**TESTING: CONCEPTS, ISSUES, AND TECHNIQUES**

**Testing: Why?**

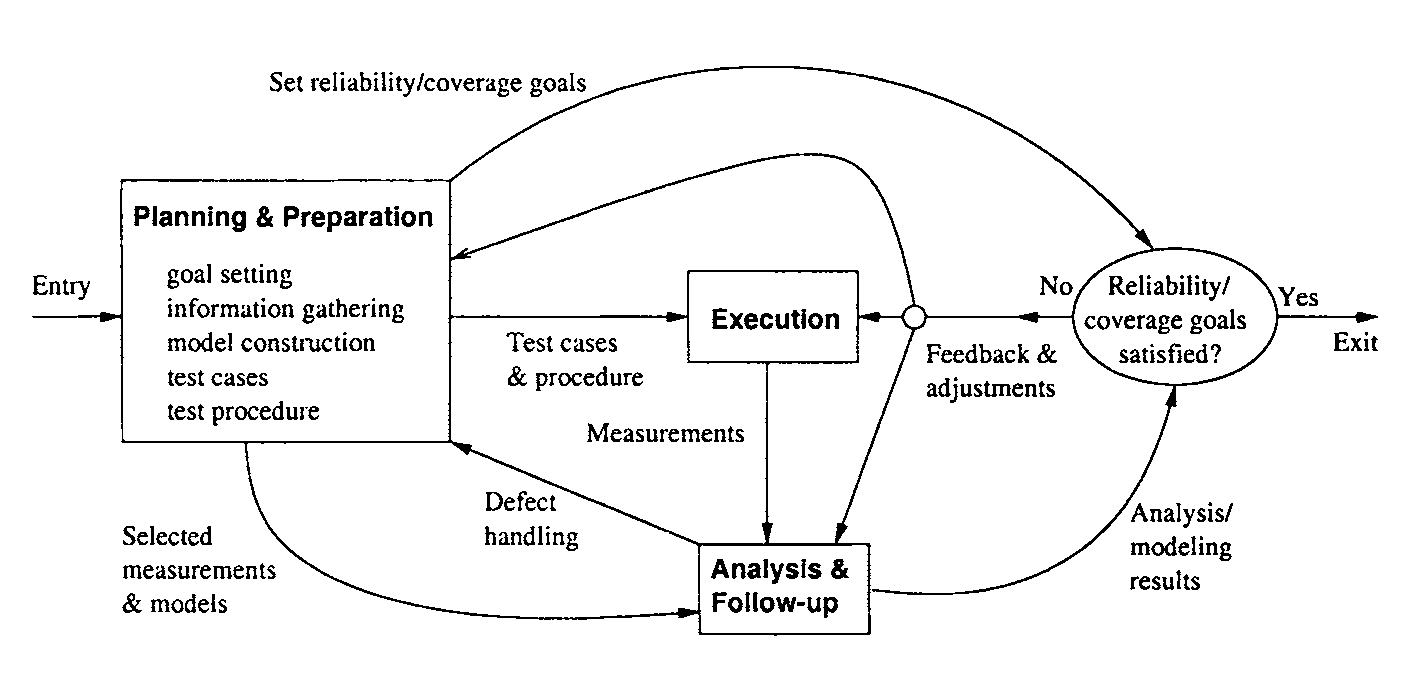
The purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users. The most natural way to show this fulfillment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in laboratory settings before the products are released or delivered. In the case of software products, such controlled experimentation through program execution is generally called testing.

**Major activities and the generic testing process**

The basic concepts of testing can be best described in the context of the major activities involved in testing. Although there are different ways to group them (Musa, 1998; Burnstein, 2003; Black, 2004), the major test activities include the following in roughly chronological order:

* Testplanning andpreparation, which set the goals for testing, select an overall testing strategy, and prepare specific test cases and the general test procedure.
* Test execution and related activities, which also include related observation and measurement of product behavior.
* Analysis and follow-up, which include result checking and analysis to determine if a failure has been observed, and if so, follow-up activities are initiated and monitored to ensure removal of the underlying causes, or faults, that led to the observed failures in the first place.

In fact, we can consider this generic testing process as an instantiation of the generic quality engineering process to testing. The major test activities are centered around test execution, or performing the actual tests.



**Figure.** Generic testing process

At a minimum, testing involves executing the software and communicating the related observations.

In act, many forms of informal testing include just this middle group of activities related to test execution, with some informal ways to communicate the results and fix the defects, but without much planning and preparation. in all forms of systematic testing, the other two activity groups, particularly test planning and preparation activities, play a much more important role in the overall testing process and activities.

The execution of a specific test case, or a sub-division of the overall test execution sequence for some systems that require continuous operation, is often referred to as a “test run”. One of the key component to effective test execution is the handling of problems to ensure that failed runs will not block the executions of other test cases. This is particularly important for systems that require continuous operation. To many people, defect fixing is not considered to be a part of testing, but rather a part of the development activities. However, re-verification of problem fixes is considered as a part of testing.

Data captured during execution and other related measurements can be used to locate and fix the underlying faults that led to the observed failures. After we have determined if a test run is a success or failure, appropriate actions can be initiated for failed runs to locate and fix the underlying faults. In addition, further analyses can be performed to provide valuable feedback to the testing process and to the overall development process in general. These analysis results provide us with assessments of the current status with respect to progress, effort, defect, and product quality, so that decisions, such as when to stop testing, can be made based on facts instead of on people’s gut feelings. In addition, some analyses can also help us identify opportunities for long-term product quality improvement. Therefore, various other activities, such as measurement, analysis, and follow-up activities, also need to be supported.

**Sub-activities in test planning and preparation**

Because of the increasing size and complexity of today’s software products, informal testing without much planning and preparation becomes inadequate. Important functions, features, and related software components and implementation details could be easily overlooked in such informal testing. Therefore, there is a strong need for planned, monitored, managed and optimized testing strategies based on1 systematic considerations for quality, formal models, and related techniques.

Test cases can be planned and prepared using such testing strategies, and test procedures need to be prepared and followed. The pre-eminent role of test planning and preparation in overall testing is also illustrated in **Figure**., by the much bigger box for related activities than those for other activities. Test planning and preparation include the following sub-activities:

***Goal setting:*** It is generally more concrete here, because the quality views and attributes have been decided by the overall quality engineering process. What remains to be done is the specific testing goals, such as reliability or coverage goals, to be used as the exit criteria.

***Test case preparation:*** This is the activity most people naturally associate with test preparation. It includes constructing new test cases or generating them automatically, selecting from existing ones for legacy products, and organizing them in some systematic ways for easy execution and management. In most systematic testing, these test cases need to be constructed, generated, or selected based on some formal models associated with formal testing techniques.

***Test procedure preparation:*** This is an important activity for test preparation. For systematic testing on a large scale for most of today’s software products and software intensive systems, a formal procedure is more of a necessity than a luxury.

It can be defined and followed to ensure effective test execution, problem handling and resolution, and the overall test process management.

**FUNCTIONAL VS. STRUCTURAL TESTING: WHAT TO TEST?**

As the primary type of objects to be tested, software programs or code exists in various forms and is written in different programming languages. They can be viewed either as individual pieces or **as** an integrated whole. Consequently, there are different levels of testing corresponding to different views of the code and different levels of abstraction, as follows:

* At the most detailed level, individual program elements can be tested. This includes testing of individual statements, decisions, and data items, typically in a small scale by focusing on an individual program unit or a small component. Depending on the different programming languages used, this unit may correspond to a function, a procedure, a subroutine or a method. **As** for the components, concepts may vary, but generally include a collection of smaller units that together accomplish something or form an object.
* At the intermediate level, various program elements or program components may be treated as an interconnected group, and tested accordingly. This could be done at component, sub-system, or system levels, with the help of some models to capture the interconnection and other relations among different elements or components.
* At the most abstract level, the whole software systems can be treated as a “blackbox”, while we focus on the functions or input-output relations instead of the internal implementation.

In each of the above abstraction levels, we may choose to focus on either the overall behavior or the individual elements that make up the objects of testing, resulting in the difference between functional testing and structural testing. The tendency is that at higher levels of abstraction, functional testing is more likely to be used; while at lower levels of abstraction, structural testing is more likely to be used. However, the other pairing is also possible, as we will see in some specific examples later.

Corresponding to these different levels of abstraction, actual testing for large software systems is typically organized and divided into various sub-phases starting from the coding phase up to post-release product support, including unit testing, component testing, integration testing, system testing, acceptance testing, beta testing, etc. Unit testing and component testing typically focus on individual program elements that are present in the unit or component. System testing and acceptance testing typically focus on the overall operations of the software system as a whole.

**Functional or black-box testing (BBT)**

Functional testing verifies the correct handling of the external functions provided by the software, through the observation of the program external behavior during execution. Becausethe software is treated as a black-box, with the external behavior observed through its input, output, and other observable characteristics, it is also commonly referred to as black-box testing (BBT). In this book, we use these two terms interchangeably.

The simplest form of BBT is to start running the software and make observations in the hope that it is easy to distinguish between expected and unexpected behavior. This form of testing is also referred to as “ad hoc” testing. Some unexpected behavior, such as a crash, is easy to detect. Once we determine that it is caused by software through repeated execution to eliminate the possibilities of hardware problems, we can pass the information to responsible parties to have the problem fixed. In fact, this is the common way through which problems experienced by actual customers are reported and fixed.

Another common form of BBT is the use of specification checklists, which list the external functions that are supposed to be present, as well as some information about the expected behavior or input-output pairing. Notice here that we used the term ***input*** to mean any action, artifact, or resource provided in the process of running a program, either at the beginning or at any time during the program execution. Similarly, we use the term ***output*** to mean any action, artifact, or result produced by the running program, either at the end or at any time during the program execution. Concrete examples of input to a calculator program might include the specific numbers entered and the action requested, such as division operation of two numbers. The output could be the actual division result, or some error message, such as when attempting to divide by zero. When problems are observed, specific follow-up actions are carried out to fix them.

More formalized and systematic BBT can be based on some formal models. These formal testing models are derived from system requirement or functional specifications. Some traditional white-box testing techniques can also be adapted to perform BBT, such as control-flow and data-flow testing for external functional units instead of for internal implementations.

In test planning, the focus is on identifying the external functions to test, and deriving input conditions to test these functions. The identified external functions are usually associated with some user expectations, from which both the input and the expected output can be derived to form the test cases. For example, for a compiler, the input is source code to be compiled, and the output is the resulting object or executable code. Part of the expected behavior is system termination, that is, the compiler should produce some output within a limited amount of time. Another part of the expected behavior is that if illegal programs are provided as input, object or executable code will not be generated, and the reason should be given. Therefore, a collection of programs to be compiled constitutes the test suite, or the collection of test cases. This test suite should typically consist of both legal and illegal programs to cover the expected spectrum of input. The testing goals may be stated explicitly as exit quality levels or implicitly as the completion of planned test cases.

The focus of test execution during BBT is to observe the external behavior, to ensure orderly execution of all the test cases, and to record execution information for analysis and follow-up activities. If the observed behavior patterns cannot be immediately identified as failures, information needs to be recorded for further analysis. In the above example of the compiler, the output produced and the execution trace should be recorded, as well as the exact set-up under which the compiler operated.

Once the execution result is obtained, either individually or as a set, analyses can be carried out to compare the specific behavior and output with the expected ones. This comparison to determine if it is expected behavior or if a failure occurred is called the testing ***oracle*** problem. Thus BBT checks whether the observed behavior conforms to user expectations or product specifications. Failures related to specific external functions can be observed, leading to follow-up activities where corresponding faults are detected and removed. The emphasis is on reducing the chances of encountering functional problems by target customers. Information recorded at test execution is used in these follow-up activities to recreate failure scenarios, to diagnose problems, to locate failure causes and identify specific faults in software design and code, and to fix them. An important followup decision, when to stop testing, can be determined either using the traditional functional coverage criteria or reliability criteria.

**Structural or white-box testing (WBT)**

Structural testing verifies the correct implementation of internal units, such as program statements, data structures, blocks, etc., and relations among them. This is done through test execution by observing the program behavior related to these specific units. Because the software is treated as a white-box, or more appropriately a glass-box or a transparent § box, where one can see through to view the internal units and their interconnections, it is also commonly referred to as white-box testing (WBT) in literature. In keeping with this convention, we also label this as WBT, with the understanding that this “white-box’’ is really transparent so that the tester can see through it. In this book, we also use the two terms, structural testing and WBT, interchangeably.

Because the connection between execution behavior and internal units needs to be made in WBT, various software tools are typically used. The simplest form of WBT is statement coverage testing through the use of various debugging tools, or debuggers, which help us in tracing through program executions. By doing so, the tester can see if a specific statement has been executed, and if the result or behavior is expected. One of the advantages is that once a problem is detected, it is also located. However, problems of omission or design problems cannot be easily detected through WBT, because only what is present in the code is tested. Another important point worth noting is that the tester needs to be very familiar with the code under testing to trace through its executions. Consequently, WBT and related activities are typically performed by the programmers themselves because of their intimate knowledge of the specific program unit under testing. This dual role also makes defect fixing easy.

Similar to the situation for BBT, more formalized and systematic WBT can be based on some formal models. These formal testing models are typically derived from system implementation details. **In** fact, the majority of the traditional testing techniques is based on program analyses and program models, and therefore is white-box in nature. WBT can also follow the generic testing process, to carry out the major test activities of planning, execution, and follow-up. However, because of the extensive amount of implementation knowledge required, and due to the possibility of combinatorial explosions to cover these implementation details, WBT is typically limited to a small scale. For small products, not much formal testing process is needed to plan and execute test cases, and to follow up on execution results. For unit testing of large products, the WBT activities are carried out in the encompassing framework where most of the planning is subject to the environment; and the environmental constraints pretty much determine what can be done. Therefore, test planning plays a much less important role in WBT than in BBT. **In** addition, defect fixing is made easy by the tight connection between program behavior and program units, and through the dual role played by the programmers as testers. Consequently, not much formal testing process is needed. The stopping criteria are also relatively simple: Once planned coverage has been achieved, such as exercising all statements, all paths, etc., testing can stop. Sometimes, internal quality measures, such as defect levels, can also be used as a stopping criterion.

**Comparing BBT with WBT**

To summarize, the key question that distinguishes black-box testing (BBT) from white-box testing (WBT) is the “perspective” question:

* ***Perspective:*** BBT views the objects of testing as a black-box while focusing on testing the input-output relations or external functional behavior; while WBT views the objects as a glass-box where internal implementation details are visible and tested. BBT and WBT can also be compared by the way in which they address the following questions:
* *Objects:* Although the objects tested may overlap occasionally, WBT is generally used to test small objects, such as small software products or small units of large software products; while BBT is generally more suitable for large software systems or substantial parts of them as a whole.
* *Timeline:* WBT is used more in early sub-phases of testing for large software systems, such as unit and component testing, while BBT is used more in late sub-phases, such as system and acceptance testing.
* *Defect focus:* In BBT, failures related to specific external functions can be observed, leading to corresponding faults being detected and removed. The emphasis is on WBT, failures related to internal implementations can be observed, leading to corresponding faults being detected and removed directly. The emphasis is on reducing internal faults so that there is less chance for failures later on no matter what kind of application environment the software is subjected to.
* *Defect detection and fiing:* Defects detected through WBT are easier to **fix** than those through BBT because of the direct connection between the observed failures and program units and implementation details in WBT. However, WBT may miss certain types of defects, such as omission and design problems, which could be detected by BBT. In general BBT is effective in detecting and fixing problems of interfaces and interactions, while WBT is effective for problems localized within a small unit.
* *Techniques:* Various techniques can be used to build models and generate test cases to perform systematic BBT, and others can be used for WBT, with some of the same techniques being able to be used for both WBT and BBT. **A** specific technique is a BBT one if external functions are modeled; while the same technique can be a WBT one if internal implementations are modeled.
* *Tester:* BBT is typically performed by dedicated professional testers, and could also be performed by third-party personnel in a setting of IV&V (independent verification and validation); while WBT is often performed by developers themselves.

**COVERAGE-BASED VS. USAGE-BASED TESTING: WHEN TO STOP**

**TESTING?**

For most of the testing situations, the answer to the question “when to stop testing?’ depends on the completion of some pre-planned activities, coverage of certain entities, or whether a pre-set goal has been achieved. We next describe the use of different exit criteria and the corresponding testing techniques.

**When to stop testing?**

The question, “when to stop testing”, can be refined into two different questions:

* On a small or a local scale, we can ask: “When to stop testing for a specific test activity?’ This question is also commonly associated with different testing subphases.
* On a global scale, we can ask: “When to stop all the major test activities?” Because the testing phase is usually the last major development phase before product release, this question is equivalent to: “When to stop testing and release the product?’ These questions may yield different answers, leading us to different testing techniques and related activities. Without a formal assessment for decision making, decision to stop testing can usually be made in two general forms:
* *Resource-based criteria,* where decision is made based on resource consumptions. The most commonly used such stopping criteria are

- “Stop when you run out of time.”

- “Stop when you run out of money.”

Such criteria are irresponsible, as far as product quality is concerned, although they

may be employed if product schedule or cost are the dominant concerns for the

product in question.

* *Activity-based criteria,* commonly in the form:

- “Stop when you complete planned test activities.”

This criterion implicitly assumes the effectiveness of the test activities in ensuring the quality of the software product. However, this assumption could be questionable without strong historical evidence based on actual data from the project concerned. Because of these shortcomings, informal decisions without using formal assessments have very limited use in managing the testing process and activities for large software systems. We next examine exit criteria based on formal analyses and assessments. On the global level, the exit from testing is associated with product release, which determined the level of quality that a customer or a user could expect. In our overall software quality engineering process, this decision is associated with achieving quality goals, as well as achieving other project goals in the overall software development process. Therefore, the most direct and obvious way to make such product release decisions is the use of various reliability assessments. When the assessment environment is similar to the actual usage environment for customers. the resulting reliability measurement would be directly meaningful to these customers.

The basic idea in using reliability criterion is to set a reliability goal in the quality planning activity during product planning and requirement analysis, and later on to compare the reliability assessment based on testing data to see if this pre-set goal has been reached. If so, the product can be released. Otherwise, testing needs to continue and product release needs to be deferred. Various models exist today to provide reliability assessments and improvement based on data from testing,

One important implication of using this criterion for stopping testing is that the reliability assessments should be close to what actual users would expect, which requires that the testing right before product release resembles actual usages by target customers. This requirement resulted in the so-called usage-based testing. On the other hand, because of the large number of customers and usage situations, exhaustive coverage of all the customer usage scenarios, sequences, and patterns is infeasible. Therefore, an unbiased statistical sampling is the best that we can hope for, which results in usage-based statistical testing (UBST) that we will describe later in this section. Some specific techniques for such testing,

For earlier sub-phases of testing, or for stopping criteria related to localized test activities, reliability definition based on customer usage scenarios and frequencies may not be meaningful. For example, many of the internal components are never directly used by actual users, and some components associated with low usage frequencies may be critical for various situations. Under these situations, the use of reliability criterion may not be meaningful or may lead to inadequate testing of some specific components. Alternative exit criteria are needed. For example, as a rule of thumb:

“Products should not be released unless every component has been tested,”

Criteria similar to this have been adopted in many organizations to test their products and related components. We call these criteria coverage criteria, which involve coverage of some specific entities, such as components, execution paths, statements, etc. The use of coverage criteria is associated with defining appropriate coverage for different testing techniques, linking what was tested with what was covered in some formal assessments.

One implicit assumption in using coverage as the stopping criterion is that everything covered is defect free with respect to this specific coverage aspect, because all defects discovered by the suite of test cases that achieved this coverage goal would have been fixed or removed before product release. This assumption is similar to the one above regarding the effectiveness of test activities, when we use the completion of planned test activities as the exit criterion. However, this assumption is more likely to be enforced because specific coverage is closely linked to specific test cases.

From the quality perspective, the coverage criteria are based on the assumption that higher levels of coverage mean higher quality, and a specific quality goal can be translated into a specific coverage goal. However, we must realize that although there is a general positive correlation between coverage and quality, the relationship between the two is not a simple one. Many other factors need to be considered before an accurate quality assessment can be made based on coverage. For example, different testing techniques and sub-phases may be effective in detecting and removing different types of defects, leading to multistage reliability growth and saturation patterns (Horgan and Mathur, 1995). Nevertheless, coverage information gives us an approximate quality estimate, and can be used as the exit criterion when actual reliability assessment is unavailable, such as in the early sub-phases of testing.

**Usage-based statistical testing (UBST) and operational profiles (OPs)**

At one extreme, actual customer usage of software products can be viewed as a form of usage-based testing. When problems are experienced by these customers, some information about the problems, can be reported to software vendors, and integrated fixes can be constructed and delivered to all the customers to prevent similar problems from occurring. However, these post-product-release defect fixing activities could be very expensive because of the massive numbers of software installations. Frequent fixes could also damage the software vendor’s reputation and long-term business viability. The so-called beta test makes use of this usage-and-fix to the advantage of software vendors, through controlled software release so that these beta customers help software development organizations improve their software quality.

In general, if the actual usage, or anticipated usage for a new product, can be captured and used in testing, product reliability could be most directly assured. In usage-based statistical testing (UBST), the overall testing environment resembles the actual operational environment for the software product in the field, and the overall testing sequence, as represented by the orderly execution of specific test cases in a test suite, resembles the usage scenarios, sequences, and patterns of actual software usage by the target customers. Because the massive number of customers and diverse usage patterns cannot be captured in an exhaustive set of test cases, statistical sampling is needed, thus the term “statistical” in the descriptive title of this strategy. For the same reason, “random testing” and “usage-based random testing” are also used in literature. However, we prefer to use the term “usage-based statistical testing” in this book to avoid the confusion between random testing and “ad hoc” testing, where no systematic strategy is implied in “ad hoc” testing.

For practical implementation of such a testing strategy, actual usage information needs to be captured in various models, commonly referred to as “operational profiles” or OPs. Different OPs are associated with different testing techniques for UBST. Two primary types of usage models or OPs are:

* Flat OPs, or Musa OPs (Musa, 1993; Musa, 1998), which present commonly used operations in a list, a histogram, or a tree-structure, together with the associated occurrence probabilities. The main advantage of the flat OP is its simplicity, both in model construction and usage**.**
* Markov chain based usage models, or Markov OPs (Mills, 1972; Mills et al., 1987b; Whittaker and Thomason, 1994; Kallepalli and Tian, 2001; Tian et al., 2003), which present commonly used operational units in Markov chains, where the state transition probabilities are history independent (Karlin and Taylor, 1975). Complete operations can be constructed by linking various states together following the state transitions, and the probability for the whole path is the product of its individual transition probabilities. Markov models based on state transitions can generally capture navigation patterns better than flat OPs, but are more expensive to maintain and to use. This testing technique.

Usage-based statistical testing (UBST) is generally applicable to the final stage of testing, typically referred to as acceptance testing right before product release, so that stopping testing is equivalent to releasing the product. Other late sub-phases of testing, such as integration and system testing, could also benefit from the knowledge of actual customer usage situations to drive effective reliability improvement before product release. Naturally, the termination criterion used to stop such testing is achievement of reliability goals.

**Coverage and coverage-based testing (CBT)**

Most traditional testing techniques, either functional testing (BBT) orstructural testing (WBT), use various forms of test coverage as the stopping criteria. Thesimplest such criterion is in the form of completing various checklists, such as a checklistof major functions based on product specifications when BBT is used, or a checklist of allthe product components or all the statements when WBT is used. Testing can be performeduntil all the items on the respective checklist have been checked off or exhausted. Formost of the systematic testing techniques, some formal models beyond simple checklistsare used. Some specific examples of such models and related coverage include:

Formally defined partitions can be used as the basis for various testing techniques**,** which are similar to checklists but ensure:

- mutual exclusion of checklist items to avoid unnecessary repetition,

- complete coverage defined accordingly.

* **A** specialized type of partitions, input domain partitions into sub-domains, can alsobe used to test these sub-domains and related boundary conditions.
* Various programming or functional states can be defined and linked together to formfinite-state machines (FSMs) to model the system as the basis for various testing techniquesto ensure state coverage and coverage of related state transitionsand execution sequences.

The above FSMs can also be extended to analyze and cover execution paths and datadependencies through various testing techniques.

The generic steps and major sub-activities for CBT model construction and test preparationare described below:

* *Defining the model:* These models are often represented by some graphs, with individualnodes representing the basic model elements and links representing theinterconnections. Some additional information may be attached to the graph as linkor node properties (commonly referred to as weights in graph theory).
* *Checking individual model elements* to make sure the individual elements, such aslinks, nodes, and related properties, have been tested individually, typically in isolation,prior to testing using the whole model. This step also represents the self-checkingof the model, to make sure that the model captures what is to be tested.
* *Dejning coverage criteria:* Besides covering the basic model elements above, someother coverage criteria are typically used to cover the overall execution and interactions.For example, with the partition-based testing, we might want to cover theboundaries in addition to individual partitions; and for FSM-based testing, we mightwant to cover state transition sequences and execution paths.
* *Derive test cases:* Once the coverage criteria are defined, we can design our test casesto achieve them. The test cases need to be sensitized, that is, with its input valuesselected to realize specific tests, anticipated results defined, and ways to check theoutcomes planned ahead of time.

Model construction and test preparation are more closely linked to individual testingtechniques, which are described when each testing technique. The other major testing related activities, including test execution, measurement,analysis, and follow-up activities, are typically similar for all testing techniques.Coverage analysis plays an importantrole in guiding testing and coverage criterion is used to determine when to stop testing.Automated tool support by for this analysis and related data collection.

**Comparing CBT with UBST**

To summarize, the key questions that distinguish coverage-based testing (CBT) from usage based statistical testing (UBST) are the “perspective” question and the related stopping criteria:

* *Perspective:* UBST views the objects of testing from a user’s perspective and focuses on the usage scenarios, sequences, patterns, and associated frequencies or probabilities; while CBT views the objects from a developer’s perspective and focuses covering functional or implementation units and related entities.
* *Stopping criteria:* UBST use product reliability goals as the exit criterion; and CBT using coverage goals - surrogates or approximations of reliability goals - as the exit criterion. CBT and UBST can also be compared by the way in which they address the following questions:
* *Objects:* Although the objects tested may overlap, CBT is generally used to test and cover small objects, such as small software products, small units of large software products, or large systems at a high level of abstraction, such major functions or components; while UBST is generally more suitable for large software systems as a whole.
* *Verification vs. validation:* Although both CBT and UBST can be used for both verification test and validation test, UBST is more likely to be used for validation test because of their relationship to customers and users.
* *Timeline:* For large software systems, CBT is often used in early sub-phases of testing, such as unit and component testing, while UBST is often used in late sub-phases of testing, such as system and acceptance testing.
* *Defect detection:* In UBST, failures that are more likely to be experienced by users are also more likely to be observed in testing, leading to corresponding faults being detected and removed for reliability improvement. In CBT, failures are more closely related to things tested, which may lead to effective fault removal but may not be directly linked to improved reliability due to different exposure ratios for software faults.
* *Testing environment:* UBST uses testing environment similar to that for in-field operation at customer installations; while CBT uses environment specifically set up for testing.
* *Techniques:* Various techniques can be used to build models and generate test cases to perform systematic CBT. When these models are augmented with usage information, typically as the probabilities associated with checklist items, partitions, states, and state transitions, they can be used as models for UBST also. This is why we cover UBST models and techniques together with corresponding basic CBT models and techniques .
* *Customer and user roles:* UBST models are constructed with extensive customer and user input; while CBT models are usually constructed without active customer or user input. UBST is also more compatible with the customer and user focus in today’s competitive market.
* *Tester:* Dedicated professional testers typically perform UBST; while CBT can be performed by either professional testers or by developers themselves.

**4. TEST ACTIVITIES, MANAGEMENT, AND**

**AUTOMATI0N**

**1. TEST PLANNING AND PREPARATION**

Test planning and preparation is the most important activity in the generic testing process for systematic testing based on formal models. Most of the key decisions about testing are made during this stage. In this section, we first examine what key questions need to be answered in the high-level test planning, and then examine individual low-level activities related to test preparation. Test planning and test preparation are sometimes treated as separate groups of activities (Black, 2004).

**1.1 Test planning: Goals, strategies, and techniques**

The high-level task for test planning is to set goals and to determine a general testing strategy. This high-level decision should be based on answers to several key questions, particularly the objectives or goals of testing under a specific environment. The answers to these questions not only determine the general types of testing to perform, but also determine the test termination or exit criteria. Overall environment needs to be considered because the environmental constraints imposed on testing also affect the choice of testing strategies.

Most of the testing we cover in this book focuses on the correctness aspect of quality. If the software is complete or nearly complete, then the above correctness-centered quality goals can be directly translated into reliability goals, which, in turn, requires us to use usage-based statistical testing. Sometimes, these quality goals can be translated indirectly into coverage goals, which can be achieved by black-box testing for the whole system. However, if only individual units and pieces are available, we might choose to focus on the implementation details and perform coverage-based white-box testing.

Therefore, we set an overall testing strategy by making the following decisions:

* *Overall objectives and goals,* which can be refined into specific goals for specific testing. Some specific goals include *reliability* for usage-based statistical testing or *coverage* for various traditional testing techniques.
* *Objects to be tested and the specific focus:* Functional testing views the software product as a black-box and focuses on testing the external functional behavior; while structural testing views the software product or component as a (transparent) white box and focuses on testing the internal implementation details.

Once the overall testing strategy has be selected, we can plan to allocate resources and staff to implement it. The available staff and resources also affect the specific models and techniques that can be used to implement the strategy. For example, simple models based on checklists and partitions generally require less resources and prior knowledge by the testing staff, while more complex formal models and related testing techniques may require more resources and expertise. Different models and techniques are also associated with different effectiveness levels or different applicability to different types of problems and situations. Consequently, appropriate testing models and related techniques can be selected to optimize some form of cost-benefit measure.

Sometimes, existing models or test suites can be used with some minor modifications or adaptations, which would require minimal additional effort in test planning and preparation. Nevertheless, the above high-level activities still need to be carried out to arrive at this decision, because indiscriminately using exiting testing strategies, techniques, models, and test suites may not fulfill the need for the new situation and end up merely wasting valuable time and resources. In what follows, we focus on the situation where new models, procedures, and test cases need to be considered in testing planning and preparation. The situation of minor adaptations in connection with regression testing as a specialized type or testing.

**1.2 Testing models and test cases**

Different models are closely linked to different testing techniques, and the modeling details can only be described together with their corresponding techniques, as we will do in Chapters 8 through 11. However, some generic steps and activities are involved in test model construction. as follows:

1. *Information source identification and data collection:* The information and data are generally affected by both what is required by specific models and what is available in the project environment. For example, in usage-based statistical testing, information about actual in-field or anticipated usage by target customers needs to be gathered to construct operational profiles as the basis of testing; while in white-box unit testing, the tested unit provides the information source which can be analyzed to construct our testing models.

*2. Analysis and initial model construction:* The information and data collected above are analyzed to construct testing models. Expertise and familiarity with the specific testing techniques and models are required for people who perform this task. This step is typically the hardest to automate because of the human intelligence and expertise required.

*3. Model validation and incremental improvement:* This is an important step, particularly for large objects or for functions or usages associated with external customers. Iterative procedure might be necessary to fix inaccuracies and other problems found in the initial model or early versions of the candidate models.

Once the testing models have been constructed and validated, they can be used to generate test cases, which can then be executed by following some planned test procedure. First, we need to define and distinguish the static test cases and the dynamic test runs, as follows:

* **A** *test case* is a collection of entities and related information that allows a test to be executed or a test run to be performed.
* **A** *test* ***run,*** is a dynamic unit of specific test activities in the overall testing sequence on a selected testing object.

Each time a static test case is invoked, we have an individual dynamic test run. Therefore, each test case can correspond to multiple test runs. In some literature and organizations, each test run is also called an *attempt.*

The information included for a test case must enable the related test run to start, continue, and finish. For most of the testing situations, the starting and finishing points correspond to the initiation and termination of the operations for the whole software system, such as the compilation of a program when the compiler is tested. But there are exceptions, such as in operating systems and telecommunication systems, where continuous operation without stopping is the expected norm. In these cases, because the specific test is an activity associated with finite time for practical purposes, the starting and finishing points need to be artificially inserted, resulting in a subsection of the system execution as a test run.

Essential among the test case information is the specific input to the software object in operation, which includes both the initial input at the start of the test run and the input to allow it to continue and to finish. In addition, the test case often includes information about the expected output throughout the test run, which, together with the specific input and timing information, defines the program behavior under this test run. Such input, output, and timing information can be captured by the set of input variables, the set of output variables, and their values over time. With the above understanding, we can view the construction of a specific test case as assigning its input values over a planned test run, which is referred to as test *sensitization* in testing literature. This assignment is typically derived from testing models we constructed in the previous step of test planning and preparation. Different criteria and steps may be involved in test sensitization when different testing techniques are used.

**1.3 Test suite preparation and management**

The collection of individual test cases that will be run in a test sequence until some stopping criteria are satisfied is called a *test suite.* Test suite preparation involves the construction and allocation of individual test cases in some systematic way based on the specific testing techniques used. For example, when usage-based statistical testing is planned, the test case allocation will be determined by the operational profiles (OPs) constructed as the testing models, in proportion to individual usage probabilities. Similarly, when coverage-based testing is planned, the specific coverage criteria would dictate the allocation of test cases. For example, in control flow testing not involving loops, the exact number of test cases is determined by the number of paths for all-path coverage.

Another way to obtain a test suite is through reuse of test cases for earlier versions of the same product. This kind of testing is commonly referred to as *regression testing.* It ensures that common functionalities are still supported satisfactorily in addition to satisfactory performance of new functionalities. Special types of formal models are typically used to make the selection from existing test cases, in connection to regression testing.

In general, all the test cases should form an integrated suite, regardless of their origins, how they are derived, and what models are used to derive them. Sometimes, the test suite may evolve over time and its formation may overlap with the actual testing. In fact, in some testing techniques, test cases can be constructed dynamically, or “on-the-fly”, during test execution. But even for such testing, some planning of the test cases and test suite is still necessary, at least to determine the specific method for dynamic test case construction and the precise stopping criteria. For most of the testing techniques we cover in this book, a significant part of test preparation must be done before actual testing starts.

In general, test cases cost time, effort, and expertise to be obtained, and are too valuable to be thrown away. It is worthwhile to spend some addition effort and resource to save them, organize them, and manage them as a test suite for easy reuse in the future. Test suite management includes managing the collection of both the existing test cases and the newly constructed ones. At a minimum, some consistent database for the test suite needs to be kept and shared by people who are working on similar areas. Some personnel information can also be kept in the test suite, such as the testers who designed specific test cases, to better supported future use of this test suite. The information contained in the test suite constitutes an indexed database with important information about individual test cases in the test suite, as well as pointers to actual test cases. The actual test cases, in turn, contains more detailed information about the exact scenario, test input, expected output and behavior, etc.

There are many ways to organize the test suite or test suites. The most common way is to organize them by sub-phases, because of the different objects, objectives, concerns,perspectives, priorities, and the testing techniques used. Various attributes can be used to describe, classify, and organize individual test cases in the suite. One concrete example is the use of the following attributes for an IBM product in its system testing phase (Tian, 1998):

* sc - scenario class
* *sn*- scenario number
* *vn*- variation number with a particular scenario

The scenario class sc corresponds to high-level functional areas or groups of functions. Within each sc, the scenario number *sn,* and the variation number *vn*within each ***sn,*** form a three-layer hierarchical organization of test cases in the suite. In addition, *sn*and ***vn***are generally ordered in rough correspondence to the expected execution order, ranging from **1** to 99, with consecutive numbers used up to a point and then skipping to 99 to indicate some ad hoc test cases - those do not fall into some systematic sequence. Therefore, less than 99 scenarios or variations within scenarios are allowed, which was more than adequate for the product tested.

**1.4 Preparation of test procedure**

In addition to preparation of individual test cases and the overall test suite, the test procedure also needs to be prepared for effective testing. The basic question is the sequencing of the individual test cases and the switch-over from one test run to another. Several concerns affect the specific test procedure to be used, including:

* ***Dependencies*** among individual test cases. Some test cases can only be run after others because one is used to set up another. This is particularly true for systems that operate continuously, where the later test run may need to start at a state set up by the earlier one.
* ***Defect detection*** related sequencing. Many problems can only be effectively detected after others have been discovered and fixed. For example, integration of several components and related testing typically focus on interface and interaction problems, which can be masked by problems in individual components. Therefore, these components need to be individually tested before integration testing starts.
* **Sequences to avoid *accidents****.* For some systems, possibly severe problems and damages may incur during testing if certain areas were not checked through related test runs prior to the current test run. For example, in embedded software for safety critical systems, one does not want to start testing safety features before testing other related functions first. This can be considered as a special case of the problem or defect related sequencing where there is a very strong economical incentive for preferring certain sequencing to others.
* ***Problem diagnosis*** related sequencing. Some execution problems observed during testing may involve complicated scenarios and many possible sources of problems. Under this situation, related test runs focused on a single aspect or limited areas can be used to help with the problem diagnosis. Better yet, if such complicated problems are expected, we should run related simpler test cases first to eliminatecertain possibilities and narrow down the problem areas. Therefore, one natural sequence for test case execution commonly used in practical testing procedures is to progress from simple and easy ones to complicated and difficult ones. The same idea has been used in defining coverage hierarchies.
* ***Natural grouping*** of test cases, such as by functional and structural areas or by usage frequencies, can also be used for test sequencing and to manage parallel testing. However, among areas where no such order exists, or when the incentive for following a certain order is not strong, we can carry out testing for them in parallel to speed up the testing process. In fact, this is what people do all the time for large-scale software testing, where parallelism and interleaving are common.

The key to test run transition in the test procedure preparation is to make sure that the next test run can start right after the current one is finished for each software installment. This consideration may place some additional requirements on individual test cases, either requiring them to leave the system in the same initial condition or in some specified final condition. In fact, the initial and final states of specific test cases can also be used to group individual test cases in the test suite. This is similar to the grouping of test cases when system configuration and environmental setup are considered in defining the operational mode in usage-based testing using Musa’s operational profiles (Musa, 1998).

When test cases are derived dynamically, test procedure would naturally involve much more dynamic elements. However, the above considerations for test procedure preparation should still be incorporated in the corresponding test procedure. In this case, not only the execution but also the generation of dynamic test cases is affected by the dependency, effectiveness and efficiency concerns.

A related topic to test procedure preparation is the assignment of people to perform certain tests. Their roles and responsibilities need to be clearly specified. In addition, allocation of time and other resources also needs to be planned ahead of time before test execution starts, in accordance with test case grouping and allocation within a test suite. One specific type of resources is the test automation tools which could significantly reduce the time, staffing, and other resources required for test execution.

**2. TEST EXECUTION, RESULT CHECKING, AND MEASUREMENT**

The key to the overall test execution is the smooth transition from one test run to another, which also requires us to allocate all the required resources to ensure that individual test runs can be started, executed, and finished, and related problems can be handled seamlessly.

General steps in test execution include:

1. Allocating test time and resources;

2. Invoking and running tests, and collecting execution information and measurements;

**3.** Checking testing results and identifying system failures.

The step to invoke and run tests is fairly straightforward with well-prepared test cases or already sensitized test cases. We can simply provide the input variable values over the whole execution duration as required and as already precisely specified in these test cases. The sequence of test runs can follow the pre-planned sequence.

In the case where test cases are generated dynamically, such as in various usage-based statistical testing approaches, much of the work we described in terms of test sensitization needs to be done at this stage. The key in handling failed test runs is to ensure that they will not block the execution of other test cases. In addition, there will be test runs related to the re-verification of fixed problems, which can be treated much the same way as other planned test cases except the newly added dependency and its impact on test sequencing: Before an integrated fix becomes available, the test case that triggered the failure observation in the first place and other closely related test cases should be suspended to avoid repeatedly observing the same failure, which adds little new information to what is already known. The same test case can be re-run after the fix is in, and closely related test cases can also continue at this point. By doing so, we avoided unnecessary repetitions/re-runs, thus improving the overall test efficiency.

Test time and resources allocation is most closely related to the test planning and preparation activities described in the previous section. Although the allocation could be planned or even carried out at the previous stage, the monitoring, adjustment, and management of these resources need to be carried out during test execution. Test time allocation and management are closely related to people’s roles and responsibility in carrying out specific testing activities. Managing other test resources primarily involves the environmental set up and related facility management. For pure software systems, this is fairly straightforward, with the environment setup to include the hardware configuration and software environment that the finished product will operate within. Sometimes, limited number of simulation programs or hardware simulators can be used for testing some product components, but the overall system testing would very much resemble the actual operational environment. Once the general system configuration is decided, the facility management is mainly the allocation and monitoring of testing time on these facilities.

For embedded software systems or for heterogeneous systems with important software components, the environment and facility management issues involve the so-called “super system”. Coordination between different branches is a major issue where people have different perspectives and concerns. In addition, various techniques, such as simulation and prototyping techniques, will be used to aid testing or sometimes to replace part of the testing. We will also see some specific techniques to deal with interface, interaction, and interoperability problems among different sub-systems as part of the safety assurance program.

**Result checking: The oracle problem**

Result checking, or the ***oracle problem,*** and the related failure identification is a difficult task, because of both the theoretical difficulties and practical limitations. In this book, we use the term ***test oracle*** to indicate any means to check the testing result. Long standing theoretical results state that result checking for testing in general is an undecidable problem.

In other words, there is no hope for algorithmic or fully automated solution to the general test oracle problem. On the practical side, the expected behavior can hardly be precisely described so that the observed behavior can be compared against. Combined with the fact that software can fail in innumerable variations, the unexpected behavior can happen in truly unexpected ways, thus making result checking difficult or nearly impossible. However, there are cases where specific types of system failures, such as irresponsive behavior or system crash, are easy to identify. In other cases, various other means, such and/or execution states, etc., can be used to help us find approximate solutions to the oracle problem, as described below:

* Sometimes heuristics guesses can be used based on product domain knowledge, for example, what other similar products would do under similar situations. Consequently, similar products, such as previous releases of the same product or competitors’ products with similar functionality, can often be used as the test oracle, to check execution results and to identify system failures.
* Knowledge of implementation details can also be used to link specific behavior to specific program units. We can also examine various product internal information and dynamic execution state to help solve the oracle problem. For example, if an external function is supported by some internal components, and these internal components were not invoked when we test for this external function, we can be almost certain there is something wrong with this test run. In addition, product experts or developers themselves can also help testers to perform this difficult task when some important problem is suspected, making effective use of these people’s product knowledge.
* Various types of consistency checking during execution, such as checking for the database consistency, can also help us determine the execution failures. Such consistency checking can usually be done through sampling of some dynamic states and related product internal information, which could be analyzed either on-line during the test execution, or off-line with the detailed dynamic execution information.

**Test measurement**

Observed failures need to be recorded and tracked until their resolution. This defect handling or defect tracking process is typically considered part of the testing process itself, and the reporting of the observed failures is consider part the test execution activity. However, the handling of defects discovered during testing is not fundamentally different from that of defects discovered during other QA activities, as we described in the general context of QA activities.

Detailed information about failure observations and the related activities is needed for problem diagnosis and defect fixing. Some specific information for failures and faults also includes various generic information about defects, covering defect type, severity, impact areas, possible cause, when-injected, etc. This information could be collected either when the failure were observed and recorded or when the faults were fixed, or even afterward. When failures are not observed, the measurement of related test runs can be used to demonstrate product reliability or correct handling of input and dynamic situations.

**Table1. A** template for test execution measurements

|  |
| --- |
| * *rid* - run identification, consisting of   - **sc** - scenario class,  - *sn* - scenario number,  - *vn* - variation number with a particular scenario,  - *an* - attempt number for the specific scenario variation   * *timing* - start time *tO* and end time *tl* * *tester* - the tester who attempted the test run * *trans* - transactions handled by the test run * ***result*** - result of the test run (1 indicates success and 0 for failure) |

Various other measurements can be taken during test execution for further analysis and follow-up actions. Successful executions also need to be recorded for various purposes, including documentation of test activities, possible use as oracle to check future execution results, etc. This is particularly important for regression testing and for legacy products that are expected to change and evolve over the whole product lifespan. The timing and other related information can be important, when it can be used as input in analysis and follow-up activitiesand in reliability analysis. In addition to the “on-line” measurement of the dynamic test runs and related failure information, the corresponding static test cases can be measured “off-line” to avoid interference with normal test execution. Various other information could also be collected, such as testing personnel, environment, configuration, test object, etc**. Table1** is an example template for test execution information collected for an IBM product during system testing (Tian, 1995). Notice that a test run here corresponds to a specific *attempt* in the hierarchically organized test suite.Each attempt or test run, numbered *an,* is drawn from a specific variation, with variation number *vn,* of a scenario numbered as *sn* that belong to a specific scenario class sc. Other information about individual test runs, such as timing, tester, workload measured in transactions, and the run result, is also recorded.

**3. ANALYSIS AND FOLLOW-UP**

The third group of major test activities is analysis and follow-up after test execution. The measurement data collected during test execution, together with other data about the testing and the overall environment, form the data input to these analyses, which, in turn, provide valuable feedback to test execution and other testing and development activities. Direct follow-up includes defect fixing and making other management decisions, such as product release and transition from one development phase or sub-phase to another. We examine these issues in this section.

**Analysis and follow-up based on individual testing runs**

Analysis of individual test runs includes result checking and failure identification we covered in the previous section as part of the test execution activities. When failures are identified, additional analyses are normally performed by software developers or “code owners” to diagnose the problem and locate the faults that caused the failures for defect removal. This activity may involve the following steps:

*Understanding* the problem by studying the execution record, particularly those involving failures.

Being able to ***recreate*** the same problem scenario and observe the same problem. This is important to confirm the problem and rule out possibilities of transient problems due to environmental disturbances or user errors. It also provides input to diagnose the problem causes.

***Problem diagnosis*** to examine what kind of problem it is, where, when, and possible causes. This may involve analyzing the above records and using some diagnostic tools or addition test runs to zoom in on possible causes or to eliminate other possibilities.

***Fault locating,*** to identify the exact location(s) of fault(s) based on information from the previous steps and product knowledge.

***Defect Jixing,*** to fix the located fault(s) by adding, removing, or correcting certain parts of the code. Sometimes, design and requirement changes could also be triggered or propagated from the above changes due to logical linkage among the different software components.

Once an integrated fix is available, the failed test cases were re-run to verify the fix. If successful, the normal test execution continues; otherwise, another round of defect fixing as described above is again initiated.

**Analysis and follow-up for overall testing results**

Various analyses can be performed on the overall testing results and related data to provide various assessments about testing, and to drive follow-up activities, including:

***Reliability analysis*** for usage-based testing, which can be used to assess current product reliability and as input to determine if the pre-set reliability goal has been achieved. If so, product release or test termination decisions can be made. If not, future reliability as well as time and resources needed to reach the reliability goal can be estimated. Sometimes, low reliability areas can be identified for focused testing and reliability improvement. This analysis and its many uses in follow-up activities.

***Coverage analysis*** for coverage-based testing, which can be used as a surrogate for reliability and used as the stopping criterion or as input to various decisions related to testing management. Specifics about this are presented when specific testing techniques and coverage hierarchies.

***Overall defect analysis,*** which can be used to examine defect distribution and to identify high-defect areas for focused remedial actions. In addition, some product internal measurements, such as size and complexity of individual components, and other measurements can also be used together with defect data to identify high-defect areas for focused quality improvement.

These analyses about overall testing results and related follow-up activities, in connection with the overall analysis and feedback for all **QA** alternatives. Possible test process and overall development process improvement based on these and other analyses and feedback is also described therein.

**4. ACTIVITIES, PEOPLE, AND MANAGEMENT**

In this section, we examine people’s roles and responsibilities in specific test activities as well as the related management issues.

**People’s role in informal vs. formal, testing**

Informal software testing and some types of formal testing could involve minimal prior knowledge of the software products or systems. One simple way to test the software is to just run it and observe its behavior. Some obvious problems can be easily recognized by people with almost no prior knowledge of computer software and software products.

Some formal forms of testing, such as usability testing, can be performed with little prior knowledge as well. For example, to test some user-friendly, “plug-and-play” software products, novice users are often asked to start using the products. Their behavior and their difficulties in using the products are observed and related information is recorded for usability assessment and improvement. In this scenario, the testing involves the actual novice users as testers, but it may also involve experienced testers who observe and record the testing information. With automated information recording, the role of the experienced tester in this situation can be eliminated.

Because of the above situations, many people have the wrong perception that testing is “easy”, and any “warm body” can perform testing. This misconception also contributes to various problems related to software management, where the least experienced and skilled people are assigned to testing groups. This problem can be corrected by a good knowledge of the technical skills and experience involved in testing, and through some organizational initiatives, such as creating a well-established and well respected career path for testers (Weyuker et al., 2000).

For the large and complex software systems used in society today, any hope of assured quality needs to be supported by testing beyond informal ad hoc testing. We need to model the software systems, their operational environment, their users and usage scenarios, sequences, and patterns, so that systematic testing can be performed to ensure that these systems satisfy their customers’ quality expectations. Test cases can be derived from these models and used systematically to detect and fix defects and to ensure software quality and reliability. All these activities are performed by individual testers or testing teams.

Various other development personnel also need to be involved in testing activities. For example, as part of the follow-up activity to testing, problems detected during testing need to be resolved by the people who are responsible for the creation of the product design or code. Therefore, software developers, particularly those designers and programmers whose code is tested, also need to be involved in testing, although mostly indirectly to follow up on failure observations and defect fixing.

Sometimes, people can play the dual role of developers and testers, when they test their own code, such as in the unit or component testing sub-phases. However, for the overall system, professional testers are typically employed to testing the integration of different components and the overall operation of the system as a whole in the integration, system, and acceptance testing sub-phases.

**Testing teams: Organization and management**

The test activities need to be managed by people with a good understanding of the testing techniques and processes. The feedback derived from analyses of measurement data needs to be used to help with various management decisions, such as product release, and to help quality improvement.

Test managers are involved in these activities. Testers and testing teams can be organized into various different structures, but basically following either a horizontal or a vertical model:

* ***A vertical model*** would organize around a product, where dedicated people perform one or more testing tasks for the product. For example, one or more teams can perform all the different types of testing for the product, from unit testing up to acceptance testing.
* ***A horizontal model*** is used in some large organizations so that a testing team only performs one kind of testing for many different products within the organization. For example, different products may share the same system testing team.

Depending on the demand for testers by different projects, staffing level may vary over time. In the vertical model, as the product development shifts from one phase to another or as the development focus shifts from one area to another, project personnel could be reassigned to perform different tasks. One common practice in industry is to use programmers to perform various testing tasks when testing phase peaks. This practice may create various problems related to staffing management: If not done carefully, it may also lead to project delays, as in Brooks’ famous observation that adding people to a late project will make it later (Brooks, 1995). The mismatch between people’s expertise and their assignments may also result in more defects passing through the testing phase to cause additional in-field problems. This fact is part of the reason for people to adopt the horizontal model where staffing level variations can generally be better managed due to the different schedules and demands by different projects.

In reality, a mixture of the two is often used in large software organizations (Tian, **1998),** with low-level testing performed by dedicated testers or testing teams, system testing shared across similar products, and general project support provided by a centralized support unit for the entire organization. The general project support includes process, technology, and tool support necessary for formal development and testing. This centralized support unit resembles the so-called experience factory that also packages experience and lessons learned from development for more effective future use (Basili, 1995). The idea of experience factory is connection to defect prevention based on process improvement.

**External participants: Users and third-party testers**

Besides the above internal participants, external participants or users may also be involved in testing. The concept of users can also be expanded to include non-human users of software as well, such as other software and hardware environments that the software product in question interacts with (Whittaker, 2001). This extended user concept is particularly relevant to embedded systems or heterogeneous systems with extensive software components.

In general, the users’ views and perspectives, their usage scenarios, sequences, and patterns, and the overall operational environment need to be captured in some models and used in testing to ensure satisfactory performance and reliability for the software products. This is particularly true for usage-based statistical testing, where active user participant is essential in model construction. Sometimes, the users can even serve informally as testers, such as in the usability testing example earlier.

For certain types of software systems, such as those used in defense industry or government, independent verification and validation (IV&V) model is extensively used, where software systems are independently tested or verified and validated using various techniques by third-party participants. This model has gained popularity for various other types of high-assurance software systems, where high reliability, high integrity, or other properties are required, resulting in the so-called certification model or certification pipeline (Voas, 1999).

Another reason for IV&V’s popularity is the increasing use and focus on software development using COTS (commercial-off-lhe shelf) components and CBSE (component based software engineering, or CBSD - component-based software development). In such paradigms, independent testing and certification of software components or reusable parts are key to the possible selection, use, and adoption of software components, parts, or subsystems.

**5. TEST AUTOMATION**

Test automation aims to automate some manual tasks with the use of some software tools. The demand for test automation is strong, because purely manual testing from start to finish can be tedious and error-prone. On the other hand, long standing theoretical results tell us that no fully automated testing is possible. Even most of the major sub-activities, such as result checking or the oracle problem, are undecidable.

However, some level of automation for individual activities is possible, and can be supported by various commercial tools or tools developed within large organizations. The key in the use of test automation to relieve people of tedious and repetitive tasks and to improve overall testing productivity is to first examine what is possible, feasible, and economical, and then to set the right expectations and goals. Various issues related to test automation include:

* specific needs and potential for automation;
* selection of existing testing tools, if available;
* possibility and cost of constructing specific test automation tools;
* availability of user training for these tools and time/effort needed;
* overall cost, including costs for tool acquisition, support, training, and usage;
* impact on resource, schedule, and project management. We examine test automation in connection with the major test activities and people’s roles and responsibilities in them.

**Automation for test execution**

Among the three major test activities, preparation, execution, and follow-up, execution is a prime candidate for automation. In fact, this is the area in which the earliest test automation tools found some unequivocal successes. For example, various semi-automatic debugging tools or debuggers allow testers to set and reset variable values and execution states during execution and observe the dynamic execution behavior at different observation points. These tools are semi-automatic because testers are still involved in test execution intervention.

Many of the modern test automation tools can be considered as enhanced debuggers that work for larger products, automate more individual testing activities, and are generally more flexible and more tailorable than earlier debuggers. Various automated task sequencing tools for job transfer from one test run to another work in much the same way as job dispatcher/scheduler in various operating systems. In fact, most such test run sequencing tools are platform-specific, and are often constructed within testing organizations using some system utilities or APIs (application program interfaces).

An additional functionality for many of the test automation tools is to allow information recording and collection. For example, in the testing of some commercial software product in IBM (Tian et al., 1995; Tian, 1998), an internal test automation tool called T3 was used to generate workload, monitor the execution, and record various execution details for a subset of test scenario classes listed in Table 7.1. The specific measurement data that need to be collected are dictated by the specific analyses to be performed. Therefore, we cover test measurement tools in conjunction with analysis tools later.

**Automation for test planning and preparation**

In test planning and preparation, the potential for automation is different for different sub-activities. The overall planning part can only be carried out by experienced personal with expertise in planning and management as well as a good understanding of testing and development technologies. Not much automation can be achieved in these sub-activities, nor is there a high demand for automation here. Similarly, test procedure planning is primarily done by experts, although the planned procedure can be later enforced and automated during actual test execution with the help of various test execution automation tools.

Test case preparation is the area where there is some realistic potential for automation. For example, in testing of legacy products, various automated analysis can be performed to compare the current version of the product with its previous versions, and to screen the existing test suites to select the ones for regression testing. For construction of new test cases, automation is also possible. For example, in the T3 tool we mentioned above for test execution support, a script can be provided to generate different workload for testing, which effectively generates test cases and related test runs dynamically from test script. However, the test scripts, which are high-level descriptions of what to test, need to be constructed in the first place by the experienced testers. These test scripts are usually much simpler and shorter, thus much less costly to generate than actual test cases. Consequently, a semi-automated test case generation is supported in this case.

In general, test scripts or test cases are based on some formal models. The model construction for different test techniques requires high levels of human intelligence and expertise, and is therefore not easily automated. However, some individual steps in model construction can be automated, such as some automated data gathering, graphical or other aids for modeling, etc. For small-scale programs, some tools can be used to generate certain models and test cases directly, much like using compilers to generate object code from source code. However, these tools cannot scale up to large software systems. In addition, in most of the models, various decisions need to be made and parameters need to be selected for specific model variations, which can only be carried out by people with proper expertise.

Once such a model is constructed, various tools can be used to generate test cases automatically. Sometimes, even if a tool is not directly available, the testing model is typically associated with some algorithms that can be at least partially implemented for automatic generation of at least some test cases. For example, once an underlying usage model in the form of a Markov chain is constructed, several algorithms can be used for usage based statistical testing to cover frequently used usage patterns, frequently visited states, and call-pairs (Avritzer and Weyuker, 1995). If there is a commercial tool or an existing tool within the organization available, the key in its adoption for test case generation is to understand what kind of model is supported and how difficult it is to construct models of this type, in order to match it with our purpose of testing.

**Automation for test measurement, analysis, and follow-up**

In terms of analyses of test results and follow-up actions, the situation is similar to test planning and preparation. Most of the follow-up actions involve problem fixing and various other remedial and improvement initiatives, very little of which can be automated. However, specific analysis activities can be supported by various analysis and modeling tools. For example, many of the reliability analysis activities described in Tian (1998) were automated. This was achieved after many rounds of studies that converged on the appropriate models and data to use. Many popular tools were discarded because they were found to be unsuitable for the type of commercial products froni IBM. This experience told us that automated analysis tools should not be indiscriminately applied, but rather based on intelligent choice based on one's own specific environment and experience. **A** general tool support strategy for QA and development process measurement, data analysis, and follow-up.

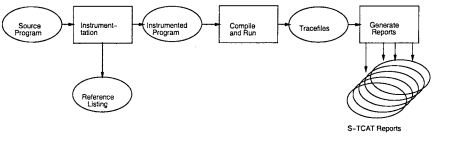
Closely related to test result analyses a.re coverage analysis for coverage-based testing and reliability analysis for usage-based testing. For traditional reliability analysis, we typically need results for individual test ru.ns and related timing information (Lyu, 1995a). Sometimes, some additional information, such as test input, environment, and personnel information can also be used to link input states to reliability or to identify problematic areas for focused reliability improvement (Tian, 1995). These data can usually be automatically collected using various test execution tools or dedicated data collection tools.

Coverage analysis usually involves the use of more detailed information and measurement data than that for reliability analysis. But, fortunately, various coverage analysis tools exist to collect coverage information during test execution. For example, several popular commercial test tools collect and analyze coverage information to provide feedback and quality assessments to the testing process or the overall development process, including:

* McCabe Test from McCabe and Associates provides control flow coverage related information and analysis.

S-TCAT (System Test Coverage Analysis Tool) from SRI (Software Research, Inc.) provides function-level coverage and call-pair information. S-TCAT can also be integrated into a tool suite called Testworks from SRI for various other testing purposes. ATAC (Automatic Test Analysis for C) from Telecodia is a data flow coverage analysis tool for white-box testing.

To use these tools for coverage analysis, the source code is usually instrumented to build an instrumented test driver. When this instrumented code is run in testing, information related to coverage in the form of raw data is collected. Later on, the raw data are analyzed for coverage. Figure 7.1 illustrates these steps with the use of S-TCAT for test coverage analysis. Each of these steps is usually automated by the tools themselves.



**Figure.**Test coverage analysis with S-TCAT

One interestingfact to notice is that, although these tools are designed for coverage-based testing, they can also be used sometimes to support usage-based testing, such as the use of S-TCAT to collect customer usage information for some IBM products (Lu and Tian, 1993b).