects as proposed by [21], [79], [80]. • Severity levels as proposed by [81] and used commonly in industry.
Limitations	• The bug classifications and the severity levels differs from one organization to other, depending upon different factors e.g. level of testing and type of application under test. • The bug classifications and severity levels must be used consistently throughout the company within different projects so as to provide a consistent baseline for comparing projects.

8 METRICS FOR SOFTWARE TEST DESIGN ATTRIBUTES

Earlier we categorized the attributes for software test design process into five categories of progress, quality, size, cost and strategy. Now we proceed to address available metrics for each attribute within the individual categories.

8.1 Metric Support for Progress

The attributes falling in the category of progress included (Figure 30):

- Tracking testing progress.
- Tracking testing defect backlog.
- Staff productivity.

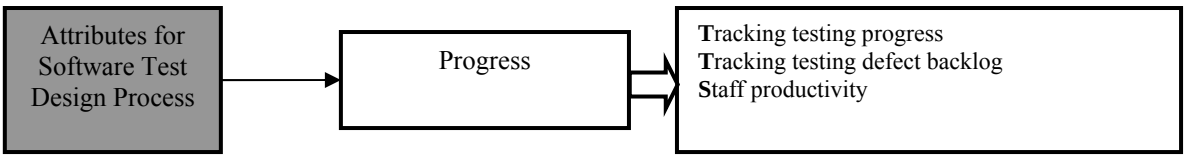

Figure 30. Attributes falling into progress category.

The following text describes the available metric support for each identified attribute within the progress category.

8.1.1 Tracking Testing Progress

Tracking testing progress attribute is discussed here in combination with the *monitoring of testing status* attribute of the test planning process.

The testing status needs to be monitored and controlled with respect to a plan for every project. Controlling and monitoring of testing occurs at the Testing Maturity Model (TMM) level 3. Controlling and monitoring are engineering management activities that are expected to be performed by professional software engineers. Monitoring compares the actual work done to the work that was planned and controlling develops and applies actions that get the project on track incase of deviations [21]. Monitoring and controlling actions complement each other in the sense that the deviations from planning are indicated by monitoring and controlling implements the corrective actions to counteract the deviations. Controlling in a sense works to achieve the project goals.

The testing milestones are planned in the schedule of the test plan. Each milestone is scheduled for completion during a certain time period in the test plan. These testing milestones can be used by the test manager to monitor the progress of the testing efforts. For example, execution of all planned system tests is one example of a testing milestone. This milestone is to be achieved within the time defined in the test schedule. Measurements need to be available for comparing the planned and actual progress towards achieving testing milestones. For the testing milestone of execution of all planned system tests, the data related to number of planned system tests currently available and the number of executed system tests at this date should be available. The testing activity being monitored can be projected using graphs that show trends over a selected period of time.

Monitoring the testing status helps a test manager to answer the following questions [21]:

- *Which tasks are on time?*
- *Which have been completed earlier then scheduled, and by how much?*
- *Which are behind schedule, and by how much?*
- *Have the scheduled milestones for this date been met?*
- *Which milestones are behind schedule, and by how much?*

For monitoring the coverage at the system testing level, the following measures are useful [21]:

- *Number of requirements or features to be tested.*
- *Number of equivalence classes identified.*
- *Number of equivalence classes actually covered.*
- *Number of degree of requirements or features actually covered.*
- *Number of features actually covered/total number of features.*

For monitoring the test case development at the system testing level, the following measures are useful [21]:

- *Number of planned test cases.*
- *Number of available test cases.*
- *Number of unplanned test cases.*

The test progress S curve compares the test progress with the plan to indicate corrective actions incase the testing activity is falling behind schedule. The advantage of this curve is that schedule slippage is difficult to be ignored. The test progress S curve shows the following set of information on one graph [34]:

- Planned number of test cases to be completed successfully by week.
- Number of test cases attempted by week.
- Number of test cases completed successfully by week.

The S-curve can also be used for release-release or project-project comparisons and one such graph is shown in [34].

A testing status report is a primary formal communication channel for informing the organization about the progress made by the testing team. A testing status report takes the form as shown in Table 20 [6].

Table 20. An example test status report.

Project Name					
Feature tested	Total tests	Number of tests complete	Percentage of tests completed	Number of successful test cases	Percentage of successful test cases

According to [43], the cost of achieving a milestone is tracked at the end or in the middle of the milestone to predict whether a cost overrun is predicted or not. If the project expenditures exceed what was initially planned, appropriate measures can be taken e.g. demanding more funds from management or notifying the customer in time of the budget overrun. The budget is to be tracked with the help of periodic status meetings in which each member describes the progress of assigned work therefore it is easier for the test manager to predict whether the activity will miss the deadline thus resulting in extra incurred cost.

Incase where the activities are completed ahead to time, the funding can be directed to areas where there are delays. The following Table 21 can be used for tracking testing budget [43].

Table 21. Tracking testing budget

Testing activity	Planned	Actual	Deviation	Deviation %

Table 22 summarizes the measurement of monitoring of testing status and tracking testing progress.

Table 22. Measurement of monitoring of testing status and tracking testing status attribute.

Attribute studied	Monitoring of testing status and tracking testing status
Purpose	To monitor and control the testing status.
Use	• To track the cost and schedule of testing. • To take corrective actions incase of schedule and cost slippage is expected.
Studies related to monitoring and controlling of testing status	• [21] mentions metrics for test case development and coverage. • [34] mentions the use of test progress S- curve for tracking test progress and for release-release comparisons. • [43] mentions the importance of periodic status meetings for tracking testing schedule and cost.
Limitations	• The tracking of progress is dependent on the milestones described in the test plan. • The development of test progress S curve is dependent on timely availability of data related to planned, attempted and successful test cases. • The generation of test status report is dependent on the metrics provided by individual testing personnel so there can be an element of distortion of data.

8.1.2 Tracking Testing Defect Backlog

Defect backlog is the number of defects that are outstanding and unresolved at one point in time with respect to the number of defects arriving. A large number of unresolved defects before testing is started will negatively affect the testing progress. Moreover, if the development team is fixing the defect backlog at the same time the testing is being carried out, then there is a chance that some defects will be reported more than once. Also the new defects reported by testing team will not be fixed in time as the development team is busy fixing the defect backlog. Therefore, defect backlog reduction should not be the focus when testing is the critical activity. The testing should be allowed to focus on detecting as many faults as possible and the development team should try to solve the critical defects that hinder the testing team in extensively testing the application. Once the testing is near to completion, the focus can shift to manage and reduce the testing defect backlog.

The defect backlog contains reported defects that are being deferred to subsequent releases as enhancements. Then there are defects that are not fixed due to resource constraints. The existing design of the system might not allow for the fixes to be made without much modification. These old defects are accumulated over time and must be a concern for the testing team.

The defect backlog metric is to be measured for each release of a software product for release to release comparisons [34]. Just reducing the number of defects in the backlog is as important as prioritizing which defects are to be fixed first. In this regards, the expertise of the testing and development departments play a part by resolving those defects that help in

achieving early system stability. A graph providing a snapshot of defect status graph is given in [82].

While tracking the defect backlog over time, [34] recommends an approach for controlling the defect backlog. The testing progress shall be monitored and tracked to compare the actual completion of planned test cases and procedures from the beginning. The defect arrivals are to be monitored and analyzed for error-prone areas in the software for improvement actions. The defect backlog shall be reduced with achievement of pre-determined targets in mind and requires strong management. The defects that affect and halt the testing progress should be given priority in fixing. Another way to prioritize the reduction in defect backlog is on the basis of impact on customer which can help the developers to focus efforts on critical problems [83].

A summary of discussion on tracking testing defect backlog overtime appear in Table 23.

Table 23. Tracking testing defect backlog overtime attribute.

Attribute studied	Tracking testing defect backlog overtime
Purpose	To assess the number of defects that remains unresolved in comparison to the number of incoming defects.
Use	• To assess the affect of defect backlog on testing progress. • To plan for reducing the defect backlog.
Measuring tracking testing defect backlog overtime	• Defect backlog reduction should not be the focus when testing is the critical activity. • The defect backlog metric is to be measured for each release of a software product for release to release comparisons • Prioritization is required while reducing the defect backlog.

8.1.3 Staff Productivity

The staff productivity attribute here is discussed in combination with the *staff productivity* attribute of test planning process.

Measurement of testing staff productivity helps to improve the contributions made by the testing professionals towards a quality testing process. It is difficult to know how to improve personal productivity levels; therefore there is a strong temptation to treat productivity as the production of a set of components over a certain period of time [24]. This might be true for other manufacturing and production industries that productivity increases with addition of more people; but as Brooks observed that adding more people on a software project will make it even more late. According to [44], this law does not hold as strongly in testing as it does in most areas of software engineering. If system testing is well-designed, which focuses on behavioral complexities rather than internal ones, then a new test engineer is able to contribute within a couple of weeks of joining the team.

Productivity is the comparison of level of outputs gained with a specific level of inputs. In terms of software engineering, productivity is the ratio of size of the output to the amount of input effort i.e. [24]

Productivity= Size/Effort

The above equation hides the difficulties inherent in measuring size and effort. The effort made by a software engineer differs as the software engineer is more productive on one day than other. Similarly, the effort has to count on the contributions from the support staff. The size measure is also hiding the true value of productivity; as productivity should be measured in the light of benefit that is delivered rather than size. As [24] writes, *it does no*

good to be very effective at building useless products. One other problem with the above productivity equation is that it treats software development productivity like manufacturing industry productivity. The mistake here is that software development is a creative activity involving lots of variation. It's not like producing cars through an assembly line.

Therefore, we need a measure of productivity that has more direct relationship to quality or utility [24]. So the equation of productivity should not be used as the only measure of personal productivity.

Despite the short comings, the productivity equation, being on a ratio scale, provides simplicity and ease of calculation for performing different statistical analysis. According to [21], the productivity measures for testers are not extensively reported. Nevertheless, a test manger is interested in learning about the productivity of its staff and how it changes as project progresses. Proper planning, design, execution and results gathering are important factors contributing to the success of a testing effort. Even if there is a perfect testing process in place and testing strategy is ideal, but if the test team productivity levels are declining then it impacts the finding of defect; most probably the faults will be discovered too late in testing.

The basic and useful measures of tester's productivity are [21]:

- Time spent in test planning.
- Time spent in test case design.
- Number of test cases developed.
- Number of test cases developed/unit time.

It is important to classify the activities that constitute a test planning and test design phase. Then the other consideration is the types of artifacts that are produced as part of the test planning and test design phase. The time taken for each artifact must be related with quality of artifact to arrive at true productivity measures.

Evaluation of tester's productivity is a difficult and often a subjective task [11]. The productivity of a tester should be evaluated in the light of specified roles and responsibilities, tasks, schedules and standards. The testers are expected to observe standards and procedures, keep schedules, meet goals and perform assigned tasks, and meet budgets [11]. Once these expectations are set, the work of the testing team should be compared against the established goals, tasks and schedules. There are some points that must be kept in mind while assessing the productivity:

- The tester might be a domain expert or a technical expert specializing in the use of automated tools.
- Testers might be new or experienced to the testing profession.
- The testing level determines the different activities to be performed at that particular level, so the testing level consideration is important for evaluating productivity at that level.
- The type of testing under measurement should be considered. For example, Table 24 outlines the set of questions that must be asked while evaluating functional test procedures [11].

Table 24. Evaluation of functional test procedures [11].

Questions for evaluating the functional test procedures
How completely are the test-procedure steps mapped to the requirements steps? Is traceability complete?
Are the test input, steps, and output (expected result) correct?
Are major testing steps omitted in the functional flow of the test procedure?
Has an analytical thought process been applied to produce effective test scenarios?

Have the test-procedure creation standards been followed?
How many revisions have been required as a result of misunderstanding or miscommunication before the test procedures could be considered effective and complete?
Have effective testing techniques been used to derive the appropriate set of test cases?

[45] outlines some ideas to assess the quality of testers:

- There are areas within which some persons are better than others. There is a need to identify the talents within the testing team and use them to the advantage of the project. For example, if one member creates a better test plan and other one is better in designing test cases, then it is often unproductive to force either of them to move from their respective areas of interest.
- It is very difficult to compare the expected results to measure the productivity of testers because of various variables involved. Therefore, observing the activities of testers is more vital than comparing the expected results.
- While testers are trying to self-improve, there are some activities that are compromised and the tester manager must be aware of them.
- The productivity of tester varies with technology.
- Capabilities are not necessarily obvious and visible.

According to [46], evaluation of individual testers is multidimensional, qualitative, multi-source, based on multiple samples and individually tailored. The primary sources of information in that case is not numeric, rather specific artifacts are reviewed, specific performances are assessed and the work of testers is discussed with others to come up with an evaluation.

A summary of discussion on staff productivity appears in Table 25.

Table 25. Summary of studies for staff productivity attribute.

Attribute studied	Staff productivity
Purpose	To estimate the effectiveness and performance levels of testing personnel.
Use	• Helps to improve the contributions made by the testing professionals towards a quality testing process.
Staff productivity measures studied	• Measures for the productivity of testing staff personnel are not widely reported and are often subjective in nature. Few of the simple measures in place in industry are time spent in test planning, time spent in test case design, number of test cases developed, number of test cases developed/unit time.
Limitations	• The productivity of a tester should be evaluated in the light of specified roles and responsibilities, tasks, schedules and standards. • The work of the testing team should be compared against the established goals, tasks and schedules. • The expertise, experience level, testing level, type of testing performed and technology used in the application under test are important factors that must be kept in mind when assessing productivity.

8.2 Metric Support for Quality

The attributes falling in the category of quality included (Figure 31):

- Effectiveness of test cases.
- Fulfillment of process goals.
- Test completeness.

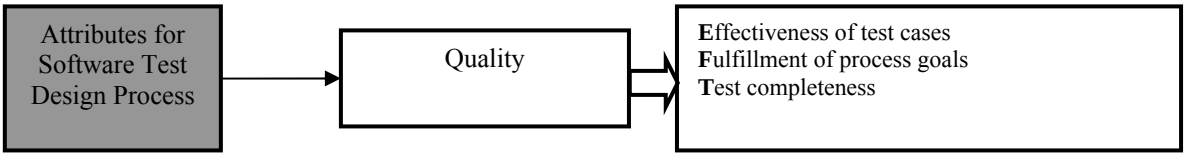

Figure 31. Attributes falling into quality category.

The following text describes the available metric support for each identified attribute within the quality category.

8.2.1 Measuring Effectiveness of Test Cases

The mere execution of test cases is not the proof that systems are adequately tested. It is of little advantage to evaluate the effectiveness of test cases when the system is in production; there needs to be an in-process evaluation of test-case effectiveness [84] so that problems are identified and corrected in the testing process before system goes in to production.

The test case specifications must be verified at the end of test design phase for conformance with requirements. While verifying the test case specifications, there are factors that affect effectiveness of test cases including designing test cases with incomplete functional specifications, poor test design and wrong interpretation of test specifications by testers [84].

Common methods available for verifying test case specifications include inspections and traceability of test case specifications to functional specifications. These methods seldom help improving the fault-detecting ability of test case specifications. Therefore, test case specifications are validated by determining how effective have been the tests in execution.

A simple in-process test case effectiveness metric is proposed by [84], which defines the test case effectiveness metric as the ratio of faults found by test cases to the total number of faults reported during a function testing cycle.

Test case effectiveness = (Faults found by test cases / Total number of faults reported)*100%

For each project, there is a baseline value which can be used to compare the value given by test case effectiveness. If the test case effectiveness is below the baseline value, it is an indication that the testing process needs improvement. The test case effectiveness can be improved by improving the overall testing process. The improvement approach given by [84] is based on the causal analysis of failures that are caught by the customers in production and that were not identified by the testing group. This approach consists of five steps, beginning with the need to understand and document the test process. The test process typically consists of test planning, test design, test preparation and execution and test evaluation and improvement [84], therefore the tasks in each of these phases shall be planned and defined. The second step involves developing an understanding of the factors that affect test case effectiveness. The common affecting factors includes incorrect and incomplete functional

specifications, incomplete test design, incorrect test specifications, incomplete test suite, incomplete and incorrect test case execution (Figure 32).

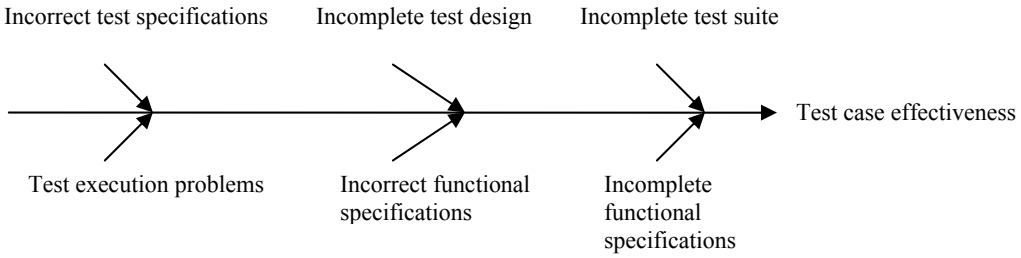

Figure 32. Factors affecting test-case effectiveness [84]

The third step of the improvement approach involves gathering of faults that were missed by testing (using a defect tracking system) and performing a causal analysis on these faults to identify the reasons causing the testing team to miss these faults. The causes can be attributed to one or more of the factors as identified in Figure 32. The fourth step identifies the main factors that are responsible for majority of faults missed by the test cases. One can use a bar chart to isolate the most important factors. The last step involves taking corrective actions to repeat the testing cycle to assess improvement in test case effectiveness. The corrective actions can take the form of inspecting functional specifications and then re-working and updating test case specifications, use of traceability matrix to ensure coverage of business rules, training sessions for testers on test case design techniques, inspection and re-work of test case specifications [84].

The test case effectiveness is related to the effectiveness of the overall testing process. There are certain metrics for assessing the effectiveness of tests. These metrics can be categorized as customer satisfaction measures, measures based on faults and coverage [6].

Customer satisfaction measures are used to determine the satisfaction level of customers after using the software. Conducting surveys is one of the common methods of measuring the customer satisfaction. There are drawbacks associated with conducting surveys for assessing test effectiveness because the customer would not differentiate between effectiveness of test from the overall quality of the software so it is difficult for the test managers to interpret the results of the survey.

There are several measures based on faults. One of them is number of faults found in testing. A problem with fault count is that it is important to analyze the severity of found defects. Also the number of found defects is dependent on number of faults that already existed, i.e. the quality of the software. Also number of found bugs might be lower because there were actually fewer faults in the software. Anther measure is the number of failures observed by the customer which can be used as a reflection of the effectiveness of the test cases. Defect removal efficiency is another powerful metric for test effectiveness, which is defined as the ratio of number of faults actually found in testing and the number of faults that could have been found in testing. There are potential issues that must be taken in to account while measuring the defect removal efficiency. For example, the severity of bugs and an estimate of time by which the customers would have discovered most of the failures are to be established. This metric is more helpful to establish test effectiveness in the longer run as compared to the current project. 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. Defect spoilage is calculated as following [6]:

Spoilage = Sum of (Number of Defects x defect age) / Total number of defects

Spoilage is more suitable for measuring the long term trend of test effectiveness. Generally, low values of defect spoilage mean more effective defect discovery processes [6].

The effectiveness of the test cases can also be judged on the basis of the coverage provided. It is a powerful metric in the sense that it is not dependent on the quality of the software and also it is an in-process metric that is applicable when the testing is actually done. Requirements and design based coverage can be measured using a matrix which maps test cases to requirements and design. Code coverage can be measured using a tool, but execution of all the code by the test cases is not the guarantee that the code works according to the requirements and design.

Some important guidelines for deriving effective test cases from requirements are given by [11].

- The details and complexities of the application must be understood and analyzed.
- The behavior of the system, flow, dependencies among the requirements must be understood.
- It must be analyzed that how a change in one part of the application affects rest of the application.
- Test procedures should not be designed from requirements specifications alone.
- The test procedures should rarely overlap as it is not effective for two testers to test the same functionality. To achieve this, the test procedures can be developed in modular fashion to reuse and combine them.
- The expected result of one test case should not invalidate the results of the other test case.
- The preconditions necessary for executing a test procedure must be determined by analyzing the sequence in which specific transactions must be tested.
- Testing of high priority requirements must be done early in the development schedule.
- The test procedures created should be prioritized based on highest-risk and highest-usage functionality.

A summary of discussion on effectiveness of test cases appear in Table 26.

Table 26. Summary of discussion on effectiveness of test cases attribute.

Attribute studied	Effectiveness of test cases
Purpose	To determine the fault finding ability (quality) of the test case.
Use	• To reduce the number of failures in production by maximizing the number of faults detected in testing. • To indicate improvement trends in the software testing process to improve effectiveness of test cases.
Metrics for test case effectiveness	• Test Case Effectiveness Metric [84]. • Customer satisfaction measure ▪ Surveys • Fault-based measures ▪ Number of failures observed. ▪ Number of faults found in testing. ▪ Defect removal efficiency. ▪ Defect age. ▪ Defect spoilage. • Coverage-based measures ▪ Requirements, design and code coverage.
Limitations or metrics for effectiveness of test cases	• The Test Case Effectiveness Metric proposed by [84] is dependent on the availability of sufficient number of failures detected by the user as it relies entirely on the defects missed by test cases. • In customer satisfaction measures like surveys,

customers will not differentiate between quality of application and quality of testing performed.
- Fault-based measures are not in-process measures so the existing project is not able to take significant benefit from it.
- Coverage based measures do not ensure that the code covered is meeting the customer requirements.

8.2.2 Measuring Fulfillment of Process Goals

The software test design process should be able to perform the activities that are expected of it. The activities comprising a test design process were outlined previously. The common way of assessing the conformance to a process is using a check list. One such checklist is given in Appendix 3 as a reference [71].

8.2.3 Measuring Test Completeness

The test completeness attribute is discussed in combination with the *test coverage* attribute for software test planning process.

While measuring the test coverage of an application, we are basically interested in measuring how much of the code and requirements are covered by the test set. We study the code coverage measures and the specification coverage measures to analyze what measures exist to establish confidence that the tests are covering both the code and requirements. We do not emphasize on other coverage measures as we believe that code coverage and specification coverage measures are more prevalent in literature and provides enough evidence of the coverage potential of a test set. Moreover, we emphasize more on specification coverage measures as these measures are more applicable at the system testing level which forms the focal point of this study.

Test coverage incase of white box testing is called code coverage which is the measure of the extent to which program source code is tested. Formally, a test coverage criterion defines *a set of entities of the program and requires that every entity in this set is covered under some test case* [56]. The basic code coverage measures are statement coverage, decision coverage, condition coverage and path coverage [57]. [58] lists down 101 coverage measures which is still not an exhaustive list of measures.

The test coverage forms differ with respect to the level of testing. For example at the function testing level, a test coverage decision of 100% statement coverage and 95% branch coverage may be mentioned in the function test plan; although this coverage is found to be not generally achievable [59]. An analysis conducted on the survey of testing in IBM revealed that 70% statement coverage is sufficient while increasing beyond a range of 70%-80% is not cost effective [59].

The advantages of measuring test coverage are that it provides the ability to design new test cases and improve existing ones to increase test coverage. Also it helps in avoiding over estimation of test coverage by testers.

There are two important questions that need to be answered when determining the sufficiency of test coverage [59]. One is that whether the test cases cover all possible output states and second is that about the adequate number of test cases to achieve test coverage.

The relationship between code coverage and number of test cases is described by the following expression [59]:

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

Where C(x) is the coverage after executing x number of test cases, N is the number of blocks in the program and p is the average number of blocks covered by a test case during

function test. The function indicates that test cases cover some blocks of code more than others while increasing test coverage beyond a threshold is not cost effective.

Test coverage for control flow and data flow can be measured using spanning sets of entities in a program flow graph. A spanning set is a minimum subset of the set of entities of the program flow graph such that a test suite covering the entities in this subset is guaranteed to cover every entity in the set [56].

At system testing level, we are more interested in knowing whether all product's features are being tested or not. A common requirements coverage metric is the percentage of requirements covered by at least one test [9]. A requirements traceability matrix maintains a mapping between tests, requirements, design and code. A requirements traceability matrix helps estimating and identifying tests that needs to change as requirements change. Tracing requirements from development to testing establishes the confidence that test cases are covering the developed functional requirements [60]. Table 27 exemplifies a requirements traceability matrix [61].

Table 27. An example requirements traceability matrix.

Requirement	Function specification	Design specification	Source code files	Test cases

In black box testing, as we do not know the internals of the actual implementation, the test coverage metrics tries to cover the specification [62]. Three specification coverage metrics namely state, transition and decision are proposed by [63]. The specification coverage measures are recommended for safety critical domains [64]. Specification coverage measures the extent to which the test cases conform to the specification requirements. These metrics provide objective and implementation-independent measures of how the black box test suite covers requirements [64]. The test cases incase of specification coverage are traceable to high-level requirements. Currently, the coverage of test cases for requirements is assessed indirectly on an executable artifact like source code or software models [64]. This poses some short comings because it is an indirect measure and secondly lack of coverage can be attributed to multiple factors. According to [64], it is possible to derive specification based coverage metrics if there exists a formal specification expressed as LTL (Linear Temporal Logic) properties. Also three potential metrics that could be used to assess requirements coverage are proposed namely requirements coverage, requirements antecedent coverage and requirements UFC (Unique First Cause Coverage). The rigorous exploration of metrics for requirements-based testing is in its initial phases [64]. Another approach to measure testing coverage independent of source code is given by [65, 66]. This approach applied model checking and mutation analysis for measuring test coverage using mutation metric. This metric involves taking a set of externally developed test cases, turning each test case into a constrained finite state machine and then scoring the set against the metric [65].

A summary of the discussion on test coverage and test completeness appear in Table 28.

Table 28. Summary of results for test coverage and test completeness attribute.

Attribute studied	Test coverage and test completeness
Purpose	To determine the percentage of code and specification covered by a test set.
Use	• Establishes the sufficiency of set of test cases. • Design of new test cases to increase test coverage. • Improvement of existing test cases to improve test coverage.

Types of test coverage metrics studied	• Code coverage measures (White-box testing).
	  ▪ Control flow measures.
	  ▪ Data flow measures.
	• Specification coverage measures (Black-box testing)
	• Requirements traceability matrix.

	• Metrics based on Linear Temporal Logic (LTL) based specification i.e. requirements coverage, requirements antecedent coverage and requirements UFC. • Mutation metric using model checking and mutation analysis.
Limitations	• Test coverage tools are difficult to use because code is to be run under the control of debugger instead of a separate testing environment, increase in the execution time of the tested program due to code measurement, lack of specific language support by coverage tools and extra effort required for measurement [59]. • Scalability of specification coverage measures is a concern that requires more theoretical and experimental investigation.

8.3 Metric Support for Cost

The attributes falling in the category of cost included (Figure 33):

- Cost effectiveness of automated tool.

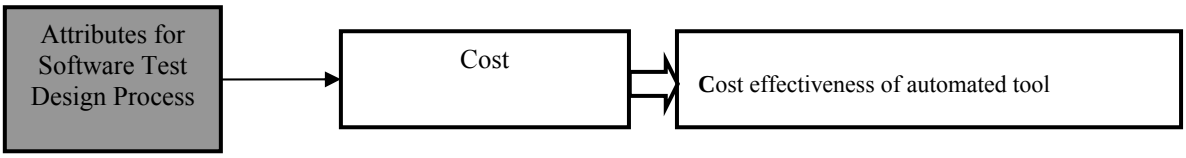

Figure 33. Attributes falling into cost category.

The following text describes the available metric support for identified attribute within the cost category.

8.3.1 Measuring Cost Effectiveness of Automated Tool

In order to avoid costly changes to test automation later on, it is important to assess the organizational needs before investing large amounts into automated testing tools. The organizational needs include the verification that selected tool works properly with selected technologies, fits in the organizational budget and time frame for introducing the tool [11]. There is a need to execute the evaluation criteria before a decision to acquire the tool is made. The evaluation criteria evaluate the benefits and drawbacks of the tool. Before evaluating a tool, it is important to know about the different types of automated testing tools. The different types of the automated testing tools are listed in Appendix 4 [11].

The evaluation criteria for a testing tool must consider its compatibility with operating systems, databases and programming languages used organization wide. The performance requirements for significant applications under test in the organization must be reviewed. In a given environment, it is difficult to find out a single tool compatible with all operating systems and programming languages. Therefore, the need of more than one tool must be established. The way application transforms the data and displays it to the user must be understood as the automated tool is to support data verification. The types of tests to be automated must be established. The schedule must be assessed so that there is enough time in the project for the testing team to get used to the new tool. A rush to introduce automation

in a project will cause opposition for automation. The cost of the testing tool is to be assessed along with the cost of training the testing team in using the automated tool, cost of automated script development and maintenance costs.

According to [21], the evaluation criteria for evaluating a testing tool includes ease of use, power, robustness, functionality, ease of insertion, quality of support, cost of the tool and fit with organizational policies and goals. The first six of these criterions are given by Firth and colleagues in their technical paper, *A Guide to the Classification and Assessment of Software Engineering Tools*. Each of these criterions is satisfied with answering several questions associated with it. The important questions to ask from each criterion are listed in Appendix 5.

It should be noted that introduction of automated testing will not result in an immediate decrease in test effort and test schedule.

Sometimes customized automated tools and in-house developed test scripts need to be developed because of lack of compatible tools in the market and high complexity of the task. There are several issues that must be considered before making a decision to build a tool. The resources, budgets and schedules for tool building must be determined. Approval is to be taken from management for this effort. The development of the automated tool is to be considered as part of software development so that serious efforts are put in to development.

In order to verify that the automated tool meets the organization's needs and to verify the vendor's claims about the compatibility of the tool with various technologies, it is best to test the tool on an application prototype [11]. The prototype should consist of features that are representative of the technologies to be used in the organization. A common problem for the automated testing tools is its inability to recognize third party controls. These types of incompatibility issues are identified earlier if the project team evaluates the tool against the project needs right from the beginning.

[21] argues that the real benefits of investing in the testing tools can be realized if the testers are properly trained and educated, organizational culture is supportive in tool evaluation and its usage, the tools are introduced incrementally and in accordance with the process maturity level of the testers. In one of his papers, R. M. Poston and M. P. Sexton emphasize the need of developing tool evaluation forms [85]. These forms are based on standard set of evaluation criteria and each criterion is weighted. These forms not only select the tools but also analyze user needs.

A summary of discussion on cost effectiveness of automated tool appears in Table 29.

Table 29. A summary of discussion on cost effectiveness of automated tool.

Attribute studied	Cost effectiveness of automated tool
Purpose	To evaluate the benefits of automated tool as compared to the costs associated with it.
Use	• To avoid later changes to test automation. • To select the tool that best fits the organizational needs.
Evaluation of the automated testing tools	The evaluation criteria for the automated tools must take care of the following: • Compatibility with operating systems, databases and programming languages used organization wide. • Assessment of performance requirements for significant applications under test. • Need for multiple tools. • Support for data verification. • Cost of training. • Types of tests to automate. • Ease of use. • Power. • Robustness. • Functionality.

- Ease of insertion.
- Quality of support.
- Cost of the tool.
- Fit with organizational policies and goals.

8.4 Metric Support for Size

The attributes falling in the category of size included (Figure 34):

- Estimation of test cases.
- Number of regression tests.
- Tests to automate.

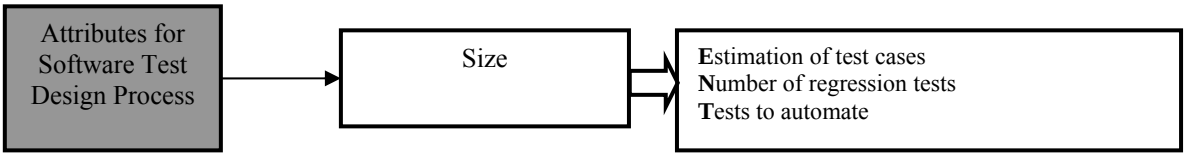

Figure 34. Attributes falling into size category.

The following text describes the available metric support for each identified attribute within the size category.

8.4.1 Estimation of Test Cases

The estimation of test cases attribute will be discussed along with the *combination of testing techniques* attribute of test design process.

8.4.2 Number of Regression Tests

Regression testing is performed on a modified program that establishes the confidence that changes and fixes against reported faults are correct and have not affected unchanged portions of the program [86]. Running all the test cases in a test suite is one of the regression testing techniques (re-test all) but it can require a large amount of effort [87]. Regression testing that is not planned and done manually makes inefficient use of resources [11]. Therefore, regression test selection techniques select some subset of the existing test suite to retest a modified program. The two major concerns for regression test selection techniques are the time required to select and run tests and the fault detection ability of the tests [88]. [86] presents several of regression test selection techniques, which are given in Appendix 6. There are different empirical studies that establish the relevance of selecting a regression testing technique ([88, 89, 90])

According to [9], there are some guidelines to follow when selecting test cases for regression testing. These guidelines uses the program complexity measures (e.g. cyclomatic complexity and Halstead's metrics) to indicate the modules whose complexity falls outside the complexity baseline set in the company. These modules are difficult to test [90], therefore are candidates for regression testing. The other guidelines recommend making use of judgment and knowledge of software design and past history to select tests for regression testing.

Moreover, the measure of the percentage of the faults that have been removed as compared to the known number of faults (defect removal percentage) helps making decision

about regression testing by selecting test cases that exercises the fixed part of the code and any code dependent on the fixed part.

The measure of the defects reported in each baseline [9] also offers clues for additional regression testing as modules that are identified as more faulty in each baseline stand candidates for more regression testing.

According to [11], regression test cases should include test cases that exercise high-risk functionality (e.g. the functionality that is affected by fault fixing) and most-often executed paths. After these areas are tested then more detailed functionality should be the target of regression testing. Moreover some practices are recommended by [11] for making the regression test suite optimized and improved. These practices are listed below:

- Include the test cases in to the regression testing suite that identified faults in prior releases of the software.
- Using number and type of faults identified by the test cases to continuously improve the regression testing suite.
- A regression impact matrix can be developed that indicates adequate coverage of the affected areas and is assisted by the information provided by the developers about areas affected by the addition of new features and fixing of faults.
- Regression tests should get updated for each iteration as parts of the application change to provide depth and coverage.
- Any changes to the requirements must be followed with modification to the regression testing suite to account for application's new functionality.
- *Regression test cases should be traceable to the original requirement* [11].
- There is a strong motivation to automate regression tests executed in a stable test environment.
- An analysis of the test results identifies fault prone software components to focus regression testing.

Regression testing is the practice that comes under the Testing Maturity Model (TMM) Level 3. [21] provides the following measurements to monitor regression testing:

- *Number of test cases reused.*
- *Number of test case 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.*

A summary of discussion on number of regression tests appear in Table 30.

Table 30. Summary of results for number of regression tests attribute.

Attribute studied	Number of regression tests
Purpose	To find out number of regression tests to be made part of the regression testing suite.
Use	• To create a regression testing suite that reveals faults resulting from changes to the code due to fault fixing. • To create a regression testing suite that executes in limited amount of time with the ability to identify maximum faults.
Regression test selection techniques	• [86] presents several techniques for regression test selection including: ▪ Linear equation techniques. ▪ The symbolic execution technique. ▪ The path analysis technique. ▪ Dataflow techniques. ▪ Program dependence graph techniques. ▪ System dependence graph techniques.

- The modification based technique.
- The firewall technique.
- The cluster identification technique.
- Slicing techniques.
- Graph walk techniques.
- The modified entity technique.

Important guidelines and metrics on number of regression tests	• Use of program complexity measures. • Use of judgment and knowledge of software design and past history of selected regression tests. • Use of defect removal percentage. • Measure the number of faults reported in each baseline. • Number of test cases reused. • Number of test case 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. • Include tests that exercise high-risk functionality. • Include tests that exercises most often executed paths. • Continuously improve the regression testing suite.

8.4.3 Tests to Automate

The tests to automate attribute is discussed in combination with t*raining needs of testing group and tool requirements* attribute for test planning process.

Training needs:

The type and distribution of faults in prior projects within the company or within prior releases of a project or within the process can identify training needs. For example, during the inspection of testing artifacts, if it is found that there is a greater proportion of incorrectly formed test cases, then this gives an indication for a need of training in test case design. The training needs are documented under the section 'staffing and training needs' of the IEEE template for software test plan [5].

Estimation by Dorothy Graham, a noted author in the field of test inspection and test certification, revealed that by the end of 1990's only 10% of testers and developers ever had any training in test techniques [12].

Establishing a technical training program is a goal at level 3 of the Testing Maturity Model (TMM) [21]. According to TMM, the training program includes in-house courses and training sessions, college/university degree program and courses and courses taken at an external commercial training center. These training programs are essential for staying in touch with new testing techniques, methods and tools.

It is the responsibility of the test managers to provide adequate levels of trainings to the sub-ordinates who lack in experience and exposure of testing applications. The test managers must assume leadership in teaching and ensure that the sub-ordinates are given appropriate levels of training according to a suitable technique. [6] mentions an example that it is written in the Fleet Marine Force Manual 1 (FMFM1) that *the commanders should see the development of their sub-ordinates as a direct reflection on themselves*.

According to [6], there is a strong relationship between increased training and improved worker productivity, profitability and shareholder value. The training needs for a project varies according to the scope of the project. The areas in which the testing group requires training involves [6]:

1. Software testing.
2. Tools.

3. Technical skills.
4. Business knowledge.
5. Communication skills.

Training in software testing can take different forms. Formal training seminars, conferences on testing and certification programs are few examples. Then there are websites and books that are dedicated to the field of software testing. The test mangers should allow enough time and resources for sub ordinates to avail these opportunities. According to Roger Pressman [6], a software engineering staff should take at least one to three weeks of methods training each year.

Use of tools requires special training for the testing staff members. The form of training can both be formal and informal training. The formal training to be conducted by the vendor of the tool and the informal ones can be conducted in-house by more experienced professionals. The training on tools is to be carefully planned because *the amount of time and training required to implement and use tools successfully is frequently underestimated* [6].

The testing group needs to be technically proficient as well to be able to better communicate with the developers, to do better functional testing and to be able to develop automated testing scripts. Therefore, the testing group needs to be part of company wide training programs.

It is very important for the testing staff to understand the business needs that are supported by the application under test. The test managers should arrange for their staff to be trained in business functions [6]. This can happen through early involvement of the testing resources during domain gathering and requirements elicitation processes. The testing staff should be given time and motivation for reading and collecting material about the business domain of the application under test.

It is also imperative for the testers to have sound communication and writing skills as they have to interact with multiple departments and have to write testing artifacts that need to be unambiguous. The testing group can enhance their communication and writing abilities by participating as speakers at a conference and by writing papers [6].

There are different methods of training that can be applied by the test managers in effectively training the sub-ordinates. It includes [6]:

1. Mentoring.
2. On-site commercial training.
3. Training in a public forum.
4. In-house training.
5. Specialty training.

Mentoring is an effective training method in which an experienced resource (mentor) takes the responsibility of training a new or less experienced resource about the methods, tools and processes in place within the organization. The resources undergoing training have chance to learn from the experience of the mentors while the mentors feel comfortable as being recognized as experts within their field.

On-site commercial training makes use of the training expertise of another company to train the resources. The advantages of this training is that, in most of the cases, this type of training is formal; the whole testing group receives the same level of training and the training can be customized to the unique needs to the testing group.

Training in a public forum is a good idea when the testing team is too small for a professional instructor to be hired. In that case, one or few of the testing resources can participate in a public training class. An advantage of this approach is that the testing resources get to interact with different people from different backgrounds and shares each other's ideas and experiences. A disadvantage of public trainings is that the training cannot be customized to the specific needs to few individual trainees.

In-house trainings take place when the testing group is too small or there is no time to hire a professional instructor. In this case, the company spends time and cost in preparing the training materials. The test resources have to take time out of their work to attend those training classes. A positive impact of in-house trainings is that it is great learning experience for the person within the testing group who is giving the training.

Specialty training programs involve web-enabled training, virtual training conferences and distance learning [6]. These training modes are often less effective as compared to face-to-face trainings.

As part of process improvement, a training checklist can be maintained in the company, as shown in Table 31[6].

Table 31. Checklist for training.

Pass	Fail	Checklist for Training
□	□	Are training needs identified according to a procedure?
□	□	Is training conducted for all personnel performing work related to quality?
□	□	Are personnel who are performing specific tasks qualified on the basis of appropriate education, training, and/or experience?
□	□	Are records kept of personnel training and experience?

Tool requirement:

Testing tools are certainly not a remedy of all problems related to software testing. But there are many occasion when appropriate tools makes the testing team more effective and productive. The testing strategy should consider making use of automated tool wherever the needs warrants its use. As Rick Craig says *testing tools are not the answer to your problems, they are just one more tool in your bag of testing tricks* [6]. Automated testing refers to the integration of testing tools in the test environment in such a way that test execution, recording and comparison of results are done with minimal human effort [6].

The automated testing comes with its set of specific requirements. Expertise is required to gain confidence over the tool's scripting language. If the automated scripts are taking longer time to create than creating manual tests, then it must be examined that how much time is saved in the execution of the automated scripts. Then if the automated script is to be repeated multiple times, it can be estimated whether it is favorable to automate the script or not [6].

Tasks that are repetitive in nature and tedious to be performed manually are prime candidates for an automated tool. The category of tests that come under repetitive tasks are as following:

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

Smoke tests are to be performed every time a build is delivered to the testing group, therefore it is logical to automate them to gain time advantages. Smoke tests are recommended to be made part of the subset of the regression test set. Performance and load tests are executed most of the times with the help of an automated tool.

The categories of tests that come under the category of tedious tasks are as following [6]:

- Code coverage.
- Mathematical calculations.

- Simulations.
- Human-intensive tasks.

The above mentioned tasks are not possible to be carried out manually on a large scale. The candidate conditions for automated testing are shown in Figure 35[6].

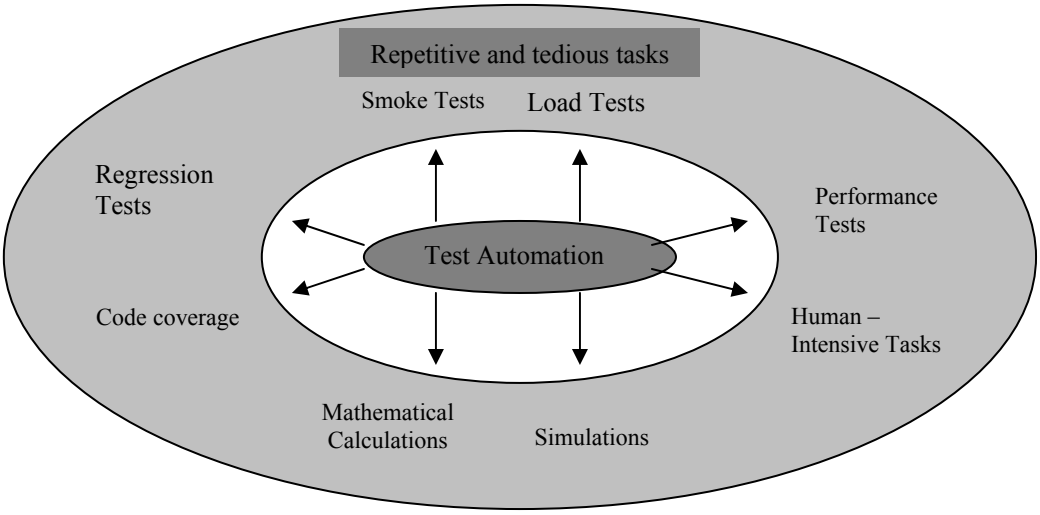

Figure 35. Candidate conditions for automated testing.

There are situations when it is not beneficial to automated tests. The tests that are to be executed only once are not good candidates for automation. If the requirements are changing then regression tests are unstable and hence automating them will not pay off the effort that is put in to automate them.

Once a tool is configured to be used for testing, it must be made sure that the testers are adequately trained to use the tool. The testers should actually know how to use new tools i.e. how to select and setup test cases and determine the results [6].

There is no single best automated testing tool available. The selection of an automated tool depends on whether the tool is to be selected for a particular type of project or for organizational wide testing purpose. If the tool is to be used organization wide, different aspects become important like application development methodologies in use, types of operating systems and types of applications. The data transformation by the application should be well-understood so that the automated tool supports data verification [11]. Moreover, the impact of automation on project schedule should be determined.

There are benefits attached with automation including speed, efficiency, accuracy and precision, resource reduction and repetitiveness [55], but there are important issues to consider before automating tests:

- The software requirements change frequently in today's environment. It must be evaluated to automate only stable tests; otherwise the requirement changes will cause costly rework to change scripts.
- Automation is not a substitute for human intuition [55].
- If the automation runs without finding a fault, it does not mean that there is no fault remaining.

A summary of the discussion on training needs and the tool requirements & tests to automate appears in Table 32.

Table 32. Summary of training needs and tool requirements & tests to automate attribute.

Attribute studied	Training needs and tool requirements & tests to automate
Purpose	To identify the training needs for the testing group, tool requirement and tests to automate.
Use	• Training needs in the areas of software testing, tools, technical skills, business knowledge and communication skills helps software testing professionals in fulfilling their duties efficiently. • The careful evaluation of requirements for the use of automated tools helps using the advantages of automation for the project.
Training needs and tool requirements	• The training needs for software testers lies in the areas of software testing, tools, technical skills, business knowledge and communication skills. • The training methods involve mentoring, on-site commercial training, training in a public forum, in-house training and specialty training. • Maintenance of an in-house training checklist is also important as part of process improvement. • Automation is suited for repetitive (regression tests, smoke tests, load tests, performance tests) and tedious tasks (code coverage, mathematical calculations, simulations, human-intensive tasks).
Limitations	• Test automation to be applied after careful evaluation of tasks that are repetitive and tedious in nature. • There is no single best automation tool available. • There are situations when automating tests does not pay off.

8.5 Metric Support for Strategy

The attributes falling in the category of strategy included (Figure 36):

- Sequence of test cases.
- Identification of areas for further testing.
- Combination of test techniques.
- Adequacy of test data.

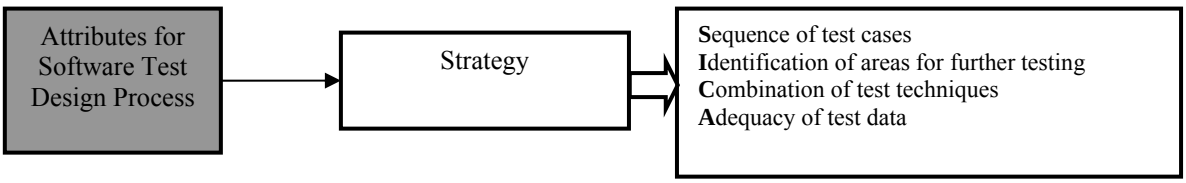

Figure 36. Attributes falling into strategy category.

The following text describes the available metric support for each identified attribute within the strategy category.

8.5.1 Sequence of Test Cases

Test case prioritization techniques enable the testers to prioritize the test cases according to some criteria so that the test cases having high priority are executed earlier. The criteria can include the following:

- To increase the rate of fault detection.
- To increase the detection of high-risk faults.
- To increase the coverage of code under test.
- To increase the reliability of the system under test at a higher rate.

The real benefit of test case prioritization is achieved when the time required to execute an entire test suite is sufficiently long, as the testing goals are met earlier [87]. Test case prioritization ensures that the testing time is spent more beneficially.

There are two broad types of test case prioritization techniques, general and version specific [87]. In general test case prioritization techniques, the ordering of the test cases is useful over a sequence of subsequent modified versions of a given program. Version specific test case prioritization techniques are applied after a set of changes have been made to a given program and is less effective on average over a succession of subsequent releases.

Test case prioritization is formally defined as a function which is applied to all the possible prioritizations of a testing suite such that it yields a higher award value such that higher award values are preferred over lower award values. There are several considerations in selecting a test case prioritization technique mentioned by [92]. One consideration is the levels of granularity i.e. test cases that are executed at either function level or statement level. The granularity of test cases affects the costs in terms of computation and storage. A second consideration is that whether or not the test case prioritization technique incorporates feedback to adjust for the test cases already executed. A third consideration is that whether the techniques make use of modified program version or are based entirely on the coverage information which do not consider the specific modifications in the modified version of a program. Finally, an important consideration is the assessment of practicality of test prioritization techniques. It is claimed that [92] techniques based on coverage at statement and function level are applicable provided code coverage tools are available. But the techniques based on fault exposing potential need further demonstration of practicality.

[92] classifies eighteen test case prioritization techniques in to three categories. The categorization of these prioritization techniques is shown in Appendix 7.

Controlled experiments and case study was conducted by [92] to evaluate the test case prioritization techniques. All the test case prioritization techniques contributed to improve the rate of fault detection. In terms of granularity of techniques, fine-granularity techniques performed better than the coarse-granularity techniques but not significantly. Moreover, the inclusion of measures for fault-proneness techniques did not showed large improvements in terms of rate of fault detection.

A study carried out by [92] measured the effectiveness of several of test case prioritization techniques in terms of the ability to improve the rate of fault-detection. The techniques chosen were random, optimal, total branch coverage, additional branch coverage, total statement coverage, additional statement coverage, total fault exposing potential and additional fault exposing potential prioritization. The results of the study showed that the use of test case prioritization techniques improved the rate of fault detection.

A summary of discussion on sequence of test cases appear in Table 33.

Table 33. Summary of discussion on sequence of test cases attribute.

Attribute studied	Sequence of test cases
Purpose	To prioritize the execution of test cases.
Use	• To enable the execution of those test cases that meet criteria for prioritization.
Studies related to test case prioritization	• There can be eighteen test case prioritization techniques divided into three categories namely comparator, statement level and function level techniques [92]. • Experiments and case studies shows that use of test case prioritization techniques improves the rate of fault detection [92, 93].

8.5.2 Measuring Identification of Areas for Further Testing

The identification of areas for further testing attribute will be discussed in combination with the *count of faults prior to testing* and *number of expected faults* attribute of software test planning process.

Counting of faults prior to testing helps indicate the areas or files likely to be beneficial to test. There has been evidence that a small number of modules contain most of the faults discovered prior to releasing the product [72]. This notion is captured by what is commonly known as the Pareto Principle or 20-80 rule [72].

During the code development phase of software development, if it can be predicted that which files or modules are likely to have largest concentrations of faults in the next release of a system [73], the effectiveness and efficiency of testing activities can be improved. The improvement in the effectiveness and efficiency is due to the fact that testers can target their efforts on those files/modules by testing them first and with greater emphasis; resulting in quick identification of faults and providing extra time for the testers to test rest of the system [74]. One way to predict the number of faults in files is using a fault-prediction model like a negative regression model. This model predicts the faults for each file of a release based on characteristics like *file size, whether the file was new to the release or changed or unchanged from the previous release, the age of the file, the number of faults in the previous release and the programming language* [73]. The results of application of the model on two large industrial systems were found to be quite accurate.

Khoshgoftaar et al. [74] used software design metrics and reuse information (i.e. whether a module is changed from previous release) to predict the actual number of faults in the module. They applied the model to a single release of a large telecommunication system with 1.3 million lines of code and 25000 files. The use of software design metrics and reuse information contributed significant fault predictive power.

Graves et al. [75] reported a study in which the fault predictions in a module were based on module's age, the changes made to the module and the ages of the module. The report does not discuss the effectiveness of the predictions but found that module size and design metrics are poor predictors of fault likelihood. This is in contrast with the study carried out by [76] in which it was stated that a high correlation exists between complexity measures and program quality discriminating between programs that are likely to contain faults and those that will contain few faults.

[77] predicts the fault proneness of software modules based entirely on the pre-coding characteristics of the system design. The result of the study showed that top twenty percent of modules identified by the predictors contained forty three percent to forty seven percentage of the faults in the implemented code.

Another approach to predict fault prone modules is using random forests [78]. *Random forests are an extension of decision tree learning* [78]. The general characteristics of this approach are that it is general, efficient for large data sets and more robust to noise. The

prediction accuracy of the proposed approach was found to be statistically significant over other methods.

A summary of the discussion on the count of faults prior to testing/expected number of faults and identification of areas for further testing appear in Table 34.

Table 34. Measuring the count of faults prior to testing/ expected number of faults/ identification of areas for further testing attribute.

Attribute studied	The count of faults prior to testing / expected number of faults / identification of areas for further testing
Purpose	To determine the areas beneficial to test on the basis of count of faults prior to testing.
Use	• Indication of areas beneficial to test. • Testers target efforts on error prone modules first and test them rigorously. • Quick identification of faults. • Extra time to test rest of the system. • Improvement in overall testing effectiveness and efficiency.
Types of studies covered related to count of faults prior to testing	• Using a negative regression model [73]. • Using software design metrics and reuse information [74]. • Using module's age, the changes made to the module and the ages of the module [75]. • Using pre-coding characteristics of the system design [77]. • Using random forests [78].
Limitations	• Extraction of information for fault prediction is not normally organized in the form of software change databases. • If a change database is maintained, it is still difficult to categorize which changes are faults. • The database might be prone to erroneous data. • The unit of analysis is not consistent e.g. some researchers use files while others use modules as being error prone, thus making is difficult for system testers who base their testing on functionality described in specification [74]. • Some of the predictions are based on the timely availability of process and design metrics.

8.5.3 Measuring Combination of Testing Techniques

The combination of testing techniques is discussed here in combination with *estimation of test cases* attribute.

System testing validates whether the program meets the specification from behavioral or functional perspective. The black box testing techniques is the dominant type of testing at the system level. In terms of black box testing, the number of test for system testing can grow very large. Therefore, an important concern is to select wisely among the black box techniques so that the black box tests are reduced to a number that can be executed within resource limitations and exposes majority of software defects [93]. As [2] writes, *since exhaustive testing is out of question, the objective should be to maximize the yield on the testing investment by maximizing the number of error found by a finite number of test cases.*

Equivalence class partitioning and boundary value analysis is commonly used to select input data for testing. The input data combinations must be selected in such a way that

detects maximum number of faults. In this situation, the tester either uses manual combinations that are important or use heuristics like orthogonal arrays [93]. These approaches do reduce the number of input combinations but their effect on fault detection capabilities is hard to determine.

There are several types of black box techniques. One study carried out by R. Torkar and S. Mankefors-Christiernin [94] divides black box testing techniques into partition testing, boundary value analysis, cause-effect graphing, error-guessing, random testing, exhaustive testing and nominal and abnormal testing. Three black box techniques namely equivalence partitioning, random testing and boundary value analysis were compared to find the efficiency with respect to finding severe faults. Equivalence partitioning was found to be the single most effective test methodology to detect faults that leads to severe failures.

Another study carried out by L. Lauterbach and W. Randall [95] compared six testing techniques namely branch testing, random testing, functional testing, code review, error analysis and structured analysis at both unit and configured source code level. The results of the study indicated that a combination of different techniques discovered as many as 20 faults in the software that were not previously found [31].

Further, a study carried out by Basili and Selby [96] compared three testing strategies namely code reading using step wise abstraction, functional testing using equivalence partitioning and boundary value analysis and structural testing with 100 percent statement coverage. One of their results showed that incase of professional programmers, code reading resulted in higher fault detection rates than functional and structural testing.

[93] introduces an approach that claims to significantly reduce the number of black box tests based on automated input-output analysis to identify relationships between program inputs and outputs. [2] is of the view to supplement black box oriented test case design methodologies with testing the logic using white box methods. The rigorous testing strategy should aim to take advantage of the distinct strengths of each test case design technique. A strategy recommended by [2] for combining test case design techniques is as following:

1. Start with cause-effect graphing if specification contains combinations of input conditions.
2. Supplement cause-effect graphing with boundary value analysis.
3. Identify valid and invalid equivalence classes for input and output.
4. Use error-guessing to include additional test cases.
5. If the test coverage criterion is not met in the first four steps, examine the program's logic to add test cases that complete the coverage criteria.

According to [97], the tests against requirements forms the initial tests followed with structural tests for branch coverage and data flow coverage. [6] identifies the four factors that contribute to the decision of selection of a testing technique, namely nature of the system, the overall risk of implementation, the level of test and skill set of testers. A combination of proven testing techniques when designing test case scenarios (functional analysis, equivalence partitioning, path analysis, boundary value analysis, and orthogonal arrays) is also recommended by [11]. Moreover, [6] identifies the techniques of black box testing as equivalence partitioning, boundary value analysis, decision tables, domain analysis, state transition diagrams and orthogonal arrays. These techniques are recommended to be applied at the system testing level (Table 35).

Table 35. Techniques at system level.

Method	System Level
Equivalence class partitioning	✓
Boundary value analysis	✓
Decision tables	✓
Domain analysis	✓
State transition diagrams	✓
Orthogonal arrays	✓

Cem Kaner and James Bach lists down a list of paradigms of black box software testing [98]. A paradigm provides a framework of thinking about an area and a testing paradigm defines types of tests that are relevant and interesting. The list of paradigms for black box testing involves the testing techniques types of domain driven, stress driven, specification driven, risk driven, random/statistical, functional, regression, scenario/use case/transaction flow, user testing and exploratory.

A summary of discussion on combination of testing techniques/estimation of test cases appear in Table 36.

Table 36. A summary of discussion on combination of testing techniques/estimation of test cases.

Attribute studied	Combination of testing techniques/estimation of test cases
Purpose	To know what combination of testing techniques are effective in finding more faults.
Use	• To reduce the number of executed tests. • To be able to execute tests within resource constraints. • To be able to select tests with fault-finding effectiveness.
Studies relevant to combination of testing techniques	• [94] showed that equivalence class partitioning performs better than random testing. • A result of the comparison of six testing techniques by [95] showed considerable variance in results and showed that a combination of testing techniques detected more faults. • Study carried out by Basili and Selby [96] showed that incase of professional programmers, code reading resulted in higher fault detection rates than functional and structural testing. • [93] introduces an approach to reduce the number of black box tests based on automated input-output analysis. • [98, 6, 11, 99] agree on using a combination of testing techniques to take advantage of the strengths of each.

8.5.4 Measuring Adequacy of Test Data

Test data needs careful execution and preparation. The pre-requisites for an effective test data are a good data dictionary and detailed design documentation [11]. Data dictionary is important because it describes the data structures, data types, length of data items and rules.

Design documentation, especially the data model helps to identify the relationship among data elements and how the functionality uses data. It is not practically possible to test every combination of input and output to a program, so the selection of the test-design techniques helps to narrow down the number of input and output combinations. For example, boundary value analysis executes tests at and around the boundary of data items.

While acquiring the test data, there are several concerns that need to be addressed:

1. Depth of test data: The size of test data needs to be determined. For example in case of a table in a database, it is required to be determined how much records are sufficient. For functional testing, a small subset of test data may be sufficient but for performance testing, production-size data needs to be acquired.
2. Breadth of test data: There needs to be variation among the test data so as to make the test case more effective in finding faults. Also if a data filed supports multiple data types, the test data must test with data representing each of the data type supported.
3. Scope of test data: The scope of test data means that it should be accurate, relevant and complete.
4. Data integrity during test execution: When a testing team is performing tests, it is important that no two members are modifying and querying the same data at the same time, so as to avoid generation of unexpected results. The test data should be able to be modified, segregated and aggregated as per requirements.
5. Conditions specific data: The test data should be ready to match specific conditions in applications. For example, for generating a student transcript in a student management system, the person needs to be an active student in the database.

There is normally a criterion to evaluate whether a test set is adequate for a given program or not. *A test adequacy criterion is a rule or a set of rules that imposes requirements on a set of test cases* [99]. An adequate test set is a finite set of test cases which meets the test adequacy criterion. The program is said to be adequately tested if it is tested with an adequate test set [99]. A test adequacy criterion is also used as stopping criteria for testing. The test adequacy criterion is related to a program or specification or both. Program-based adequacy criteria are based on the coverage of structural elements of the code by the test data e.g. control-flow and data-flow coverage. Specification-based adequacy criteria are based on the coverage of the specification and the test data ensures coverage of all the functions.

According to [100], there are two classic papers on test adequacy. One is *Toward a Theory of Test Data Selection* by J. B. Goodenough and S. L. Gerhart (1975) and the other is *The Evaluation of Program-Based Software Test Data Adequacy Criteria* by E. J. Weyuker (1988). Gerhart and Goodenough laid the foundations of test adequacy criterion by defining it as a predicate that defines what properties of a program must be exercised to constitute a thorough test [100]. Weyuker defined 11 axioms for program-based test adequacy criteria, a critique of this effort is that the paper is theoretical with few hints for practitioners [100].

9 EPILOGUE

This chapter consists of recommendations reflecting the thesis study, conclusions and identification of areas where further work might be needed.

9.1 Recommendations

The thesis attempted to consolidate the metric support for the attributes in software test planning and test design processes. The purpose for the thesis of different metrics is to provide a wide variety of metric options for project managers and testing team. The metrics studied against each attribute needs to be assessed for their practicality in terms of project's context and benefits to the testing team. The organization needs to evaluate these metrics in terms of the benefits that it expects to achieve from them. A metric would be successful if it is clearly needed.

Moreover, there should be a strong motivation for the use of metrics in improving the software test planning and design processes. The metrics are not to measure individuals and this is a common mistake resulting from misinterpretation of the metric results. In this regards, it is important to set clear goals to achieve from the metrics. It is very important that the metrics are interpreted and analyzed not only by the data it represents but also considering associated factors affecting the results. The metrics should also be inline with the organizational strategy because the strategy would help the selection of attributes that needs measurement.

An important aspect for implementing metrics is that the data collection process should be timely; therefore, the managers need to keep the members informed by giving feedback about the data that they have collected. The team members need encouragement in reporting the data that is useful for the organization. Therefore, the data collection shall support the team members, without affecting their primary tasks. Another important point that must be addressed is that the level of effort required for measurement needs to be well-understood and ideally be minimal. It is useful to express the objectives that are expected to be achieved out of testing in measurable form. These objectives need to be explicitly mentioned in the test plan.

Moreover, it is important to analyze the assumptions in the calculation of the metrics. The metrics need to be continuously assessed for its role in the design of effective test cases and efficacy of testing. In other words, the metrics need to be strongly linked with the improvement goals so that the team does not lack motivation in collecting metrics data. The development goals shall be communicated to the employees. Metrics data collection and analysis are to be automated to an extent possible so as to provide the project managers with timely feedback on progress and quality. The metrics need to be introduced incrementally, perhaps initially for one development team.

The team shall be involved and closely associated to the metric program so that accurate metric data can be collected. The metrics to be collected shall be transparent to the team members so that they can acknowledge the benefits obtained. Moreover, the metrics need to be simpler that establishes the relationship between the measurements and problems to be addressed.

9.2 Conclusions

Today, the majority of the metrics in software testing are based on test execution phase and on the basis of number of faults found in test execution. There is an apparent gap when we are interested in metrics for test planning and test design processes. Therefore, by focusing on the metric support in the test planning and test design processes, this thesis has contributed in filling part of this gap.

The measurable attributes of software test planning and test design are not mentioned in literature in a consolidated form. By identifying seventeen attributes for software test planning and thirty one attributes for software test design processes, the thesis serves to consolidate the attributes that are useful for measurement. The thesis partitioned the attributes in different categories, for software test planning there are four categories while there are five for software test design process. Such a categorization is a contribution towards assigning classes to the identified attributes, which may entice further research within each category.

The thesis discussed metric support for the identified attributes for the test planning and test design processes. Again, there were different studies contributing to the measurement of each attribute. The thesis presented the different ways to measure each of the attributes with the intention to provide a variety of methods for the reader to think of and choose according to the context of the situation at hand.

An interesting aspect resulting from this thesis is that although measurements help informed decision making, but a degree of subjectivity, expert judgment and analogy still plays an important role in the final decision relating to software test planning and test design. Therefore, reaching a decision must interplay between knowledgeable judgment and metric results.

An organization can build a metrics program for its software test planning and test design processes on the foundations of attributes identified in the thesis. It is expected that such an effort will lead to informed decision making about different software testing activities, in addition to the formulation of a baseline for measuring improvement.

9.3 Further Work

During the course of thesis, several interesting related areas were found to be useful for further investigation. These issues were not covered in the thesis due to time constraints. A list of these areas is given below:

9.3.1 Metrics for Software Test Execution and Test Review Phases

Initial study in this master thesis categorized software testing in to four phases. An investigation in to the metric support available for software test execution and test review phases would be interesting. Some interesting questions to ask could be that what are the measurable attributes of the software test execution and test review phases and what is the metric support for the identified attributes?

9.3.2 Metrics Pertaining to Different Levels of Testing

The study in this thesis focused on metric support at the system testing level. This leaves an interesting area to investigate the metric support at other levels of testing i.e. unit level, integration level and user acceptance level.

9.3.3 Integration of Metrics in Effective Software Metrics Program

Another interesting area to explore further is to investigate how the identified metrics can be integrated into an organization wide metrics program.

9.3.4 Data Collection for Identified Metrics

The identified metrics for software testing needs to be supported by an investigation in to how to collect the metrics data in an optimal way and what are the different issues related to collecting data on different attributes.

9.3.5 Validity and Reliability of Measurements

Furthermore, a study into the validity of metrics and the degree to which they contribute to the understanding of the measured attributes could be interesting. Also an area of further research could be an investigation in to the reliability of measurements and degree of variation in the metric results.

9.3.6 Tool Support for Metric Collection and Analysis

A study of the tools to support metrics collection and analysis promises to be useful for project managers in deciding which tools to use for the project needs.

TERMINOLOGY

Metric	Number or symbol assigned to attributes to define them according to rules [24].
Attribute	Feature or property or characteristic of an entity;
Software testing	An evaluation process to determine the presence of software errors.
Unit testing	Testing of internal processing logic and data structures in individual modules.
Integration testing	Testing related to interfaces between modules.
System testing	Determines software meets requirements as mentioned in the SRS.
Acceptance testing	Software testing done by customers or users.
Regression testing	Determines that software meets the requirements after changes.
Fault	Resulting due to human error.
Failure	Resulting due to a fault in run time.
Test planning	Planning for testing strategy, resource utilization, responsibilities, risks and priorities
Test design	Designing test objectives, selecting of test case design techniques, preparing test data, developing test procedures, setting up the test environment and supporting tools.
The suspension criteria	Conditions temporarily suspending software testing.
The exit criteria	Conditions indicating transition of testing activities from one level to the next.
Scope of testing	Determining the items, features, procedures, functions, objects, clusters and sub-systems to be tested [21].
Test coverage	Percentage of code, requirements, design, or interface covered by a test set.
Smoke tests	A set of test cases ensuring that the software is in a stable condition for testing to continue forward.
Defect backlog	Accumulated number of faults unresolved prior to test design.
Test data	Data input to tests.

REFERENCES

[1] B. Marick. New Models for Test Development. *Testing Foundations*. `www.testing.com/writings/new-models.pdf`, September 2006.

[2] G. J. Myers. *The Art of Software Testing*. John Willey & Sons, Inc., New York, USA, 1976.

[3] E. W. Dijkstra. Structured Programming. *In J.N.Buxton and B.Randell (eds.), Software Engineering Techniques, Brussels, Belgium, NATO Science Committee*, 1970.

[4] E. Miller. The Philosophy of Testing. *In Program Testing Techniques*, *IEEE Computer Society Press,* 1977.

[5] IEEE Standard 829-1998. IEEE Standard for Software Test Documentation. *IEEE*, 1998.

[6] R. D. Craig, S. P. Jaskiel. *Systematic Software Testing*. Artech House Publishers, Boston-London, 2002.

[7] IEEE Standard 1059-1993. IEEE Guide for Software Verification and Validation Plans. *IEEE*, 1993.

[8] R. S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw Hill Education Asia, 2005.

[9] S. R. Rakitin. *Software Verification and Validation for Practitioners and Managers*. Artech House Inc. Boston-London, 2001.

[10] ANSI/IEEE Standard 1008-1987. IEEE Standard for Software Unit Testing. *IEEE*, 1997.

[11] E. Dustin. *Effective Software Testing-50 Specific Ways to Improve Your Testing*. Addison Wesley, 2002.

[12] M. L. Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. John Willey & Sons, 2003.

[13] M. Rätzmann, C. D. Young, *Software Testing and Internationalization.* Galileo Press GmbH, Bonn, 2002.

[14] ISEB Foundation Certificate in Software Testing. *SIM Group Ltd., SQS Group AG*, 2002.

[15] J. Tian. *Software Quality Engineering- Testing, Quality Assurance, and Quantifiable Improvement*, IEEE Computer Society, 2005.

[16] E. Dustin, J. Rashka, J. Paul. *Automated Software Testing.* Addison-Wesley, 1999.

[17] J. Seo, B. Choi. Tailoring Test Process by Using the Component-Based Development Paradigm and the XML Technology. *In IEEE Software Engineering Conference,* 2000.

[18] The Wikipedia, the free encyclopedia. Article on Software Testing `http://en.wikipedia.org/wiki/Software_testing`, September 2006.

[19] I. Somerville. *Software Engineering*. Sixth Edition. Addison-Wesley, 2001.

[20] D. Kranzlmuller. Event Graph Analysis for Debugging Massively Parallel Programs. *Institute of Graphics and Parallel Processing, Johannes Kepler Universitat Linz, Austria*. `http://www.gup.uni-linz.ac.at/~dk/thesis/html/relateda2.html`, September 2006.

[21] I. Burnstein, T. Suwanassart, R. Carlson. Developing a Testing Maturity Model for Software Test Process Evaluation and Improvement. *In IEEE, Test Conference*, 1996.

[22] QAI Consulting Organization. Emphasizing Software Test Process Improvement, `http://www.qaiindia.com/Resources_Art/journal_emphasizing.htm` , September 2006.

[23] D. J. Paulish, A. D. Carleton. Case Studies of Software Process Improvement Measurement. *IEEE*, 1994.

[24] N. E. Fenton, S. L. Pfleeger. *Software Metrics - A Rigorous & Practical Approach.* Second Edition. PWS Publishing Company, 1997.

[25] S. Morasca, L. C. Briand. Towards a Theoretical Framework for Measuring Software Attributes. *IEEE*, 1997.

[26] C. Kaner. Software Engineering Metrics: What Do They Measure and How Do We Know? *10th International Software Metrics Symposium*, 2004.

[27] R. E. Park, W. B. Goethert, W. A. Florac. Goal Driven Software Measurement-A Guidebook. *CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University*, August 1996.

[28] K. H. Moller, D. J. Paulish. *Software Metrics: A Practitioner's Guide to Improved Product Development.* IEEE CS Press, Los Alamitos, 1993.

[29] IEEE Standard 1061-1998. IEEE Standard for Software Quality Metrics Methodology. *IEEE*, 1998.

[30] I. Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., 2003.

[31] R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall Inc., New Jersey, 1992.

[32] A. D. Carleton et al. Software Measurement for DoD Systems: Recommendations for Initial Core Measures. *Tech. Report CMUISEI-92-019, ESC-TR-92-019*, *Software Engineering Institute, Carnegie Mellon University, Pittsburgh*, 1992.

[33] L. M. Laird, M. C. Brennan. S*oftware Measurement and Estimation: A Practical Approach.* John Wiley & Sons, Inc., New Jersey, 2006.

[34] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Second Edition. Addison Wesley, 2002.

[35] Software Engineering Program-Software Measurement Guidebook. *National Aeronautics and Space Administration (NASA) Washington, DC.*, August 1995.

[36] S. Bradshaw. Test Metrics: A Practical Approach to Tracking & Interpretation, *Qestcon Technologies, A division of Howard Systems International, Inc*.

[37] R. Chillarege. Software Testing Best Practices. *Center for Software Engineering, Copyright IBM Research- Technical Report RC 21457 Log 96856,* 1999.

[38] Bazman's Testing Pages, a website containing articles and white papers on software testing
`http://members.tripod.com/bazman/`, November 2006.

[39] National Archives and Records Administration. Testing Management Plan. *Integrated Computer Engineering, Inc. a subsidiary of American Systems Corporation (ASC)*, 2003.

[40] R. Craig. Test Strategies and Plans. *Copyright Software Quality Engineering, Inc.*, 1999

[41] K. Iberle. Divide and Conquer: Making Sense of Test Planning. *In the International Conference on Software Testing, Analysis and Review, STARWEST*, 1999.

[42] R. Shewale. Unit Testing Presentation. *A StickyMinds Article*.
`http://www.stickyminds.com/getfile.asp?ot=XML&id=6124&fn=XDD6124filelistfilename1%2Eppt`, November 2006.

[43] R. Fantina. *Practical Software Process Improvement*. Artech House Inc., 2005.

[44] R. Black. *Managing the Testing Process*. Second Edition. Wiley Publishing, Inc., 2002.

[45] J. Bach. Testing Testers — Things to Consider When Measuring Performance. *A StickyMinds Article*. `http://www.stickyminds.com`, November 2006.

[46] C. Kaner. Measuring the Effectiveness of Software Testers. *Progressive Insurance*, July 2006.

[47] The Standish Group. Chaos Report.
`www.projectsmart.co.uk/docs/chaos_report.pdf`, 1995.

[48] K. Molokken and M. Jorgensen. A Review of Surveys on Software Effort Estimation. *Simula Research Laboratory*, 2003.

[49] N. Bajaj, A. Tyagi, R. Agarwal. Software Estimation – A Fuzzy Approach. *ACM SIGSOFT Software Engineering Notes Volume 31 Number 3*, May 2006.

[50] J. P. Lewis. Limits to Software Estimation. *ACM SIGSOFT Software Engineering Notes Volume 26 Number 4*, July 2001.

[51] D. V. Ferens, D. S. Christensen. Does Calibration Improve the Predictive Accuracy of Software Cost Models? *CrossTalk*, April 2000.

[52] K. Iberle, S. Bartlett. Estimating Tester to Developer Ratios (or Not). *Hewlett-Packard and STEP Technology*. `www.kiberle.com/pnsqc1/estimate.doc`, November 2006.

[53] L. M. Laird, M. C. Brennan. Practical Software Measurement and Estimation: A Practical Approach. *Copyright IEEE Computer Society*, 2006.

[54] W. E. Lewis. *Software Testing and Continuous Quality Improvement*. Second Edition. Auerbach Publications, 2005.

[55] R. Patton. *Software Testing*. Sams Publishing, July 2006.

[56] M. Marre´, A. Bertolino. Using Spanning Sets for Coverage Testing. *In IEEE Transactions on Software Engineering, Volume 29, Number 11*, November 2003.

[57] S. Cornett. Code Coverage Analysis. *Bull Seye Testing Technology*. `http://www.bullseye.com/coverage.html`, Dec 2005.

[58] C. Kaner. Software Negligence and Testing Coverage. `http://www.kaner.com/coverage.htm`, 1996.

[59] P. Piwowarski, M. Ohba, J. Caruso. Coverage Measurement Experience During Function Test. International Business Machines Corporation. *In IEEE Software Engineering Proceedings,* 1993.

[60] Requirements Traceability for Quality Management. *Compuware Corporation Whitepaper.* `www.softwarebusinessonline.com/images/WhitePaper_Compuware.pdf`, 2006.

[61] V. Karthikeyan. Traceability Matrix. *A StickyMinds Article*. `http://www.stickyminds.com/r.asp?F=DART_6051`, 2006.

[62] T. Pyhälä, K. Heljanko. Specification Coverage Aided Test Selection. *In Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03), IEEE,* 2003.

[63] M. P. E. Heimdahl, D. George, R. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? *In Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04), IEEE,* 2004.

[64] M. W. Wahlen, A. Rajan, M. P. E. Heimdahl, S. P. Miller. Coverage Metrics for Requirements Based Testing. *In Proceedings of the 2006 International Symposium on Software Testing and Analysis,* 2006.

[65] P.E. Ammann, P. E. Black. A Specification Based Coverage Metric to Evaluate Test Sets. *In Proceedings of 4th International Symposium on High-Assurance Systems Engineering*, *IEEE,* 1999.

[66] P.E. Ammann, P. E. Black, W. Majurski. Using Model Checking to Generate Tests from Specifications. *In Proceedings of 2nd International Conference on Formal Engineering Methods, IEEE,* 1998.

[67] S. McConnell. Daily Build and Smoke Test. *IEEE Software, Volume 13, Number 4*, July 1996.

[68] A. Dada. Smoke Tests to Signal Test Readiness. *Sun Microsystems*. `https://glassfish.dev.java.net/quality/BetterSoftware-Presentation-Smoketests.pdf`, November 2006.

[69] J. Bach. Test Plan Evaluation Model. *Satisfice, Inc.*, 1999.

[70] B. Berger. Evaluating Test Plans with Rubrics. *International Conference on Software Testing Analysis and Review*, 2004.

[71] R. Black. *Critical Testing Processes: Plan, Prepare, Perform*, Perfect. Addison Wesley, 2003.

[72] N. E. Fenton, N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering, Volume 26, Number 8*, 2000.

[73] T. J. Ostrand, E. J. Weuker, R. M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering, Volume 31, Number 4*, 2005.

[74] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, N. Goeal. Early Quality Prediction: A Case Study in Telecommunications. *IEEE Software, Volume 13, Issue 1*, 1996.

[75] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy. Predicting the Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering, Volume 26, Number 27*, 2000.

[76] J. C. Munson, T. J. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering, Volume 18*, *Number 5,* 1992.

[77] N. Ohlsson, H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions on Software Engineering, Volume 22*, *Number 12,* 1996.

[78] L. Guo, Y. Ma, B. Cukic, H. Singh. Robust Prediction of Fault-Proneness by Random Forests. *In Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.

[79] J. Gray. Why do Computers Stop and What Can be Done About it? *In Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[80] K. Vaidyanathan, K. S. Trivedi. Extended Classification of Software Faults Based on Aging. *Duke University Durham USA*, 2001.

[81] Bazman's Testing Pages, a website containing articles and white papers on software testing. `http://members.tripod.com/~bazman/classification.html?button7=Classification+of+Errors+by+Severity`, November 2006.

[82] S. Sanyal, K. Aida, K. Gaitanos, G. Wowk, S. Lahiri. Defect Tracking and Reliability Modeling For a New Product Release. *IBM Canada Limited Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada*.

[83] T. Pearse, T. Freeman, P.Oman. Using Metrics to Manage the End-Game of a Software Project. *In Proceedings of Sixth International Symposium on Software Metrics, IEEE,* l999.

[84] Y. Chernak. Validating and Improving Test-Case Effectiveness. *IEEE Software*, 2001.

[85] R. M. Poston, M. P. Sexton. Evaluating and Selecting Testing Tools. *IEEE Software*, 1992.

[86] G. Rothermel, M. J. Harrold. Analyzing Regression Test Selection Techniques. *In the Proceedings of IEEE Transactions on Software Engineering, Volume 22, Number 8,* 1996.

[87] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. Prioritizing Test Cases for Regression Testing. *In the Proceedings of IEEE Transactions on Software Engineering, Volume 27, Number 10,* 2001.

[88] T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *IEEE,* 1998.

[89] A. G. Malishevsky, G. Rothermel, S. Elbaum. Modeling the Cost-Benefits Tradeoffs for Regression Testing Techniques. *In the Proceedings of International Conference on Software Maintenance (ICSM), IEEE,* 2002.

[90] D. S. Rosenblum, E. J. Weyuker. Predicting the Cost Effectiveness of Regression Testing Strategies. *In the Proceedings of 4th ACM SIGSOFT Symposium on Foundations of Software Engineering,* 1996.

[91] T. J. McCabe, C. W. Butler. Design Complexity Measurement and Testing. *In the Communications of the ACM, Volume 32, Issue 12,* 1989.

[92] S. Elbaum, A. G. Malishevsky, G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *In the Proceedings of IEEE Transactions on Software Engineering, Volume 28, Number 2,* 2002.

[93] P. J. Schroeder, B. Korel. Black-box Test Reduction Using Input-Output Analysis. *In the Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA*, 2000.

[94] R. Torkar, S. Mankefors-Christiernin. Fault Finding Effectiveness in Common Black Box Testing Techniques: A Comparative Study. *In Proceedings of 3rd Conference on Software Engineering Research and Practice in Sweden (SERPS'03),* 2003.

[95] L. Lauterbach, W. Randall. Experimental Evaluation of Six Test Techniques. *In the Proceedings of the Fourth Annual Conference on Systems Integrity, Software Safety and Process Security, IEEE,* 1989.

[96] V. R. Basili, R. W. Selby. Comparing the Effectiveness of Software Testing Strategies. *In the Proceedings of IEEE Transactions on Software Engineering Volume 13 Number 12,* 1987.

[97] B. Beizer. *Software Testing Techniques*. Second Edition, The Coriolis Group, 1990.

[98] C. Kaner, J. Bach. Paradigms of Black Box Software Testing. 1999.
`www.kaner.com/pdfs/swparadigm.pdf`

[99] S. Kim, J. A. Clark, J. A. McDermid. Investigating the Applicability of Traditional Test Adequacy Criteria for Object-Oriented Programs. *Department of Computer Science, University of York, United Kingdom*. November 2006.
`www.cs.york.ac.uk/~jac/papers/TRAD.pdf`

[100] T. Xie. Software Testing and Analysis. Special Topics on Software Testing and Analysis, *NC State University*, 2006.
`http://ase.csc.ncsu.edu/courses/csc591t/2006spring/wrap/GUItesting.pdf`.

[101] J. W. Creswell. *Research Design. Qualitative, Quantitative and Mixed Method Approaches.* Second Edition, Sage Publications, 2002.

APPENDIX 1. TEST PLAN RUBRIC

	Good	Average	Bad
Theory of Objective	Clearly identifies realistic test objectives. Describes the most efficient tests.	Describes credible test objectives	There are no objectives, or they are irrelevant. Objectives documented in the test plan are incongruent with the culture of the organization.
Theory of Scope	Scope may be implicit or documented, but in either case the test scope is specific and unambiguous.	Identifies some scope of testing and is understood by only some people.	Mix-ups in the test execution because of wrong scope assumptions. Scope of testing is not defined within the project.
Theory of Coverage	For this test plan, the test coverage is complete relative to the test scope. Planned tests are laid out in a way that clearly connects with test objective.	The breadth of test coverage is sufficient. Planned tests are somewhat connected to test objectives.	Test coverage is correlated to neither the test objectives nor the test scope. Exit criteria are not defined or inappropriately defined.
Theory of Risk	Test plan identifies plausible testing risks, assumptions and constraints.	Risks have inappropriate priorities. Risks are either exaggerated or underplayed. Most risks to the successful completion of the project have been addressed, in one way or another.	There is no understanding of project or product risk throughout the organization. Obvious risks have not been addressed.
Theory of Data	Efficient method to generate enough valid and invalid test data is known and planned for.	Some method to generate test data is specified.	There is no method of capturing data. No formal plan of generating test data exists.
Theory of Originality	There is high information in every word. There are things to	A test plan template is used as a starting point, and some original content has	Template placeholders are not filled in. Instructions to the

	pay attention to on every page.	been added.	tester on how to use the template is included as content of the project's test plan.
Theory of Communication	Tester used multiple modes to communicate test plan to, and receive feedback from appropriate stakeholders.	All feedback is incorporated, not necessarily the best feedback. There is signoff that appears binding.	No Distribution, no opportunity for feedback. Test plan contradicts itself after incorporating feedback from multiple stakeholders.
Theory of Usefulness	Test plan helped testers early discover and react to the right problems.	Things went wrong not mentioned in the plan.	Test plan was successfully used against the company in a lawsuit.
Theory of Completeness	There is "Just Enough" testing in the plan without over or under testing.	Most test items have been identified.	Huge holes are discovered in the plan (not necessarily documented) and have not been filled after review.
Theory of Insightfulness	The test plan shows understanding of what is interesting and challenging in testing this specific project.	The test plan is effective, but bland.	The tester infuses only one-dimensional, linear, hierarchical, or sequential thought into the test plan.

APPENDIX 2. A CHECKLIST FOR TEST PLANNING PROCESS

Step #	*Step*	*Done*
1.	*Research, devise, collect, and document the strategy, tactics, and internal workings of the test subproject.*	□
2.	*Negotiate and document the collaborative workings between the test subproject and the overall project.*	□
3.	*Finalize and document the remaining logistical and planning details, such as the risks to the test subproject itself and definitions of testing and project terms. Annotate any referenced documents. Write a one-page executive summary of the test subproject.*	□
4.	*Circulate the plan for private (offline) review, often to the test team first and then to the wider collection of stakeholders and participants. Gather input and revise the plan, iterating steps 1 through 3 as needed. Assess any changes to the estimated schedule and budget (that exceed retained slack or contingency) resulting from the planning process, and obtain management support for such changes.*	□
5.	*Circulate the plan for public review. Hold a review meeting with all the stakeholders. Gather any final adjustments needed, and obtain commitment that, with any modifications agreed upon in the review meeting, the plan shall be the plan of record for the test subproject.*	□
6.	*Revise the estimated schedule and budget based on new knowledge gleaned from the planning process, including resource use. If this results in a slip in the schedule or an increase in the budget beyond any retained slack or contingency, escalate it to management for resolution. Negotiations about the new budget and schedule may cause iteration of the previous steps or reworking of the estimate.*	□
7.	*Check the test plan(s) into the project library or configuration management system. Place the document under change control.*	□

APPENDIX 3. A CHECKLIST FOR TEST DESIGN PROCESS

Step #	*Step*	*Done*
8.	*Create or update an outline of test suites that can cover the risks to system quality for which testing has been identified as a recommended action.*	□
9.	*Select an appropriate test suite and discover a new set of interesting test conditions within the most critical still-uncovered risk area; e.g., conditions that might provoke a particular set of failures or model actual usage. Define at a high level how to assess the correctness of results under those test conditions.*	□
10.	*Select the appropriate test techniques, given the test conditions to be explored, the verification of expected results desired, the time and budget available, the testability features of the system under test, the test environment, and the skills present in the test team.*	□
11.	*Design, develop, acquire, enhance, configure, and document the testware, the test environment, and the test execution process to produce those conditions and verify those behaviors using the selected test techniques. Augment theoretical test techniques with empirical techniques, especially customer/user data, field failure reports (previous and competitors' releases, similar products, and similar sets of risks to system quality), sales and marketing predictions on future usage, and other customer-centered inputs.*	□
12.	*Should any testware, test environment, or test execution process elements prove unattainable during step 4 because of resource, schedule, or other limitations, repeat step 4 iteratively to accommodate any such limitations.*	□
13.	*Test the test system, using static techniques (e.g., reviews) and dynamic techniques (e.g., running the test).*	□
14.	*Check the testware, test execution process description, test environment configuration information, test system test documentation (from step 6), and any other documentation or files produced into the project library or configuration management system. Link the version of the test system to the version of the system under test. Place the item(s) under change control.*	□
15.	*Update the quality risks list based on what was learned in developing this latest set of tests. Evaluate coverage of the quality risks with the new test system. Identify remaining uncovered quality risks for which testing action is recommended.*	□
16.	*Repeat steps 2 through 8 until all quality risks are covered to the extent testing action was recommended. Ensure that the quality risk documentation in the project repository is updated. If schedule or budget restrictions curtail development without all critical quality risks covered, produce a report of uncovered quality risks and escalate that report to management.*	□
17.	*If time and resources allow, iterate steps 2 through 8 using structural analysis (e.g., McCabe complexity or code coverage) to identify areas needing further testing.*	□

APPENDIX 4. TYPES OF AUTOMATED TESTING TOOLS

Type of tool	Description	Key points
Test-Procedure Generators	Generate test procedures from requirements/design/object models	Best used with requirements management tool, creates test procedures by statistical, algorithmic and heuristics means.
Code (Test) Coverage Analyzers and Code Instrumentors	Identify untested code and support dynamic testing	Measures the effectiveness of test suites and tests.
Memory-Leak Detection	Verify that an application is properly managing its memory resources	Provides run time error detection.
Metrics-Reporting Tools	Read source code and display metrics information, such as complexity of data flow, data structure, and control flow.	Can provide metrics about code size in terms of numbers of modules, operands, operators, and lines of code.
Usability-Measurement Tools	User profiling, task analysis, prototyping, and user walk-throughs	It is not a replacement of human verification of interface.
Test-Data Generators	Generate test data	Provides quick population of database.
Test-Management Tools	Provide such test-management functions as test-procedure documentation and storage and traceability	Provides support for all phases of testing life cycle.
Network-Testing Tools	Monitoring, measuring, testing, and diagnosing performance across entire network	Provides performance coverage for server, client and network.
GUI-Testing Tools (Capture/Playback)	Automate GUI tests by recording user interactions with online systems, so they may be replayed automatically	The recorded scripts are modified to form reusable and maintainable scripts.
Load, Performance, and Stress Testing Tools	Load/performance and stress testing	Provides for running of multiple client machines simultaneously for response times and stress scenarios.
Specialized Tools	Architecture-specific tools that provide specialized testing of specific architectures or technologies, such as embedded systems	Examples include automated link testers for web application testing.

APPENDIX 5. TOOL EVALUATION CRITERIA AND ASSOCIATED QUESTIONS

Tool evaluation criterion	Questions
Ease of use	• Is the tool easy to use? • Is its behavior predictable? • Is there meaningful feedback for the user? • Does it have adequate error handling capabilities? • Is the interface compatible with other tools already existing in the organization?
Power	• Does it have many useful features? • Do the commands allow the user to achieve their goals? • Does the tool operate at different levels of abstraction? • Does it perform validation checks on objects or structures?
Robustness	• Is the tool reliable? • Does it recover from failures without major loss of information? • Can the tool evolve and retain compatibility between old and new versions?
Functionality	• What functional category does the tool fit? • Does it perform the task it is designed to perform? • Does it support the methodologies used by the organization? • Does it produce the correct outputs?
Ease of Insertion	• How easy will it be to incorporate the tool into the organizational environment? • Will the user have proper background to use it? • Is the time required to learn the tool acceptable? • Are results available to the user without a long set-up process? • Does the tool run on the operating system used by the organization? • Can data be exchanged between this tool and others already in use? • Can the tool be supported in a cost-effective manner?
Quality of support	• What is the tool's track record? • What is the vendor history? • What type of contract, licensing, or rental agreement is involved? • Who does maintenance, installation and training? • What is the nature of the documentation? • Will the vendor supply list of previous purchases? • Will they provide a demonstration model?
Cost of the tool	• Does organization have enough budget to purchase the tool? • Is a cost/benefit analysis conducted?
Fit with organizational policies and goals	• Is the tool purchase aligned with organizational policies and goal?

APPENDIX 6. REGRESSION TEST SELECTION TECHNIQUES

Linear equation techniques: In these techniques, linear equations are used to express relationships between tests and program segments. Program segments are portions of code covered by test execution. The linear equations are obtained from matrices that *track program segments reached by test cases, segments reachable from other segments and definite-use information about the segments* [86].

The symbolic execution technique: In this technique, input partitions and data-driven symbolic execution are used to select and execute regression tests [86]. This technique selects all test cases that exercise the new or modified code.

The path analysis technique: This technique makes use of exemplar paths i.e. acyclic paths from program entry to program exit [86]. This technique compares the exemplar paths in original program to the modified program and selects all tests that traverse modified exemplar paths [86].

Dataflow techniques: Dataflow techniques identifies the definition-use pairs that are new and modified in changed program and selects tests that exercises these pairs.

Program dependence graph techniques: These techniques compare the program dependence graph nodes and flow edges of original and modified programs to select regression tests.

System dependence graph techniques: These techniques select tests that exercise components in the original program that have common execution patterns with respect to new or affected components in modified program [86].

The modification based technique: This technique determines the program components that are data or control dependent on modified code and may be affected by a modification. *Testing is complete for the modification when each influenced component has been reached by some test that exercised the modification* [86].

The firewall technique:

This technique places a firewall around the modified code modules in such a way that unit and integration tests are selected for the modified modules that lie within the firewall.

The cluster identification technique:

This technique selects tests that execute new, deleted, and modified clusters (sub-graphs of a control flow graph).

Slicing techniques: There are four types of slicing techniques that are used for regression test selection. These techniques are execution slice, dynamic slice, relevant slice, and approximate relevant slice [86].

Graph walk techniques: These techniques make use of comparison of control flow graphs of the original and modified programs. The techniques selects those test cases from the original suite that that reaches the new or modified code or the tests that formerly reached code that has been deleted from the original program [86].

The modified entity technique: *This technique selects all test cases associated with changed entities* [86]. Entities are defined as the executable (e.g. functions) and non-executable (e.g. storage locations) portions of code.

APPENDIX 7. TEST CASE PRIORITIZATION TECHNIQUES

Category	Prioritization technique	Description
Comparator Techniques	Random ordering	Random ordering of test cases in test suite.
	Optimal ordering	Ordering to optimize rate of fault detection by knowing programs with known faults and determining which fault each test case exposes.
Statement Level Techniques	Total Statement Coverage Prioritization	Prioritization in descending order of the number of statements covered by each test case.
	Additional Statement Coverage Prioritization	Prioritization based on feedback about coverage attained so far in testing to focus on statements not yet covered.
	Total Fault Exposing Potential (FEP) Prioritization	Prioritization based on the fault exposing potential of the test case at statement level.
	Additional Fault Exposing Potential (FEP) prioritization	Prioritization based on feedback about the FEP of previous test cases at statement level.
Function Level Techniques	Total Function Coverage Prioritization	Prioritization of test cases based on the total number of functions they execute.
	Additional Function Coverage Prioritization	Prioritization based on feedback about coverage of functions attained so far in testing to focus on functions not yet covered
	Total Fault Exposing Potential (Function level) Prioritization	Prioritization based on the fault exposing potential of the test case at function level.
	Additional Fault Exposing Potential (Function Level) Prioritization	Prioritization based on feedback about the FEP of previous test cases at function level.
	Total Fault Index (FI) Prioritization	Prioritization based on the association of changes with fault-proneness using fault index, a metric for fault proneness.
	Additional Fault Index (FI) Prioritization	Prioritization based on incorporating feedback in to the Total Fault Index prioritization.
	Total FI with FEP Coverage Prioritization	Prioritization based on the combined total fault index prioritization and total fault exposing potential prioritization.
	Additional FI with FEP Prioritization	Prioritization based on incorporating feedback on total FI with total FEP prioritization.
	Total DIFF Prioritization	Prioritization based on the association of changes with fault-proneness using DIFF, a technique to compute syntactic differences between two versions of a program.

	Additional DIFF Prioritization	Prioritization based on incorporating feedback in to the total DIFF prioritization.
	Total DIFF with FEP Prioritization	Prioritization based on the combined total DIFF prioritization and total fault exposing potential prioritization.
	Additional DIFF with FEP Prioritization	Prioritization based on incorporating feedback on total DIFF with total FEP prioritization.

APPENDIX 8. ATTRIBUTES OF SOFTWARE TEST DESIGN PROCESS

No.	Software Test Design Attribute	Purpose
1.	Tracking testing progress	To track the progress of test design activity.
2.	Sequence of test cases	To prioritize the execution of test cases.
3.	Tracking testing defect backlog	To assess the impact of testing defect backlog on test design process.
4.	Staff productivity	To assess staff productivity during test design activity.
5.	Effectiveness of test cases	To determine the fault finding ability (quality) of the test case.
6.	Fulfillment of process goals	To assess accomplishment of software test design process goals.
7.	Test completeness	To track the test coverage of test cases.
8.	Estimation of test cases	To estimate the total number of test cases.
9.	Number of regression tests	To measure the number of test cases in regression testing suite.
10.	Identification of areas for further testing	To identify risky areas requiring more testing.
11.	Combination of test techniques	To identify the right combination of test techniques.
12.	Adequacy of test data	To define adequate test data.
13.	Tests to automate	To decide upon which tests to automate.
14.	Cost effectiveness of automated tool	To evaluate the benefits of automated tool as compared to the cost associated with it.

APPENDIX 9. ATTRIBUTES OF SOFTWARE TEST PLANNING PROCESS

No.	Software Test Planning Attribute	Purpose
15.	Test coverage	To assess the percentage of code, requirements, design or interface covered by a test set.
16.	The suspension criteria of testing	To establish conditions for suspending testing.
17.	Count of faults prior to testing	To identify training needs for the resources and process improvement opportunities.
18.	Duration of testing	To estimate a testing schedule.
19.	Expected number of faults	To gauge the quality of software.
20.	Resource requirements i.e. number of testers required	To estimate the number of testers required for carrying out the system testing activity.
21.	Testing cost estimation	To establish the estimates for system testing budgets
22.	Training needs of testing group and tool requirement	To identify the training needs for the testing group and tool requirement.
23.	Effectiveness of smoke tests	To establish that application is stable enough for testing.
24.	The exit criteria	To flag exit criteria for testing.
25.	The quality of the test plan	To improve the quality of the test plan produced.
26.	Scope of testing	To determine how much of the software is to be tested.
27.	Bug classification	To determine the severity of bugs.
28.	Monitoring of testing status	To keep track of testing schedule and cost.
29.	Staff productivity	To assess tester's productivity.
30.	Fulfillment of process goals	To assess accomplishment of process goals.
31.	Tracking of planned and unplanned submittals	To assess the impact of incomplete submittals on test planning process.

APPENDIX 10. HEURISTICS FOR EVALUATING TEST PLAN QUALITY ATTRIBUTES

Following heuristics are suggested [69]:

1. Testing should be optimized to find critical problems first because early detection of critical faults reduces the risk of a bad fix.
2. The testing strategy should also place focus on less critical problems because we are not perfect in our technical risk analysis.
3. The test plan should mention the testing environment, how the product will be operated, how will it be observed and how evaluations will be made. These factors are important otherwise there is a probability of not being able to find important problems.
4. The test strategy should focus on diverse set of testing techniques to increase the chances of detecting as many faults as possible. Similarly, methods for evaluating test coverage should look in to different coverage dimensions including structural, functional and requirements.
5. The test strategy should focus on how to generate and design the test data to avoid on-the-fly test data generation.
6. The testing strategy should have flexibility to incorporate exploratory testing so as to uncover unanticipated problems.
7. Since requirements can be incomplete and ambiguous, testing strategy should also focus on implicit and implied requirements in addition to explicitly written requirements.
8. The test plan should also mention the communication channels with development and technical support because they can provide useful information for effective risk analysis.
9. The test plan should highlight, if there are any aspects that can improve the testability of the product and communicate those to the development team.
10. The test plan should be configurable and adjustable to the special needs and non-routine aspects of the project.
11. A balance is required between automated and manual testing. Each one of them is suitable in circumstances best suited for them.
12. A test schedule should not be rigid but should highlight the dependencies on the *progress of development, the testability of the product, time required to report problems, and the project team's assessment of risk* [69].
13. In order to reduce the chances of reduction in testing time, the testing should be planned to be carried out in parallel with development.
14. The test plan should highlight the number of testing cycles required so that developers know when to expect the feedback on the fixes they make. It is important to speed up quality improvement [69].
15. The testing strategy should benefit from other sources of establishing quality, e.g. inspections to evaluate the testing process.
16. The test plan should include a review mechanism for the test documentation so as to reveal improvement opportunities.

APPENDIX 11. CHECKING FULFILLMENT OF TEST PLANNING GOALS

Following points serve as an example to check fulfillment of test planning goals:

1. Is the description of all test related documents available to all interested parties?
2. Are prescribed templates available to stakeholders?
3. Are the testing methods and tools to be used for testing described?
4. Are the testing methods specific to a project described?
5. At what levels the test planning will be carried out i.e. unit, integration, system and acceptance?
6. Are both white box and black box testing methods to be applied described in the plan?
7. Is the test plan reviewed?
8. Is the test plan placed under change control?
9. Is the author of the test plan experienced and trained in writing test plan?
10. Does the test plan mentions fault classification?
11. Are test related measurements collected during the test planning process?
12. Is the test plan signed-off for approval?
13. Are there appropriate planning tools available for test planning?
14. Are any test related risks identified in developing test plans?
15. Are estimates from prior projects available to be used for test planning [21]?
16. Is the test planning process reviewed?