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

**Testing: Why?**

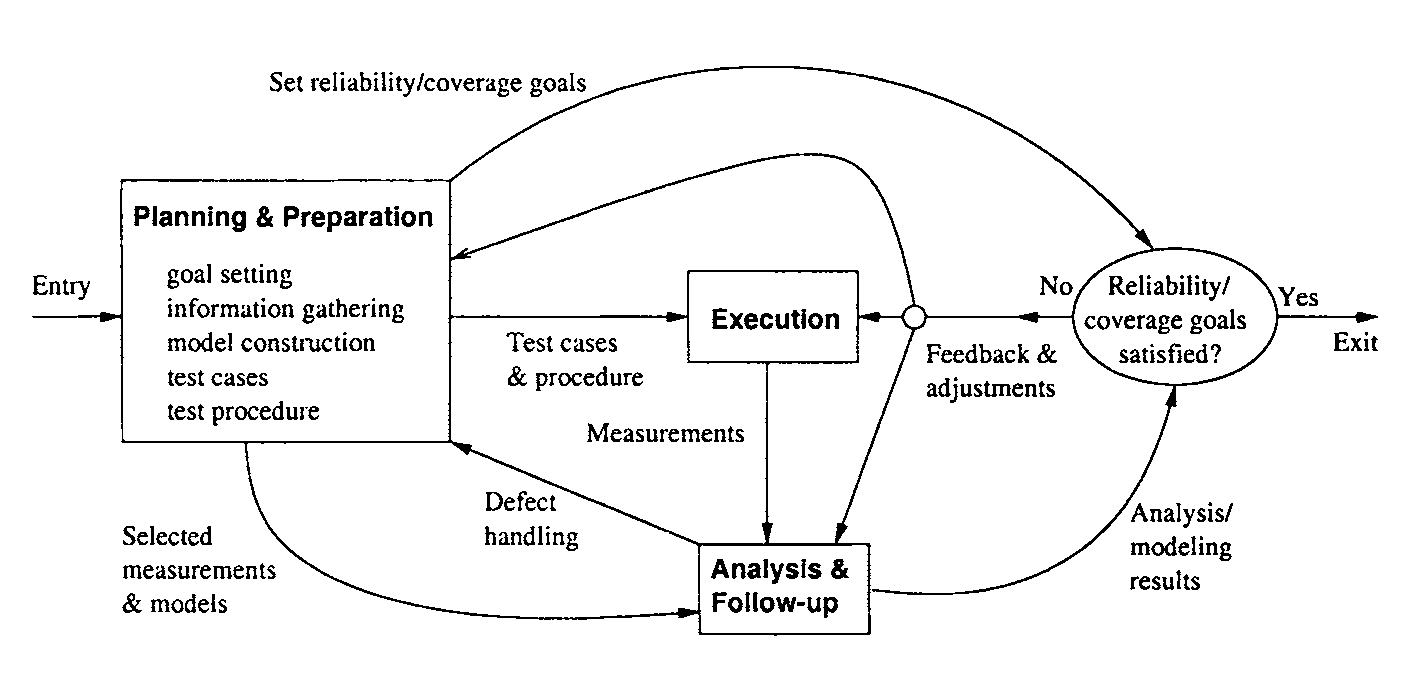
The purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users. The most natural way to show this fulfillment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in 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.