CS1016 – SOFTWARE TESTING

L T P C
3 0 0 3

UNIT I TESTING BASICS **8**

Testing as an engineering activity – Role of process in software quality – Testing as a process – Basic definitions – Software testing principles – The tester's role in a software development organization – Origins of defects – Defect classes – The defect repository and test design – Defect examples – Developer / tester support for developing a defect repository.

UNIT II TEST CASE DESIGN **11**

Introduction to testing design strategies – The smarter tester – Test case design strategies – Using black box approach to test case design – Random testing – Equivalence class partitioning – Boundary value analysis – Other black box test design approaches – Black box testing and COTS – Using white box approach to test design – Test adequacy criteria – Coverage and control flow graphs – Covering code logic – Paths – Their role in white box based test design – Additional white box test design approaches – Evaluating test adequacy criteria.

UNIT III LEVELS OF TESTING **9**

The need for levels of testing – Unit test – Unit test planning – Designing the unit tests – The class as a testable unit – The test harness – Running the unit tests and recording results – Integration tests – Designing integration tests – Integration test planning – System test – The different types – Regression testing – Alpha, beta and acceptance tests.

UNIT IV TEST MANAGEMENT **9**

Basic concepts – Testing and debugging goals and policies – Test planning – Test plan components – Test plan attachments – Locating test items – Reporting test results – The role of three groups in test planning and policy development – Process and the engineering disciplines – Introducing the test specialist – Skills needed by a test specialist – Building a testing group.

UNIT V CONTROLLING AND MONITORING **8**

Defining terms – Measurements and milestones for controlling and monitoring – Status meetings – Reports and control issues – Criteria for test completion – SCM – Types of reviews – Developing a review program – Components of review plans – reporting review results. **Total: 45**

TEXT BOOKS

1. Ilene Burnstein, "Practical Software Testing", Springer International Edition, 2003.
2. Edward Kit, "Software Testing in the Real World – Improving the Process", Pearson Education, 1995.

REFERENCES

1. Elfriede Dustin, "Effective Software Testing", Pearson Education, 2003.
2. Renu Rajani and Pradeep Oak, "Software Testing – Effective Methods, Tools and Techniques", Tata McGraw Hill, 2003.

UNIT I TESTING BASICS

1.1 Testing as an engineering activity

This is an exciting time to be a software developer. Software systems are becoming more challenging to build. They are playing an increasingly important role in society. People with software development skills are in demand. New methods, techniques, and tools are becoming available to support development and maintenance tasks. Because software now has such an important role in our lives both economically and socially, there is pressure for software professionals to focus on quality issues. Poor quality software that can cause loss of life or property is no longer acceptable to society. Failures can result in catastrophic losses. Conditions demand software development staffs with interest and training in the areas of software product and process quality. Highly qualified staff ensure that software products are built on time, within budget, and are of the highest quality with respect to attributes such as reliability, correctness, usability, and the ability to meet all user requirements.

Using an engineering approach to software development implies that:

- the development process is well understood;
- projects are planned;
- life cycle models are defined and adhered to;
- standards are in place for product and process;
- measurements are employed to evaluate product and process quality;
- components are reused;
- validation and verification processes play a key role in quality determination;
- engineers have proper education, training, and certification.

1.2 Role of process in software quality

The need for software products of high quality has pressured those in the profession to identify and quantify quality factors such as usability, testability, maintainability, and reliability, and to identify engineering practices that support the production of quality products having these favorable attributes. Among the practices identified that contribute to the development of high-quality software are project planning, requirements management, development of formal specifications, structured design with use of information hiding and encapsulation, design and code reuse,inspections and reviews, product and process measures, education and training of software professionals, development and application of CASE tools, use of effective testing techniques, and integration of testing activities into the entire life cycle. In addition to identifying these individual best technical and managerial practices, software researchers realized that it was important to integrate them within the context of a high-quality software development process.

Process, in the software engineering domain, is the set of methods, practices, standards, documents, activities, policies, and procedures that software engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents, code, and manuals.

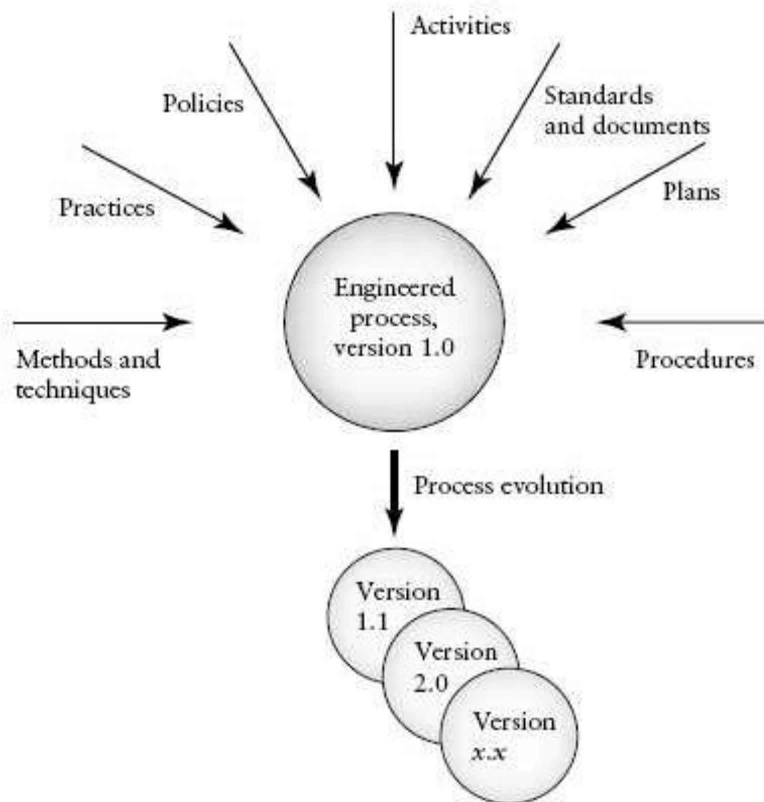


FIG. 1.2

Components of an engineered process.

It also was clear that adding individual practices to an existing software development process in an ad hoc way was not satisfactory. The software development process, like most engineering artifacts, must be engineered. That is, it must be designed, implemented, evaluated, and maintained. As in other engineering disciplines, a software development process must evolve in a consistent and predictable manner, and the best technical and managerial practices must be integrated in a systematic way. These models allow an organization to evaluate its current software process and to capture an understanding of its state. Strong support for incremental process improvement is provided by the models, consistent with historical process evolution and the application of quality principles. The models have received much attention from industry, and resources have been invested in process improvement efforts with many successes recorded.

All the software process improvement models that have had wide acceptance in industry are high-level models, in the sense that they focus on the software process as a whole and do not offer adequate support to evaluate and improve specific software development sub processes such as design and testing. Most software engineers would agree that testing is a vital component of a quality software process, and is one of the most challenging and costly activities carried out during software development and maintenance.

1.3 Testing as a process

The software development process has been described as a series of phases, procedures, and steps that result in the production of a software product. Embedded within the software development process are several other processes including testing. Some of these are shown in Figure 1.3. Testing itself is related to two other processes called verification and validation as shown in Figure 1.3.

Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements.

Validation is usually associated with traditional execution-based testing, that is, exercising the code with test cases.

Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [11].

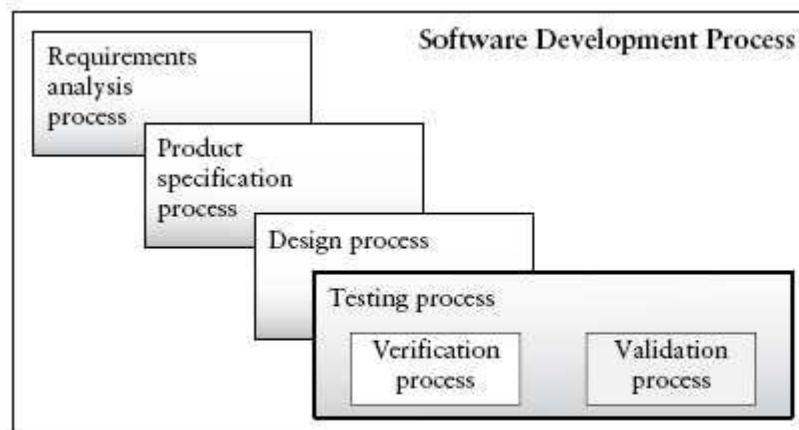


FIG. 1.3

Example processes embedded in the software development process.

Verification is usually associated with activities such as inspections and reviews of software deliverables. Testing itself has been defined in several ways. Two definitions are shown below.

Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software.

Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

Note that these definitions of testing are general in nature. They cover both validation and verification activities, and include in the testing domain all of the following: technical reviews, test planning, test tracking, test case design, unit test, integration test, system test, acceptance test, and usability test. The definitions also describe testing as a dual-purpose process—one that reveals defects, as well as one that is used to evaluate quality attributes of the software such as reliability, security, usability, and correctness.

Also note that testing and debugging, or fault localization, are two very different activities. The debugging process begins *after* testing has been carried out and the tester has noted that the software is not behaving as specified.

Debugging, or fault localization is the process of (1) locating the fault or defect, (2) repairing the code, and (3) retesting the code.

Testing as a process has economic, technical and managerial aspects. Economic aspects are related to the reality that resources and time are available to the testing group on a limited basis. In fact, complete testing is in many cases not practical because of these economic constraints. An organization must structure its testing process so that it can deliver software on time and within budget, and also satisfy the client's requirements. The technical aspects of testing relate to the techniques, methods, measurements, and tools used to insure that the software under test is as defect-free and reliable as possible for the conditions and constraints under which it must operate. Testing is a process, and as a process it must managed. Minimally that means that an organizational policy for testing must be defined and documented. Testing procedures and steps must be defined and documented. Testing must be planned, testers should be trained, the process should have associated quantifiable goals that can be measured and monitored. Testing as a process should be able to evolve to a level where there are mechanisms in place for making continuous improvements.

1.4 Basic definitions

Errors

An error is a mistake, misconception, or misunderstanding on the part of a software developer.
In the category of developer we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a design notation, or a programmer might type a variable name incorrectly.

Faults (Defects)

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

Faults or defects are sometimes called “bugs.” Use of the latter term trivializes the impact faults have on software quality. Use of the term “defect” is also associated with software artifacts such as requirements and design documents. Defects occurring in these artifacts are also caused by errors and are usually detected in the review process.

Failures

A failure is the inability of a software system or component to perform its required functions within specified performance requirements **.**

During execution of a software component or system, a tester, developer, or user observes that it does not produce the expected results. In some cases a particular type of misbehavior indicates a certain type of fault is

Test case

A test case in a practical sense is a test-related item which contains the following information:
1. *A set of test inputs.* These are data items received from an external source by the code under test. The external source can be hardware, software, or human.
2. *Execution conditions.* These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
3. *Expected outputs.* These are the specified results to be produced by the code under test.

Test

A test is a group of related test cases, or a group of related test cases and test procedures**.**

Test Oracle

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

A program, or a document that produces or specifies the expected outcome of a test, can serve as an oracle.Examples include a specification (especially one that contains pre- and post conditions), a design document, and a set of requirements. Other sources are regression test suites. The suites usually contain components with correct results for previous versions of the software. If some of the functionality in the new version overlaps the old version, the appropriate oracle information can be extracted. A working trusted program can serve as its own oracle in a situation where it is being ported to a new environment. In this case its intended behavior should not change in the new environment.

Test Bed

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.This includes the entire testing environment, for example, simulators, emulators,memory checkers, hardware probes, software tools, and all other items needed to support execution of the tests.

Software Quality

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.
2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

In order to determine whether a system, system component, or process is of high quality we use what are called quality attributes. the degree to which they possess a given quality attribute with quality metrics.

Quality metric

A metric is a quantitative measure of the degree to which a system, system component,or process possesses a given attribute.

There are product and process metrics. A very commonly used example of a software product metric is software size, usually measured in lines of code (LOC). Two examples of commonly used process metrics are costs and time required for a given task. Quality metrics are a special kind of metric.

A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute.

Some examples of quality attributes with brief explanations are the following:

correctness—the degree to which the system performs its intended function
reliability—the degree to which the software is expected to perform its required functions under stated conditions for a stated period of time
usability—relates to the degree of effort needed to learn, operate, prepare input, and interpret output of the software
integrity—relates to the system's ability to withstand both intentional and accidental attacks
portability—relates to the ability of the software to be transferred from one environment to another
maintainability—the effort needed to make changes in the software
interoperability—the effort needed to link or couple one system to another.

Another quality attribute that should be mentioned here is testability.
1. the amount of effort needed to test the software to ensure it performs according to specified requirements (relates to number of test cases needed),
2. the ability of the software to reveal defects under testing conditions (some software is designed in such a way that defects are well hidden during ordinary testing conditions).
Testers must work with analysts, designers and, developers throughout the software life system to ensure that testability issues are addressed.

Software Quality Assurance Group

The software quality assurance (SQA) group in an organization has ties to quality issues. The group serves as the customers' representative and advocate. Their responsibility is to look after the customers' interests.

The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software conforms to established technical requirements.

Review

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

The composition of a review group may consist of managers, clients,developers, testers and other personnel depending on the type of artifact under review. A special type of review called an audit is usually conducted by a Software Quality Assurance group for the purpose of assessing compliance with specifications, and/or standards, and/or contractual agreements.

1.5 Software testing principles

Principles play an important role in all engineering disciplines and are usually introduced as part of an educational background in each branch of engineering. Figure 1.1 shows the role of basic principles in various engineering disciplines. Testing principles are important to test specialists/ engineers because they provide the foundation for developing testing knowledge and acquiring testing skills. They also provide guidance for defining testing activities as performed in the practice of a test specialist.A principle can be defined as:

1. a general or fundamental, law, doctrine, or assumption;
2. a rule or code of conduct;
3. the laws or facts of nature underlying the working of an artificial device.

Extending these three definitions to the software engineering domain we can say that software engineering principles refer to laws, rules, or doctrines that relate to software systems, how to build them, and how they behave. In the software domain, principles may also refer to rules or codes of conduct relating to professionals who design, develop, test, and maintain software systems. Testing as a component of the software engineering discipline also has a specific set of principles that serve as guidelines for the tester. They guide testers in defining how to test software systems, and provide rules of conduct for testers as professionals. Glenford Myers has outlined such a set of *execution-based* testing principles in his pioneering book, *The Art of Software Testing* [9]. Some of these principles are described below. Principles 1-8, and 11 are derived directly from Myers' original set. The author has reworded these principles, and also has made modifications to the original set to reflect the evolution of testing from an art, to a quality-related process within the context of an engineering discipline. Note that the principles as stated below only relate to execution-based testing. Principles relating to reviews, proof of correctness, and certification as testing activities are not covered.

Principle 1. Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.

Software engineers have made great progress in developing methods to prevent and eliminate defects. However, defects do occur, and they have a negative impact on software quality. Testers need to detect these defects before the software becomes operational. This principle supports testing as an execution-based activity to detect defects. It also supports the separation of testing from debugging since the intent of the latter is to locate defects and repair the software. The term "software component" is used in this context to represent any unit of software ranging in size and complexity from an individual procedure or method, to an entire software system. The term "defects" as used in this and in subsequent principles represents any deviations in the software that have a negative impact on its functionality, performance, reliability, security, and/or any other of its specified quality attributes.

Principle 2. When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yetundetected defect(s).

Principle 2 supports careful test design and provides a criterion with which to evaluate test case design and the effectiveness of the testing effort when the objective is to detect defects. It requires the tester to consider the goal for each test case, that is, which specific type of defect is to be detected by the test case. In this way the tester approaches testing in the same way a scientist approaches an experiment. In the case of the scientist there is a hypothesis involved that he/she wants to prove or disprove by means of the experiment. In the case of the tester, the hypothesis is related to the suspected occurrence of specific types of defects. The goal for the
test is to prove/disprove the hypothesis, that is, determine if the specific defect is present/absent. Based on the hypothesis, test inputs are selected, correct outputs are determined, and the test is run. Results are analyzed to prove/disprove the hypothesis. The reader should realize that many resources are invested in a test, resources for designing the test cases, running the tests, and recording and analyzing results. A tester can justify the expenditure of the resources by careful test design so that principle 2 is supported.

Principle 3. Test results should be inspected meticulously.

Testers need to carefully inspect and interpret test results. Several erroneous and costly scenarios may occur if care is not taken. For example: A failure may be overlooked, and the test may be granted a "pass" status when in reality the software has failed the test. Testing may continue based on erroneous test results. The defect may be revealed at some later stage of testing, but in that case it may be more costly and difficult to locate and repair.

• A failure may be suspected when in reality none exists. In this case the test may be granted a "fail" status. Much time and effort may be spent on trying to find the defect that does not exist. A careful reexamination of the test results could finally indicate that no failure has occurred.

• The outcome of a quality test may be misunderstood, resulting in unnecessary rework, or oversight of a critical problem.

Principle 4. A test case must contain the expected output or result.

It is often obvious to the novice tester that test inputs must be part of a test case. However, the test case is of no value unless there is an explicit statement of the expected outputs or results, for example, a specific variable value must be observed or a certain panel button that must light up. Expected outputs allow the tester to determine (i) whether a defect has been revealed, and (ii) pass/fail status for the test. It is very important to have a correct statement of the output so that needless time is not spent due to misconceptions about the outcome of a test. The specification of test inputs and outputs should be part of test design activities. In the case of testing for quality evaluation, it is useful for quality goals to be expressed in quantitative terms in the requirements document if possible, so that testers are able to compare actual software attributes as determined by the tests with what was specified.

Principle 5. Test cases should be developed for both valid and invalid input conditions.

A tester must not assume that the software under test will always be provided with valid inputs. Inputs may be incorrect for several reasons. For example, software users may have misunderstandings, or lack information about the nature of the inputs. They often make typographical errors even when complete/correct information is available. Devices may also provide invalid inputs due to erroneous conditions and malfunctions. Use of test cases that are based on invalid inputs is very useful for revealing defects since they may exercise the code in unexpected ways and identify unexpected software behavior. Invalid inputs also help developers and testers evaluate the robustness of the software, that is, its ability to recover when unexpected events occur (in this case an erroneous input). Principle 5 supports the need for the independent test group called for in Principle 7 for the following reason. The developer of a software component may be biased in the selection of test inputs for the component and specify only valid inputs in the test cases to demonstrate that the software works correctly. An independent tester is more apt to select invalid inputs as well.

Principle 6. The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.

What this principle says is that the higher the number of defects already detected in a component, the more likely it is to have additional defects when it undergoes further testing. For example, if there are two components A and B, and testers have found 20 defects in A and 3 defects in B, then the probability of the existence of additional defects in A is higher than B. This empirical observation may be due to several causes. Defects often occur in clusters and often in code that has a high degree of complexity and is poorly designed. In the case of such components developers and testers need to decide whether to disregard the current version of the component and work on a redesign, or plan to expend additional testing resources on this component to insure it meets its requirements. This issue is especially important for components that implement mission or safety critical functions.

Principle 7. Testing should be carried out by a group that is independent of the development group.

This principle holds true for psychological as well as practical reasons. It is difficult for a developer to admit or conceive that software he/she has created and developed can be faulty. Testers must realize that (i) developers have a great deal of pride in their work, and (ii) on a practical level it may be difficult for them to conceptualize where defects could be found. Even when tests fail, developers often have difficulty in locating the defects since their mental model of the code may overshadow their view of code as it exists in actuality. They may also have misconceptions or misunderstandings concerning the requirements and specifications relating to the software. The requirement for an independent testing group can be interpreted by an organization in several ways. The testing group could be implemented as a completely separate functional entity in the organization. Alternatively, testers could be members of a Software Quality Assurance Group, or even be a specialized part of the development group, but in the latter case especially, they need the capability to be objective. Reporting management that is separate from development can support their objectivity and independence. As a member of any of these groups, the principal duties and training of the testers should lie in testing rather than in development. Finally, independence of the testing group does not call for an adversarial relationship between developers and testers. The testers should not play "gotcha" games with developers. The groups need to cooperate so that software of the highest quality is released to the customer.

Principle 8. Tests must be repeatable and reusable.

Principle 2 calls for a tester to view his/her work as similar to that of an experimental scientist. Principle 8 calls for experiments in the testing domain to require recording of the exact conditions of the test, any special events that occurred, equipment used, and a careful accounting of the results. This information is invaluable to the developers when the code is returned for debugging so that they can duplicate test conditions. It is also useful for tests that need to be repeated after defect repair. The repetition and reuse of tests is also necessary during regression test (the retesting of software that has been modified) in the case of a new release of the software. Scientists expect experiments to be repeatable by others, and testers should expect the same!

Principle 9. Testing should be planned.

Test plans should be developed for each level of testing, and objectives for each level should be described in the associated plan. The objectives should be stated as quantitatively as possible. Plans, with their precisely specified objectives, are necessary to ensure that adequate time and resources are allocated for testing tasks, and that testing can be monitored and managed. Test planning activities should be carried out throughout the software life cycle (Principle 10). Test planning must be coordinated with project planning. The test manager and project manager must work together to coordinate activities. Testers cannot plan to test a component on a given date unless the developers have it available on that date. Test risks must be evaluated. For example, how probable are delays in delivery of software components, which components are likely to be

complex and difficult to test, do the testers need extra training with new tools? A test plan template must be available to the test manager to guide development of the plan according to organizational policies and standards. Careful test planning avoids wasteful “throwaway” tests and unproductive and unplanned “test-patch-retest” cycles that often lead to poor-quality software and the inability to deliver software on time and within budget.

Principle 10. Testing activities should be integrated into the software life cycle.

It is no longer feasible to postpone testing activities until after the code has been written. Test planning activities as supported by Principle 10, should be integrated into the software life cycle starting as early as in the requirements analysis phase, and continue on throughout the software life cycle in parallel with development activities. In addition to test planning, some other types of testing activities such as usability testing can also be carried out early in the life cycle by using prototypes. These activities can continue on until the software is delivered to the users. Organizations can use process models like the V-model or any others that support the integration of test activities into the software life cycle [11].

Principle 11. Testing is a creative and challenging task [12].

Difficulties and challenges for the tester include the following:
• A tester needs to have comprehensive knowledge of the software engineering discipline.
• A tester needs to have knowledge from both experience and education as to how software is specified, designed, and developed.
• A tester needs to be able to manage many details.
• A tester needs to have knowledge of fault types and where faults of a certain type might occur in code constructs.
• A tester needs to reason like a scientist and propose hypotheses that relate to presence of specific types of defects.
• A tester needs to have a good grasp of the problem domain of the software that he/she is testing. Familiarly with a domain may come from educational, training, and work-related experiences.
• A tester needs to create and document test cases. To design the test cases the tester must select inputs often from a very wide domain.

1.6 The tester’s role in a software development organization

Testing is sometimes erroneously viewed as a destructive activity. The tester’s job is to reveal defects, find weak points, inconsistent behavior, and circumstances where the software does not work as expected. As a tester you need to be comfortable with this role. Given the nature of the tester’s tasks, you can see that it is difficult for developers to effectively test their own code (Principles 3 and 8). Developers view their own code as their creation, their “baby,” and they think that nothing could possibly be wrong with it! This is not to say that testers and developers are adversaries. In fact, to be most effective as a tester requires extensive programming experience in order to understand how code is constructed, and where, and what kind of, defects are likely to occur. Your goal as a tester is to work with the developers to produce high-quality

software that meets the customers' requirements. Teams of testers and developers are very common in industry, and projects should have an appropriate developer/tester ratio. The ratio will vary depending on available resources, type of project, and TMM level. For example, an embedded realtime system needs to have a lower developer/tester ratio (for example, 2/1) than a simple data base application (4/1 may be suitable). At higher TMM levels where there is a well-defined testing group, the developer/ tester ratio would tend to be on the lower end (for example 2/1 versus 4/1) because of the availability of tester resources. Even in this case, the nature of the project and project scheduling issues would impact on the ratio. In addition to cooperating with code developers, testers also need to work along side with requirements engineers to ensure that requirements are testable, and to plan for system and acceptance test (clients are also involved in the latter). Testers also need to work with designers to plan for integration and unit test. In addition, test managers will need to cooperate with project managers in order to develop reasonable test plans, and with upper management to provide input for the development and maintenance of organizational testing standards, polices, and goals. Finally, testers also need to cooperate with software quality assurance staff and software engineering process group members. In view of these requirements for multiple working relationships, communication and team working skills are necessary for a successful career as a tester. and marketing staff need to realize that testers add value to a software product in that they detect defects and evaluate quality as early as possible in the software life cycle. This ensures that developers release code with few or no defects, and that marketers can deliver software that satisfies the customers' requirements, and is reliable, usable, and correct. Low-defect software also has the benefit of reducing costs such as support calls, repairs to operational software, and ill will which may escalate into legal action due to customer dissatisfaction. In view of their essential role, testers need to have a positive view of their work. Management must support them in their efforts and recognize their contributions to the organization.

1.7 Origins of defects

The term *defect* and its relationship to the terms *error* and *failure* in the context of the software development domain has been discussed in Chapter 2. Defects have detrimental affects on software users, and software engineers work very hard to produce high-quality software with a low number of defects. But even under the best of development circumstances errors are made, resulting in defects being injected in the software during the phases of the software life cycle. Defects as shown in Figure 3.1 stem from the following sources [1,2]:

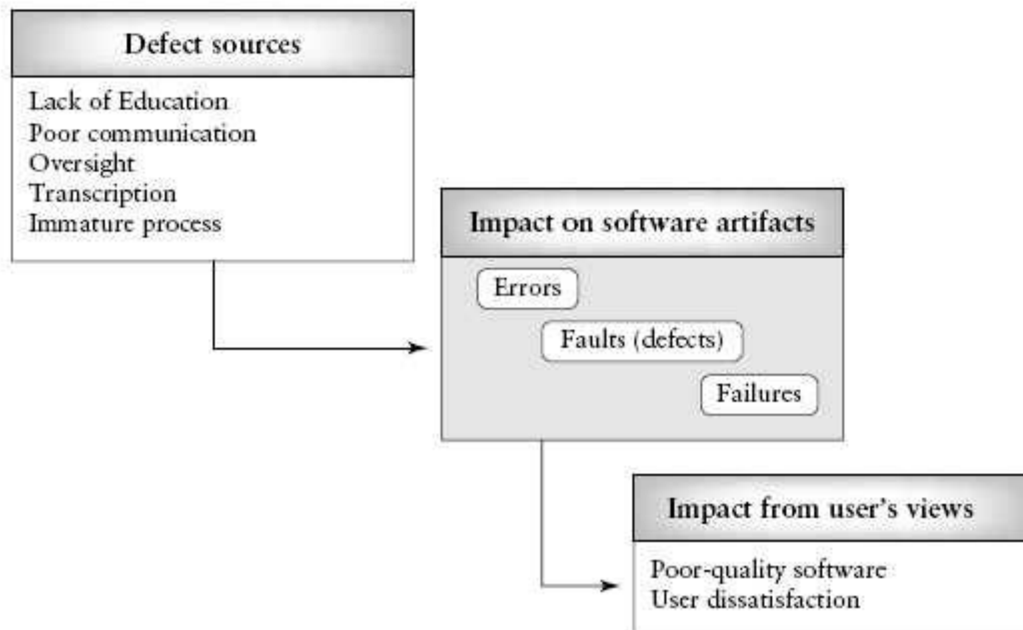


FIG. 3.1

Origins of defects.

1. *Education:* The software engineer did not have the proper educational background to prepare the software artifact. She did not understand how to do something. For example, a software engineer who did not understand the precedence order of operators in a particular programming language could inject a defect in an equation that uses the operators for a calculation.
2. *Communication:* The software engineer was not informed about something by a colleague. For example, if engineer 1 and engineer 2 are working on interfacing modules, and engineer 1 does not inform engineer 2 that a no error checking code will appear in the interfacing module he is developing, engineer 2 might make an incorrect assumption relating to the presence/absence of an error check, and a defect will result.
3. *Oversight:* The software engineer omitted to do something. For example, a software engineer might omit an initialization statement.
4. *Transcription:* The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.
5. *Process:* The process used by the software engineer misdirected her actions. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects.

When defects are present due to one or more of these circumstances, the software may fail, and the impact on the user ranges from a minor inconvenience to rendering the software unfit for use. Our goal as testers is to discover these defects preferably before the software is in operation.One of the ways we do this is by designing test cases that have a high probability of revealing defects. How do we develop these test cases? One approach is to think of software testing as an experimental activity. The results of the test experiment are analyzed to determine whether the software has behaved correctly. In this experimental scenario a tester develops hypotheses about possible defects (see Principles 2 and 9). Test cases are then designed based on the hypotheses. The tests are run and results analyzed to prove, or disprove, the hypotheses.

Myers has a similar approach to testing. He describes the successful test as one that reveals the presence of a (hypothesized) defect. He compares the role of a tester to that of a doctor who is in the process of constructing a diagnosis for an ill patient. The doctor develops hypotheses about possible illnesses using her knowledge of possible diseases, and the patients' symptoms. Tests are made in order to make the correct diagnosis. A successful test will reveal the problem and the doctor can begin treatment. Completing the analogy of doctor and ill patient, one could view defective software as the ill patient. Testers as doctors need to have knowledge about possible defects (illnesses) in order to develop defect hypotheses. They use the hypotheses to:

• design test cases;
• design test procedures;
• assemble test sets;
• select the testing levels (unit, integration, etc.)appropriate for the tests;
• evaluate the results of the tests.

A successful testing experiment will prove the hypothesis is true—that is, the hypothesized defect was present. Then the software can be repaired (treated).A very useful concept related to this discussion of defects, testing, and diagnosis is that of a fault model.

A fault (defect) model can be described as a link between the error made (e.g., a missing requirement, a misunderstood design element, a typographical error), and the fault/defect in the software.

Digital system engineers describe similar models that link physical defects in digital components to electrical (logic) effects in the resulting digital system [4,5]. Physical defects in the digital world may be due to manufacturing errors, component wear-out, and/or environmental effects.

The fault models are often used to generate a fault list or dictionary. From that dictionary faults can be selected, and test inputs developed for digital components. The effectiveness of a test can be evaluated in the context of the fault model, and is related to the number of faults as expressed in the model, and those actually revealed by the test. This view of test effectiveness (success) is similar to the view expressed by Myers stated above.

Although software engineers are not concerned with physical defects, and the relationships between software failures, software defects, and their origins are not easily mapped, we often use the fault model concept and fault lists accumulated in memory from years of experience to

design tests and for diagnosis tasks during fault localization (debugging) activities. A simple example of a fault model a software engineer might have in memory is “an incorrect value for a variable was observed because the precedence order for the arithmetic operators used to calculate its value was incorrect.” This could be called “an incorrect operator precedence order” fault. An error was made on the part of the programmer who did not understand the order in which the arithmetic operators would execute their operations. Some incorrect assumptions about the order were made.

The defect (fault) surfaced in the incorrect value of the variable. The probable cause is a lack of education on the part of the programmer. Repairs include changing the order of the operators or proper use of parentheses. The tester with access to this fault model and the frequency of occurrence of this type of fault could use this information as the basis for generating fault hypotheses and test cases. This would ensure that adequate tests were performed to uncover such faults.

In the past, fault models and fault lists have often been used by developers/ testers in an informal manner, since many organizations did not save or catalog defect-related information in an easily accessible form. To increase the effectiveness of their testing and debugging processes, software organizations need to initiate the creation of a defect database, or defect repository. The defect repository concept supports storage and retrieval of defect data from all projects in a centrally accessible location. A defect classification scheme is a necessary first step for developing the repository. The defect repository can be organized by projects and for all projects defects of each class are logged, along their frequency of occurrence, impact on operation, and any other useful comments. Defects found both during reviews and execution-based testing should be cataloged.

1.8 Defect classes, the defect repository and test design

Defects can be classified in many ways. It is important for an organization to adapt a single classification scheme and apply it to all projects. No matter which classification scheme is selected, some defects will fit into more than one class or category. Because of this problem, developers,testers, and SQA staff should try to be as consistent as possible when recording defect data. The defect types and frequency of occurrence should be used to guide test planning, and test design. Execution-based testing strategies should be selected that have the strongest possibility of detecting particular types of defects. It is important that tests for new and modified software be designed to detect the most frequently occurring defects. The reader should keep in mind that execution-based testing will detect a large number of the defects that will be described; however, software reviews as described in Chapter 10 are also an excellent testing tool for detection of many of the defect types that will be discussed in the following sections.

Defects, as described in this text, are assigned to four major classes reflecting their point of origin in the software life cycle—the development phase in which they were injected. These classes are: requirements/ specifications, design, code, and testing defects as summarized in Figure 3.2. It should be noted that these defect classes and associated subclasses focus on defects that are the major focus of attention to execution-based testers. The list does not include other defects types that are best found in software reviews, for example, those defects related to conformance to styles and standards. The review checklists in Chapter 10 focus on many of these types of defects.

2 .1.1 Requirements and Specification Defects

The beginning of the software life cycle is critical for ensuring high quality in the software being developed. Defects injected in early phases can persist and be very difficult to remove in later phases. Since many requirements documents are written using a natural language representation, there are very often occurrences of ambiguous, contradictory, unclear, redundant, and imprecise requirements. Specifications in many organizations are also developed using natural language representations, and these too are subject to the same types of problems as mentioned above.

However, over the past several years many organizations have introduced the use of formal specification languages that, when accompanied by tools, help to prevent incorrect descriptions of system behavior. Some specific requirements/specification defects are:

1 . Functional Description Defects

The overall description of what the product does, and how it should behave (inputs/outputs), is incorrect, ambiguous, and/or incomplete.

2 . Feature Defects

Features may be described as distinguishing characteristics of a software component or system.

Features refer to functional aspects of the software that map to functional requirements as described by the users and clients. Features also map to quality requirements such as performance and reliability. Feature defects are due to feature descriptions that are missing, incorrect, incomplete, or superfluous.

3 . Feature Interaction Defects

These are due to an incorrect description of how the features should interact. For example, suppose one feature of a software system supports adding a new customer to a customer database. This feature interacts with another feature that categorizes the new customer. The classification feature impacts on where the storage algorithm places the new customer in the database, and also affects another feature that periodically supports sending advertising information to customers in a specific category. When testing we certainly want to focus on the interactions between these features.

4 . Interface Description Defects

These are defects that occur in the description of how the target software is to interface with external software, hardware, and users. For detecting many functional description defects, black box testing techniques, which are based on functional specifications of the software, offer the best approach. In Chapter 4 the reader will be introduced to several black box testing techniques

such as equivalence class partitioning, boundary value analysis, state transition testing, and cause-and-effect graphing, which are useful for detecting functional types of detects. Random testing and error guessing are also useful for detecting these types of defects. The reader should note that many of these types of defects can be detected early in the life cycle by software reviews. Black box-based tests can be planned at the unit, integration, system, and acceptance levels to detect requirements/specification defects. Many feature interaction and interfaces description defects are detected using black box-based test designs at the integration and system levels.

D e s i g n D e f e c t s

Design defects occur when system components, interactions between system components, interactions between the components and outside soft ware/hardware, or users are incorrectly designed. This covers defects in the design of algorithms, control, logic, data elements, module interface descriptions, and external software/hardware/user interface descriptions.

When describing these defects we assume that the detailed design description for the software modules is at the pseudo code level with processing steps, data structures, input/output parameters, and major control structures defined. If module design is not described in such detail then many of the defects types described here may be moved into the coding defects class.

1 . Algorithmic and Processing Defects

These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. For example, the pseudo code may contain a calculation that is incorrectly specified, or the processing steps in the algorithm written in the pseudo code language may not be in the correct order. In the latter case a step may be missing or a step may be duplicated.

Another example of a defect in this subclass is the omission of error condition checks such as division by zero. In the case of algorithm reuse, a designer may have selected an inappropriate algorithm for this problem (it may not work for all cases).

2 . Control, Logic, and Sequence Defects

Control defects occur when logic flow in the pseudo code is not correct. For example, branching to soon, branching to late, or use of an incorrect branching condition. Other examples in this subclass are unreachable pseudo code elements, improper nesting, improper procedure or function calls. Logic defects usually relate to incorrect use of logic operators, such as less than (_), greater than (_), etc. These may be used incorrectly in a Boolean expression controlling a branching instruction.

3 . Data Defects

These are associated with incorrect design of data structures. For example, a record may be lacking a field, an incorrect type is assigned to a variable or a field in a record, an array may not have the proper number of elements assigned, or storage space may be allocated incorrectly.

Software reviews and use of a data dictionary work well to reveal these types of defects.

4 . Module Interface Description Defects
These are defects derived from, for example, using incorrect, and/or inconsistent parameter types, an incorrect number of parameters, or an incorrect ordering of parameters.

5 . Functional Description Defects
The defects in this category include incorrect, missing, and/or unclear design elements. For example, the design may not properly describe the correct functionality of a module. These defects are best detected during a design review.

6 . External Interface Description Defects
These are derived from incorrect design descriptions for interfaces with COTS components, external software systems, databases, and hardware devices (e.g., I/O devices). Other examples are user interface description defects where there are missing or improper commands, improper sequences of commands, lack of proper messages, and/or lack of feedback messages for the user.

Coding Defects

Coding defects are derived from errors in implementing the code. Coding defects classes are closely related to design defect classes especially if pseudo code has been used for detailed design. Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designers. Others may have transcription or omission origins. At times it may be difficult to classify a defect as a design or as a coding defect. It is best to make a choice and be consistent when the same defect arises again.

1 . Algorithmic and Processing Defects
Adding levels of programming detail to design, code-related algorithmic and processing defects would now include unchecked overflow and underflow conditions, comparing inappropriate data types, converting one data type to another, incorrect ordering of arithmetic operators (perhaps due to misunderstanding of the precedence of operators), misuse or omission of parentheses, precision loss, and incorrect use of signs.

2 . Control, Logic and Sequence Defects
On the coding level these would include incorrect expression of case statements, incorrect iteration of loops (loop boundary problems), and missing paths.

3 . Typographical Defects
These are principally syntax errors, for example, incorrect spelling of a variable name, that are usually detected by a compiler, self-reviews, or peer reviews.

4 . Initialization Defects
These occur when initialization statements are omitted or are incorrect. This may occur because of misunderstandings or lack of communication between programmers, and/or programmers and designers, carelessness, or misunderstanding of the programming environment.

5 . Data-Flow Defects

There are certain reasonable operational sequences that data should flow through. For example, a variable should be initialized, before it is usedin a calculation or a condition. It should not be initialized twice before there is an intermediate use. A variable should not be disregarded before it is used. Occurrences of these suspicious variable uses in the code may, or may not, cause anomalous behavior. Therefore, in the strictest sense of the definition for the term “defect,” they may not be considered as true instances of defects. However, their presence indicates an error has occurred and a problem exists that needs to be addressed.

6 . Data Defects

These are indicated by incorrect implementation of data structures. For example, the programmer may omit a field in a record, an incorrect type or access is assigned to a file, an array may not be allocated the proper number of elements. Other data defects include flags, indices, and constants set incorrectly.

7 . Module Interface Defects

As in the case of module design elements, interface defects in the code may be due to using incorrect or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters. In addition to defects due to improper design, and improper implementation of design, programmers may implement an incorrect sequence of calls or calls to nonexistent modules.

8 . Code Documentation Defects

When the code documentation does not reflect what the program actually does, or is incomplete or ambiguous, this is called a code documentation defect. Incomplete, unclear, incorrect, and outof-date code documentation affects testing efforts. Testers may be misled by documentation defects and thus reuse improper tests or design new tests that are not appropriate for the code. Code reviews are the best tools to detect these types of defects.

9 . External Hardware, Software Interfaces Defects

These defects arise from problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling, data exchanges with hardware, protocols, formats, interfaces with build files, and timing sequences (race conditions may result).

Many initialization, data flow, control, and logic defects that occur in design and code are best addressed by white box testing techniques applied at the unit (single-module) level. For example, data flow testing is useful for revealing data flow defects, branch testing is useful for detecting control defects, and loop testing helps to reveal loop-related defects. White box testing approaches are dependent on knowledge of the internal structure of the software, in contrast to black box approaches, which are only dependent on behavioral specifications. The reader will
be introduced to several white box-based techniques in Chapter 5. Many design and coding defects are also detected by using black box testing techniques. For example, application of decision tables is very useful for detecting errors in Boolean expressions. Black box tests as

described in Chapter 4 applied at the integration and system levels help to reveal external hardware and software interface defects. The author will stress repeatedly throughout the text that a combination of both of these approaches is needed to reveal the many types of defects that are likely to be found in software.

Testing Defects

Defects are not confined to code and its related artifacts. Test plans, test cases, test harnesses, and test procedures can also contain defects. Defects in test plans are best detected using review techniques.

1 . Test Harness Defects

In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code. Chapter 6 has a more detailed discussion of the need for this code. The test harness code should be carefully designed, implemented, and tested since it a work product and much of this code can be reused when new releases of the software are developed. Test harnesses are subject to the same types of code and design defects that can be found in all other types of software.

2 . Test Case Design and Test Procedure Defects

These would encompass incorrect, incomplete, missing, inappropriate test cases, and test procedures. These defects are again best detected in test plan reviews as described in Chapter 10. Sometimes the defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

1.10 Defect Examples: The Coin Problem

The following examples illustrate some instances of the defect classes that were discussed in the previous sections. A simple specification, a detailed design description, and the resulting code are shown, and defects in each are described. Note that these defects could be injected via one or more of the five defect sources discussed at the beginning of this chapter. Also note that there may be more than one category that fits a given defect. Figure 3.3 shown a sample informal specification for a simple program that calculates the total monetary value of a set of coins. The program could be a component of an interactive cash register system to support retail store clerks. This simple example shows requirements/ specification defects, functional description defects, and interface description defects.

The functional description defects arise because the functional description is ambiguous and incomplete. It does not state that the input, number_of_coins, and the output, number_of_dollars and number _of_cents, should all have values of zero or greater. The number_of_coins cannot be negative, and the values in dollars and cents cannot be negative in the real-world domain. As a consequence of these ambiguities and specification incompleteness, a checking routine may be omitted from the design, allowing the final program to accept negative values for the input

number_of_coins for each of the denominations, and consequently it may calculate an invalid value for the results. A more formally stated set of preconditions and postconditions would be helpful here, and would address some of the problems with the specification. These are also useful for designing black box tests.

A precondition is a condition that must be true in order for a software component to operate properly.

In this case a useful precondition would be one that states for example:number_of_coins __0
A postcondition is a condition that must be true when a software component completes its operation properly.
A useful postcondition would be:
number_of_dollars, number_of_cents __ 0.

Specification for Program calculate_coin_value

This program calculates the total dollars and cents value for a set of coins. The user inputs the amount of pennies, nickels, dimes, quarters, half-dollars, and dollar coins held. There are six different denominations of coins. The program outputs the total dollar and cent values of the coins to the user.

Inputs: `number_of_coins` is an integer
Outputs: `number_of_dollars` is an integer
`number_of_cents` is an integer

FIG. 3.3
A sample specification with defects.

In addition, the functional description is unclear about the largest number of coins of each denomination allowed, and the largest number of dollars and cents allowed as output values. Interface description defects relate to the ambiguous and incomplete description of user-software interaction. It is not clear from the specification how the user interacts with the program to provide input, and how the output is to be reported. Because of ambiguities in the user interaction description the software may be difficult to use. Likely origins for these types of specification defects lie in the nature of the development process, and lack of proper education and training. A poor-quality development process may not be allocating the proper time and resources to specification development and review. In addition, software engineers may not have the proper education and training to develop a quality specification. All of these specification defects, if not detected and repaired, will propagate to the design and coding phases. Black box testing techniques, which we will study in Chapter 4, will help to reveal many of these functional weaknesses. Figure 3.4 shows the specification transformed in to a design description. There are numerous design defects, some due to the ambiguous and incomplete nature of the specification; others are newly introduced. Design defects include the following:

Control, logic, and sequencing defects. The defect in this subclass arises from an incorrect "while" loop condition (should be less than or equal to six)

Algorithmic, and processing defects. These arise from the lack of error checks for incorrect and/or invalid inputs, lack of a path where users can correct erroneous inputs, lack of a path for recovery from input errors. The lack of an error check could also be counted as a functional design defect since the design does not adequately describe the proper functionality for the program.

```
Design Description for Program calculate_coin_values

Program calculate_coin_values
number_of_coins is  integer
total_coin_value is integer
number_of_dollars is integer
number_of_cents is integer
coin_values is array of  six integers representing
each coin value in cents
initialized to: 1,5,10,25,25,100
begin

initialize total_coin_value to zero
initialize loop_counter to one
while loop_counter is less then six
begin
       output "enter number of coins"
       read (number_of_coins )
       total_coin_value = total_coin_value +
       number_of_coins * coin_value[loop_counter]
       increment loop_counter
end
number_dollars =  total_coin_value/100
number_of_cents = total_coin_value - 100 * number_of_dollars
output (number_of_dollars, number_of_cents)
end
```

FIG. 3.4

A sample design specification with defects.

Data defects. This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read 1,5,10,25,50,100.

External interface description defects. These are defects arising from the absence of input messages or prompts that introduce the program to the user and request inputs. The user has no way of knowing in which order the number of coins for each denomination must be input, and when to stop inputting values. There is an absence of help messages, and feedback for user if he wishes to change an input or learn the correct format and order for inputting the number of coins. The output description and output formatting is incomplete. There is no description of what the outputs means in terms of the problem domain. The user will note that two values are output, but has no clue as to their meaning. The control and logic design defects are best addressed by white

box- based tests, (condition/branch testing, loop testing). These other design defects will need a combination of white and black box testing techniques for detection.
Figure 3.5 shows the code for the coin problem in a "C-like" programming language. Without effective reviews the specification and design defects could propagate to the code. Here additional defects have been introduced in the coding phase.

Control, logic, and sequence defects. These include the loop variable increment step which is out of the scope of the loop. Note that incorrect loop condition (*i* _ 6) is carried over from design and should be counted as a design defect.

Algorithmic and processin g defects. The division operator may cause problems if negative values are divided, although this problem could be eliminated with an input check.

Data Flow defects. The variable total_coin_value is not initialized. It is used before it is defined.
(This might also be considered a data defect.)

Data Defects. The error in initializing the array coin_values is carried over from design and should be counted as a design defect.

External Hardware, Software Interface Defects. The call to the external func tion "scanf" is incorrect. The address of the variable must be provided (&number_of_coins).

Code Documentation Defects. The documentation that accompanies this code is incomplete and ambiguous. It reflects the deficiencies in the external interface description and other defects that occurred during speci fication and design. Vital information is missing for anyone who will need to repair, maintain or reuse this code.

```
/*****************************************************************
program calculate_coin_values  calculates the dollar and cents
value of a set of coins of different dominations input by the user
denominations are pennies, nickels, dimes, quarters, half dollars,
and dollars
*****************************************************************/
main ()
{
int total_coin_value;
int number_of_coins = 0;
int number_of_dollars = 0;
int number_of-cents = 0;
int coin_values = {1,5,10,25,25,100};
{
int  i = 1;
while ( i < 6)
{
       printf("input number of coins\n");
       scanf ("%d", number_of_coins);
       total_coin_value = total_coin_value +
        (number_of_coins * coin_value{i]);
}
i = i + 1;
number_of_dollars =  total_coin_value/100;
number_of_cents = total_coin_value - (100 * number_of_dollars);
printf("%d\n", number_of_dollars);
printf("%d\n", number_of-cents);
}

/*****************************************************************/
```

FIG. 3.5

A code example with defects.

The control, logic, and sequence, data flow defects found in this example could be detected by using a combination of white and black box testing techniques. Black box tests may work well to reveal the algorithmic and data defects. The code documentation defects require a code review for detection. The external software interface defect would probably be caught by a good compiler.

The poor quality of this small program is due to defects injected during several of the life cycle phases with probable causes ranging from lack of education, a poor process, to oversight on the part of the designers and developers. Even though it implements a simple function the program is unusable because of the nature of the defects it contains. Such software is not acceptable to users; as testers we must make use of all our static and dynamic testing tools as described in subsequent chapters to ensure that such poor-quality software is not delivered to our user/client group. We must work with analysts, designers and code developers to ensure that quality issues are addressed early the software life cycle. We must also catalog defects and try to eliminate them by improving education, training, communication, and process.

1.11 Developer/Tester Support for Developing a Defect Repository

The focus of this chapter is to show with examples some of the most common types of defects that occur during software development. It is important if you are a member of a test organization to illustrate to management and your colleagues the benefits of developing a defect repository to store defect information. As software engineers and test specialists we should follow the examples of engineers in other disciplines who have realized the usefulness of defect data. A requirement for repository development should be a part of testing and/or debugging policy statements. You begin with development of a defect classification scheme and then initiate the collection defect data from organizational projects. Forms and templates will need to be designed to collect the data. Examples are the test incident reports as described in Chapter 7, and defect fix reports as described in Chapter 4. You will need to be conscientious about recording each defect after testing, and also recording the frequency of occurrence for each of the defect types. Defect monitoring should continue for each on-going project. The distribution of defects will change as you make changes in your processes. The defect data is useful for test planning, a TMM level 2 maturity goal. It helps you to select applicable testing techniques,
design (and reuse) the test cases you need, and allocate the amount of resources you will need to devote to detecting and removing these defects. This in turn will allow you to estimate testing schedules and costs.

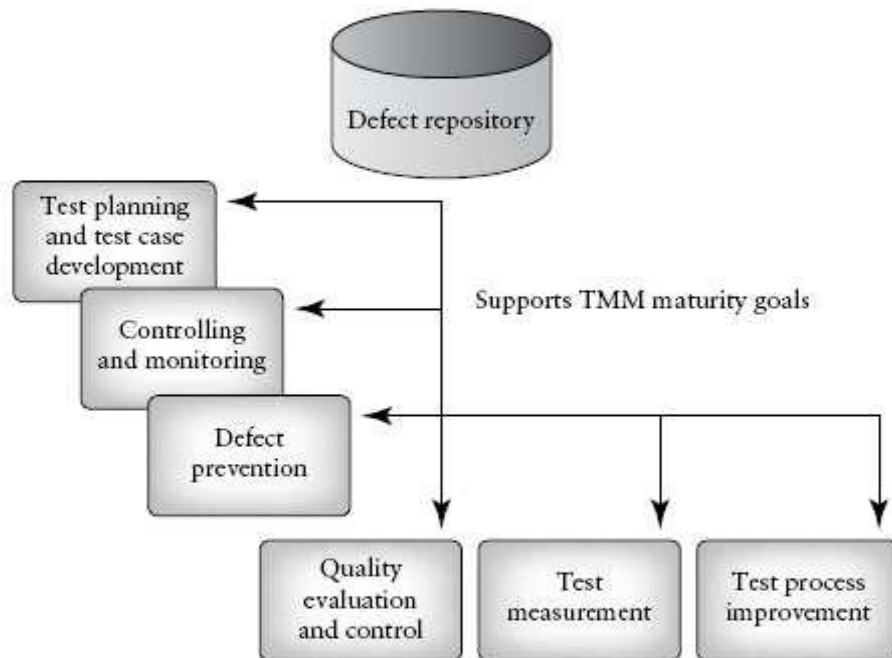


FIG. 3.6

The defect repository, and support for TMM maturity goals.

The defect data can support debugging activities as well. In fact, as Figure 3.6 shows, a defect repository can help to support achievement and continuous implementation of several TMM maturity goals including controlling and monitoring of test, software quality evaluation and control, test measurement, and test process improvement. Chapter 13 will illustrate the application of this data to defect prevention activities and process improvement. Other chapters will describe the role of defect data in various testing activities.

CS1016 - SOFTWARE TESTING

IMPORTANT QUESTIONS

Unit I
Part-A Questions

1. Compare Validation and Verification.
2. Define Software quality.
3. Define:Process
4. Define:Testing and debugging
5. Compare:Errors,faults and failures
6. Define:metrics
7. Define the role of SQA Group.
8. Define: Defect repository

Part-B Questions

1. Explain the Software testing principles.
2. Describe the defect classes in detail with example.
3. Explain defect repository.

UNIT II TEST CASE DESIGN

2.1 Introduction to Testing Design Strategies

As a reader of this text, you have a goal to learn more about testing and how to become a good tester. You might be a student at a university who has completed some software engineering courses. Upon completing your education you would like to enter the profession of test specialist. Or you might be employed by an organization that has test process improvement as a company goal. On the other hand, you may be a consultant who wants to learn more about testing to advise your clients. It may be that you play several of these roles. You might be asking yourself, Where do I begin to learn more about testing? What areas of testing are important? Which topics need to be addressed first? The Testing Maturity Model provides some answers to these questions. It can serve as a learning tool, or framework, to learn about testing. Support for this usage of the TMM lies in its structure. It introduces both the technical and managerial aspects of testing in a manner that allows for a natural evolution of the testing process, both on the personal and organizational levels.

In this chapter we begin the study of testing concepts using the TMM as a learning framework. We begin the development of testing skills necessary to support achievement of the maturity goals at levels 2-3 of the Testing Maturity Model. TMM level 2 has three maturity goals, two of which are managerial in nature. These will be discussed in subsequent chapters. The technically oriented maturity goal at level 2 which calls for an organization to "institutionalize basic testing techniques and methods" addresses important and basic technical issues related to execution-based testing. Note that this goal is introduced at a low level of the TMM, indicating its importance as a basic building block upon which additional testing strengths can be built. In order to satisfy this maturity goal test specialists in an organization need to acquire technical knowledge basic to testing and apply it to organizational projects.

Chapters 4 and 5 introduce you to fundamental test-related technical concepts related to execution-based testing. The exercises at the end of the chapter help to prepare you for their application to real-world problems. Testing strategies and methods are discussed that are both basic and practical. Consistent application of these strategies, methods, and techniques by testers across the whole organization will support test process evolution to higher maturity levels, and can lead to improved software quality.

2.2 The Smart Tester

Software components have defects, no matter how well our defect prevention activities are implemented. Developers cannot prevent/eliminate all defects during development. Therefore, software must be tested before it is delivered to users. It is the responsibility of the testers to design tests that (i) reveal defects, and (ii) can be used to evaluate software performance, usability, and reliability. To achieve these goals, testers must select a finite number of test cases, often from a very large execution domain. Unfortunately, testing is usually performed under budget and time constraints. Testers often are subject to enormous pressures from management and marketing because testing is not well planned, and expectations are unrealistic. The smart tester must plan for testing, select the test cases, and monitor the process to insure that the resources and time allocated for the job are utilized effectively. These are formidable tasks, and to carry them out effectively testers need proper education and training and the ability to enlist management support.

Novice testers, taking their responsibilities seriously, might try to test a module or component using all possible inputs and exercise all possible software structures. Using this approach, they reason, will enable them to detect all defects. However an informed and educated tester knows that is not a realistic or economically feasible goal. Another approach might be for the tester to select test inputs at random, hoping that these tests will reveal critical defects. Some testing experts believe that randomly generated test inputs have a poor performance record .

The author believes that goal of the smart tester is to understand the functionality, input/output domain, and the environment of use for the code being tested. For certain types of testing, the tester must also understand in detail how the code is constructed. Finally, a smart tester needs to use knowledge of the types of defects that are commonly injected during development or maintenance of this type of software. Using this information, the smart tester must then intelligently select a subset of test inputs as well as combinations of test inputs that she believes have the greatest possibility of revealing defects within the conditions and constraints placed on the testing process. This takes time and effort, and the tester must chose carefully to maximize use of resources [1,3,5]. This chapter, as well as the next, describes strategies and practical methods to help you design test cases so that you can become a smart tester.

2.3 Test Case Design Strategies

A smart tester who wants to maximize use of time and resources knows that she needs to develop what we will call effective test cases for execution-based testing. By an effective test case we mean one that has a good possibility of revealing a defect (see Principle 2 in Chapter 2). The ability to develop effective test cases is important to an organization evolving toward a higher-quality testing process. It has many positive consequences. For example, if test cases are effective there is (i) a greater probability of detecting defects, (ii) a more efficient use of organizational resources, (iii) a higher probability for test reuse, (iv) closer adherence to testing and project schedules and budgets, and, (v) the possibility for delivery of a higher-quality software product. What are the approaches a tester should use to design effective test cases? To answer the question we must adopt the view that software is an engineered product. Given this view there are two basic strategies that can be used to design test cases. These are called the black box (sometimes called functional or specification) and white box (sometimes called clear or glassbox) test strategies. The approaches are summarized in Figure 4.1.

Using the black box approach, a tester considers the software-under test to be an opaque box. There is no knowledge of its inner structure (i.e., how it works). The tester only has knowledge of what it does. The size of the software-under-test using this approach can vary from a simple module, member function, or object cluster to a subsystem or a complete Software system. The description of behavior or functionality for the software-under-test may come from a formal specification, an Input/Process/Output Diagram (IPO), or a well-defined set of pre and post conditions. Another source for information is a requirements specification document that usually describes the functionality of the software-under-test and its inputs and expected outputs. The tester provides the specified inputs to the software-under-test, runs the test and then determines if the outputs produced are equivalent to those in the specification. Because the black box approach only considers software behavior and functionality, it is often called functional or specification-based testing. This approach is especially useful for revealing requirements and specification defects.

The white box approach focuses on the inner structure of the software to be tested. To design test cases using this strategy the tester must have knowledge of that structure. The code, or a suitable pseudo codelike representation must be available. The tester selects test cases to exercise specific internal structural elements to determine if they are working properly. For example, test cases are often designed to exercise all statements or true/false branches that occur in a module or member function. Since designing, executing, and analyzing the results of white

box testing is very time consuming, this strategy is usually applied to smaller-sized pieces of software such as a module or member function. The reasons for the size restriction will become more apparent in Chapter 5 where the white box strategy is described in more detail. White box testing methods are especially useful for revealing design and code-based control, logic and sequence defects, initialization defects, and data flow defects.

The smart tester knows that to achieve the goal of providing users with low-defect, high-quality software, both of these strategies should be used to design test cases. Both support the tester with the task of selecting the finite number of test cases that will be applied during test. Neither approach by itself is guaranteed to reveal all defects types we have studied in Chapter 3. The approaches complement each other; each may be useful for revealing certain types of defects. With a suite of test cases designed using both strategies the tester increases the chances of revealing the many different type of defects in the software under test. The tester will also have an effective set of reusable test cases for regression testing (re-test after changes), and for testing new releases of the software.

There is a great deal of material to introduce to the reader relating to both of these strategies. To facilitate the learning process, the material has been partitioned into two chapters. This chapter focuses on black box methods, and Chapter 5 will describe white box methods and how to apply them to design test cases.

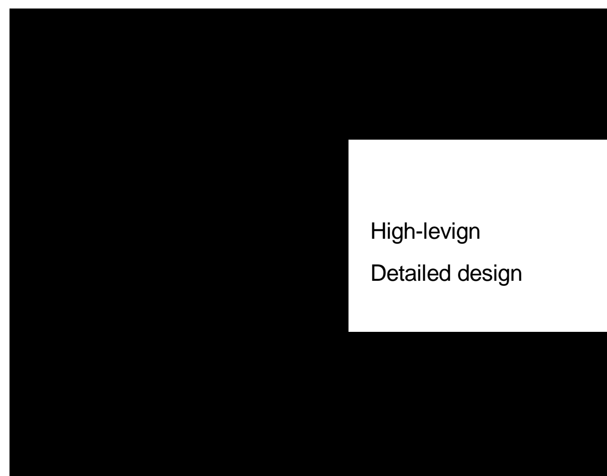
Test Strategy	Tester's View	Knowledge Sources	Methods
Black box		Requirements document Specifications	Equivalence class Partitioning Boundary value analysis State transition testing Cause and effect graphing
White box		High-levign Detailed design	Statement testing Branch testing Path testing Data flow testing Mutation testing

Fig 4.1 The two basic testing strategies.

2.4 Using black box approach to test case design

Given the black box test strategy where we are considering only inputs and outputs as a basis for designing test cases, how do we choose a suitable set of inputs from the set of all possible valid and invalid inputs? Keep in mind that infinite time and resources are not available to exhaustively test all possible inputs. This is prohibitively expensive even if the target software is a simple software unit. As a example, suppose you tried to test a single procedure that calculates the square root of a number. If you were to exhaustively test it you would have to try all positive input values. This is daunting enough! But, what about all negative numbers, fractions? These are also possible inputs. The number of test cases would rise rapidly to the point of infeasibilty. The goal for the smart tester is to effectively use the resources available by developing a set of test cases that gives the maximum yield of defects for the time and effort spent. To help achieve this goal using the black box approach we can select from several methods. Very often combinations of the methods are used to detect different types of defects. Some methods have greater practicality than others.

2.5 Random Testing

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing. For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen. Given this approach, some of the issues that remain open are the following:

- Are the three values adequate to show that the module meets its specification when the tests are run? Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects? For example, should positive integers at the beginning or end of the domain be specifically selected as inputs?
- Should any values outside the valid domain be used as test inputs? For example, should test data include floating point values, negative values, or integer values greater than 100?

More structured approaches to black box test design address these issues.

Use of random test inputs may save some of the time and effort that more thoughtful test input selection methods require. However, the reader should keep in mind that according to many testing experts, selecting test inputs randomly has very little chance of producing an effective set of test data [1]. There has been much discussion in the testing world about whether such a statement is accurate. The relative effectiveness of random versus a more structured approach to generating test inputs has been the subject of many research papers. Readers should refer to references [2-4] for some of these discussions. The remainder of this chapter and the next will illustrate more structured approaches to test case design and selection of inputs. As a final note there are tools that generate random test data for stress tests. This type of testing can be very useful especially at the system level. Usually the tester specifies a range for the random value generator, or the test inputs are generated according to a statistical distribution associated with a pattern of usage.

2.6 Equivalence Class Partitioning

If a tester is viewing the software-under-test as a black box with well- defined inputs and outputs, a good approach to selecting test inputs is to use a method called equivalence class partitioning. Equivalence class partitioning results in a partitioning of the input domain of the software under test. The technique can also be used to partition the output domain, but this is not a common usage. The finite number of partitions or equivalence classes that result allow the tester to select a given member of an equivalence class as a representative of that class. It is assumed that all members of an equivalence class are processed in an equivalent way by the target software.

Using equivalence class partitioning a test value in a particular class is equivalent to a test value of any other member of that class. Therefore, if one test case in a particular equivalence class reveals a defect, all the other test cases based on that class would be expected to reveal the same defect. We can also say that if a test case in a given equivalence class did not detect a particular type of defect, then no other test case based on that class would detect the defect (unless a subset of the equivalence class falls into another equivalence class, since classes may overlap in some cases). A more formal discussion of equivalence class partitioning is given in Beizer [5].

Based on this discussion of equivalence class partitioning we can say that the partitioning of the input domain for the software-under-test using this technique has the following advantages:

1. It eliminates the need for exhaustive testing, which is not feasible.

2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.

3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

Most equivalence class partitioning takes place for the input domain. How does the tester identify equivalence classes for the input domain? One approach is to use a set of what Glen Myers calls "interesting" input conditions [1]. The input conditions usually come from a description in the specification of the software to be tested. The tester uses the conditions to partition the input domain into equivalence classes and then develops a set of tests cases to cover (include) all the classes. Given that only the information in an input/output specification is needed, the tester can begin to develop black box tests for software early in the software life cycle in parallel with analysis activities (see Principle 11, Chapter 2). The tester and the analyst interact during the analysis phase to develop (i) a set of testable requirements, and (ii) a correct and complete input/output specification. From these the tester develops, (i) a high-level test plan, and (ii) a preliminary set of black box test cases for the system. Both the plan and the test cases undergo further development in subsequent life cycle phases. The V-Model as described in Chapter 8 supports this approach.

There are several important points related to equivalence class partitioning that should be made to complete this discussion.

1. The tester must consider both valid and invalid equivalence classes. Invalid classes represent erroneous or unexpected inputs.

2. Equivalence classes may also be selected for output conditions.

3. The derivation of input or outputs equivalence classes is a heuristic process. The conditions that are described in the following paragraphs only give the tester guidelines for identifying the partitions. There are no hard and fast rules. Given the same set of conditions, individual testers may make different choices of equivalence classes. As a tester gains experience he is more able to select equivalence classes with confidence.

4. In some cases it is difficult for the tester to identify equivalence classes. The conditions/boundaries that help to define classes may be absent, or obscure, or there may seem to be a very large or very small number of equivalence classes for the problem domain. These difficulties may arise from an ambiguous, contradictory, incorrect, or incomplete specification and/or requirements description. It is the duty of the tester to seek out the analysts and meet with them to clarify these documents. Additional contact with the user/client group may be required. A tester should also realize that for some software problem domains defining equivalence classes is inherently difficult, for example, software that needs to utilize the tax code.

Myers suggests the following conditions as guidelines for selecting input equivalence classes [1]. Note that a condition is usually associated with a particular variable. We treat each condition separately. Test cases, when developed, may cover multiple conditions and multiple variables.

List o f Conditions

1. ‘‘If an input condition for the software-under-test is specified as a range of values, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one outside each end of the range.’’

For example, suppose the specification for a module says that an input, the length of a widget in millimeters, lies in the range 1-499; then select one valid equivalence class that includes all values from 1 to 499. Select a second equivalence class that consists of all values less than 1, and a third equivalence class that consists of all values greater than 499.

2.‘‘If an input condition for the software-under-test is specified as a number of values, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number.’’

For example, if the specification for a real estate-related module say that a house can have one to four owners, then we select one valid equivalence class that includes all the valid number of owners, and then two invalid equivalence classes for less than one owner and more than four owners.

3.‘If an input condition for the software-under-test is specified as a set of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set.’’

For example, if the specification for a paint module states that the colors RED, BLUE, GREEN and YELLOW are allowed as inputs, then select one valid equivalence class that includes the set RED, BLUE, GREEN and YELLOW, and one invalid equivalence class for all other inputs.

‘4‘If an input condition for the software-under-test is specified as a “must be” condition, select one valid equivalence class to represent the “must be” condition and one invalid class that does not include the “must be” condition.’’

For example, if the specification for a module states that the first character of a part identifier must be a letter, then select one valid equivalence class where the first character is a letter, and one invalid class where the first character is not a letter.

5‘‘If the input specification or any other information leads to the belief that an element in an equivalence class is not handled in an identical way by the software-under-test, then the class should be further partitioned into smaller equivalence classes.’’

To show how equivalence classes can be derived from a specification, consider an example in Figure 4.2. This is a specification for a module that calculates a square root.

The specification describes for the tester conditions relevant to the

Function square_root

message (x:real)

when x >_0.0

reply (y:real)

where y >_0.0 & approximately (y*y,x)

otherwise reply exception imaginary_square_root

end function

Fig 4.2 A specification of a square root function.

input/output variables *x* and *y.* The input conditions are that the variable *x* must be a real number and be equal to or greater than 0.0. The conditions for the output variable *y* are that it must be a real number equal to or greater than 0.0, whose square is approximately equal to *x.* If *x* is not equal to or greater than 0.0, then an exception is raised. From this information the tester can easily generate both invalid and valid equivalence classes and boundaries. For example, input equivalence classes for this module are the following:

EC1. The input variable *x* is real, valid.

EC2. The input variable *x* is not real, invalid.

EC3. The value of *x* is greater than 0.0, valid.

EC4. The value of *x* is less than 0.0, invalid.

Because many organizations now use some type of formal or semiformal specifications, testers have a reliable source for applying the input/output conditions described by Myers.

After the equivalence classes have been identified in this way, the next step in test case design is the development of the actual test cases. A good approach includes the following steps.

1. Each equivalence class should be assigned a unique identifier. A simple integer is sufficient.

2. Develop test cases for all valid equivalence classes until all have been covered by (included in) a test case. A given test case may cover more than one equivalence class.

3. Develop test cases for all invalid equivalence classes until all have been covered individually. This is to insure that one invalid case does not mask the effect of another or prevent the execution of another.

An example of applying equivalence class partitioning will be shown in the next section.

2.7 Boundary Value Analysis

Equivalence class partitioning gives the tester a useful tool with which to develop black box based-test cases for the software-under-test. The method requires that a tester has access to a specification of input/output behavior for the target software. The test cases developed based on equivalence class partitioning can be strengthened by use of an technique called boundary value analysis. With experience, testers soon realize that many defects occur directly on, and above and below, the edges of equivalence classes. Test cases that consider these boundaries on both the input and output spaces as shown in Figure 4.3 are often valuable in revealing defects. Whereas equivalence class partitioning directs the tester to select test cases from any element of an equivalence class, boundary value analysis requires that the tester select elements close to the edges, so that both the upper and lower edges of an equivalence class are covered by test cases. As in the case of equivalence class partitioning, the ability to develop high quality test cases with the use of boundary values requires experience. The rules-of-thumb described below are useful for getting started with boundary value analysis.

1. If an input condition for the software-under-test is specified as a *range* of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range. For example if a specification states that an input value for a module must lie in the range between _1.0 and _1.0, valid tests that include values for ends of the range, as well as invalid test cases for values just above and below the ends, should be included. This would result in input values of _1.0, _1.1, and 1.0, 1.1.

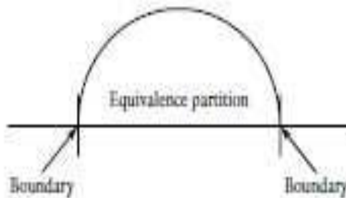


FIG. 4.3
Boundaries of an equivalence partition.

2. If an input condition for the software-under-test is specified as a *number* of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum. For example, for the real-estate module mentioned previously that specified a house can have one to four owners, tests that include

0,1 owners and 4,5 owners would be developed. The following is an example of applying boundary value analysis to output equivalence classes. Suppose a table of 1 to 100 values is to be produced by a module. The tester should select input data to generate an output table of size 0,1, and 100 values, and if possible 101 values.

3. If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set. It is important for the tester to keep in mind that equivalence class partitioning and boundary value analysis apply to testing both inputs and outputs of the software-under-test, and, most importantly, conditions are *not* combined for equivalence class partitioning or boundary value analysis. Each condition is

considered separately, and test cases are developed to insure coverage of all the individual conditions. An example follows.

An Example of the Application of Equivalence Class Partitioning and Boundary Value Analysis

Suppose we are testing a module that allows a user to enter new widget identifiers into a widget data base. We will focus only on selecting equivalence classes and boundary values for the inputs. The input specification for the module states that a widget identifier should consist of 3-15 alphanumeric characters of which the first two must be letters. We have three separate conditions that apply to the input: (i) it must consist of alphanumeric characters, (ii) the range for the total number of characters is between 3 and 15, and, (iii) the first two characters must be letters. Our approach to designing the test cases is as follows. First we will identify input equivalence classes and give them each an identifier. Then we will augment these with the results from boundary value analysis. Tables will be used to organize and record our findings. We will label the equivalence classes with an identifier ECxxx, where xxx is an integer whose value is one or greater. Each class will also be categorized as valid or invalid for the input domain.

First we consider condition 1, the requirement for alphanumeric characters. This is a "must be" condition. We derive two equivalence classes.

EC1. Part name is alphanumeric, valid.

EC2. Part name is not alphanumeric, invalid.

Then we treat condition 2, the range of allowed characters 3-15.

EC3. The widget identifier has between 3 and 15 characters, valid.

EC4. The widget identifier has less than 3 characters, invalid.

EC5. The widget identifier has greater than 15 characters, invalid.

Finally we treat the "must be" case for the first two characters.

EC6. The first 2 characters are letters, valid.

EC7. The first 2 characters are not letters, invalid.

Condition	Valid equivalence classes	Invalid equivalence classes
1	EC1	EC2
2	EC3	EC4, EC5
3	EC6	EC7

TABLE 4.1
Example equivalence class reporting table.

Note that each condition was considered separately. Conditions are *not* combined to select equivalence classes. The tester may find later on that a specific test case covers more than one equivalence class. The equivalence classes selected may be recorded in the form of a table as shown in Table 4.1. By inspecting such a table the tester can confirm that all the conditions and associated valid and invalid equivalence classes have been considered. Boundary value analysis is now used to refine the results of equivalence class partitioning. The boundaries to focus on are those in the allowed length for the widget identifier. An experienced tester knows that the module could have defects related to handling widget identifiers that are of length equal to, and directly adjacent to, the lower boundary of 3 and the upper boundary of 15. A simple set of abbreviations can be used to represent the bounds groups. For example:

BLB—a value just below the lower bound

LB—the value on the lower boundary

ALB—a value just above the lower boundary

BUB—a value just below the upper bound

UB—the value on the upper bound

AUB—a value just above the upper bound

For our example module the values for the bounds groups are:

BLB—2 **BUB**—14

LB—3 **UB**—15

ALB—4 **AUB**—16

Note that in this discussion of boundary value analysis, values just above the lower bound (ALB) and just below the upper bound (BUB) were selected. These are both valid cases and may be omitted if the tester does not believe they are necessary. The next step in the test case design process is to select a set of actual input values that covers all the equivalence classes and the boundaries. Once again a table can be used to organize the results. Table 4.2 shows the inputs for the sample module. Note that the table has the module name, identifier, a date of creation for the test input data, and the author of the test cases.

Table 4.2 only describes the tests for the module in terms of inputs derived from equivalence classes and boundaries. Chapter 7 will describe the components required for a complete test case. These include test inputs as shown in Table 4.2, along with test conditions and expected outputs.

Test logs are used to record the actual outputs and conditions when execution is complete. Actual outputs are compared to expected outputs to determine whether the module has passed or failed the test. Note that by inspecting the completed table the tester can determine whether all the equivalence classes and boundaries have been covered by actual input test cases. For this example the tester has selected a total of nine test cases. The reader should also note then when selecting inputs based on equivalence classes, a representative value at the midpoint of the bounds of each relevant class should be included as a typical case. In this example, a test case was selected with 9 characters, the average of the range values of 3 and 15 (test case identifier 9). The set of test cases

presented here is not unique: other sets are possible that will also cover all the equivalence classes and bounds. Based on equivalence class partitioning and boundary value analysis these test cases should have a high possibility of revealing defects in the module as opposed to

selecting test inputs at random from the input domain. In the latter case there is no way of estimating how productive

the input choices would be. This approach is also a better alternative to exhaustive testing where many combinations of characters, both valid and invalid cases, would have to be used. Even for this simple module exhaustive testing would not be feasible.

2.8 Other Black Box Test Design Approaches

There are alternative methods to equivalence class partitioning/boundary value analysis that a tester can use to design test cases based on the functional specification for the software to be tested. Among these are causeand effect graphing, state transition testing, and error guessing. Equivalence class partitioning combined with boundary value analysis is a practical approach to designing test cases for software written in both procedural and object-oriented languages since specifications are usually available for both member functions associated with an object and traditional procedures and functions to be written in procedural languages. However, it must be emphasized that use of equivalence class partitioning should be complimented by use of white box and, in many cases, other black box test design approaches. This is an important point for the tester

to realize. By combining strategies and methods the tester can have more confidence that the test cases will reveal a high number of defects for the effort expended. White box approaches to test design will be described in the next chapter. We will use the remainder of this section to give a description of other black box techniques.

Module name: Insert_Widget
Module identifier: AP62-Mod4
Date: January 31, 2000
Tester: Michelle Jordan

Test case identifier	Input values	Valid equivalence classes and bounds covered	Invalid equivalence classes and bounds covered
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc*	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6 (typical case)	

TABLE 4.2
Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module.

Cause - and - Effect Graphing

A major weakness with equivalence class partitioning is that it does not allow testers to combine conditions. Combinations can be covered in some cases by test cases generated from the classes. Cause-and-effect graphing is a technique that can be used to combine conditions and derive

an effective set of test cases that may disclose inconsistencies in a specification. However, the specification must be transformed into a graph that resembles a digital logic circuit. The tester is not required to have a background in electronics, but he should have knowledge of Boolean logic. The graph itself must be expressed in a graphical language [1]. Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming. The graph must be converted to a decision table that the tester uses to develop test cases. Tools are available for the latter process and allow the derivation of test cases to be more practical using this approach. The steps in developing test cases with a cause-and-effect graph are as follows [1]:

1. The tester must decompose the specification of a complex software component into lowerlevel units.

2. For each specification unit, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules.

3. From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs. An example of the notation is shown in Figure 4.4. Myers shows additional examples of graph notations [1].

4. The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.

5. The graph is then converted to a decision table.

6. The columns in the decision table are transformed into test cases. The following example illustrates the application of this technique. Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string. The specification states that the user must input the length of the string and the character to search for. If the string length is out-of-range an error message will appear. If the character appears in the string, its position will be reported. If the character is not in the string the message "not found" will be output.The input conditions, or causes are as follows:

C1: Positive integer from 1 to 80

C2: Character to search for is in string

The output conditions, or effects are: E1:

Integer out of range

E2: Position of character in string

E3: Character not found

The rules or relationships can be described as follows:

If C1 and C2, then E2.

If C1 and not C2, then E3. If

not C1, then E1.

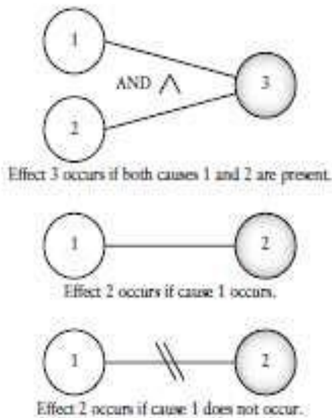


FIG. 4.4
Samples of cause-and-effect graph notations.

Based on the causes, effects, and their relationships, a cause-and-effect graph to represent this information is shown in Figure 4.5. The next step is to develop a decision table. The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes. Columns list each combination of causes, and each column represents a test case. Given *n* causes this could lead to a decision table with 2*n* entries, thus indicating a possible need for many test cases. In this example, since we have only two causes, the size and complexity of the decision table is not a big problem. However, with specifications having large numbers of causes and effects the size of the decision table can be large. Environmental constraints and unlikely combinations may reduce the number of entries and subsequent test cases.

A decision table will have a row for each cause and each effect. The entries are a reflection of the rules and the entities in the cause and effect graph. Entries in the table can be represented by a "1" for a cause or effect that is present, a "0" represents the absence of a cause or effect,and a "—" indicates a "don't care" value. A decision table for our simple example is shown in Table 4.3 where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases. The tester can use the decision table to consider combinations of inputs to generate the actual tests. In this example, three test cases are called for. If the existing string is "abcde," then possible tests are the following:

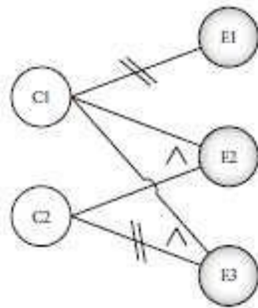


FIG. 4.5
Cause-and-effect graph for the character search example.

Inputs	Length	Character to search for	outputs
T1	5	c	3
T2	5	w	Not found
T3	90		Integer out of range

One advantage of this method is that development of the rules and the graph from the specification allows a thorough inspection of the specification. Any omissions, inaccuracies, or inconsistencies are likely to be detected. Other advantages come from exercising combinations of test data that may not be considered using other black box testing techniques. The major problem is developing a graph and decision table when there are many causes and effects to consider. A possible solution to this is to decompose a complex specification into lower-level, simpler components and develop cause-and-effect graphs and decision tables for these. Myers has a detailed description of this technique with examples [1]. Beizer [5] and Roper [9] also have discussions of this technique. Again, the possible complexity of the graphs and tables make it apparent that tool support is necessary for these time-consuming tasks. Although an effective set of test cases can be derived, some testers believe that equivalence class partitioning—if performed in a careful and systematic way—will generate a good set of test cases, and may make more effective useof a tester's time.

	T1	T2	T3
C1	1	1	0
C2	1	0	—
E1	0	0	1
E2	1	0	0
E3	0	1	0

TABLE 4.3

Decision table for character search example.

State transition testing

State transition testing is useful for both procedural and object-oriented development. It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes. This view gives the tester an additional opportunity to develop test cases to detect defects that may not be revealed using the input/output condition as well as cause-and-effect views presented by equivalence class partitioning and cause-and-effect graphing. Some useful definitions related to state concepts are as follows:

A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.

A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

During the specification phase a state transition graph (STG) may be generated for the system as a whole and/or specific modules. In object oriented development the graph may be called a state chart. STG/state charts are useful models of software (object) behavior. STG/state charts are commonly depicted by a set of nodes (circles, ovals, rounded rectangles) which represent states. These usually will have a name or number to identify the state. A set of arrows between nodes indicate what inputs or events will cause a transition or change between the two linked states. Outputs/actions occurring with a state transition are also depicted on a link or arrow. A simple state transition diagram is shown in Figure 4.6. S1 and S2 are the two states of interest. The black dot represents a pointer to the initial state from outside the machine. Many STGs also have "error" states and "done" states, the latter to indicate a final state for the system. The arrows display inputs/actions that cause the state transformations in the arrow directions. For example, the transition from S1 to S2 occurs with input, or event B. Action 3 occurs as part of this state transition. This is represented by the symbol "B/act3." It is often useful to attach to the STG the system or component variables that are affected by state transitions. This is valuable information for the tester as we will see in subsequent paragraphs. For large systems and system components, state transition graphs can become very complex. Developers can nest them to represent different

levels of abstraction. This approach allows the STG developer to group a set of related states together to form an encapsulated state that can be represented as a single entity on the original STG. The STG developer must ensure that this new state has the proper connections to the unchanged states from the original STG. Another way to simplify the STG is to use a state table representation which may be more concise. A state table for the STG in Figure 4.6 is shown in Table 4.4. The state table lists the inputs or events that cause state transitions.

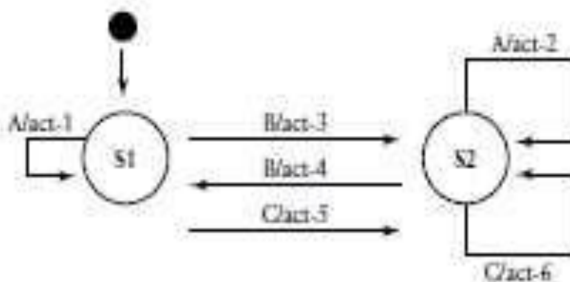


FIG. 4.6
Simple state transition graph.

For each state and each input the next state and action taken are listed. Therefore, the tester can consider each entity as a representation of a state transition. As testers we are interested in using

an existing STG as an aid to designing effective tests. Therefore this text will not present a discussion

of development and evaluation criteria for STGs. We will assume that the STGs have been prepared by developers or analysts as a part of the requirements specification. The STGs should be subject to a formal inspection when the requirement/specification is reviewed. This step is required

for organization assessed at TMM level 3 and higher. It is essential that testers be present at the reviews. From the tester's view point the review should ensure that (i) the proper number of states are represented, (ii) each state transition (input/output/action) is correct, (iii) equivalent states are identified, and (iv) unreachable and dead states are identified. Unreachable states are those that no input sequence will reach, and may indicate missing transitions. Dead states are those that once entered cannot be exited. In rare cases a dead state is legitimate, for example, in software that controls a destructible device. After the STG has been reviewed formally the tester should plan appropriate test cases. An STG has similarities to a control flow graph in that it has paths, or successions of transitions, caused by a sequence of inputs. Coverage of all paths does not guarantee complete testing and may not be practical. A simple approach might be to develop tests that insure that all states are entered. A more practical and systematic approach suggested by Marik consists of testing every possible state transition [10]. For the simple state machine in Figure 4.6 and Table 4.4 the transitions to be tested are:

Input A in S1

Input A in S2

Input B in S1

Input B in S2

Input C in S1

Input C in S2

The transition sequence requires the tester to describe the exact inputs for each test as the next step. For example the inputs in the above transitions might be a command, a menu item, a signal from a device or a button that is pushed. In each case an exact value is required, for example, the command might be ead, the signal might be ot or the button might be ff. The exact sequence of inputs must also be described, as well as the expected sequence of state

changes, and actions. Providing these details makes state-based tests easier to execute, interpret, and maintain. In addition, it is best to design each test specification so that the test begins in the start state, covers intermediate states, and returns to the start state. Finally, while the tests are being executed it is very useful for the tester to have software probes that report the current state (defining a state variable may be necessary) and the incoming event. Making state- related variables visible during each transition is also useful. All of these probes allow the tester to monitor the tests and detect incorrect transitions and any discrepancies in intermediate results.

For some STGs it may be possible that a single test case specification sequence could use (exercise) all of the transitions. There is a difference of opinion as to whether this is a good approach [5,10]. In most cases it is advisable to develop a test case specification that exercises many transitions, especially those that look complex, may not have been tried before, or that look ambiguous or unreachable. In this way more defects in the software may be revealed. For further exploration of state-based testing the following references are suggested, [5,10,11].

Error Guessing

Designing test cases using the error guessing approach is based on the tester's/developer's past experience with code similar to the code-under- test, and their intuition as to where defects may lurk in the code. Code similarities may extend to the structure of the code, its domain, the design

approach used, its complexity, and other factors. The tester/developer is sometimes able to make an educated uess as to which types of defects may be present and design test cases to reveal them. Some examples of obvious types of defects to test for are cases where there is a possible

division by zero, where there are a number of pointers that are manipulated, or conditions around array boundaries. Error guessing is an ad hoc approach to test design in most cases. However, if defect data for similar code or past releases of the code has been carefully recorded, the defect

types classified, and failure symptoms due to the defects carefully noted, this approach can have some structure and value. Such data would be available to testers in a TMM level 4 organization.

Black Box Testing and Commercial Off-the-Shelf (COTS) Components

As software development evolves into an engineering discipline, the reuse of software components will play an increasingly important role. Reuse of components means that developers need not reinvent the wheel; instead they can reuse an existing software component with the required functionality. The reusable component may come from a code reuse library within their organization or, as is most likely, from an outside vendor who specializes in the development of specific types of software components. Components produced by vendor organizations are known as commercial off-the-shelf, or COTS, components. The following data illustrate the growing usage of COTS components. In 1997, approximately 25% of the component portfolio of a typical corporation consisted of COTS components. Estimates for 1998 were about 28% and during the next several years the number may rise to 40% [12].

Using COTS components can save time and money. However, the COTS component must be evaluated before becoming a part of a developing system. This means that the functionality, correctness, and reliability of the component must be established. In addition, its suitability

for the application must be determined, and any unwanted functionality must be identified and addressed by the developers. Testing is one process that is not eliminated when COTS components are used for development!When a COTS component is purchased from a vendor it is basically a black box. It can range in size from a few lines of code, for example, a device driver, to thousands of lines of code, as in a telecommunication subsystem. It most cases, no source code is available, and if it is, it is very expensive to purchase. The buyer usually receives an executable version of the component, a description of its functionality, and perhaps a statement of how it was tested. In some cases if the component has been widely adapted, a statement of reliability will also be included. With this limited information, the developers and testers must make a decision on whether or not to use the component. Since the view is mainly as a black box, some of the techniques discussed in this chapter are applicable for testing the COTS components.

If the COTS component is small in size, and a specification of its inputs/outputs and functionality is available, then equivalence class partitioning and boundary value analysis may be useful for detecting defects and establishing component behavior. The tester should also use this approach for identifying any unwanted or unexpected functionality or side effects that could have a detrimental effect on the application. Assertions, which are logic statements that describe

correct program behavior, are also useful for assessing COTS behavior [13]. They can be associated with

program components, and monitored for violations using assertion support tools. Large-sized COTS components may be better served by using random or statistical testing guided by usage profiles.

Usage profiles are characterizations of the population of intended uses of the software in its intended environment .

These are not strictly black box in nature. As in the testing of newly developing software, the testing of COTS components requires the development of test cases, test oracles, and auxiliary code called a test harness (described in Chapter 6). In the case of COTS components, additional code, called glue software, must be developed to bind the COTS component to other modules for smooth system functioning. This glue software must also be tested. All of these activities add to the costs of reuse and must be considered when project plans are developed. Researchers are continually working on issues related to testing and certification of COTS components.

Certification refers to third-party assurance that a product (in our case a software product), process, or service meets a specific set of requirements.

2.9 Using white box approach to test design

In the previous chapter the reader was introduced to a test design approach that considers the software to be tested as a black box with a well-defined set of inputs and outputs that are described in a specification. In this chapter a complementary approach to test case design will be examined where the tester has knowledge of the internal logic structure of the software under test. The tester's goal is to determine if all the logical and data elements in the software unit are functioning properly. This is called the white box, or glass box, approach to test case design. The knowledge needed for the white box test design approach often becomes available to the tester in the later phases of the software life cycle, specifically during the detailed design phase of development. This is in contrast to the earlier availability of the knowledge necessary for black box test design. As a consequence, white box test design follows black box design as the

test efforts for a given project progress in time. Another point of contrast between the two approaches is that the black box test design strategy can be used for both small and large software components, whereas white box-based test design is most useful when testing small components. This is because the level of detail required for test design is very high, and the granularity of the items testers must consider when developing the test data is very small. These points will become more apparent as the discussion of the white box approach to test design continues.

2.10 Test adequacy criteria

The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements such as statements and branches. The tester develops test cases that exercise these structural elements to determine if defects exist in the program structure. The term exercise is used in this context to indicate that the target structural elements are executed when the test cases are run. By exercising all of the selected structural elements the tester hopes to improve the chances for detecting defects. Testers need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data, and for deciding when the testing efforts are adequate enough to terminate

the process with confidence that the software is working properly. Such a framework exists in the form of test adequacy criteria. Formally a test data adequacy criterion is a stopping rule [1,2]. Rules of this type can be used to determine whether or not sufficient testing has been carried out.

The criteria can be viewed as representing minimal standards for testing a program.

The application scope of adequacy criteria also includes:

(i) helping testers to select properties of a program to focus on during test;

(ii) helping testers to select a test data set for a program based on the selected properties; (iii) supporting testers with the development of quantitative objectives for testing;

(iv) indicating to testers whether or not testing can be stopped for that program.

A program is said to be adequately tested with respect to a given criterion if all of the target structural elements have been exercised according to the selected criterion. Using the selected adequacy criterion a tester can terminate testing when he/she has exercised the target structures, and have some confidence that the software will function in manner acceptable to the user.

If a test data adequacy criterion focuses on the structural properties of a program it is said to be a program-based adequacy criterion. Program-based adequacy criteria are commonly applied in white box testing. They use either logic and control structures, data flow, program text, or faults as the focal point of an adequacy evaluation [1]. Other types of test data adequacy criteria focus on program specifications. These are called specification-based test data adequacy criteria. Finally, some test data adequacy criteria ignore both program structure and specification in the selection and evaluation of test data. An example is the random selection criterion.

Adequacy criteria are usually expressed as statements that depict the property, or feature of interest, and the conditions under which testing can be stopped (the criterion is satisfied). For example, an adequacy criterion that focuses on statement/branch properties is expressed as the

following:

A test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.

In addition to statement/branch adequacy criteria as shown above, other types of program-based test data adequacy criteria are in use; for example, those based on (i) exercising program paths from entry to exit, and (ii) execution of specific path segments derived from data flow combinations such as definitions and uses of variables (see Section 5.5). As we will see in later sections of this chapter, a hierarchy of test data adequacy criteria exists; some criteria presumably have better defect detecting abilities than others.

The concept of test data adequacy criteria, and the requirement that certain features or properties of the code are to be exercised by test cases, leads to an approach called coverage analysis, which in practice is used to set testing goals and to develop and evaluate test data. In

the context of coverage analysis, testers often refer to test adequacy criteria as coverage criteria [1]. For example, if a tester sets a goal for a unit specifying that the tests should be statement adequate, this goal is often expressed as a requirement for complete, or 100%, statement coverage. It follows from this requirement that the test cases developed must insure that all the statements in the unit are executed at least once. When a coverage-related testing goal is expressed as a percent, it is often called the degree of coverage. The planned degree of coverage is specified in the test plan and then measured when the tests are actually executed by a coverage tool. The planned degree of coverage is usually specified as 100% if the tester wants to completely satisfy the commonly applied test adequacy, or coverage criteria. Under some circumstances, the planned degree of coverage may be less than 100% possibly due to the following:

• The nature of the unit

—Some statements/branches may not be reachable.

—The unit may be simple, and not mission, or safety, critical, and so complete coverage is thought to be unnecessary.

• The lack of resources

—The time set aside for testing is not adequate to achieve 100% coverage.

—There are not enough trained testers to achieve complete coverage for all of the units.

—There is a lack of tools to support complete coverage.

• Other project-related issues such as timing, scheduling, and marketing constraints

The following scenario is used to illustrate the application of coverage analysis. Suppose that a tester specifies branches as a target property for a series of tests. A reasonable testing goal would be satisfaction of the branch adequacy criterion. This could be specified in the test plan as a requirement for 100% branch coverage for a software unit under test. In this case the tester

must develop a set of test data that insures that all of the branches (true/false conditions) in the unit will be executed at least once by the test cases. When the planned test cases are executed under the control of a coverage tool, the actual degree of coverage is measured.

If there are, for example, four branches in the software unit, and only two are executed by the planned set of test cases, then the degree of branch coverage is 50%. All four of the branches must be executed by a test set in order to achieve the planned testing goal. When a coverage goal is not met, as in this example, the tester develops additional test cases and re- executes the code. This cycle continues until the desired level of coverage is achieved. The greater the degree of coverage, the more adequate the test set. When the tester achieves 100% coverage according to the selected criterion, then the test data has satisfied that criterion; it is said to be adequate for that criterion. An implication of this process is that a higher degrees of coverage will lead to greater numbers of detected defects.

It should be mentioned that the concept of coverage is not only associated with white box testing. Coverage can also be applied to testing with usage profiles (see Chapter 12). In this case the testers want to ensure that all usage patterns have been covered by the tests. Testers also use

coverage concepts to support black box testing. For example, a testing goal might be to exercise, or cover, all functional requirements, all equivalence classes, or all system features. In contrast to black box approaches, white box-based coverage goals have stronger theoretical and practical

support.

2.11 Coverage and Control Flow Graphs

The application of coverage analysis is typically associated with the use of control and data flow models to represent program structural elements and data. The logic elements most commonly considered for coverage are based on the flow of control in a unit of code. For example,

(i) program statements;

(ii) decisions/branches (these influence the program flow of control);

(iii) conditions (expressions that evaluate to true/false, and do not contain any other true/false-valued expressions);

(iv) combinations of decisions and conditions;

(v) paths (node sequences in flow graphs).

These logical elements are rooted in the concept of a program prime. A program prime is an atomic programming unit. All structured programs can be built from three basic primes-sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops). Graphical representations for these three primes are shown in Figure 5.1.

Using the concept of a prime and the ability to use combinations of primes to develop structured code, a (control) flow diagram for the soft- ware unit under test can be developed. The flow graph can be used by the tester to evaluate the code with respect to its testability, as well as to develop white box test cases. This will be shown in subsequent sections of this chapter. A flow graph representation for the code example in Figure 5.2 is found in Figure 5.3. Note that in the flow graph the nodes represent sequential statements, as well as decision and looping predicates. For simplicity, sequential statements are often omitted or combined as a block that indicates that if the first statement in the block is executed, so are all the following statements in the block. Edges in the graph represent transfer of control. The direction of the transfer depends on the outcome of the condition in the predicate (true or false).

There are commercial tools that will generate control flow graphs from code and in some cases from pseudo code. The tester can use tool support for developing control flow graphs especially for complex pieces of code. A control flow representation for the software under test facilitates the design of white box-based test cases as it clearly shows the logic elements needed to design the test cases using the coverage criterion of choice.Zhu has formally described a set of program-based coverage criteria in the context of test adequacy criteria and control/data flow models [1].

This chapter will presents control-flow, or logic-based, coverage concepts in a less formal but practical manner to aid the tester in developing test data sets, setting quantifiable testing goals, measuring results, and evaluating the adequacy of the test outcome. Examples based on the logic

elements listed previously will be presented. Subsequent sections will describe data flow and fault-based coverage criteria.

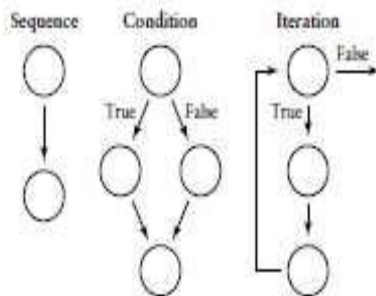


FIG. 5.1

Representation of program primes.

2.12 Covering Code Logic

Logic-based white box-based test design and use of test data adequacy/ coverage concepts provide two major payoffs for the tester: (i) quantitative coverage goals can be proposed, and (ii) commercial tool support is readily available to facilitate the tester's work (see Chapter 14). As de-

scribed in Section 5.1, testers can use these concepts and tools to decide on the target logic elements (properties or features of the code) and the degree of coverage that makes sense in terms of the type of software, its mission or safety criticalness, and time and resources available. For ex-

ample, if the tester selects the logic element program statements, this indicates that she will want to design tests that focus on the execution of program statements. If the goal is to satisfy the statement adequacy/ coverage criterion, then the tester should develop a set of test cases so that when the module is executed, all (100%) of the statements in the module are executed at least once. In terms of a flow graph model of the code, satisfying this criterion requires that all the nodes in the graph are exercised at least once by the test cases. For the code in Figure 5.2

and its corresponding flow graph in Figure 5.3 a tester would have to develop test cases that exercise nodes 1-8 in the flow graph. If the tests achieve this goal, the test data would satisfy the statement adequacy criterion. In addition to statements, the other logic structures are also associated with corresponding adequacy/coverage criteria. For example, to achieve complete (100%) decision (branch) coverage test cases must be designed

/* pos_sum nds the sum of all positive numbers (greater than zero) stored in an integer

array a. Input parameters are num_of_entries, an integer, and a, an array of integers with

num_of_entries elements. The output parameter is the integer sume */

1. pos_sum(a, num_of_entries, sum)

2. sum 0

3. inti 1

4. while (i < num_of_entries)

5. if a[i] > 0

6. sum sum a[i]

endif

7. i i 1

end while

8. end pos_sum

FIG. 5.2 ***Code sample with branch and loop.***

so that each decision element in the code (if-then, case, loop) executes with all possible outcomes at least once. In terms of the control flow model, this requires that all the edges in the corresponding flow graph must be exercised at least once. Complete decision coverage is considered to be a stronger coverage goal than statement coverage since its satisfaction results in satisfying statement coverage as well (covering all the edges in a flow graph will ensure coverage of the nodes). In fact, the statement coverage goal is so weak that it is not considered

to be very useful for revealing defects. For example, if the defect is a missing statement it may remain undetected by tests satisfying complete statement coverage. The reader should be aware that in spite of the weakness, even this minimal coverage goal is not required in many test plans. Decision (branch) coverage for the code example in Figure 5.2, requires test cases to be developed for the two decision statements, that is, the four true/false edges in the control flow graph of Figure 5.3. Input values must ensure execution the true/false possibilities for the decisions in line 4 (while loop) and line 5 (if statement). Note that the if statement has a full else component, that is, there is no else part. However, we include a test that covers both the true and false conditions for the statement.

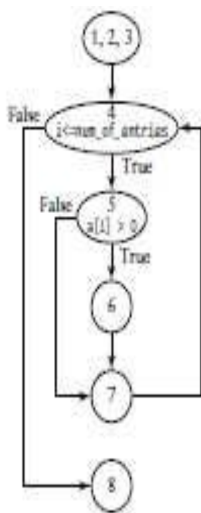


FIG. 5.3

A control flow graph representation for the code in Figure 5.2.

A possible test case that satisfies 100% decision coverage is shown in Table 5.1. The reader should note that the test satisfies both the branch adequacy criterion and the statement adequacy criterion, since all the statements 1-8 would be executed by this test case. Also note that for this

code example, as well as any other code component, there may be several sets of test cases that could satisfy a selected criterion. This code example represents a special case in that it was feasible to achieve both branch and statement coverage with one test case. Since one of the inputs, , is an array, it was possible to assign both positive and negative values to the elements of , thus allowing coverage of both the true/false branches of the if statement. Since more than one iteration of the while loop was also possible, both the true and false branches of this loop could also be covered by one test case. Finally, note that the code in the example does not

contain any checks on the validity of the input parameters. For simplicity it is assumed that the calling module does the checking.

In Figure 5.2 we have simple predicates or conditions in the branch and loop instructions. However, decision statements may contain multiple conditions, for example, the statement

Decision or branch	Value of variable i	Value of predicate	Test case: Value of a, num_of_entries
			a=1, −45,3 num_of_entries = 3
while	1	True	
	4	False	
if	1	True	
	2	False	

TABLE 5.1
A test case for the code in Figure 5.2 that satisfies the decision coverage criterion.

If (x MIN and y MAX and (not INT Z))

has three conditions in its predicate: (i) x MIN, (ii) y MAX, and (iii) not INT Z. Decision coverage only requires that we exercise at least once all the possible outcomes for the branch or loop predicates as a *whole, not for each individual condition contained in a compound pred*icate. There are other coverage criteria requiring at least one execution of the all possible conditions and combinations of decisions/conditions. The names of the criteria reflect the extent of condition/decision coverage. For example, condition coverage requires that the tester insure that each individual condition in a compound predicate takes on all possible values at least once during execution of the test cases. More stringent coverage criteria also require exercising all possible combinations of decisions and conditions in the code. All of the coverage criterion described so far can be arranged in a hierarchy of strengths from weakest to strongest as follows: statement, decision, decision/condition. The implication for this approach to test design is that the stronger the criterion, the more defects will be revealed by the tests. Below is a simple example showing the test cases for a decision statement with a compound predicate.

if(age <65 and married true)

do X

do Y ....

else

do Z

Condition 1: Age less than 65
Condition 2: Married is true

Test cases for simple decision coverage

Value for age	Value for married	Decision outcome (compound predicate as a whole)	Test case ID
30	True	True	1
75	True	False	2

Note that these tests would not exercise the possible outcome for married as false. A defect in the logical operator for condition 2, for example, may not be detected. Test cases 2 and 3 shown as follows would cover both possibilities.

Test cases for condition coverage

Value for age	Value for married	Condition 1 outcome	Condition 2 outcome	Test case ID
75	True	False	True	2
30	False	True	False	3

Note that the tests result in each condition in the compound predicate taking on a true/false outcome. However, all possible outcomes for the decision as a whole are not exercised so it would not satisfy decision/condition coverage criteria. Decision/condition coverage requires that every condition will be set to all possible outcomes and the decision as a whole will be set to all possible outcomes. A combination of test cases 1, 2, and 3 would satisfy this criterion.

Test cases for decision condition coverage

Value for age	Value for married	Condition 1 outcome	Condition 2 outcome	Decision outcome (compound predicate as a whole)	Test case ID
30	True	True	True	True	1
75	True	False	True	False	2
30	False	True	False	False	3

The criteria described above do not require the coverage of all the possible combinations of conditions. This is represented by yet another criterion called multiple condition coverage where all possible combinations of condition outcomes in each decision must occur at least once when the test cases are executed. That means the tester needs to satisfy the following combinations for the example decision statement:

Condition 1	Condition 2
True	True
True	False
False	True
False	False

In most cases the stronger the coverage criterion, the larger the number of test cases that must be developed to insure complete coverage. For code with multiple decisions and conditions the complexity of test case design increases with the strength of the coverage criterion. The tester must decide, based on the type of code, reliability requirements, and resources available which criterion to select, since the stronger the criterion selected the more resources are usually required to satisfy it.

2.13 Paths: Their Role in White Box-Based Test Design

In Section 5.2 the role of a control flow graph as an aid to white box test design was described. It was also mentioned that tools were available to generate control flow graphs. These tools typically calculate a value for a software attribute called McCabe's Cyclomatic Complexity V(G) from a flow graph. The cyclomatic complexity attribute is very useful to a tester [3]. The complexity value is usually calculated from the control flow graph (G) by the formula

$$V(G) = E - N + 2 \qquad (1)$$

The value E is the number of edges in the control flow graph and N is the number of nodes. This formula can be applied to flow graphs where there are no disconnected components [4]. As an example, the cyclomatic complexity of the flow graph in Figure 5.3 is calculated as follows:

$$E = 7, N = 6$$
$$V(G) = 7 - 6 + 2 = 3$$

The cyclomatic complexity value of a module is useful to the tester in several ways. One of its uses is to provide an approximation of the number of test cases needed for branch coverage in a module of structured code. If the testability of a piece of software is defined in terms of the number of test cases required to adequately test it, then McCabes' cyclomatic complexity provides an approximation of the testability of a module. The tester can use the value of V(G) along with past project data to approximate the testing time and resources required to test a software module. In addition, the cyclomatic complexity value and the control flow graph give the tester another tool for developing white box test cases using the concept of a path. A definition for this term is given below.

A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.

A path may go through a given segment of the control flow graph one or more times. We usually designate a path by the sequence of nodes it encompasses. For example, one path from the graph in Figure 5.3 is

1-2-3-4-8

where the dashes represent edges between two nodes. For example, the sequence 4-8 represents the edge between nodes 4 and 8 Cyclomatic complexity is a measure of the number of so-called in-dependent paths in the graph. An independent path is a special kind of path in the flow graph. Deriving a set of independent paths using a flow graph can support a tester in identifying the control flow features in the code and in setting coverage goals. A tester identifies

a set of independent paths for the software unit by starting out with one simple path in the flow graph and iteratively adding new paths to the set by adding new edges at each iteration until there are no more new edges to add. The independent paths are defined as any new path through the graph that introduces a new edge that has not be traversed before the path is defined.

A set of independent paths for a graph is sometimes called a basis set. For most software modules it may be possible to derive a number of basis sets. If we examine the flow graph in Figure 5.3, we can derive the following set of independent paths starting with the first path identified above.

(i) 1-2-3-4-8

(ii) 1-2-3-4-5-6-7-4-8

(iii) 1-2-3-4-5-7-4-8

The number of independent paths in a basis set is equal to the cyclomatic complexity of the graph. For this example they both have a value of 3. Recall that the cyclomatic complexity for a flow graph also gives us an approximation (usually an upper limit) of the number of tests needed to achieve branch (decision) coverage. If we prepare white box test cases so that the inputs cause the execution of all of these paths, we can be reasonably sure that we have achieved complete statement and decision coverage for the module. Testers should be aware that although identifying the independent paths and calculating cyclomatic complexity in a module of structured code provides useful support for achieving decision coverage goals, in some cases the number of independent paths in the basis set can lead to an overapproximation of the number of test cases needed for decision (branch) coverage. This is illustrated by the code example of Figure 5.2, and the test case as shown in Table 5.1.

To complete the discussion in this section, one additional logic-based testing criterion based on the path concept should be mentioned. It is the strongest program-based testing criterion, and it calls for complete path coverage; that is, every path (as distinguished from independent paths) in

a module must be exerc ised by the test set at least once. This may not be a practical goal for a tester. For example, even in a small and simple unit of code there may be many paths between

the entry and exit nodes. Adding even a few simple decision statements increases the number of paths.

Every loop multiplies the number of paths based on the number of possible iterations of the loop since each iteration constitutes a different path through the code. Thus, complete path coverage for even a simple module may not be practical, and for large and complex modules it is not feasible.

In addition, some paths in a program may be unachievable, that is, they cannot be executed no matter what combinations of input data are used. The latter makes achieving complete path coverage an impossible task. The same condition of unachievability may also hold true for some branches or statements in a program. Under these circumstances coverage goals are best expressed in terms of the number of feasible or achievable paths, branches, or statements respectively.

As a final note, the reader should not confuse the coverage based on independent path testing as equivalent to the strongest coverage goal—complete path coverage. The basis set is a special set of paths and does not represent all the paths in a module; it serves as a tool to aid the tester in achieving decision coverage.

2.14 Additional White Box Test Design Approaches

In addition to methods that make use of software logic and control structures to guide test data generation and to evaluate test completeness there are alternative methods that focus on other characteristics of the code. One widely used approach is centered on the role of variables (data) in

the code. Another is fault based. The latter focuses on making modifications to the software, testing the modified version, and comparing results. These will be described in the following sections of this chapter.

Data Flow and White Box Test Design

In order to discuss test data generation based on data flow information, some basic concepts that define the role of variables in a software component need to be introduced.

We say a variable is defined in a statement when its value is assigned or changed.

For example in the statements

```
Y = 26 * X
Read (Y)
```

the variable Y is defined, that is, it is assigned a new value. In data flow notation this is indicated as a def for the variable Y.

We say a variable is used in a statement when its value is utilized in a statement. The value of the variable is not changed.

A more detailed description of variable usage is given by Rapps and Weyuker [4]. They describe a predicate use (p-use) for a variable that indicates its role in a predicate. A computational use (c-use) indicates the variable's role as a part of a computation. In both cases the variable value is un-

changed. For example, in the statement

Y =26*X

the variable X is used. Specifically it has a c-use. In the statement

if (X >98)

Y= max

X has a predicate or p-use. There are other data flow roles for variables such as undefined or dead, but these are not relevant to the subsequent discussion. An analysis of data flow patterns for specific variables is often very useful for defect detection. For example, use of a variable without a definition occurring first indicates a defect in the code. The variable has not been initialized. Smart compilers will identify these types of defects. Testers and developers can utilize data flow tools that will identify and display variable role information. These should also be used prior to code reviews to facilitate the work of the reviewers.

Using their data flow descriptions, Rapps and Weyuker identified several data-flow based test adequacy criteria that map to corresponding coverage goals. These are based on test sets that exercise specific path segments, for example:

All def

All p-uses

All c-uses/some p-uses

All p-uses/some c-uses

All uses

All def-use paths

The strongest of these criteria is all def-use paths. This includes all p- and c-uses.

```
1    sum=0                        sum, def
2    read (n),                    n, def
3    i=1                          i, def
4    while (i <= n)               i, n p-use
5         read (number)           number, def
6         sum = sum + number      sum, def, sum, number, c-use
7         i=i+1                   i, def, c-use
8    end while
9    print (sum)                  sum, c-use
```

FIG. 5.4

Sample code with data flow information.

We say a path from a variable definition to a use is called a def-use path.

To satisfy the all def-use criterion the tester must identify and classify occurrences of all the variables in the software under test. A tabular summary is useful. Then for each variable, test data is generated so that all definitions and all uses for all of the variables are exercised during test. As an example we will work with the code in Figure 5.4 that calculates the sum of n numbers.

The variables of interest are sum, i, n, and number. Since the goal is to satisfy the all def-use criteria we will need to tabulate the def-use occurrences for each of these variables. The data flow role for each variable in each statement of the example is shown beside the statement in italics.

Tabulating the results for each variable we generate the following tables. On the table each defuse pair is assigned an identifier. Line numbers are used to show occurrence of the def or use. Note that in some statements a given variable is both defined and used.

Table for n

pair id	def	use
1	2	4

Table for number

pair id	def	use
1	5	6

Table for sum

pair id	def	use
1	1	6
2	1	9
3	6	6
4	6	9

Table for i

pair id	def	use
1	3	4
2	3	7
3	7	7
4	7	4

After completion of the tables, the tester then generates test data to exercise all of these def-use pairs In many cases a small set of test inputs will cover several or all def-use paths. For this example two sets of test data would cover all the def-use pairs for the variables:

Test data set 1: n 0

Test data set 2: n 5, number 1,2,3,4,5

Set 1 covers pair 1 for n, pair 2 for sum, and pair 1 for i. Set 2 covers pair 1 for n, pair 1 for number, pairs 1,3,4 for sum, and pairs 1,2,3,4 for *i.*

Note even for this small piece of code there are four tables and four def-use pairs for two of the variables.

As with most white box testing methods, the data flow approach is most effective at the unit level of testing. When code becomes more complex and there are more variables to consider it becomes more time consuming for the tester to analyze data flow roles, identify paths, and design the tests. Other problems with data flow oriented testing occur in the handling of dynamically bound variables such as pointers. Finally, there are no commercially available tools that provide strong support for data flow testing, such as those that support control-flow based testing. In the latter case, tools that determine the degree of coverage, and which portions of the code are yet uncovered, are of particular importance. These are not available for data flow methods. For examples of prototype tools.

Loop Testing

Loops are among the most frequently used control structures. Experienced software engineers realize that many defects are associated with loop constructs. These are often due to poor programming practices and lack of reviews. Therefore, special attention should be paid to loops during testing. Beizer has classified loops into four categories: simple, nested, concatenated, and unstructured [4]. He advises that if instances of unstructured loops are found in legacy code they should be redesigned to reflect structured programming techniques. Testers can then focus on the

remaining categories of loops.

Loop testing strategies focus on detecting common defects associated with these structures. For example, in a simple loop that can have a range of zero to n iterations, test cases should be developed so that there are:

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely; (ii)

one iteration of the loop;

(iii) two iterations of the loop;

(iv) k iterations of the loop where k n;

(v) n 1 iterations of the loop;

(vi) n 1 iterations of the loop (if possible).

If the loop has a nonzero minimum number of iterations, try one less than the minimum. Other cases to consider for loops are negative values for the loop control variable, and n 1 iterations of the loop if that is possible. Zhu has described a historical loop count adequacy criterion that states that in the case of a loop having a maximum of n iterations, tests that execute the loop zero times, once, twice, and so on up to n times are required [1].

Beizer has some suggestions for testing nested loops where the outer loop control variables are set to minimum values and the innermost loop is exercised as above. The tester then moves up one loop level and finally tests all the loops simultaneously. This will limit the number of tests to

perform; however, the number of test under these circumstances is still large and the tester may have to make trade-offs. Beizer also has suggestions for testing concatenated loops.

Mutation Testing

Mutation testing is another approach to test data generation that requires knowledge of code struc-

ture, but it is classified as a fault-based testing approach. It considers the possible faults that could occur in a software component as the basis for test data generation and evaluation of testing effectiveness.

Mutation testing makes two major assumptions:

1. The competent programmer hypothesis. This states that a competent programmer writes programs that are nearly correct. Therefore we can assume that there are no major construction errors in the program; the code is correct except for a simple error(s).

2. The coupling effect. This effect relates to questions a tester might have about how well mutation testing can detect complex errors since the changes made to the code are very simple. DeMillo has commented on that issue as far back as 1978 [10]. He states that test data that can distinguish all programs differing from a correct one only by simple errors are sensitive enough to distinguish it from programs with more complex errors.

Mutation testing starts with a code component, its associated test cases, and the test results. The original code component is modified in a simple way to provide a set of similar components that are called mutants. Each mutant contains a fault as a result of the modification. The original test data is then run with the mutants. If the test data reveals the fault in the mutant (the result of the modification) by producing a different output as a result of execution, then the mutant is said to be killed. If the mutants do not produce outputs that differ from the original with the test data, then the test data are not capable of revealing such defects. The tests cannot distinguish the original from the mutant. The tester then must develop additional test data to reveal the fault and kill the mutants. A test data adequacy criterion that is applicable here is the following:

A test set T is said to be mutation adequate for program P provided that for every in equivalent mutant Pi of P there is an element t in T such that Pi(t) is not equal to P(t).

The term T represents the test set, and t is a test case in the test set. For the test data to be adequate according to this criterion, a correct program must behave correctly and all incorrect programs behave incorrectly for the given test data.

Mutations are simple changes in the original code component, for example: constant replacement, arithmetic operator replacement, data statement alteration, statement deletion, and logical operator replace- ment. There are existing tools that will easily generate mutants. Tool users

need only to select a change operator. To illustrate the types of changes made in mutation testing we can make use of the code in Figure 5.2. A first mutation could be to change line 7 from

$i = i + 1$ to $i = i + 2$.

If we rerun the tests used for branch coverage as in Table 5.1 this mutant will be killed, that is, the output will be different than for the original code. Another change we could make is in line 5, from

if $a[i] > 0$ to if $a[i] < 0$.

This mutant would also be killed by the original test data. Therefore, we can assume that our original tests would have caught this type of defect. However, if we made a change in line 5 to read

$$\text{if } a[i] > = 0,$$

this mutant would not be killed by our original test data in Table 5.1. Our inclination would be to augment the test data with a case that included a zero in the array elements, for example:

$$a = 0, 45, 3, \text{SIZE} = 3.$$

However, this test would not cause the mutant to be killed because adding a zero to the output variable sum does not change its final value. In this case it is not possible to kill the mutant. When this occurs, the mutant is said to be equivalent to the original program. To measure the mutation adequacy of a test set T for a program P we can use what is called a mutation score (MS), which is calculated as

$$MS(P,T) = \frac{\text{\# of dead mutants}}{\text{\# total mutants} - \text{\# of equivalent mutants}}$$

Equivalent mutants are discarded from the mutant set because they do not contribute to the adequacy of the test set. Mutation testing is useful in that it can show that certain faults as represented in the mutants are not likely to be present since they would have been revealed by test data. It also helps the tester to generate hy- potheses about the different types of possible

faults in the code and to develop test cases to reveal them. As previously mentioned there are tools to support developers and testers with producing mutants. In fact, many hundreds of mutants can be produced easily. However, running the tests, analyzing results, and developing additional tests, if needed, to kill the mutants are all time consuming. For these reasons mutation testing is usually applied at the unit level. However, recent research in an area called interface mutation (the application of mutation testing to evaluate how well unit interfaces have been tested) has suggested that it can be applied effectively at the integration test level as well .Mutation testing as described above is called strong mutation testing. There are variations that reduce the number of mutants produced. One of these is called weak mutation testing which focuses on specific code components .

2.15 Evaluating Test Adequacy Criteria

Most of the white box testing approaches we have discussed so far are associated with application of an adequacy criterion. Testers are often faced with the decision of which criterion to apply to a given item under test given the nature of the item and the constraints of the test environment (time, costs, resources) One source of information the tester can use to select an appropriate criterion is the test adequacy criterion hierarchy as shown in Figure 5.5 which describes a subsumes relationship among the criteria. Satisfying an adequacy criterion at the higher levels of the hierarchy implies a greater thoroughness in testing [1,14-16]. The criteria at the top of the hierarchy are said to subsume those at the lower levels. For example, achieving all definition-use (def-use) path adequacy means the tester has also achieved both branch and statement adequacy. Note from the hierarchy that statement adequacy is the weakest of the test adequacy criteria. Unfortunately, in many organizations achieving a high level of statement coverage is not even included as a minimal testing goal.

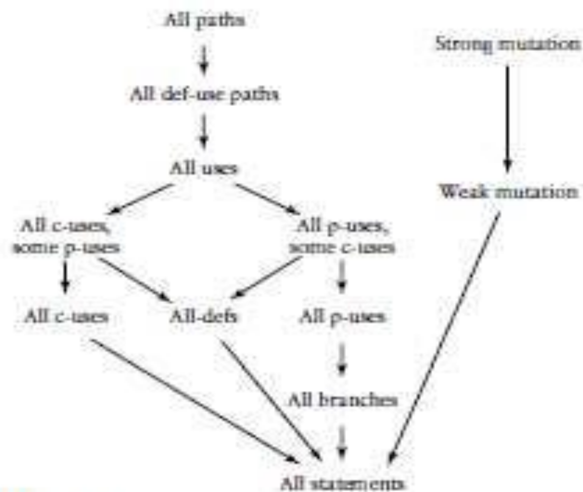


FIG. 5.5
A partial ordering for test adequacy criteria.

As a conscientious tester you might at first reason that your testing goal should be to develop tests that can satisfy the most stringent criterion. However, you should consider that each adequacy criterion has both strengths and weaknesses. Each, is effective in revealing certain types of defects. Application of the so-called Stronger criteria usually requires more tester time and resources. This translates into higher testing costs. Testing conditions, and the nature of the software should guide your choice of a criterion.

Support for evaluating test adequacy criteria comes from a theoretical treatment developed by Weyuker . She presents a set of axioms that allow testers to formalize properties which should be satisfied by any good program-based test data adequacy criterion. Testers can use the axioms to

• recognize both strong and weak adequacy criteria; a tester may decide to use a weak criterion, but should be aware of its weakness with respect to the properties described by the axioms;

• focus attention on the properties that an effective test data adequacy criterion should exhibit;

• select an appropriate criterion for the item under test;

• stimulate thought for the development of new criteria; the axioms are the framework with which to evaluate these new criteria.

The axioms are based on the following set of assumptions :

(i) programs are written in a structured programming language;

(ii) programs are SESE (single entry/single exit);

(iii) all input statements appear at the beginning of the program;

(iv) all output statements appear at the end of the program.

The axioms/properties described by Weyuker are the following :

1. Applicability Property

For every program there exists an adequate test set. What this axiom means is that for all programs we should be able to design an adequate test set that properly tests it. The test set may be very large so the tester will want to select representable points of the specification domain to test it. If we test on all representable points, that is called an exhaustive test set. The exhaustive test set will surely be adequate since there will be no other test data that we can generate. However, in past discussions we have ruled out exhaustive testing because in most cases it is too expensive,

time consuming, and impractical.

2. Non exhaustive Applicability Property

For a program P and a test set T, P is adequately tested by the test set *T, and T is not an exhaustive test set. To paraphrase, a tester does not* need an exhaustive test set in order to adequately test a program.

3. Monotonicity Property

If a test set T is adequate for program P, and if T is equal to, or a subset of T , then T is adequate for program P.”

4. Inadequate Empty Set

In empty test set is not an adequate test for any program. If a program is not tested at all, a tester cannot claim it has been adequately tested! Note that these first four axioms are very general and apply to all programs independent of programming language and equally apply to uses of both program- and specification-based testing. For some of the next group of axioms this is not true.

5. Antiextensionality Property

There are programs P and Q such that P is equivalent to Q, and T is adequate for P, but T is not adequate for Q. We can interpret this axiom as saying that just because two programs are semantically equivalent (they may perform the same function) does not mean we should test them the same way. Their implementations (code structure) may be very different. The reader should note that if programs have equivalent specifications then their test sets may coincide using black box testing techniques, but this axiom applies to program-based testing and it is the differences that may occur in program code that make it necessary to test P and Q with

different test sets.

6. General Multiple Change Property

There are programs P and Q that have the same shape, and there is a test set T such that T is adequate for P, but is not adequate for Q. Here Weyuker introduces the concept of shape to express a syntactic equivalence. She states that two programs are the same shape if one can be transformed into the other by applying the set of rules shown below any number of times:

(i) replace relational operator r1 in a predicate with relational operator *r2;*

(ii) replace constant c1 in a predicate of an assignment statement with constant c2;

(iii) replace arithmetic operator a1 in an assignment statement with arithmetic operator a2.

Axiom 5 says that semantic closeness is not sufficient to imply that two programs should be tested in the same way. Given this definition of shape, Axiom 6 says that even the syntactic closeness of two programs is not strong enough reason to imply they should be tested in the same way.

7. Antidecomposition Property

There is a program P and a component Q such that T is adequate for *P, T is the set of vectors of values that variables can assume on entrance* to Q for some t in T, and T is not adequate for Q.This axiom states that although an encompassing program has been adequately tested, it does not follow that each of its components parts has been properly tested. Implications for this axiom are:

1. a routine that has been adequately tested in one environment may not have been adequately tested to work in another environment, the environment being the enclosing program.

2. although we may think of P, the enclosing program, as being more complex than Q it may not be. Q may be more semantically complex; it may lie on an unexecutable path of P, and thus would have the null set, as its test set, which would violate Axiom 4.

8. Anticomposition Property

There are programs P and Q, and test set T, such that T is adequate for *P, and the set of vectors of values that variables can assume on entrance* to Q for inputs in T is adequate for Q, but T is not adequate for P; Q (the composition of P and Q). Paraphrasing this axiom we can say that adequately testing each individual program component in isolation does not necessarily mean that we have adequately tested the entire program (the program as a whole). When we integrate two separate program components, there are interactions that cannot arise in the isolated components. Axioms 7 and 8 have special impact on the testing of object oriented code. These issues are covered in Chapter 6.

9. Renaming Property

If P is a renaming of Q, then T is adequate for P only if T is adequate for Q. A program P is a renaming of Q if P is identical to Q expect for the fact that all instances of an identifier, let us say a in Q have been replaced in P by an identifier, let us say b, where b does not occur in *Q, or if there is a set of such renamed identifiers. This axiom simply* says that an inessential change in a program such as changing the names of the variables should not change the nature of the test data that are needed to adequately test the program.

10. Complexity Property

For every n, there is a program P such that P is adequately tested by a size n test set, but not by any size n 1 test set.This means that for every program, there are other programs that require more testing.

11. Statement Coverage Property

If the test set T is adequate for P, then T causes every executable statement of P to be executed.Ensuring that their test set executed all statements in a program is a minimum coverage goal for a tester. A tester soon realizes that if some portion of the program has never been executed, then that portion could contain defects: it could be totally in error and be working improperly. Testing would not be able to detect any defects in this portion of the code. However, this axiom implies that a tester needs to be able to determine which statements of a program are executable. It is possible that not all of program statements are executable. Unfortunately, there is no algorithm to support the tester in the latter task, but

Weyuker believes that developers/testers are quite good at determining whether or not code is, or is not, executable [2]. Issues relating to infeasible (unexecutable) paths, statements, and branches have been discussed in Section 5.4.

The first eight axioms as described by Weyuker exposed weaknesses in several well-known program-based adequacy criteria. For example , both statement and branch adequacy criteria were found to fail in satisfying several of the axioms including the applicability axiom. Some data flow adequacy criteria also failed to satisfy the applicability axiom. An additional three axioms/properties (shown here as 9-11) were added to the original set to provide an even stronger framework for evaluating test adequacy criteria. Weyuker meant for these axioms to be used as a tool by testers to understand the strengths and weaknesses of the criteria they select. Note that each criterion has a place on the Subsumes hierarchy as shown in Figure 5.5. A summary showing several criteria and eight of the axioms they satisfy, and fail to satisfy, is shown in Table 5.2 [11].

Weyuker's goal for the research community is to eventually develop criteria that satisfy all of the axioms. Using these new criteria, testers will be able to have greater confidence that the code under test has been adequately tested. Until then testers will need to continue to use exiting criteria such as branch- and statement-based criteria. However, they should be aware of inherent weaknesses of each, and use combinations of criteria and different testing techniques to adequately test a program.

Unit II

Part-A Questions

1. Define Smart tester
2. What is white box testing?
3. Define:Black box testing.
4. Define:Random testing.
5. Write a note on COTS components.
6. Write a note on: Equivalence class partitioning, Boundary value analysis.

Part-B Questions

1. Explain about the following methods of black box testing with example.

   (1) Equivalence class partitioning.

   (2) Boundary value analysis.

2. Explain COTS components.
3. Write a note on the following:

   (1) Loop testing (2)Mutation testing.

4. Explain evaluating test adequacy criteria.

UNIT III LEVELS OF TESTING

3.1 The need for levels of testing

Execution-based software testing, especially for large systems, is usually carried out at different levels. In most cases there will be 3-4 levels, or major phases of testing: unit test, integration test, system test, and some type of acceptance test as shown in Figure 6.1. Each of these may consist of one or more sublevels or phases. At each level there are specific testing goals. For example, at unit test a single component is tested. A principal goal is to detect functional and structural defects in the unit. At the integration level several components are tested as a group, and the tester investigates component interactions. At the system level the system as a whole is tested and a principle goal is to evaluate attributes such as usability, reliability, and performance. System test begins when all of the components have been integrated successfully. It usually requires the bulk of testing resources. Laboratory equipment, special software, or special hardware may be necessary, especially for real-time, embedded, or distributed systems. At the system level the tester looks for defects, but the focus is on evaluating performance, usability, reliability, and other quality-related requirements.

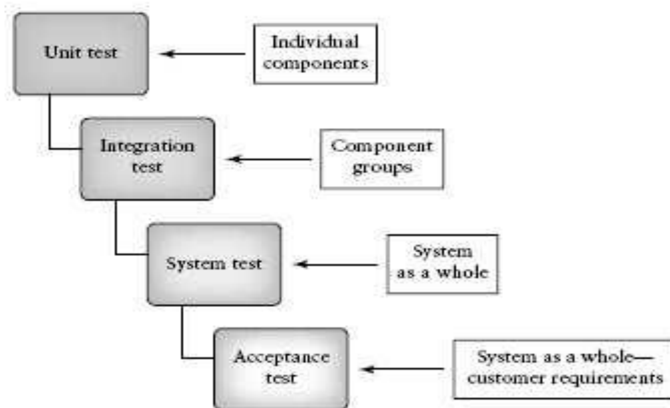


FIG. 6.1
Levels of testing.

The approach used to design and develop a software system has an impact on how testers plan and design suitable tests. There are two major approaches to system development—bottom-up, and top-down. These approaches are supported by two major types of programming languages— procedure-oriented and object-oriented.

3.2 Unit test

A unit is the smallest possible testable software component.It can be characterized in several ways. For example, a unit in a typical procedure-oriented software system:
• performs a single cohesive function;
• can be compiled separately;
• is a task in a work breakdown structure (from the manager's pointof view);
• contains code that can fit on a single page or screen.

A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language. In object-oriented systems both the method and the class/object have been suggested by researchers as the choice for a unit [1-5]. The relative merits of each of these as the selected component for unit test are described in sections that follow. A unit may also be a small-sized COTS component purchased from an outside vendor that is

undergoing evaluation by the purchaser, or a simple module retrieved from an in-house reuse library.

Fig. 6.2

Some components suitable for unit test.

3.3 Unit test planning

A general unit test plan should be prepared. It may be prepared as a component of the master test plan or as a stand-alone plan. It should be developed in conjunction with the master test plan and the project plan for each project. Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units. Components of a unit test plan are described in detail the *IEEE Standard for Software Unit Testing* . This standard is rich in information and is an excellent guide for the test planner. A brief description of a set of development phases for unit test planning is found below. In each phase a set of activities is assigned based on those found in the IEEE unit test standard . The phases allow a steady evolution of the unit test plan as more information becomes available. The reader will note that the unit test plan contains many of the same components as the master test plan that will be described in Chapter 7. Also note that a unit test plan is developed to cover all the units within a software project; however, each unit will have its own associated set of tests.

Phase 1: Describe Unit Test Approach and Risks
In this phase of unit testing planning the general approach to unit testing is outlined. The test planner:
(i) identifies test risks;
(ii) describes techniques to be used for designing the test cases for the units;
(iii) describes techniques to be used for data validation and recording of test results;
(iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.

During this phase the planner also identifies completeness requirements—
what will be covered by the unit test and to what degree (states, functionality, control, and data flow patterns). The planner also identifies termination conditions for the unit tests. This includes coverage requirements, and special cases. Special cases may result in abnormal termination of unit test (e.g., a major design flaw). Strategies for handling these special cases need to be documented. Finally, the planner estimates resources needed for unit test, such as hardware, software, and staff, and develops a tentative schedule under the constraints identified at that time.

Phase 2: Identify Unit Features to be Tested
This phase requires information from the unit specification and detailed design description. The planner determines which features of each unit will be tested, for example: functions, performance requirements, states, and state transitions, control structures, messages, and data flow patterns. If some features will not be covered by the tests, they should be mentioned and the risks of not testing them be assessed. Input/output characteristics associated with each unit should also be identified, such as variables with an allowed ranges of values and performance at a certain level.

Phase 3: Add Levels of Detail to the Plan

In this phase the planner refines the plan as produced in the previous two phases. The planner adds new details to the approach, resource, and scheduling portions of the unit test plan. As an example, existing test cases that can be reused for this project can be identified in this phase. Unit availability and integration scheduling information should be included in the revised version of the test plan. The planner must be sure to include a description of how test results will be recorded. Test-related documents that will be required for this task, for example, test logs, and test incident reports, should be described, and references to standards for these documents provided. Any special tools required for the tests are also described. The next steps in unit testing consist of designing the set of test cases, developing the auxiliary code needed for testing, executing the tests, and recording and analyzing the results.

3.4 Designing the unit tests

Part of the preparation work for unit test involves unit test design. It is important to specify (i) the test cases (including input data, and expected outputs for each test case), and, (ii) the test procedures (steps required run the tests). These items are described in more detail in Chapter 7. Test case data should be tabularized for ease of use, and reuse. Suitable tabular formats for test cases are found in Chapters 4 and 5. To specifically support object-oriented test design and the organization of test data, Berard has described a test case specification notation. He arranges the components of a test case into a semantic network with parts, Object_ID, Test_Case_ID, Purpose, and List_of_Test_Case_Steps. Each of these items has component parts. In the test design specification Berard also includes lists of relevant states, messages (calls to methods), exceptions, and interrupts. As part of the unit test design process, developers/testers should also describe the relationships between the tests. Test suites can be defined that bind related tests together as a group. All of this test design information is attached to the unit test plan. Test cases, test procedures, and test suites may be reused from past projects if the organization has been careful to store them so that they are easily retrievable and reusable.

Test case design at the unit level can be based on use of the black and white box test design strategies described in Chapters 4 and 5. Both of these approaches are useful for designing test cases for functions and procedures. They are also useful for designing tests for the individual methods (member functions) contained in a class. Considering the relatively small size of a unit, it makes sense to focus on white box test design for procedures/functions and the methods in a class. This approach gives the tester the opportunity to exercise logic structures and/or data flow sequences, or to use mutation analysis, all with the goal of evaluating the structural integrity of the unit. Some black box-based testing is also done at unit level; however, the bulk of black box testing is usually done at the integration and system levels and beyond. In the case of a smaller-sized COTS component selected for unit testing, a black box test design approach may be the only option. It should be mentioned that for units that perform mission/safely/business critical functions, it is often useful and prudent to design stress, security, and performance tests at the unit level if possible.

3.5 The class as a testable unit

If an organization is using the object-oriented paradigm to develop software systems it will need to select the component to be considered for unit test. As described in Section 6.1, the choices consist of either the individual method as a unit or the class as a whole. Each of these choices requires special consideration on the part of the testers when designing and running the unit tests, and when retesting needs to be done. For example, in the case of the method as the selected unit to test, it may call other methods within its own class to support its functionality. Additional code, in the form of a test harness, must be built to represent the called methods within the class. Building such a test harness for each individual method often requires developing code equivalent to that already existing in the class itself (all of its other methods). This is costly; however, the tester needs to consider that testing each individual method in this way helps to ensure that all statements/branches have been executed at least once, and that the basic functionality of the method is correct. This is especially important for mission or safety critical methods.

In spite of the potential advantages of testing each method individually, many developers/testers consider the class to be the component of choice for unit testing. The process of testing classes as units is sometimes called component test . A class encapsulates multiple interacting methods operating on common data, so what we are testing is the intraclass interaction of the methods. When testing on the class level we are able detect not only traditional types of defects, for example, those due to control or data flow errors, but also defects due to the nature of objectoriented systems, for example, defects due to encapsulation, inheritance, and polymorphism errors. We begin to also look for what Chen calls object management faults, for example, those associated with the instantiation, storage, and retrieval of objects .

This brief discussion points out some of the basic trade-offs in selecting the component to be considered for a unit test in object-oriented systems. If the class is the selected component, testers may need to address special issues related to the testing and retesting of these components. Some of these issues are raised in the paragraphs that follow.

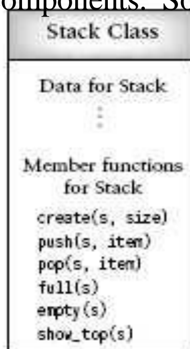


Fig. 6.3
Sample stack class with multiple methods.

Issue 1: Adequately Testing Classes

The potentially high costs for testing each individual method in a class have been described. These high costs will be particularly apparent when there are many methods in a class; the numbers can reach as high as 20 to 30. If the class is selected as the unit to test, it is possible to reduce these costs since in many cases the methods in a single class serve as drivers and stubs for one another. This has the effect of lowering the complexity of the test harness that needs to be developed. However, in some cases driver classes that represent outside classes using the methods of the class under test will have to be developed. In addition, if it is decided that the class is the smallest component to test, testers must decide if they are able to adequately cover all necessary features of each method in class testing. Some researchers believe that coverage objectives and test data need to be developed for each of the methods, for example, the *create, pop, push, empty, full,* and *show_top* methods associated with the stack class shown in Figure 6.3. Other researchers believe that a class can be adequately tested as a whole by observation of method interactions using a sequence of calls to the member functions with appropriate parameters.

Again, referring to the stack class shown in Figure 6.3, the methods *push, pop, full, empty,* and *show_top* will either read or modify the state of the stack. When testers unit (or component) test this class what they will need to focus on is the operation of each of the methods in the class and the interactions between them. Testers will want to determine, for example, if *push* places an item in the correct position at the top of the stack. However, a call to the method *full* may have to be made first to determine if the stack is already full. Testers will also want to determine if *push* and *pop* work together properly so that the stack pointer is in the correct position after a sequence of calls to these methods. To properly test this class, a sequence of calls to the methods needs to be specified as part of component test design. For example, a test sequence for a stack that can hold three items might be:

create(s,3), empty(s), push(s,item-1), push(s,item-2), push(s,item-3),
full(s), show_top(s), pop(s,item), pop(s,item), pop(s,item), empty(s), . . .

The reader will note that many different sequences and combination of calls are possible even for this simple class. Exhaustively testing every possible sequence is usually not practical. The tester must select those sequences she believes will reveal the most defects in the class. Finally, a tester might use a combination of approaches, testing some of the critical methods on an individual basis as units, and then testing the class as a whole.

Issue 2: Observation of Object States and State Changes

Methods may not return a specific value to a caller. They may instead change the state of an object. The state of an object is represented by a specific set of values for its attributes or state variables. State-based testing as described in Chapter 4 is very useful for testing objects.

Methods will often modify the state of an object, and the tester must ensure that each state transition is proper. The test designer can prepare a state table (using state diagrams developed for the requirements specification) that specifies states the object can assume, and then in the table indicate sequence of messages and parameters that will cause the object to enter each state. When the tests are run the tester can enter results in this same type of table. For example, the first call to the method *push* in the stack class of Figure 6.3, changes the state of the stack so that

empty is no longer true. It also changes the value of the stack pointer variable, *top.* To determine if the method *push* is working properly the value of the variable *top* must be visible both before and after the invocation of this method. In this case the method *show_top* within the class may be called to perform this task. The methods *full* and *empty* also probe the state of the stack. A sample augmented sequence of calls to check the value of *top* and the *full/empty* state of the three-item stack is:

empty(s), push(s,item-1), show_top(s), push(s,item-2),
show_top(s), push(s,item-3), full(s), show_top(s), pop(s,item),
show_top(s), pop(s,item), show_top(s), empty(s), . . .

Issue 3: The Retesting of Classes—I

One of the most beneficial features of object-oriented development is encapsulation. This is a technique that can be used to hide information. A program unit, in this case a class, can be built with a well-defined public interface that proclaims its services (available methods) to client classes. The implementation of the services is private. Clients who use the services are unaware of implementation details. As long as the interface is unchanged, making changes to the implementation should not affect the client classes. A tester of object-oriented code would therefore conclude that only the class with implementation changes to its methods needs to be retested. Client classes using unchanged interfaces need not be retested. In an object-oriented system, if a developer changes a class implementation that class needs to be retested as well as all the classes that depend on it. If a superclass, for example, is changed, then it is necessary to retest all of its subclasses. In addition, when a new subclass is added (or modified), we must also retest the methods inherited from each of its ancestor superclasses. The new (or changed) subclass introduces an unexpected form of dependency because there now exists a new context for the inherited components.

Issue 4: The Retesting of Classes—II

Classes are usually a part of a class hierarchy where there are existing inheritance relationships. Subclasses inherit methods from their superclasses. Very often a tester may assume that once a method in a superclass has been tested, it does not need retested in a subclass that inherits it. However, in some cases the method is used in a different context by the subclass and will need to be retested. In addition, there may be an overriding of methods where a subclass may replace an inherited method with a locally defined method. Not only will the new locally defined method have to be retested, but designing a new set of test cases may be necessary. This is because the two methods (inherited and new) may be structurally different. The antiextentionality axiom as discussed in Chapter 5 expresses this need .

The following is an example of such as case using the shape class in Figure 6.4. Suppose the shape superclass has a subclass, triangle, and triangle has a subclass, equilateral triangle. Also suppose that the method *display* in shape needs to call the method *color* for its operation. Equilateral triangle could have a local definition for the method *display.* That method could in turn use a local definition for *color* which has been defined in triangle. This local definition of the *color* method in triangle has been tested to work with the inherited *display* method in shape, but not with the locally defined *display* in equilateral triangle. This is a new context that must be

retested. A set of new test cases should be developed. The tester must carefully examine all the relationships between members of a class to detect such occurrences.

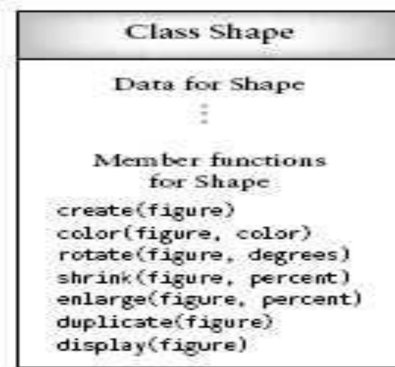


Fig. 6.4
Sample shape class.

3.6 The test harness

In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world. Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit. This code called the test harness, is developed especially for test and is in addition to the code that composes the system under development. The role is of the test harness is shown in Figure 6.5 and it is defined as follows:

The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls.

The development of drivers and stubs requires testing resources. The drivers and stubs must be

Fig. 6.5
The test harness.

tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software. Drivers and stubs can be developed at several levels of functionality. For example, a driver could have the following options and combinations of options:

(i) call the target unit;
(ii) do 1, and pass inputs parameters from a table;
(iii) do 1, 2, and display parameters;
(iv) do 1, 2, 3 and display results (output parameters).

The stubs could also exhibit different levels of functionality. For example a stub could:

(i) display a message that it has been called by the target unit;
(ii) do 1, and display any input parameters passed from the target unit;
(iii) do 1, 2, and pass back a result from a table;
(iv) do 1, 2, 3, and display result from table.

Drivers and stubs as shown in Figure 6.5 are developed as procedures and functions for traditional imperative-language based systems. For object-oriented systems, developing drivers and stubs often means the design and implementation of special classes to perform the required testing tasks. The test harness itself may be a hierarchy of classes. For example, in Figure 6.5 the driver for a procedural system may be designed as a single procedure or main module to call the unit under test; however, in an object-oriented system it may consist of several test classes to emulate all the classes that call for services in the class under test. Researchers such as Rangaraajan and Chen have developed tools that generate test cases using several different approaches, and classes of test harness objects to test object-oriented code .

The test planner must realize that, the higher the degree of functionally for the harness, the more resources it will require to design, implement, and test. Developers/testers will have to decide

depending on the nature of the code under test, just how complex the test harness needs to be. Test harnesses for individual classes tend to be more complex than those needed for individual procedures and functions since the items being tested are more complex and there are more interactions to consider.

3.7 Running the unit tests and recording results

Unit tests can begin when (i) the units becomes available from the developers (an estimation of availability is part of the test plan), (ii) the test cases have been designed and reviewed, and (iii) the test harness, and any other supplemental supporting tools, are available. The testers then proceed to run the tests and record results. Chapter 7 will describe documents called test logs that can be used to record the results of specific tests. The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format such as shown in Table 6.1. These forms can be included in the test summary report, and are of value at the weekly status meetings that are often used to monitor test progress. It is very important for the tester at any level of testing to carefully record, review, and check test results. The tester must determine from the results whether the unit has passed or failed the test. If the test is failed, the nature of the

Unit Test Worksheet

Unit Name: ______________________

Unit Identifier: ______________________

Tester: ______________________

Date: ______________________

Test case ID	Status (run/not run)	Summary of results	Pass/fail

TABLE 6.1

Summary work sheet for unit test results.

problem should be recorded in what is sometimes called a test incident report (see Chapter 7). Differences from expected behavior should be described in detail. This gives clues to the developers to help them locate any faults. During testing the tester may determine that additional tests are required. For example, a tester may observe that a particular coverage goal has not been achieved. The test set will have to be augmented and the test plan documents should reflect these changes. When a unit fails a test there may be several reasons for the failure. The most likely reason for the failure is a fault in the unit implementation (the code). Other likely causes that need to be carefully investigated by the tester are the following:

• a fault in the test case specification (the input or the output was not specified correctly);
• a fault in test procedure execution (the test should be rerun);
• a fault in the test environment (perhaps a database was not set up properly);
• a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

The causes of the failure should be recorded in a test summary report, which is a summary of testing activities for all the units covered by the unit test plan.

Ideally, when a unit has been completely tested and finally passes all of the required tests it is ready for integration. Under some circumstances unit may be given a conditional acceptance for integration test. This may occur when the unit fails some tests, but the impact of the failure is not significant with respect to its ability to function in a subsystem, and the availability of a unit is critical for integration test to proceed on schedule. This a risky procedure and testers should evaluate the risks involved. Units with a conditional pass must eventually be repaired. When testing of the units is complete, a test summary report should be prepared. This is a valuable document for the groups responsible for integration and system tests. It is also a valuable component of the project history. Its value lies in the useful data it provides for test process improvement and defect prevention. Finally, the tester should insure that the test cases, test procedures, and test harnesses are preserved for future reuse.

3.8 Integration tests

Integration test for procedural code has two major goals:

(i) to detect defects that occur on the interfaces of units;
(ii) to assemble the individual units into working subsystems and finally a complete system that is ready for system test.

In unit test the testers attempt to detect defects that are related to the functionality and structure of the unit. There is some simple testing of unit interfaces when the units interact with drivers and stubs. However, the interfaces are more adequately tested during integration test when each unit is finally connected to a full and working implementation of those units it calls, and those that call it. As a consequence of this assembly or integration process, software subsystems and finally a completed system is put together during the integration test. The completed system is then ready for system testing.

With a few minor exceptions, integration test should only be performed on units that have been reviewed and have successfully passed unit testing. A tester might believe erroneously that since a unit has already been tested during a unit test with a driver and stubs, it does not need to be retested in combination with other units during integration test. However, a unit tested in isolation may not have been tested adequately for the situation where it is combined with other modules. This is also a consequences of one of the testing axioms found in Chapter 4 called anticomposition.

Integration testing works best as an iterative process in proceduraloriented system. One unit at a time is integrated into a set of previously integrated modules which have passed a set of integration tests. The interfaces and functionally of the new unit in combination with the previously integrated units is tested. When a subsystem is built from units integrated in this stepwise manner, then performance, security, and stress tests can be performed on this subsystem.

Integrating one unit at a time helps the testers in several ways. It keeps the number of new interfaces to be examined small, so tests can focus on these interfaces only. Experienced testers know that many defects occur at module interfaces. Another advantage is that the massive failures that often occur when multiple units are integrated at once is avoided. This approach also helps the developers; it allows defect search and repair to be confined to a small known number of components and interfaces. Independent subsystems can be integrated in parallel as long as the required units are available. The integration process in object-oriented systems is driven by assembly of the classes into cooperating groups. The cooperating groups of classes are tested as a whole and then combined into higher-level groups. Usually the simpler groups are tested first, and then combined to form higher-level groups until the system is assembled.

3.9 Designing integration tests

Integration tests for procedural software can be designed using a black or white box approach. Both are recommended. Some unit tests can be reused. Since many errors occur at module interfaces, test designers need to focus on exercising all input/output parameter pairs, and all calling relationships. The tester needs to insure the parameters are of the correct type and in the correct order. The author has had the personal experience of spending many hours trying to locate a fault that was due to an incorrect ordering of parameters in the calling routine. The tester must also insure that once the parameters are passed to a routine they are used correctly. For example, in Figure 6.9, Procedure_b is being integrated with Procedure_a. Procedure_a calls Procedure_b with two input parameters in3, in4. Procedure_b uses those parameters and then returns a value for the output parameter out1. Terms such as *lhs* and *rhs* could be any variable or expression. The reader should interpret the use of the variables in the broadest sense. The parameters could be involved in a number of *def* and/or *use* data flow patterns. The actual usage

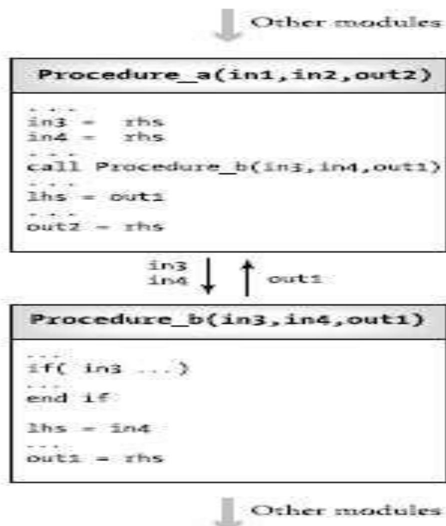


Fig. 6.9
Example integration of two procedures.

patterns of the parameters must be checked at integration time. Data flow-based (def-use paths) and control flow (branch coverage) test data generation methods are useful here to insure that the

input parameters, in3, in4, are used properly in Procedure_b. Again data flow methods (def-use pairs) could also be used to check that the proper sequence of data flow operations is being carried out to generate the correct value for out1 that flows back to Procedure_a.

Black box tests are useful in this example for checking the behavior of the pair of procedures. For this example test input values for the input parameters in1 and in2 should be provided, and the outcome in out2 should be examined for correctness. For conventional systems, input/output parameters and calling relationships will appear in a structure chart built during detailed design. Testers must insure that test cases are designed so that all modules in the structure chart are called at least once, and all called modules are called by every caller. The reader can visualize these as coverage criteria for integration test. Coverage requirements for the internal logic of each of the integrated units should be achieved during unit tests. Some black box tests used for module integration may be reusable from unit testing. However, when units are integrated and subsystems are to be tested as a whole, new tests will have to be designed to cover their functionality and adherence to performance and other requirements (see example above).
Sources for development of black box or functional tests at the integration level are the requirements documents and the user manual. Testers need to work with requirements analysts to insure that the requirements are testable, accurate, and complete. Black box tests should be developed to insure proper functionally and ability to handle subsystem stress. For example, in a transaction-based subsystem the testers want to determine the limits in number of transactions that can be handled. The tester also wants to observe subsystem actions when excessive amounts of transactions are generated. Performance issues such as the time requirements for a transaction should also be subjected to test. These will be repeated when the software is assembled as a whole and is undergoing system test.

Integration testing of clusters of classes also involves building test harnesses which in this case are special classes of objects built especially for testing. Whereas in class testing we evaluated intraclass method interactions, at the cluster level we test interclass method interaction as well. We want to insure that messages are being passed properly to interfacing objects, object state transitions are correct when specific events occur, and that the clusters are performing their required functions. Unlike procedural-oriented systems, integration for object-oriented systems usually does not occur one unit at a time. A group of cooperating classes is selected for test as a cluster. In their object-oriented testing framework the method is the entity selected for unit test. The methods and the classes they belong to are connected into clusters of classes that are represented by a directed graph that has two special types of entities.These are method-message paths, and atomic systems functions that represent input port events. A method-message path is described as a sequence of method executions linked by messages. An atomic system function is an input port event (start event) followed by a set of method messages paths and terminated by an output port event (system response).Murphy et al. define clusters as classes that are closely coupled and work together to provide a unified behavior [5]. Some examples of clusters are groups of classes that produce a report, or monitor and control a device.Scenarios of operation from the design document associated with a cluster are used to develop test cases. Murphy and his co-authors have developed a tool that can be used for class and cluster testing.

3.10 Integration test planning

Integration test must be planned. Planning can begin when high-level design is complete so that the system architecture is defined. Other documents relevant to integration test planning are the requirements document, the user manual, and usage scenarios. These documents contain structure charts, state charts, data dictionaries, cross-reference tables, module interface descriptions, data flow descriptions, messages and event descriptions, all necessary to plan integration tests. The strategy for integration should be defined. For procedural-oriented system the order of integration of the units of the units should be defined. This depends on the strategy selected. Consider the fact that the testing objectives are to assemble components into subsystems and to demonstrate that the subsystem functions properly with the integration test cases. For object-oriented systems a working definition of a cluster or similar construct must be described, and relevant test cases must be specified. In addition, testing resources and schedules for integration should be included in the test plan.The plan includes the following items:

(i) clusters this cluster is dependent on;
(ii) a natural language description of the functionality of the cluster to be tested;
(iii) list of classes in the cluster;
(iv) a set of cluster test cases.

As stated earlier in this section, one of the goals of integration test is to build working subsystems, and then combine these into the system as a whole. When planning for integration test the planner selects subsystems to build based upon the requirements and user needs. Very often subsystems selected for integration are prioritized. Those that represent key features, critical features, and/or user-oriented functions may be given the highest priority. Developers may want to show clients that certain key subsystems have been assembled and are minimally functional.

3.11 System test - The different types

When integration tests are completed, a software system has been assembled and its major subsystems have been tested. At this point the developers/ testers begin to test it as a whole. System test planning should begin at the requirements phase with the development of a master test plan and requirements-based (black box) tests. System test planning is a complicated task. There ar e many components of the plan that need to be prepared such as test approaches, costs, schedules, test cases, and test procedures. All of these are examined and discussed in Chapter 7.

System testing itself requires a large amount of resources. The goal is to ensure that the system performs according to its requirements. System test evaluates both functional behavior and quality requirements such as reliability, usability, performance and security. This phase of testing is especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage. After system test the software will be turned over to users for evaluation during acceptance test or alpha/beta test. The organization will want to be sure that the quality of the software has been measured and evaluated before users/clients are invited to use the system. In fact system test serves as a good rehearsal scenario for acceptance test.

Because system test often requires many resources, special laboratory equipment, and long test times, it is usually performed by a team of testers. The best scenario is for the team to be part of an independent testing group. The team must do their best to find any weak areas in the software; therefore, it is best that no developers are directly involved. There are several types of system tests as shown on Figure 6.10. The types are as follows:

- Functional testing
- Performance testing
- Stress testing
- Configuration testing
- Security testing
- Recovery testing

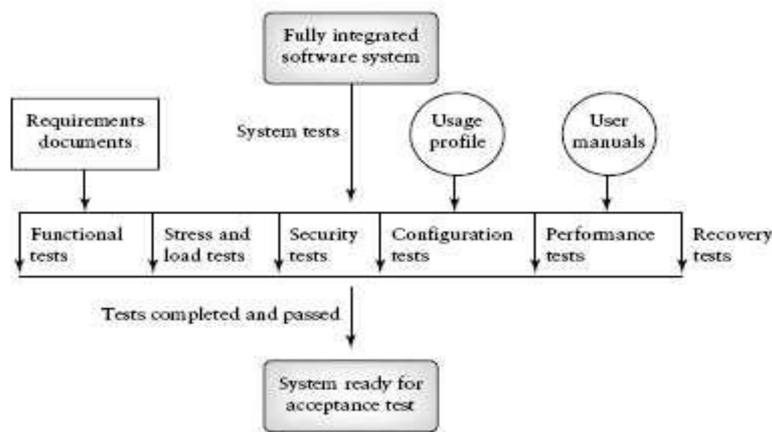


Fig. 6.10
Types of system tests.

F u n c t i o n a l T e s t i n g

System functional tests have a great deal of overlap with acceptance tests. Very often the same test sets can apply to both. Both are demonstrations of the system's functionality. Functional tests at the system level are used to ensure that the behavior of the system adheres to the requirements specification. All functional requirements for the system must be achievable by the system. For example, if a personal finance system is required to allow users to set up accounts, add, modify, and delete entries in the accounts, and print reports, the function-based system and acceptance tests must ensure that the system can perform these tasks. Clients and users will expect this at acceptance test time. Functional tests are black box in nature. The focus is on the inputs and proper outputs for each function. Improper and illegal inputs must also be handled by the system. System behavior under the latter circumstances tests must be observed. All functions must be tested. In fact, the tests should focus on the following goals.

- All types or classes of legal inputs must be accepted by the software.
- All classes of illegal inputs must be rejected (however, the system should remain available).

• All possible classes of system output must exercised and examined.
• All effective system states and state transitions must be exercised and examined.
• All functions must be exercised.

PerformanceTesting

An examination of a requirements document shows that there are two major types of requirements:

1. *Functional requirements.* Users describe what functions the software should perform. We test for compliance of these requirements at the system level with the functional-based system tests.
2. *Quality requirements.* There are nonfunctional in nature but describe quality levels expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughput, and delays. The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test whether there are any hardware or software factors that impact on the system's performance. Performance testing allows testers to tune the system; that is, to optimize the allocation of system resources. For example, testers may find that they need to reallocate memory pools, or to modify the priority level of certain system operations. Testers may also be able to project the system's future performance levels. This is useful for planning subsequent releases.

Performance objectives must be articulated clearly by the users/clients in the requirements documents, and be stated clearly in the system test plan. The objectives must be quantified. For example, a requirement that the system return a response to a query in "a reasonable amount of time" is not an acceptable requirement; the time requirement must be specified in quantitative way. Results of performance tests are quantifiable. At the end of the tests the tester will know, for example, the number of CPU cycles used, the actual response time in seconds (minutes, etc.), he actual number of transactions processed per time period. These can be evaluated with respect to requirements objectives.

StressTesting

When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing. For example, if an operating system is required to handle 10 interrupts/second and the load causes 20 interrupts/second, the system is being stressed. The goal of stress test is to try to break the system; find the circumstances under which it will crash. This is sometimes called "breaking the system." An everyday analogy can be found in the case where a suitcase being tested for strength and endurance is stomped on by a multiton elephant!

Stress testing is important because it can reveal defects in real-time and other types of systems, as well as weak areas where poor design could cause unavailability of service. For example, system prioritization orders may not be correct, transaction processing may be poorly designed and waste memory space, and timing sequences may not be appropriate for the required tasks. This is particularly important for real-time systems where unpredictable events may occur resulting in input loads that exceed those described in the requirements documents. Stress testing

often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation of the software system.

System limits and threshold values are exercised. Hardware and software interactions are stretched to the limit. All of these conditions are likely to reveal defects and design flaws which may not be revealed under normal testing conditions. Stress testing is supported by many of the resources used for performance test as shown in Figure 6.11. This includes the load generator. The testers set the load generator parameters so that load levels cause stress to the system. For example, in our example of a telecommunication system, the arrival rate of calls, the length of the calls, the number of misdials, as well as other system parameters should all be at stress levels. As in the case of performance test, special equipment and laboratory space may be needed for the stress tests. Examples are hardware or software probes and event loggers. The tests may need to run for several days. Planners must insure resources are available for the long time periods required. The reader should note that stress tests should also be conducted at the integration, and if applicable at the unit level, to detect stress-related defects as early as possible in the testing process. This is especially critical in cases where redesign is needed.

Stress testing is important from the user/client point of view. When system operate correctly under conditions of stress then clients have confidence that the software can perform as required. Beizer suggests that devices used for monitoring stress situations provide users/clients with visible and tangible evidence that the system is being stressed.

Configuration Testing

Typical software systems interact with hardware devices such as disc drives, tape drives, and printers. Many software systems also interact with multiple CPUs, some of which are redundant. Software that controls realtime processes, or embedded software also interfaces with devices, but these are very specialized hardware items such as missile launchers, and nuclear power device sensors. In many cases, users require that devices be interchangeable, removable, or reconfigurable. For example, a printer of type X should be substitutable for a printer of type Y, CPU A should be removable from a system composed of several other CPUs, sensor A should be replaceable with sensor B. Very often the software will have a set of commands, or menus, that allows users to make these configuration changes. Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur. Configuration testing also requires many resources including the multiple hardware devices used for the tests. If a system does not have specific requirements for device configuration changes then large-scale configuration testing is not essential.

According to Beizer configuration testing has the following objectives:
• Show that all the configuration changing commands and menus work properly.
• Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.
• Show that the systems' performance level is maintained when devices are interchanged, or when they fail.

Several types of operations should be performed during configuration test. Some sample operations for testers are:
(i) rotate and permutate the positions of devices to ensure physical/ logical device permutations work for each device (e.g., if there are two printers A and B, exchange their positions); **(ii)** induce malfunctions in each device, to see if the system properly handles the malfunction; **(iii)** induce multiple device malfunctions to see how the system reacts. These operations will help to reveal problems (defects) relating to hardware/software interactions when hardware exchanges, and reconfigurations occur. Testers observe the consequences of these operations and determine whether the system can recover gracefully particularly in the case of a malfunction.

Security Testing

Designing and testing software systems to insure that they are safe and secure is a big issue facing software developers and test specialists. Recently, safety and security issues have taken on additional importance due to the proliferation of commercial applications for use on the Internet. If Internet users believe that their personal information is not secure and is available to those with intent to do harm, the future of e-commerce is in peril! Security testing evaluates system characteristics that relate to the availability, integrity, and confidentially of system data and services. Users/clients should be encouraged to make sure their security needs are clearly known at requirements time, so that security issues can be addressed by designers and testers. Computer software and data can be compromised by:
(i) criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy;
(ii) errors on the part of honest developers/maintainers who modify, destroy, or compromise data because of misinformation, misunderstandings,and/or lack of knowledge.

Both criminal behavior and errors that do damage can be perpetuated by those inside and outside of an organization. Attacks can be random or systematic. Damage can be done through various means such as:
(i) viruses;
(ii) trojan horses;
(iii) trap doors;
(iv) illicit channels.

The effects of security breaches could be extensive and can cause:
(i) loss of information;
(ii) corruption of information;
(iii) misinformation;
(iv) privacy violations;
(v) denial of service.

Physical, psychological, and economic harm to persons or property can result from security breaches. Developers try to ensure the security of their systems through use of protection mechanisms such as passwords, encryption, virus checkers, and the detection and elimination of trap doors. Developers should realize that protection from unwanted entry and other security-

oriented matters must be addressed at design time. A simple case in point relates to the characteristics of a password. Designers need answers to the following: What is the minimum and maximum allowed length for the password? Can it be pure alphabetical or must it be a mixture of alphabetical and other characters? Can it be a dictionary word? Is the password permanent, or does it expire periodically? Users can specify their needs in this area in the requirements document. A password checker can enforce any rules the designers deem necessary to meet security requirements.

Password checking and examples of other areas to focus on during security testing are described below.

Password Checking—Test the password checker to insure that users will select a password that meets the conditions described in the password checker specification. Equivalence class partitioning and boundary value analysis based on the rules and conditions that specify a valid password can be used to design the tests.

Legal and Illegal Entry with Passwords—Test for legal and illegal system/data access via legal and illegal passwords.

Password Expiration—If it is decided that passwords will expire after a certain time period, tests should be designed to insure the expiration period is properly supported and that users can enter a new and appropriate password.

Encryption—Design test cases to evaluate the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.

Browsing—Evaluate browsing privileges to insure that unauthorized browsing does not occur. Testers should attempt to browse illegally and observe system responses. They should determine what types of private information can be inferred by both legal and illegal browsing.

Trap Doors—Identify any unprotected entries into the system that may allow access through unexpected channels (trap doors). Design tests that attempt to gain illegal entry and observe results. Testers will need the support of designers and developers for this task. In many cases an external "tiger team" as described below is hired to attempt such a break into the system.

Viruses—Design tests to insure that system virus checkers prevent or curtail entry of viruses into the system. Testers may attempt to infect the system with various viruses and observe the system response. If a virus does penetrate the system, testers will want to determine what has been damaged and to what extent.

Even with the backing of the best intents of the designers, developers/ testers can never be sure that a software system is totally secure even after extensive security testing. If security is an especially important issue, as in the case of network software, then the best approach if resources permit, is to hire a so-called "tiger team" which is an outside group of penetration experts who attempt to breach the system security. Although a testing group in the organization can be

involved in testing for security breaches, the tiger team can attack the problem from a different point of view. Before the tiger team starts its work the system should be thoroughly tested at all levels. The testing team should also try to identify any trap doors and other vulnerable points. Even with the use of a tiger team there is never any guarantee that the software is totally secure.

R e c o v e r y T e s t i n g

Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses. This type of testing is especially important for transaction systems, for example, on-line banking software. A test scenario might be to emulate loss of a device during a transaction. Tests would determine if the system could return to a wellknown state, and that no transactions have been compromised. Systems with automated recovery are designed for this purpose. They usually have multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device. They also have a so-called “checkpoint” system that meticulously records transactions and system states periodically so that these are preserved in case of failure. This information allows the system to return to a known state after the failure. The recovery testers must ensure that the device monitoring system and the checkpoint software are working properly.

Beizer advises that testers focus on the following areas during recovery testing :
1. *Restart.* The current system state and transaction states are discarded.The most recent checkpoint record is retrieved and the system initialized to the states in the checkpoint record. Testers must insure that all transactions have been reconstructed correctly and that all devices are in the proper state. The system should then be able to begin to process new transactions.
2. *Switchover.* The ability of the system to switch to a new processor must be tested. Switchover is the result of a command or a detection of a faulty processor by a monitor. In each of these testing situations all transactions and processes must be carefully examined to detect:
(i) loss of transactions;
(ii) merging of transactions;
(iii) incorrect transactions;
(iv) an unnecessary duplication of a transaction.

A good way to expose such problems is to perform recovery testing under a stressful load. Transaction inaccuracies and system crashes are likely to occur with the result that defects and design flaws will be revealed.

3.12 Regression testing

Regression testing is not a level of testing, but it is the *retesting* of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes. Regression testing can occur at any level of test, for example, when unit tests are run the unit may pass a number of these tests until one of the tests does reveal a defect. The unit is repaired and then retested with all the old test cases to ensure that the changes have not affected its functionality. Regression tests are especially important when multiple software releases are developed. Users

want new capabilities in the latest releases, but still expect the older capabilities to remain in place. This is where regression testing plays a role. Test cases, test procedures, and other test-related items from previous releases should be available so that these tests can be run with the new versions of the software. Automated testing tools support testers with this very time-consuming task.

3.12 Alpha, beta and acceptance tests.

In the various testing activities that have been described so far, users have played a supporting role for the most part. They have been involved in requirements analysis and reviews, and have played a role in test planning. This is especially true for acceptance test planning if the software is being custom made for an organization. The clients along with test planners design the actual test cases that will be run during acceptance test. Users/clients may also have participated in prototype evaluation, usage profile development, and in the various stages of usability testing. After the software has passed all the system tests and defect repairs have been made, the users take a more active role in the testing process. Developers/testers must keep in mind that the software is being developed to satisfy the users requirements, and no matter how elegant its design it will not be accepted by the users unless it helps them to achieve their goals as specified in the requirements. Alpha, beta, and acceptance tests allow users to evaluate the software in terms of their expectations and goals.

When software is being developed for a specific client, acceptance tests are carried out after system testing. The acceptance tests must be planned carefully with input from the client/users. Acceptance test cases are based on requirements. The user manual is an additional source for test cases. System test cases may be reused. The software must run under real-world conditions on operational hardware and software. The software-under-test should be stressed. For continuous systems the software should be run at least through a 25-hour test cycle. Conditions should be typical for a working day. Typical inputs and illegal inputs should be used and all major functions should be exercised. If the entire suite of tests cannot be run for any reason, then the full set of tests needs to be rerun from the start.

Acceptance tests are a very important milestone for the developers. At this time the clients will determine if the software meets their requirements. Contractual obligations can be satisfied if the client is satisfied with the software. Development organizations will often receive their final payment when acceptance tests have been passed.

Acceptance tests must be rehearsed by the developers/testers. There should be no signs of unprofessional behavior or lack of preparation. Clients do not appreciate surprises. Clients should be received in the development organization as respected guests. They should be provided with documents and other material to help them participate in the acceptance testing process, and to evaluate the results. After acceptance testing the client will point out to the developers which requirement have/have not been satisfied. Some requirements may be deleted, modified, or added due to changing needs. If the client has been involved in prototype evaluations then the changes may be less extensive. If the client is satisfied that the software is usable and reliable, and they give their approval, then the next step is to install the system at the client’s site. If the

client’s site conditions are different from that of the developers, the developers must set up the system so that it can interface with client software and hardware. Retesting may have to be done to insure that the software works as required in the client’s environment. This is called installationtest.

If the software has been developed for the mass market (shrinkwrapped software), then testing it for individual clients/users is not practical or even possible in most cases. Very often this type of software undergoes two stages of acceptance test. The first is called alpha test. This test takes place at the developer’s site. A cross-section of potential users and members of the developer’s organization are invited to use the software. Developers observe the users and note problems. Beta test sends the software to a cross-section of users who install it and use it under realworld working conditions. The users send records of problems with the software to the development organization where the defects are repaired sometimes in time for the current release.

Unit III

Part-A Questions

1. List the types of testing and its need.
2. What are the goals of unit testing?
3. Define: Integration testing.
4. Define:test harness.
5. Define:System testing.List the types of System testing.
6. Give an note on:alpha, beta, acceptance testing

Part-B Questions

1. Explain elaborately about the various types of system testing.
2. Discuss the importance of following:

(i) Security Testing (ii) Alpha Testing

(iii) Beta Testing (iv) Acceptance testing

UNIT IV TEST MANAGEMENT

4.1 Introductory concepts

This chapter focuses on preparing the reader to address two fundamental maturity goals at level 2 of the TMM: (i) developing organizational goals/ policies relating to testing and debugging, and (ii) test planning. These maturity goals are managerial in nature. They are essential to support testing as a managed process. According to R. Thayer, a managed process is one that is planned, monitored, directed, staffed, and organized . At TMM level 2 the planning component of a managed process is instituted. At TMM levels 3 and 4 the remaining managerial components are integrated into the process. By instituting all of the managerial components described by Thayer in an incremental manner, an organization is able to establish the high-quality testing process described at higher levels of the TMM. The test specialist has a key role in developing and implementing these managerial components. In this chapter concepts and tools are introduced to build test management skills, thus supporting the reader in his/her development as a test specialist. The development, documentation, and institutionalization of goals and related policies is important to an organization. The goals/policies may be business-related, technical, or political in nature. They are the basis for decision making; therefore setting goals and policies requires the participation and support of upper management. Technical staff and other interested parties also participate in goal and policy development. Simple examples of the three types of goals mentioned are shown below.

1. *Business goal:* to increase market share 10% in the next 2 years in the area of financial software.
2. *Technical goal:* to reduce defects by 2% per year over the next 3 years.
3. *Business/technical goal:* to reduce hotline calls by 5% over the next 2 years.
4. *Political goal:* to increase the number of women and minorities in high management positions by 15% in the next 3 years.

Planning is guided by policy, supports goal achievement, and is a vital part of all engineering activities. In the software domain, plans to achieve goals associated with a specific project are usually developed by a project manager. In the testing domain, test plans support achieving testing goals for a project, and are either developed by the project manager as part of the overall project plan, or by a test or quality specialist in conjunction with the project planner. Test planning requires the planner to articulate the testing goals for a given project, to select tools and

techniques needed to achieve the goals, and to estimate time and resources needed for testing tasks so that testing is effective, on time, within budget, and consistent with project goals.

4.2 Testing and debugging goals and policies

A goal can be described as (i) a statement of intent, or (ii) a statement of a accomplishment that an individual or an organization wants to achieve.

A goal statement relates to an area where an individual, group, or organization wants to make improvements. Goals project future states of an organization, a group, or an individual. In an organization there is often a hierarchy of goals. At the top level are general organizational goals. There are intermediate-level goals that may be associated with a particular organizational functional unit. Individual projects have specific goals. These usually reflect organizational goals. There are personal-level goals as well. Each individual in an organization has a set of goals for self-improvement so that he or she can more effectively contribute to the project, functional unit, and organization as a whole.

Goal statements can express expectations in quantitative terms or be more general in nature. For the testing goals below, goals 1 and 2 express what is to be achieved in a more quantitative manner than goals 3 and 4.

1. One-hundred percent of testing activities are planned.
2. The degree of automation for regression testing is increased from 50% to 80% over the next 3 years.
3. Testing activities are performed by a dedicated testing group.
4. Testing group members have at least a bachelor-level degree and have taken a formal course in software testing.

In general, quantitative goals are more useful. These are measurable goals, and give an organization, group, or individual the means to evaluate progress toward achieving the goal. In the testing domain, goal statements should provide a high-level vision of what testing is to accomplish in the organization with respect to quality of process and product. In addition to general testing goal statements, lower-level goal statements should be developed for all levels of testing. Goals for the education and training of testing personnel should also be included with testing goal statements. Test plans should express testing goals for each project. These reflect overall organizational testing goals as well as specific goals for the project.

The TMM itself is built on a hierarchy of high-level testing maturity goals and subgoals which support the growth of an effective software testing process and promote high software quality. TheTMMcan be used by decision-makers in an organization to develop both long- and shortterm testing goals based on the TMM goal hierarchy.

A policy can be defined as a high-level statement of principle or course of action that is used to govern a set of activities in an organization.

Because a policy provides the vision and framework for decision making, it is important to have the policy formally adopted by the organization, documented, and available for all interested parties. An intraorganizational web site is suggested as a location for policy statements. This would allow for updates and visibility within the organization. A policy statement should be formulated by a team or task force consisting of upper management, executive personnel, and technical staff. In the case of testing, a testing policy statement is used to guide the course of testing activities and test process evolution. It should be agreed upon as workable by all concerned.

Testing policy statements reflect, integrate, and support achievement of testing goals. These goals in turn often target increasing software quality and improving customer satisfaction. Test policies also provide high-level guidance as to how testing is to be done in the organization, how its effectiveness will be evaluated, who will be responsible, and what choices of resources are possible. They should be explicit enough to guide decisions on all important testing issues, for example, how to test, what to test, and who will test. Policies are not written in stone, and as an organization grows in maturity its policies will change and mature. The task force should establish documented procedures for policy change. A brief outline of a sample testing policy statement appropriate for a TMM level 2 organization follows.

Testing Policy: Organization X

Our organization, the X Corporation, realizes that testing is an important component of the software development process and has a high impact on software quality and the degree of customer satisfaction. To ensure that our testing process is effective and that our software products meet the client's requirements we have developed and adopted the following testing policy statement.

1. Delivering software of the highest quality is our company goal. The presence of defects has a negative impact on software quality. Defects affect the correctness, reliability, and usability of a software product, thus rendering it unsatisfactory to the client. We define a testing activity as a set of tasks whose purpose is to reveal functional and quality- related defects in a software deliverable. Testing activities include traditional execution of the developing software, as well as reviews of the software deliverables produced at all stages of the life cycle. The aggregation of all testing activities performed in a systematic manner supported by organizational policies, procedures, and standards constitutes the testing process.

2. A set of testing standards must be available to all interested parties on an intraorganizational web site. The standards contain descriptions of all test-related documents, prescribed templates, and the methods, tools, and procedures to be used for testing. The standards must specify the types of projects that each of these items is to be associated with.

3. In our organization the following apply to all software development/ maintenance projects:

• Execution-based tests must be performed at several levels such as unit , integration, system, and acceptance tests as appropriate for each software product.

• Systematic approaches to test design must be employed that include application of both white and black box testing methods.
• Reviews of all major product deliverables such as requirements and design documents, code, and test plans are required.
• Testing must be planned for all projects. Plans must be developed for all levels of executionbased testing as well as for reviews of deliverables.

Test plan templates must be included in organizational standards documents and implemented online. A test plan for a project must be compatible with the project plan for that project. Test plans must be approved by the project manager and technical staff. Acceptance test plans must also be approved by the client.

• Testing activities must be monitored using measurements and milestones to ensure that they are proceeding according to plan.
• Testing activities must be integrated into the software life cycle and carried out in parallel with other development activities. The extended modified V-model as shown in the testing standards document has been adopted to support this goal.
• Defects uncovered during each test must be classified and recorded.
• There must be a training program to ensure that the best testing practices are employed by the testing staff.

4. Because testing is an activity that requires special training and an impartial view of the software, it must be carried out by an independent testing group. Communication lines must be established to support cooperation between testers and developers to ensure that the software is reliable, safe, and meets client requirements.

5. Testing must be supported by tools, and, test-related measurements must be collected and used to evaluate and improve the testing process and the software product.

6. Resources must be provided for continuos test process improvement.

7. Clients/developer/tester communication is important, and clients must be involved in acceptance test planning, operational profile development, and usage testing when applicable to the project. Clients must sign off on the acceptance test plan and give approval for all changes in the acceptance test plan.

8. A permanent committee consisting of managerial and technical staff must be appointed to be responsible for distribution and maintenance of organizational test policy statements. Whatever the nature of the test policy statement, it should have strong support and continual commitment from management. After the policy statement has been developed, approved, and distributed, a subset of the task force should be appointed to permanently oversee policy implementation and change.

DebuggingPolicy:OrganizationX

Our organization, the X Corporation, is committed to delivering highquality software to our customers. Effective testing and debugging processes are essential to support this goal. It is our policy to separate testing and debugging, and we consider them as two separate processes. Each has different psychologies, goals, and requirements. The resources, training, and tools needed are different for both. To support the separation of these two processes we have developed individual testing and debugging policies. Our debugging policy is founded on our quality goal to remove all defects from our software that impact on our customers' ability to use our software effectively, safely, and economically. To achieve this goal we have developed the following debugging policy statement.

1. Testing and debugging are two separate processes. Testing is the process used to detect (reveal) defects. Debugging is the process dedicated to locating the defects, repairing the code, and retesting the software. Defects are anomalies that impact on software functionality as well as on quality attributes such as performance, security, ease of use, correctness, and reliability.

2. Since debugging is a timely activity, all project schedules must allow for adequate time to make repairs, and retest the repaired software.

3. Debugging tools, and the training necessary to use the tools, must be available to developers to support debugging activities and tasks.

4. Developers/testers and SQA staff must define and document a set of defect classes and defect severity levels. These must be must be available to all interested parties on an intraorganizational web site, and applied to all projects.

5. When failures are observed during testing or in operational software they are documented. A problem, or test incident, report is completed by the developer/tester at testing time and by the users when a failure/ problem is observed in operational software. The problem report is forwarded to the development group. Both testers/developers and SQA staff must communicate and work with users to gain an understanding of the problem. A fix report must be completed by the developer when the defect is repaired and code retested. Standard problem and fix report forms must be available to all interested parties on an intraorganizational web site, and applied to all projects.

7. All defects identified for each project must be cataloged according to class and severity level and stored as a part of the project history.

8. Measurement such as total number of defects, total number of defects/ KLOC, and time to repair a defect are saved for each project.

9. A permanent committee consisting of managerial and technical staff must be appointed to be responsible for distribution and maintenance of organizational debugging policy statements.

4.3 Test planning

A plan can be defined in the following way.

A plan is a document that provides a framework or approach for achieving a set of goals.

In the software domain, plans can be strictly business oriented, for example, long-term plans to support the economic growth of an organization, or they can be more technical in nature, for example, a plan to develop a specific software product. Test plans tend to be more technically oriented. However, a software project plan that may contain a test plan as well will often refer to business goals. In this chapter we focus on planning for execution-based software testing (validation testing).
Test planning is an essential practice for any organization that wishes to develop a test process that is repeatable and manageable. Pursuing the maturity goals embedded in the TMM structure is not a necessary precondition for initiating a test-planning process. However, a test process improvement effort does provide a good framework for adopting this essential practice. Test planning should begin early in the software life cycle, although for many organizations whose test processes are immature this practice is not yet in place. Models such as the V-model, or the Extended/ Modified V-model (Figure 1.5), help to support test planning activities that begin in the requirements phase, and continue on into successive software development phases [2,3].

In order to meet a set of goals, a plan describes what specific tasks must be accomplished, who is responsible for each task, what tools, procedures, and techniques must be used, how much time and effort is needed, and what resources are essential. A plan also contains milestones.

Milestones are tangible events that are expected to occur at a certain time in the project's lifetime. Managers use them to determine project status.

Tracking the actual occurrence of the milestone events allows a manager to determine if the project is progressing as planned. Finally, a plan should assess the risks involved in carrying out the project. Test plans for software projects are very complex and detailed documents. The planner usually includes the following essential high-level items.

1. *Overall test objectives.* As testers, why are we testing, what is to be achieved by the tests, and what are the risks associated with testing this product?

2. *What to test (scope of the tests).* What items, features, procedures, functions, objects, clusters, and subsystems will be tested?

3. *Who will test.* Who are the personnel responsible for the tests?

4. *How to test.* What strategies, methods, hardware, software tools, and techniques are going to be applied? What test documents and deliverable should be produced?

5. *When to test.* What are the schedules for tests? What items need to be available?

6. *When to stop testing.* It is not economically feasible or practical to plan to test until all defects have been revealed. This is a goal that testers can never be sure they have reached. Because of budgets, scheduling, and customer deadlines, specific conditions must be outlined in the test plan that allow testers/managers to decide when testing is considered to be complete.

Test plans can be organized in several ways depending on organizational policy. There is often a hierarchy of plans that includes several levels of quality assurance and test plans. The complexity of the hierarchy depends on the type, size, risk-proneness, and the mission/safety criticality of software system being developed. All of the quality and testing plans should also be coordinated with the overall software project plan. A sample plan hierarchy is shown in Figure 7.1. At the top of the plan hierarchy there may be a software quality assurance plan. This plan gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, configuration control, and supplier control.

FIG. 7.1
A hierarchy of test plans.

Below that in the plan hierarchy there may be a master test plan that includes an overall description of all execution-based testing for the software system. A master verification plan for reviews inspections/ walkthroughs would also fit in at this level. The master test plan itself may be a component of the overall project plan or exist as a separate document. Depending on organizational policy, another level of the hierarchy could contain a separate test plan for unit, integration, system, and acceptance tests. In some organizations these are part of the master test plan. The level-based plans give a more detailed view of testing appropriate to that level. The *IEEE Software Engineering Standards Collection* has useful descriptions for many of these plans and other test and quality-related documents such as verification and validation plans.

The persons responsible for developing test plans depend on the type of plan under development. Usually staff from one or more groups cooperates in test plan development. For example, the master test plan for execution-based testing may be developed by the project manager, especially if there is no separate testing group. It can also be developed by a tester or software quality assurance manager, but always requires cooperation and input from the project manager. It is essential that development and testing activities be coordinated to allow the project to progress smoothly. The type and organization of the test plan, the test plan hierarchy, and who

is responsible for development should be specified in organizational standards or software quality assurance documents.

The remainder of this chapter focuses on the development of a general- purpose execution-based test plan that will be referred to as a "test plan." The description of the test plan contents is based on a discussion of recommended test plan components appearing in the ***IEEE Standard for Software Test Documentation: IEEE/ANSI Std 829-1983*** . This standard also contains examples of other test-related documents described in this chapter. The reader should note that the IEEE test plan description serves as a guideline to test planners. The actual templates and documents developed by test planners should be tailored to meet organizational needs and conform to organizational goals and policies.

4.4 Test plan components

This section of the text will discuss the basic test plan components as described in IEEE Std 829-1983 [5]. They are shown in Figure 7.2. These components should appear in the master test plan and in each of the levelbased test plans (unit, integration, etc.) with the appropriate amount of detail. The reader should note that some items in a test plan may appear in other related documents, for example, the project plan. References to such documents should be included in the test plan, or a copy of the appropriate section of the document should be attached to the test plan.

Test Plan Components
1. Test plan identifier
2. Introduction
3. Items to be tested
4. Features to be tested
5. Approach
6. Pass/fail criteria
7. Suspension and resumption criteria
8. Test deliverables
9. Testing Tasks
10. Test environment
11. Responsibilities
12. Staffing and training needs
13. Scheduling
14. Risks and contingencies
15. Testing costs
16. Approvals

FIG. 7.2
Components of a test plan.

1 . Test Plan I d e n t i f i e r

Each test plan should have a unique identifier so that it can be associated with a specific project and become a part of the project history. The project history and all project-related items should be stored in a project database or come under the control of a configuration management system. Organizational standards should describe the format for the test plan identifier and how to specify versions, since the test plan, like all other software items, is not written in stone and is

subject to change. A mention was made of a configuration management system. This is a tool that supports change management. It is essential for any software project and allows for orderly change control. If a configuration management system is used, the test plan identifier can serve to identify it as a configuration item .

2 . Introduction

In this section the test planner gives an overall description of the project, the software system being developed or maintained, and the soft ware items and/or features to be tested. It is useful to include a high-level description of testing goals and the testing approaches to be used. References to related or supporting documents should also be included in this section, for example, organizational policies and standards documents, the project plan, quality assurance plan, and software configuration plan. If test plans are developed as multilevel documents, that is, separate documents for unit, integration, system, and acceptance test, then each plan must reference the next higher level plan for consistency and compatibility reasons.

3 . Items to Be Tested

This is a listing of the entities to be tested and should include names, identifiers, and version/revision numbers for each entity. The items listed could include procedures, classes, modules, libraries, subsystems, and systems. References to the appropriate documents where these items and their behaviors are described such as requirements and design documents, and the user manual should be included in this component of the test plan. These references support the tester with traceability tasks. The focus of traceability tasks is to ensure that each requirement has been covered with an appropriate number of test cases. In this test plan component also refer to the transmittal media where the items are stored if appropriate; for example, on disk, CD, tape. The test planner should also include items that will *not* be included in the test effort.

4 . Features to Be Tested

In this component of the test plan the tester gives another view of the entities to be tested by describing them in terms of the features they encompass. Chapter 3 has this definition for a feature.

Features may be described as distinguishing characteristics of a software component or system.

They are closely related to the way we describe software in terms of its functional and quality requirements . Example features relate to performance,reliability, portability, and functionality requirements for thesoftware being tested. Features that will *not* be tested should be identified and reasons for their exclusion from test should be included.

5 . Approach

This section of the test plan provides broad coverage of the issues to be addressed when testing the target software. Testing activities are described. The level of descriptive detail should be sufficient so that the major testing tasks and task durations can be identified. More details will

appear in the accompanying test design specifications. The planner should also include for each feature or combination of features, the approach that will be taken to ensure that each is adequately tested. Tools and techniques necessary for the tests should be included.

6 . Item Pass/Fail Criteria

Given a test item and a test case, the tester must have a set of criteria to decide on whether the test has been passed or failed upon execution. The master test plan should provide a general description of these criteria. In the test design specification section more specific details are given for each item or group of items under test with that specification. A definition for the term "failure" was given in Chapter 2. Another way of describing the term is to state that a failure occurs when the actual output produced by the software does not agree with what was expected, under the conditions specified by the test. The differences in output behavior (the failure) are caused by one or more defects. The impact of the defect can be expressed using an approach based on establishing severity levels. Using this approach, scales are used to rate failures/defects with respect to their impact on the customer/user (note their previous use for stop-test decision making in the preceding section). For example, on a scale with values from 1 to 4, a level 4 defect/failure may have a minimal impact on the customer/user, but one at level 1 will make the system unusable.

7 . Suspension and Resumption Criteria

In this section of the test plan, criteria to suspend and resume testing are described. In the simplest of cases testing is suspended at the end of a working day and resumed the following morning. For some test items this condition may not apply and additional details need to be provided by the test planner. The test plan should also specify conditions to suspend testing based on the effects or criticality level of the failures/defects observed. Conditions for resuming the test after there has been a suspension should also be specified. For some test items resumption may require certain tests to be repeated.

8 . Test Deliverables

Execution-based testing has a set of deliverables that includes the test plan along with its associated test design specifications, test procedures, and test cases. The latter describe the actual test inputs and expected outputs. Deliverables may also include other documents that result from testing such as test logs, test transmittal reports, test incident reports, and a test summary report. These documents are described in subsequent sections of this chapter. Preparing and storing these documents requires considerable resources. Each organization should decide which of these documents is required for a given project.

Another test deliverable is the test harness. This is supplementary code that is written specifically to support the test efforts, for example, module drivers and stubs. Drivers and stubs are necessary for unit and integration test. Very often these amount to a substantial amount of code. They should be well designed and stored for reuse in testing subsequent releases of the software. Other support code, for example, testing tools that will be developed especially for this project, should also be described in this section of the test plan.

9 . Testing Tasks

In this section the test planner should identify all testing-related tasks and their dependencies. Using a Work Breakdown Structure (WBS) is useful here.

A Work Breakdown Structure is a hierarchical or treelike representation of all the tasks that are required to complete a project.

High-level tasks sit at the top of the hierarchical task tree. Leaves are detailed tasks sometimes called work packages that can be done by 1-2 people in a short time period, typically 3-5 days. The WBS is used by project managers for defining the tasks and work packages needed for project planning. The test planner can use the same hierarchical task model but focus only on defining testing tasks. Rakos gives a good description of the WBS and other models and tools useful for both project and test management .

10. The Testing Environment

Here the test planner describes the software and hardware needs for the testing effort. For example, any special equipment or hardware needed such as emulators, telecommunication equipment, or other devices should be noted. The planner must also indicate any laboratory space containing the equipment that needs to be reserved. The planner also needs to specify any special software needs such as coverage tools, databases, and test data generators. Security requirements for the testing environment may also need to be described.

11. Responsibilities

The staff who will be responsible for test-related tasks should be identified. This includes personnel who will be:

- transmitting the software-under-test;
- developing test design specifications, and test cases;
- executing the tests and recording results;
- tracking and monitoring the test efforts;
- checking results;
- interacting with developers;
- managing and providing equipment;
- developing the test harnesses;
- interacting with the users/customers.

This group may include developers, testers, software quality assurance staff, systems analysts, and customers/users.

12. Staffing and Training Needs

The test planner should describe the staff and the skill levels needed to carry out test-related responsibilities such as those listed in the section above. Any special training required to perform a task should be noted.

13. Scheduling

Task durations should be established and recorded with the aid of a task networking tool. Test milestones should be established, recorded, and scheduled. These milestones usually appear in the project plan as well as the test plan. They are necessary for tracking testing efforts to ensure that actual testing is proceeding as planned. Schedules for use of staff, tools, equipment, and laboratory space should also be specified. A tester will find that PERT and Gantt charts are very useful tools for these assignments.

14. Risks and Contingencies

Every testing effort has risks associated with it. Testing software with a high degree of criticality, complexity, or a tight delivery deadline all impose risks that may have negative impacts on project goals. These risks should be: (i) identified, (ii) evaluated in terms of their probability of occurrence, (iii) prioritized, and (iv) contingency plans should be developed that can be activated if the risk occurs.

An example of a risk-related test scenario is as follows. A test planner, lets say Mary Jones, has made assumptions about the availability of the software under test. A particular date was selected to transmit the test item to the testers based on completion date information for that item in the project plan. Ms. Jones has identified a risk: she realizes that the item may not be delivered on time to the testers. This delay may occur for several reasons. For example, the item is complex and/or the developers are inexperienced and/or the item implements a new algorithm and/or it needs redesign. Due to these conditions there is a high probability that this risk could occur. A contingency plan should be in place if this risk occurs. For example, Ms. Jones could build some flexibility in resource allocations into the test plan so that testers and equipment can operate beyond normal working hours. Or an additional group of testers could be made available to work with the original group when the software is ready to test. In this way the schedule for testing can continue as planned, and deadlines can be met.

It is important for the test planner to identify test-related risks, analyze them in terms of their probability of occurrence, and be ready with a contingency plan when any high-priority riskrelated event occurs. Experienced planners realize the importance of risk management.

15. Testing Costs

The IEEE standard for test plan documentation does not include a separate cost component in its specification of a test plan. This is the usual case for many test plans since very often test costs are allocated in the overall project management plan. The project manager in consultation with developers and testers estimates the costs of testing. If the test plan is an independent document prepared by the testing group and has a cost component, the test planner will need tools and techniques to help estimate test costs. Test costs that should included in the plan are:

• costs of planning and designing the tests;
• costs of acquiring the hardware and software necessary for the tests (includes development of the test harnesses);
• costs to support the test environment;
• costs of executing the tests;
• costs of recording and analyzing test results;
• tear-down costs to restore the environment.

Other costs related to testing that may be spread among several projects are the costs of training the testers and the costs of maintaining the test database. Costs for reviews should appear in a separate review plan.

When estimating testing costs, the test planner should consider organizational, project, and staff characteristics that impact on the cost of testing. Several key characteristics that we will call "test cost impact items" are briefly described below.

The nature of the organization; its testing maturity level, and general maturity. This will determine the degree of test planning, the types of testing methods applied, the types of tests that are designed and implemented, the quality of the staff, the nature of the testing tasks, the availability of testing tools, and the ability to manage the testing effort. It will also determine the degree of support given to the testers by the project manager and upper management.

The nature of the software product being developed. The tester must understand the nature of the system to be tested. For example, is it a real time, embedded, mission-critical system, or a business application? In general, the testing scope for a business application will be smaller than one for a mission or safely critical system, since in case of the latter there is a strong possibility that software defects and/or poor software quality could result in loss of life or property. Mission- and safety-critical software systems usually require extensive unit and integration tests as well as many types of system tests (refer to Chapter 6). The level of reliability required for these systems is usually much higher than for ordinary applications.

For these reasons, the number of test cases, test procedures, and test scripts will most likely be higher for this type of software as compared to an average application. Tool and resource needs will be greater as well.

The scope of the test requirements. This includes the types of tests required, integration, performance, reliability, usability, etc. This characteristic directly relates to the nature of the software product. As described above, mission/safety-critical systems, and real-time embedded systems usually require more extensive system tests for functionality, reliability, performance, configuration, and stress than a simple application. These test requirements will impact on the number of tests and test procedures required, the quantity and complexity of the testing tasks, and the hardware and software needs for testing.

The level of tester ability. The education, training, and experience levels of the testers will impact on their ability to design, develop, execute, and analyze test results in a timely and effective manner. It will also impact of the types of testing tasks they are able to carry out.

Knowledge of the project problem domain. It is not always possible for testers to have detailed knowledge of the problem domain of the software they are testing. If the level of knowledge is poor, outside experts or consultants may need to be hired to assist with the testing efforts, thus impacting on costs.

The level of tool support. Testing tools can assist with designing, and executing tests, as well as collecting and analyzing test data. Automated support for these tasks could have a positive impact on the productivity of the testers; thus it has the potential to reduce test costs. Tools and hardware environments are necessary to drive certain types of system tests, and if the product requires these types of tests, the cost should be folded in.

Training requirements. State-of-the-art tools and techniques do help improve tester productivity but often training is required for testers so that they have the capability to use these tools and techniques properly and effectively. Depending on the organization, these training efforts may be included in the costs of testing. These costs, as well as tool costs, could be spread over several projects.

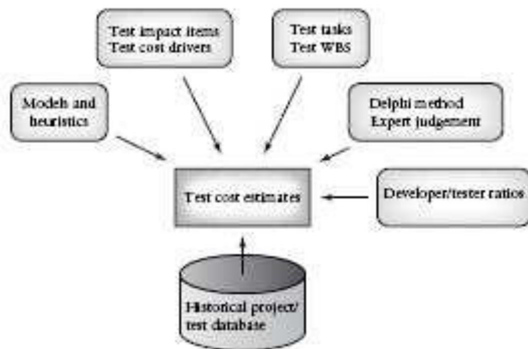


FIG. 7.3
Some approaches to test cost estimation.

Project planners have cost estimation models, for example, the COCOMO model, which they use to estimate overall project costs. At this time models of this type have not been designed specifically for test cost estimation.

4.5 Test plan attachments

The previous components of the test plan were principally managerial in nature: tasks, schedules, risks, and so on. A general discussion of technical issues such as test designs and test cases for the items under test appears in Section 5 of the test plan, ''Approach.'' The reader may be puzzled as to where in the test plan are the details needed for organizing and executing the tests.

For example, what are the required inputs, outputs, and procedural steps for each test; where will the tests be stored for each item or feature; will it be tested using a black box, white box, or functional approach? The following components of the test plan contain this detailed information. These documents are generally attached to the test plan.

Requirement identifier	Requirement description	Priority (scale 1–10)	Review status	Test ID
SR-25-13.5	Displays opening screens	8	Yes	TC-25-2 TC-25-5
SR-25-52.2	Checks the validity of user password	9	Yes	TC-25-18 TC-25-23

TABLE 7.3
Example of entries in a requirements traceability matrix.

T e s t D e s i g n S p e c i f i c a t i o n s

The IEEE standard for software test documentation describes a test design specification as a test deliverable that specifies the requirements of the test approach . It is used to identity the features covered by this design and associated tests for the features. The test design specification also has links to the associated test cases and test procedures needed to test the features, and also describes in detail pass/fail criteria for the features. The test design specification helps to organize the tests and provides the connection to the actual test inputs/outputs and test steps.

To develop test design specifications many documents such as the requirements, design documents, and user manual are useful. For requirements-based test, developing a requirements traceability matrix is valuable. This helps to insure all requirements are covered by tests, and connects the requirements to the tests. Examples of entries in such a matrix are shown in Table 7.3. Tools called requirements tracers can help to automate traceability tasks . These will be described in Chapter 14. A test design specification should have the following components according to the IEEE standard . They are listed in the order in which the IEEE recommends they appear in the document. The test planner should be sure to list any related documents that may also contain some of this material.

Test Design Specification Identifier
Give each test design specification a unique identifier and a reference to its associated test plan.

Features to Be Tested
Test items, features, and combination of features covered by this test design specification are listed. References to the items in the requirements and/or design document should be included.

Approach Refinements
In the test plan a general description of the approach to be used to test each item was described. In this document the necessary details are added. For example, the specific test techniques to be

used to generate test cases are described, and the rational is given for the choices. The test planner also describes how test results will be analyzed. For example, will an automated comparator be used to compare actual and expected results? The relationships among the associated test cases are discussed. This includes any shared constraints and procedural requirements.

Test Case Identification
Each test design specification is associated with a set of test cases and a set of set procedures. The test cases contain input/output information, and the test procedures contain the steps necessary to execute the tests. A test case may be associated with more than one test design specification.

Pass/Fail Criteria
In this section the specific criteria to be used for determining whether the item has passed/failed a test is given.

T e s t C a s e S p e c i f i c a t i o n s

This series of documents attached to the test plan defines the test cases required to execute the test items named in the associated test design specification. There are several components in this document. IEEE standards require the components to appear in the order shown here, and references should be provided if some of the contents of the test case specification appear in other documents .

Much attention should be placed on developing a quality set of test case specifications. Strategies and techniques, as described in Chapters 4 and 5 of this text, should be applied to accomplish this task. Each test case must be specified correctly so that time is not wasted in analyzing the results of an erroneous test. In addition, the development of test software and test documentation represent a considerable investment of resources for an organization. They should be considered organizational assets and stored in a test repository. Ideally, the test-related deliverables may be recovered from the test repository and reused by different groups for testing and regression testing in subsequent releases of a particular product or for related products. Careful design and referencing to the appropriate test design specification is important to support testing in the current project and for reuse in future projects.

Test Case Specification Identifier
Each test case specification should be assigned a unique identifier.

Test Items
This component names the test items and features to be tested by this test case specification. References to related documents that describe the items and features, and how they are used should be listed: for example the requirements, and design documents, the user manual.

Input Specifications
This component of the test design specification contains the actual inputs needed to execute the test. Inputs may be described as specific values, or as file names, tables, databases, parameters passed by the operating system,and so on. Any special relationships between the inputs should be identified.

Output Specifications
All outputs expected from the test should be identified. If an output is to be a specific value it should be stated. If the output is a specific feature such as a level of performance it also should be stated. The output specifications are necessary to determine whether the item has passed/failed the test.

Special Environmental Needs
Any specific hardware and specific hardware configurations needed to execute this test case should be identified. Special software required to execute the test such as compilers, simulators, and test coverage tools should be described, as well as needed laboratory space and equipment.

Special Procedural Requirements
Describe any special conditions or constraints that apply to the test procedures associated with this test.

Intercase Dependencies
In this section the test planner should describe any relationships between this test case and others, and the nature of the relationship. The test case identifiers of all related tests should be given.

T e s t P r o c e d u r e S p e c i f i c a t i o n s

A procedure in general is a sequence of steps required to carry out a specific task.

In this attachment to the test plan the planner specifies the steps required to execute a set of test cases. Another way of describing the test procedure specification is that it specifies the steps necessary to analyze a software item in order to evaluate a set of features. The test procedure specification has several subcomponents that the IEEE recommends being included in the order shown below. As noted previously, reference to documents where parts of these components are described must be provided.

Test Procedure Specification Identifier
Each test procedure specification should be assigned a unique identifier.

Purpose
Describe the purpose of this test procedure and reference any test cases it executes.

Specific Requirements
List any special requirements for this procedure, like software, hardware, and special training.

Procedure Steps
Here the actual steps of the procedure are described. Include methods, documents for recording (logging) results, and recording incidents. These will have associations with the test logs and test incident reports that result from a test run. A test incident report is only required when an unexpected output is observed. Steps include [5]:

(i) setup: to prepare for execution of the procedure;
(ii) start: to begin execution of the procedure;
(iii) proceed: to continue the execution of the procedure;
(iv) measure: to describe how test measurements related to outputs will be made;
(v) shut down: to describe actions needed to suspend the test when unexpected events occur;
(vi) restart: to describe restart points and actions needed to restart the procedure from these points;
(vii) stop: to describe actions needed to bring the procedure to an orderly halt;
(viii) wrap up: to describe actions necessary to restore the environment;
(ix) contingencies: plans for handling anomalous events if they occur during execution of this procedure.

4.6 Locating test items

Suppose a tester is ready to run tests on an item on the date described in the test plan. She needs to be able to locate the item and have knowledge of its current status. This is the function of the Test Item Transmittal Report. This document is not a component of the test plan, but is necessary to locate and track the items that are submitted for test. Each Test Item Transmittal Report has a unique identifier. It should contain the following information for each item that is tracked.

(i) version/revision number of the item;
(ii) location of the item;
(iii) persons responsible for the item (e.g., the developer);
(iv) references to item documentation and the test plan it is related to;
(v) status of the item;
(vi) approvals—space for signatures of staff who approve the transmittal.

4.7 Reporting test results

The test plan and its attachments are test-related documents that are prepared *prior* to test execution. There are additional documents related to testing that are prepared during and after execution of the tests. The *IEEE Standard for Software Test Documentation* describes the following documents .

Test Log

The test log should be prepared by the person executing the tests. It is a diary of the events that take place during the test. It supports the concept of a test as a repeatable experiment [14]. In the experimental world of engineers and scientists detailed logs are kept when carrying out experimental work. Software engineers and testing specialists must follow this example to allow others to duplicate their work.

The test log is invaluable for use in defect repair. It gives the developer a snapshot of the events associated with a failure. The test log, in combination with the test incident report which should be generated in case of anomalous behavior, gives valuable clues to the developer whose task it is to locate the source of the problem. The combination of documents helps to prevent incorrect decisions based on incomplete or erroneous test results that often lead to repeated, but ineffective, test-patch-test cycles.

Retest that follows defect repair is also supported by the test log. In addition, the test log is valuable for (i) regression testing that takes place in the development of future releases of a software product, and (ii) circumstances where code from a reuse library is to be reused. In all these cases it is important that the exact conditions of a test run are clearly documented so that it can be repeated with accuracy.

Test Log Identifier
Each test log should have a unique identifier.

Description
In the description section the tester should identify the items being tested, their version/revision number, and their associated Test Item/Transmittal Report. The environment in which the test is conducted should be described including hardware and operating system details.

Activity and Event Entries
The tester should provide dates and names of test log authors for each event and activity. This section should also contain:

1. *Execution description:* Provide a test procedure identifier and also the names and functions of personnel involved in the test.
2. *Procedure results:* For each execution, record the results and the location of the output. Also report pass/fail status.
3. *Environmental information:* Provide any environmental conditions specific to this test.
4. *Anomalous events:* Any events occurring before/after an unexpected event should be recorded. If a tester is unable to start or compete a test procedure, details relating to these happenings should be recorded (e.g., a power failure or operating system crash).
5. *Incident report identifiers:* Record the identifiers of incident reports generated while the test is being executed.

Test Incident Report

The tester should record in a test incident report (sometimes called a problem report) any event that occurs during the execution of the tests that is unexpected, unexplainable, and that requires a follow-up investigation. The *IEEE Standard for Software Test Documentation* recommends the following sections in the report:

1. *Test Incident Report identifier:* to uniquely identify this report.

2. *Summary:* to identify the test items involved, the test procedures, test cases, and test log associated with this report.

3. *Incident description:* this should describe time and date, testers, observers, environment, inputs, expected outputs, actual outputs, anomalies, procedure step, environment, and attempts to repeat. Any other information useful for the developers who will repair the code should be included.

4. *Impact:* what impact will this incident have on the testing effort, the test plans, the test procedures, and the test cases? A severity rating should be inserted here.

Test Summary Report

This report is prepared when testing is complete. It is a summary of the results of the testing efforts. It also becomes a part of the project's historical database and provides a basis for lessons learned as applied to future projects. When a project postmortem is conducted, the Test Summary Report can help managers, testers, developers, and SQA staff to evaluate the effectiveness of the testing efforts. The IEEE test documentation standard describes the following sections for the Test Summary Report :

1. *Test Summary Report identifier:* to uniquely identify this report.

2. *Variances:* these are descriptions of any variances of the test items from their original design. Deviations and reasons for the deviation from the test plan, test procedures, and test designs are discussed.

3. *Comprehensiveness assessment:* the document author discusses the comprehensiveness of the test effort as compared to test objectives and test completeness criteria as described in the test plan. Any features or combination of features that were not as fully tested as was planned should be identified.

4. *Summary of results:* the document author summarizes the testing results. All resolved incidents and their solutions should be described. Unresolved incidents should be recorded.

5. *Evaluation:* in this section the author evaluates each test item based on test results. Did it pass/fail the tests? If it failed, what was the level of severity of the failure?

6. *Summary of activities:* all testing activities and events are summarized.

Resource consumption, actual task durations, and hardware and software tool usage should be recorded.

7. *Approvals:* the names of all persons who are needed to approve this document are listed with space for signatures and dates.

Figure 7.4 shows the relationships between all the test-related documents

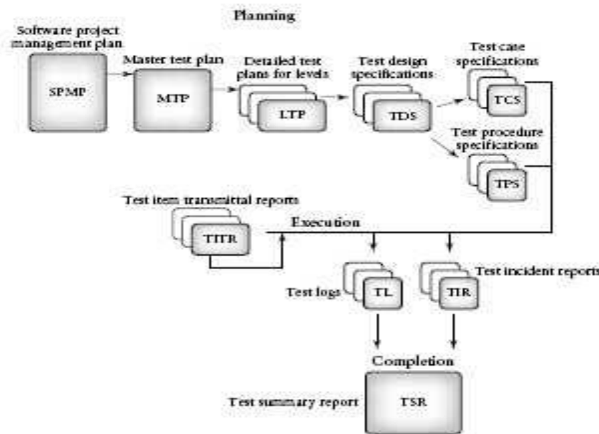


FIG. 7.4
Test-related documents as recommended by IEEE [5].

4.8 The role of three groups in test planning and policy development

Recall that in theTMMframework three groups were identified as critical players in the testing process. They all work together toward the evolution of a quality testing process. These groups were managers, developers/ testers, and users/clients. In TMM terminology they are called the three critical views (CV). Each group views the testing process from a different perspective that is related to their particular goals, needs, and requirements. The manager's view involves commitment and support for those activities and tasks related to improving testing process quality. The developer/tester's view encompasses the technical activities and tasks that when applied, constitute best testing practices. The user/client view is defined as a cooperating or supporting view. The developers/testers work with client/user groups on quality-related activities and tasks that concern user-oriented needs. The focus is on soliciting client/user support, consensus, and participation in activities such as requirements analysis, usability testing, and acceptance test planning.

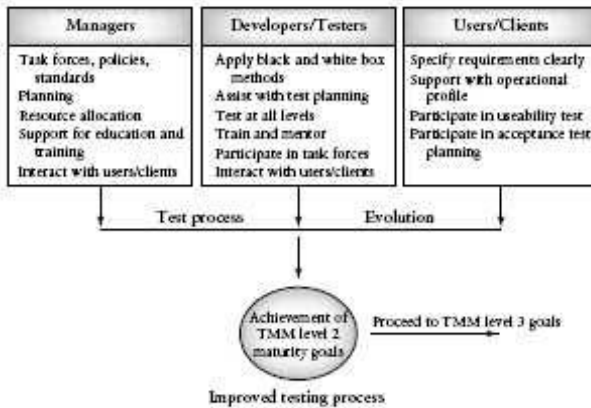


FIG. 7.5
Reaching TMM level 2: summary of critical group roles.

*Developer*s have an important role in the development of testing goals and policies. (Recall that at TMM level 2 there is no requirement for a dedicated testing group.) They serve as members of the goal/policy development teams. As representatives of the technical staff they must ensure that the policies reflect best testing practices, are implementable, receive management support, and support among technical personnel. The activities, tasks, and responsibilities for the developers/testers include:

• Working with management to develop testing and debugging policies and goals.
• Participating in the teams that oversee policy compliance and change management.
• Familiarizing themselves with the approved set of testing/debugging goals and policies, keeping up-to-date with revisions, and making suggestions for changes when appropriate.
• When developing test plans, setting testing goals for each project at each level of test that reflect organizational testing goals and policies.
• Carrying out testing activities that are in compliance with organizational policies.

Users and clients play an indirect role in the formation of an organization's testing goals and polices since these goals and policies reflect the organizations efforts to ensure customer/client/user satisfaction. Feedback from these groups and from the marketplace in general has an influence on the nature of organizational testing goals and policies. Successful organizations are sensitive to customer/client/user needs. Their policies reflect their desire to insure that their software products meet the customer's requirements. This allows them to maintain, and eventually increase, their market share of business.

Upper management supports this goal by:
• Establishing an organizationwide test planning committee with funding.
Ensuring that the testing policy statement and quality standards support test planning with commitment of resources, tools, templates, and training.
• Ensuring that the testing policy statement contains a formal mechanism for user input to the test planning process, especially for acceptance and usability testing.
• Ensuring that all projects are in compliance with the test planning policy.

• Ensuring that all developers/testers complete all the necessary posttest documents such as test logs and test incident reports.

Project managers support the test planning maturity goal by preparing the test plans for each

4.9 Process and the engineering disciplines

What we are now witnessing is the evolution of software development from a craft to an engineering discipline. Computer science students are now being introduced to the fundamentals of software engineering. As the field matures, they will be able to obtain a degree and be certified in the area of software engineering As members of this emerging profession we must realize that one of our major focuses as engineers is on designing, implementing, managing, and improving the processes related to software development. Testing is such a process. If you are a member of a TMM level 1 organization, there is a great opportunity for you become involved in process issues. You can serve as the change agent, using your education in the area of testing to form a process group or to join an existing one. You can initiate the implementation of a defined testing process by working with management and users/clients toward achievement of the technical and managerial-oriented maturity goals at TMM level 2. Minimally you can set an example on a personal level by planning your own testing activities. If the project manager receives effective personal test plans from each developer or test specialist, then the quality of the overall test plan will be improved. You can also encourage management in your organization to develop testing goals and policies, you can participate in the committees involved, and you can help to develop test planning standards that can be applied organizationwide. Finally, you can become proficient in, and consistently apply, black and white box testing techniques, and promote testing at the unit, integration, and system levels. You need to demonstrate the positive impact of these practices on software quality, encourage their adaptation in the organization, and mentor your colleagues, helping them to appreciate, master, and apply these practices.

4.10 Introducing the test specialist

By supporting a test group an organization acquires *leadership* in areas that relate to testing and quality issues. For example, there will be staff with the necessary skills and motivation to be responsible for:

- maintenance and application of test policies;
- development and application of test-related standards;
- participating in requirements, design, and code reviews;
- test planning;
- test design;
- test execution;
- test measurement;
- test monitoring (tasks, schedules, and costs);
- defect tracking, and maintaining the defect repository;
- acquisition of test tools and equipment;
- identifying and applying new testing techniques, tools, and methodologies;

• mentoring and training of new test personnel;
• test reporting.

The staff members of such a group are called test specialists or test engineers.

4.11 Skills needed by a test specialist

Given the nature of technical and managerial responsibilities assigned to the tester that are listed in Section 8.0, many managerial and personal skills are necessary for success in the area of work. On the *personal* and *managerial* level a test specialist must have:

• organizational, and planning skills;
• the ability to keep track of, and pay attention to, details;
• the determination to discover and solve problems;
• the ability to work with others and be able to resolve conflicts;
• the ability to mentor and train others;
• the ability to work with users and clients;
• strong written and oral communication skills;
• the ability to work in a variety of environments;
• the ability to think creatively

The first three skills are necessary because testing is detail and problem oriented. In addition, testing involves policymaking, a knowledge of different types of application areas, planning, and the ability to organize and monitor information, tasks, and people. Testing also requires inter actions with many other engineering professionals such as project managers, developers, analysts, process personal, and software quality assurance staff. Test professionals often interact with clients to prepare certain types of tests, for example acceptance tests. Testers also have to prepare test-related documents and make presentations. Training and mentoring of new hires to the testing group is also a part of the tester's job. In addition, test specialists must be creative, imaginative, and experimentoriented.

They need to be able to visualize the many ways that a software item should be tested, and make hypotheses about the different types of defects that could occur and the different ways the software could fail. On the *technical* level testers need to have:

• an education that includes an understanding of general software engineering principles, practices, and methodologies;
• strong coding skills and an understanding of code structure and behavior;
• a good understanding of testing principles and practices;
• a good understanding of basic testing strategies, methods, and techniques;
• the ability and experience to plan, design, and execute test cases and test procedures on multiple levels (unit, integration, etc.);
• a knowledge of process issues;
• knowledge of how networks, databases, and operating systems are organized and how they work;

• a knowledge of configuration management;
• a knowledge of test-related documents and the role each documents plays in the testing process;
• the ability to define, collect, and analyze test-related measurements;
• the ability, training, and motivation to work with testing tools and equipment;
• a knowledge of quality issues.

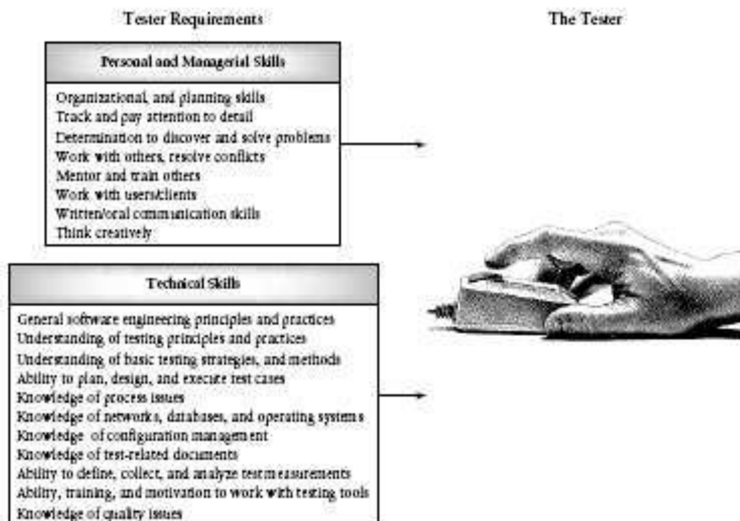


FIG. 8.1
Test specialist skills.

In order to carry out testing tasks testers need to have knowledge of how requirements, specifications, and designs are developed and how different methodologies can be applied. They should understand how errors and defects are introduced into the software artifacts even at early stages of the life cycle. Testers should have strong programming backgrounds to help them visualize how code works, how it behaves, and the possible defects it could contain. They also need coding experience to support the development of the test harnesses which often involve a considerable coding effort in themselves.

Testers must have a knowledge of both white and black box techniques and methods and the ability to use them to design test cases. Organizations need to realize that this knowledge is a necessary prerequisite for tool use and test automation. Testers need to understand the need for multilevel tests and approaches used for testing at each level.

4.12 Building a testing group

It was mentioned that organizing, staffing, and directing were major activities required to manage a project and a process.These apply to managing the testing process as well. Staffing activities include filling positions, assimilating new personnel, education and training, and staff

evaluation. Directing includes providing leadership, building teams, facilitating communication, motivating personnel, resolving conflicts, and delegating authority. Organizing includes selecting organizational structures, creating positions, defining responsibilities, and delegating authority. Hiring staff for the testing group, organizing the testing staff members into teams, motivating the team members, and integrating the team into the overall organizational structure are organizing, staffing, and directing activities your organization will need to perform to build a managed testing process.

Establishing a specialized testing group is a major decision for an organization. The steps in the process are summarized in Figure 8.2. To initiate the process, upper management must support the decision to establish a test group and commit resources to the group. Decisions must be made on how the testing group will be organized, what career paths are available, and how the group fits into the organizational structure (see Section 8.3). When hiring staff to fill test specialist positions, management should have a clear idea of the educational and skill levels required for each testing position and develop formal job descriptions to fill the test group slots. When the job description has been approved and distributed, the interviewing process takes place. Interviews should be structured and of a problem-solving nature. The interviewer should prepare an extensive list of questions to determine the interviewee's technical background as well as his or her personal skills and motivation. Zawacki has developed a general guide for selecting technical staff members that can be used by test managers . Dustin describes the kinds of questions that an interviewer should ask when selecting a test specialist [2]. When the team has been selected and is up and working on projects, the team manager is responsible for keeping the test team positions filled (there are always attrition problems). He must continually evaluate team member performance. Bartol and Martin have written a paper that contains guidelines for evaluation of employees that can be applied to any type of team and organization .They describe four categories for employees based on their performance: (i) retain, (ii) likely to retain, (iii) likely to release, (iv) and release. For each category, appropriate actions need to be taken by the manager to help employee and employer.

FIG. 8.2
Steps in forming a test group.

Structure of test group

It is important for a software organization to have an independent testing group. The group should have a formalized position in the organizational hierarchy. A reporting structure should be established and resources allocated to the group. will eventually need to upgrade their testing function to the best case scenario which is a permanent centralized group of dedicated testers with the skills described earlier in this chapter. This group is solely responsible for testing work. The group members are assigned to projects throughout the organization where they do their testing work. When the project is completed they return to the test organization for reassignment. They report to a test manager or test director, not a project manager. In such an organization testers are viewed as assets. They have defined career paths to follow which contributes to long-term job satisfaction. Since they can be assigned to a project at its initiation, they can give testing support throughout the software life cycle. Because of the permanent nature of the test organization there is a test infrastructure that endures. There is a test knowledge base of test processes, test procedures, test tools, and test histories (lessons learned). Dedicated staff is responsible for maintaining a test case and test harness library.

A test organization is expensive, it is a strategic commitment. Given the complex nature of the software being built, and its impact on society, organizations must realize that a test organization is necessary and that it has many benefits. By investing in a test organization a company has access to a group of specialists who have the responsibilities and motivation to:

• maintain testing policy statements;
• plan the testing efforts;
• monitor and track testing efforts so that they are on time and within budget;
• measure process and product attributes;
• provide management with independent product and process quality information;
• design and execute tests with no duplication of effort;
• automate testing;
• participate in reviews to insure quality; are meet.

The duties of the team members may vary in individual organizations. The following gives a brief description of the duties for each tester that are common to most organizations.

The Test Manager

In most organizations with a testing function, the test manager (or test director) is the central person concerned with all aspects of testing and quality issues. The test manager is usually responsible for test policy making, customer interaction, test planning, test documentation, controlling and monitoring of tests, training, test tool acquisition, participation in inspections and walkthroughs, reviewing test work, the test repository, and staffing issues such as hiring, firing, and evaluation of the test team members. He or she is also the liaison with upper management, project management, and the quality assurance and marketing staffs.

The Test Lead

The test lead assists the test manager and works with a team of test engineers on individual projects. He or she may be responsible for duties such as test planning, staff supervision, and

status reporting. The test lead also participates in test design, test execution and reporting, technical reviews, customer interaction, and tool training.

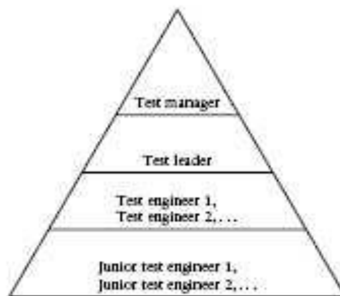


FIG. 8.3
The test team hierarchy.

The Test Engineer
The test engineers design, develop, and execute tests, develop test harnesses, and set up test laboratories and environments. They also give input to test planning and support maintenance of the test and defect repositories.

The Junior Test Engineer
The junior test engineers are usually new hires. They gain experience by participating in test design, test execution, and test harness development. They may also be asked to review user manuals and user help facilities defect and maintain the test and defect repositories.

Unit IV

Part-A Questions

1. What are the goals of testing and degugging?
2. List the Skills needed by a test specialist.
3. Give the hierarchy of test plans.
4. Define:Test group.

Part-B Questions

1. Explain the steps in forming a test group .
2. Explain in brief about test cost impact items.
3. Explain elaborately about the basic test plan components as described in IEEE 829-1983.
4. **Explain the** Testing and debugging goals and policies.

UNIT V CONTROLLING AND MONITORING

5.1 Defining terms

Project monitoring (or tracking) refers to the activities and tasks managers engage in to periodically check the status of each project. Reports are prepared that compare the actual work done to the work that was planned.

Monitoring requires a set of tools, forms, techniques, and measures. A precondition for monitoring a project is the existence of a project plan.

Project controlling consists of developing and applying a set of corrective actions to get a project on track when monitoring shows a deviation from what was planned.

If monitoring results show deviations from the plan have occurred, controlling mechanisms must be put into place to direct the project back on its proper track. Controlling a project is an important activity which is done to ensure that the project goals will be achieved occurring to the plan. Many managerial experts group the two activities into one called “controlling”.

Thayer partitions what he calls “project controlling” into six major tasks. The following is a modified description of the tasks suggested by Thayer. The description has been augmented by the author to include supplemental tasks that provide additional support for the controlling and monitoring functions.

1. *Develop standards of performance.* These set the stage for defining goals that will be achieved when project tasks are correctly accomplished.
2. *Plan each project.* The plan must contain measurable goals, milestones, deliverables, and well-defined budgets and schedules that take into consideration project types, conditions, and constraints.
3. *Establish a monitoring and reporting system.* In the monitoring and reporting system description the organization must describe the measures to be used, how/when they will be collected, what questions they will answer, who will receive the measurement reports, and how these will be used to control the project. Each project plan must describe the monitoring and reporting mechanisms that will be applied to it. If status meetings are required, then their frequency, attendees, and resulting documents must be described.
4. *Measure and analyze results.* Measurements for monitoring and controlling must be collected, organized, and analyzed. They are then used to compare the actual achievements with standards, goals, and plans.
5. *Initiate corrective actions for projects that are off track.* These actions may require changes in the project requirements and the project plan.
6. *Reward and discipline.* Reward those staff who have shown themselves to be good performers, and discipline, retrain, relocate those that have consistently performed poorly.
7. *Document the monitoring and controlling mechanisms.* All the methods, forms, measures, and tools that are used in the monitoring and controlling process must be documented in organization standards and be described in policy statements.
8. *Utilize a configuration management system.* A configuration management system is needed to manage versions, releases, and revisions of documents, code, plans, and reports.

It was Thayer's intent that these activities and actions be applied to monitor and control software development projects. However, these activities/ actions can be applied to monitor and control testing efforts as well.

5.2 Measurements and milestones for controlling and monitoring

All processes should have measurements (metrics) associated with them. The measurements help to answer questions about status and quality of the process, as well as the products that result from its implementation. Measurements in the testing domain can help to track test progress, evaluate the quality of the software product, manage risks, classify and prevent defects, evaluate test effectiveness, and determine when to stop testing. Level 4 of the TMM calls for a formal test measurement program. However, to establish a baseline process, to put a monitoring program into place, and to evaluate improvement efforts, an organization needs to define, collect, and use measurements starting at the lower levels of the TMM.

To begin the collection of meaningful measurements each organization should answer the following questions:
• Which measures should we collect?
• What is their purpose (what kinds of questions can they answer)?
• Who will collect them?
• Which forms and tools will be used to collect the data?
• Who will analyze the data?
• Who to have access to reports?
When these question have been addressed, an organization can start to collect simple measurements beginning at TMM level 1 and continue to add measurements as their test process evolves to support test process evaluation and improvement and process and product quality growth. In this chapter we are mainly concerned with monitoring and controlling of the testing process as defined in Section 9.0, so we will confine ourselves to discussing measurements that are useful for this purpose. Chapter 11 will provide an in-depth discussion of how to develop a full-scale measurement program applicable to testing. Readers will learn how measurements support test process improvement and product quality goals.

The following sections describe a collection of measurements that support monitoring of test over time. Each measurement is shown in italics to highlight it. It is recommended that measurements followed by an asterisk (*) be collected by all organizations, even those atTMMlevel 1. The reader should note that it is not suggested that all of the measurements listed be collected by an organization. TheTMMlevel, and the testing goals that an organization is targeting, affect the appropriateness of these measures. As a simple example, if a certain degree of branch coverage is not a testing objective for a organization at this time, then this type of measurement is not relevant. However, the organization should strive to include such goals in their test polices and plans in the future. Readers familiar with software metrics concepts should note that most of the measures listed in this chapter are mainly process measures; a few are product measures. Other categories for the measures listed here are (i) explicit, those that are measured directly from the process or product itself, and (ii) derived, those that are a result of the

combination of explicit and/or other derived measures. Note that the ratios described are derived measures.

Now we will address the question of how a testing process can be monitored for each project. A test manager needs to start with a test plan. What the manager wants to measure and evaluate is the actual work that was done and compare it to work that was planned. To help support this goal, the test plan must contain testing milestones as described in Chapter 7.

Milestones are tangible events that are expected to occur at a certain time in the project's lifetime. Managers use them to determine project status.

Test milestones can be used to monitor the progress of the testing efforts associated with a software project. They serve as guideposts or goals that need to be meet. A test manger uses current testing effort data to determine how close the testing team is to achieving the milestone of interest. Milestones usually appear in the scheduling component of the test plan (see Chapter 7). Each level of testing will have its own specific milestones. Some examples of testing milestones are:

- completion of the master test plan;
- completion of branch coverage for all units (unit test);
- implementation and testing of test harnesses for needed integration of major subsystems;
- execution of all planned system tests;
- completion of the test summary report.

Each of these events will be scheduled for completion during a certain time period in the test plan. Usually a group of test team members is responsible for achieving the milestone on time and within budget. Note that the determination of whether a milestone has been reached depends on availability of measurement data. For example, to make the above milestones useful and meaningful testers would need to have measurements in place such as:

- degree of branch coverage accomplished so far;
- number of planned system tests currently available;
- number of executed system tests at this date.

Test planners need to be sure that milestones selected are meaningful for the project, and that completion conditions for milestone tasks are not too ambiguous. For example, a milestone that states "unit test is completed when all the units are ready for integration" is too vague to use for monitoring progress. How can a test manager evaluate the condition, "ready"? Because of this ambiguous completion condition, a test manager will have difficulty determining whether the milestone has been reached. During the monitoring process measurements are collected that relates to the status of testing tasks (as described in the test plan), and milestones. Graphs using test process data are developed to show trends over a selected time period. The time period can be days, weeks, or months depending on the activity being monitored. The graphs can be in

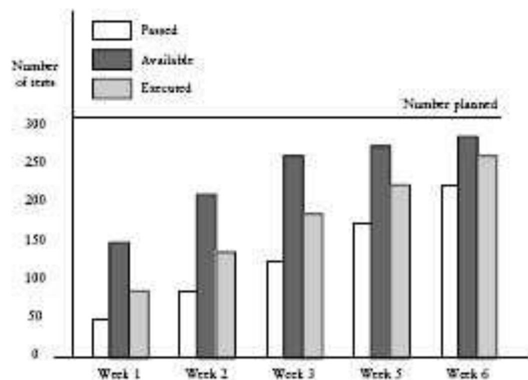


FIG. 9.1
Graph showing trends in test execution.

the form of a bar graph as shown in Figure 9.1 which illustrates trends for test execution over a 6-week period. They can also be presented in the form of *x,y* plots where the *y*-axis would be the number of tests and the *x*-axis would be the weeks elapsed from the start of the testing process for the project. These graphs, based on current measurements, are presented at the weekly status meetings and/or at milestone reviews that are used to discuss progress. At the status meetings, project and test leaders present up-to-date measurements, graphs and plots showing the status of testing efforts.

Testing milestones met/not met and problems that have occurred are discussed. Test logs, test incident reports, and other test-related documents may be examined as needed. Managers will have questions about the progress of the test effort. Mostly, they will want to know if testing is proceeding according to schedules and budgets, and if not, what are the barriers. Some of the typical questions a manager might ask at a status meeting are:
Have all the test cases been developed that were planned for this date?
• What percent of the requirements/features have been tested so far?
• How far have we proceeded on achieving coverage goals: Are we ahead or behind what we scheduled?
• How many defects/KLOC have been detected at this time?Howmany repaired? How many are of high severity?
• What is the earned value so far? Is it close to what was planned (see Section 9.1.3)?
• How many available test cases have been executed? How many of these were passed?
• How much of the allocated testing budget has been spent so far? Is it more or less than we estimated?
• How productive is tester X? How many test cases has she developed? How many has she run? Was she over, or under, the planned amount?
The measurement data collected helps to answer these questions. In fact, links between measurements and question are described in the Goals/ Questions/Metrics (GQM) paradigm reported by Basili [2]. In the case of testing, a major *goal* is to monitor and control testing efforts (a maturity goal at TMM level 3). An organizational team (developers/testers, SQA staff, project/test managers) constructs a set of likely *questions* that test/project managers are likely to ask in order to monitor and control the testing process. The sample set of questions previously

described is a good starting point. Finally, the team needs to identify a set of *measurements* that can help to answer these questions. A sample set of measures is provided in the following sections. Any organizational team can use them as a starting point for selecting measures that help to answer testrelated monitoring and controlling questions. Four key items are recommended to test managers for monitoring and controlling the test efforts for a project. These are:

(i) testing status;
(ii) tester productivity;
(iii) testing costs;
(iv) errors, faults, and failures.

In the next sections we will examine the measurements required to track these items. Keep in mind that for most of these measurements the test planner should specify a planned value for the measure in the test plan. During test the actual value will be measured during a specific time period, and the two then compared.

Measurements for Monitoring Testing Status

Monitoring testing status means identifying the current state of the testing process. The manager needs to determine if the testing tasks are being completed on time and within budget. Given the current state of the testing effort some of the questions under consideration by a project or test manager would be the following:

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

The following set of measures will help to answer these questions. The test status measures are partitioned into four categories as shown in Figure 9.2. A test plan must be in place that describes, for example, planned coverage goals, the number of planned test cases, the number of requirements to be tested, and so on, to allow the manager to compare actual measured values to those expected for a given time period.

1. Coverage Measures

As test efforts progress, the test manager will want to determine how much coverage has been actually achieved during execution of the tests, and how does it compare to planned coverage. Depending on coverage goals for white box testing, a combination of the following are recommended.

*Degree of statement, branch, data flow, basis path, etc., coverage (planned, actual)**

Tools can support the gathering of this data. Testers can also use ratios such as: *Actual degree of coverage/planned degree of coverage* to monitor coverage to date. For black box coverage the following measures can be useful:

*Number of requirements or features to be tested**
Number of equivalence classes identified
Number of equivalence classes actually covered
*Number or degree of requirements or features actually covered**

Testers can also set up ratios during testing such as:
*Number of features actually covered/total number of features**
This will give indication of the work completed to this date and the work that still needs to be done.

Test Case Development
The following measures are useful to monitor the progress of test case development, and can be applied to all levels of testing. Note that some are explicit and some are derived. The number of estimated test cases described in the master test plan is:
Number of planned test cases
The number of test cases that are complete and are ready for execution is:
Number of available test cases
In many cases new test cases may have to be developed in addition to those that are planned. For example, when coverage goals are not meet by the current tests, additional tests will have to be designed. If mutation testing is used, then results of this type of testing may require additional tests to kill the mutants. Changes in requirements could add new test cases to those that were planned. The measure relevant here is:
Number of unplanned test cases
In place of, or in addition to, test cases, a measure of the number planned, available, and unplanned test procedures is often used by many organizations to monitor test status.

Test Execution
As testers carry out test executions, the test manager will want to determine if the execution process is going occurring to plan. This next group of measures is appropriate.
*Number of available test cases executed**
*Number of available tests cases executed and passed**
Number of unplanned test cases executed
Number of unplanned test cases executed and passed.
For a new release where there is going to be regression testing then these are useful:
Number of planned regression tests executed
Number of planned regression tests executed and passed
Testers can also set up ratios to help with monitoring test execution. For example:
Number of available test cases executed/number of available test cases
Number of available test cases executed/number of available test cases executed and passed
These would be derived measures.

Test Harness Development
It is important for the test manager to monitor the progress of the development of the test harness code needed for unit and integration test so that these progress in a timely manner according to the test schedule. Some useful measurements are:
*Lines of Code (LOC) for the test harnesses (planned, available)**
Size is a measure that is usually applied by managers to help estimate the amount of effort needed to develop a software system. Size is measured in many different ways, for example, lines of code, function points, and feature points. Whatever the size measure an organization uses to measure its code, it can be also be applied to measure the size of the test harness, and to

estimate the effort required to develop it. We use lines of code in the measurements listed above as it is the most common size metric and can be easily applied to estimating the size of a test harness. Ratios such as:
Available LOC for the test harness code/planned LOC for the test harnesses are useful to monitor the test harness development effort over time.

M e a s u r e m e n t s t o M o n i t o r T e s t e r P r o d u c t i v i t y

Managers have an interest in learning about the productivity of their staff, and how it changes as the project progresses. Measuring productivity in the software development domain is a difficult task since developers are involved in many activities, many of which are complex, and not all are readily measured. In the past the measure LOC/hour has been used to evaluate productivity for developers. But since most developers engage in a variety of activities, the use of this measure for productivity is often not credible. Productivity measures for testers have been sparsely reported. The following represent some useful and basic measures to collect for support in test planning and monitoring the activities of testers throughout the project. They can help a test manger learn how a tester distributes his time over various testing activities. For each developer/tester, where relevant, we measure both planned and actual:
Time spent in test planning
*Time spent in test case design**
*Time spent in test execution**
Time spent in test reporting
*Number of test cases developed**
*Number of test cases executed**
Productivity for a tester could be estimated by a combination of:
*Number of test cases developed/unit time**
*Number of tests executed/unit time**
*Number of LOC test harness developed/unit time**
Number of defects detected in testing/unit time
The last item could be viewed as an indication of testing efficiency. This measure could be partitioned for defects found/hour in each of the testing phases to enable a manager to evaluate the efficiency of defect detection for each tester in each of these activities. For example:
Number of defects detected in unit test/hour
Number of defects detected in integration test/hour, etc.
The relative effectiveness of a tester in each of these testing activities could be determined by using ratios of these measurements. Marks suggests as a tester productivity measure [3]:
Number of test cases produced/week
All of the above could be monitored over the duration of the testing effort for each tester. Managers should use these values with caution because a good measure of testing productivity has yet to be identified. Two other comments about these measures are:
1. Testers perform a variety of tasks in addition to designing and running test cases and developing test harnesses. Other activities such as test planning, completing documents, working on quality and process issues also consume their time, and those must be taken into account when productivity is being considered.

2. Testers should be aware that measurements are being gathered based on their work, and they should know what the measurements will be used for. This is one of the cardinal issues in implementing a measurement program. All involved parties must understand the purpose of collecting the data and its ultimate use.

Measurements for Monitoring Testing Costs

Besides tracking project schedules, recall that managers also monitor costs to see if they are being held within budget. One good technique that project managers use for budget and resource monitoring is called earned value tracking. This technique can also be applied to monitor the use of resources in testing. Test planners must first estimate the total number of hours or budget dollar amount to be devoted to testing. Each testing task is then assigned a value based on its estimated percentage of the total time or budgeted dollars. This gives a relative value to each testing task, with respect to the entire testing effort. That value is credited only when the task is completed. For example, if the testing effort is estimated to require 200 hours, a 20-hour testing task is given a value of 20/200*100 or 10%. When that task is completed it contributes 10% to the cumulative earned value of the total testing effort. Partially completed tasks are not given any credit. Earned values are usual presented in a tabular format or as a graph. An example will be given in the next section of this chapter. The graphs and tables are useful to present at weekly test status meetings.

To calculate planned earned values we need the following measurement data:
Total estimated time or budget for the overall testing effort
Estimated time or budget for each testing task
Earned values can be calculated separately for each level of testing. This would facilitate monitoring the budget/resource usage for each individual testing phase (unit, integration, etc.). We want to compare the above measures to:
*Actual cost/time for each testing task**
We also want to calculate:
Earned value for testing tasks to date
and compare that to the planned earned value for a specific date. Section 9.2 shows an earned value tracking form and contains a discussion of how to apply earned values to test tracking. Other measures useful for monitoring costs such as the number of planned/actual test procedures (test cases) are also useful for tracking costs if the planner has a good handle on the relationship between these numbers and costs (see Chapter 7).
Finally, the ratio of:
Estimated costs for testing/Actual costs for testing can be applied to a series of releases or related projects to evaluate and promote more accurate test cost estimation and higher test cost effectiveness through test process improvement.

Measurements for Monitoring Errors, Faults, and Failures

Monitoring errors, faults, and failures is very useful for:
• evaluating product quality;
• evaluating testing effectiveness; making stop-test decisions;
• defect casual analysis;
• defect prevention;

• test process improvement;
• development process improvement.

Test logs, test incident reports, and problem reports provide test managers with some of the raw data for this type of tracking. Test managers usually want to track defects discovered as the testing process continues over time to address the second and third items above. The other items are useful to SQA staff, process engineers, and project managers. At higher levels of the TMM where defect data has been carefully stored and classified, mangers can use past defect records from similar projects or past releases to compare the current project defect discovery rate with those of the past. This is useful information for a stop-test decision (see Section 9.3). To strengthen the value of defect/failure information, defects should be classified by type, and severity levels should be established depending on the impact of the defect/failure on the user. If a failure makes a system inoperable it has a higher level of severity than one that is just annoying. A example of a severity level rating hierarchy is shown in Figure 9.3.

Some useful measures for defect tracking are:

Total number of incident reports (for a unit, subsystem, system)*

*Number of incident reports resolved/unresolved (for all levels of test)**

*Number of defects found of each given type**

Number of defects causing failures of severity level greater than X found (where X is an appropriate integer value)

*Number of defects/KLOC (This is called the defect volume. The division by KLOC normalizes the defect count)**

*Number of failures**

Number of failures over severity level Y (where Y is an appropriate integer value)

*Number of defects repaired**

Estimated number of defects (from historical data)

Other failure-related data that are useful for tracking product reliability will be discussed in later chapters.

M o n i t o r i n g T e s t E f f e c t i v e n e s s

To complete the discussion of test controlling and monitoring and the role of test measurements we need to address what is called test effectiveness. Test effectiveness measurements will allow managers to determine if test resources have been used wisely and productively to remove defects and evaluate product quality. Test effectiveness evaluations allow managers to learn which testing activities are or are not productive. For those areas that need improvement, responsible staff should be assigned to implement and monitor the changes. At higher levels of the TMM members of a process improvement group can play this role. The goal is to make process changes that result in improvements to the weak areas. There are several different views of test effectiveness. One of these views is based on use of the number of defects detected. For example, we can say that our testing process was effective if we have successfully revealed all defects that have a major impact on the users. We can make such an evaluation in several ways, both before and after release.

1. *Before release.* Compare the numbers of defects found in testing for this software product to the number expected from historical data. The ratio is:

Number of defects found during test/number of defects estimated

This will give some measure of how well we have done in testing the current software as compared to previous similar products. Did we find more or fewer errors given the test resources and time period? This is not the best measure of effectiveness since we can never be sure that the current release contains the same types and distribution of defects as the historical example.

2. *After release.* Continue to collect defect data after the software has been released in the field. In this case the users will prepare problem reports that can be monitored. Marks suggests we use measures such as "field fault density" as a measure of test effectiveness. This is equal to: *Number of defects found/thousand lines of new and changed code*.

This measure is applied to new releases of the software.

Another measure suggested is a ratio of:

Pre-ship fault density/Post-ship fault density .

This ratio, sometimes called the "defect removal efficiency," gives an indication of how many defects remain in the software when it is released. As the testing process becomes more effective, the number of predelivery defects found should increase and postdelivery defects found should fall. The value of the postship fault density (number of faults/KLOC) is calculated from the problem reports returned to the development organization, so testers need to wait until after shipment to calculate this ratio. Testers must examine the problem reports in detail when using the data.

There may be duplicate reports especially if the software is released to several customers. Some problem reports are due to misunderstandings; others may be requests for changes not covered in the requirements. All of these should be eliminated from the count. Other measurements for test effectiveness have been proposed. For example,:

Number of defects detected in a given test phase/total number of defects found in testing.

For example, if unit test revealed 35 defects and the entire testing effort revealed 100 defects, then it could be said that unit testing was 35% effective. If this same software was sent out to the customer and 25 additional defects were detected, then the effectiveness of unit test would then be 25/125, or 20%. Testers can also use this measure to evaluate test effectiveness in terms of the severity of the failures caused by the defects. In the unit test example, perhaps it was only 20% effective in finding defects that caused severe failures. The fault seeding technique as described in Section 9.3 could also be applied to evaluate test effectiveness. If you know the number of seeded faults injected and the number of seeded faults you have already found, you can use the ratio to estimate how effective you have been in using your test resources to date. Another useful measure, called the "detect removal leverage (DRL)" described in Chapter 10 as a review measurement, can be applied to measure the relative effectiveness of: reviews versus test phases, and test phases with respect to one another. The DRL sets up ratios of defects found. The ratio denominator is the base line for comparison. For example, one can compare:

DRL (integration/unit test) _ Number of defects found integration test

Number of defects found in unit test

Section 10.7 gives more details on the application of this metric. The costs of each testing phase relative to its defect detecting ability can be expressed as:

Number of defects detected in testing phase X

Costs of testing in testing phase X

Instead of actual dollar amounts, tester hours, or any other indicator of test resource units could also be used in the denominator. These ratios could calculated for all test phases to compare their relative effectiveness. Comparisons could lead to test process changes and improvements. An additional approach to measuring testing effectiveness is described by Chernak [8]. The main objectives of Chernak's research are (i) to show how to determine if a set of test cases (a test suite) is sufficiently effective in revealing defects, and (ii) to show how effectiveness measures can lead to process changes and improvements. The effectiveness metric called the TCE is defined as follows:
Number of defects found by the test cases
TCE _Total number of defects _ 100
The total number of defects in this equation is the sum of the defects found by the test cases, plus the defects found by what Chernak calls side effects. Side effect are based on so-called "testescapes." These are software defects that a test suite does not detect but are found by chance in the testing cycle.

Test escapes occur because of deficiencies in the testing process. They are identified when testers find defects by executing some steps or conditions that are not described in a test case specification. This happens by accident or because the tester gets a new idea while performing the assigned testing tasks. Under these conditions a tester may find additional defects which are the test-escapes. These need to be recorded, and a casual analysis needs to be done to develop corrective actions. The use of Chernak's metric depends on finding and recording these types of defects. Not all types of projects are candidates for this type of analysis. From his experience, Chernak suggests that client-server business applications may be appropriate projects. He also suggests that a baseline value be selected for the TCE and be assigned for each project.

When the TCE value is at or above the baseline then the conclusion is that the test cases have been effective for this test cycle, and the testers can have some confidence that the product will satisfy the uses needs. All test case escapes, especially in the case of a TCE below the specified baseline, should be studied using Pareto analysis and Fishbone diagram techniques (described in Chapter 13), so that test design can be improved, and test process deficiencies be removed. Chernak illustrates his method with a case study (a client-server application) using the baseline TCE to evaluate test effectiveness and make test process improvements. When the TCE in the study was found to be below the baseline value (_ 75 for this case), the organization analyzed all the test-escapes, classified them by cause, and built a Pareto diagram to describe the distribution of causes.

Incomplete test design and incomplete functional specifications were found to be the main causes of test-escapes. The test group then addressed these process issues, adding both reviews to their process and sets of more "negative" test cases to improve the defect-detecting ability of their test suites.The TMM level number determined for an organization is also a metric that can be used to monitor the testing process. It can be viewed as a high-level measure of test process effectiveness, proficiency, and overall maturity. A mature, testing process is one that is effective. The TMM level number that results from a TMM assessment is a measurement that gives an organization information about the state of its testing process. A lower score on theTMMlevel number scale indicates a less mature, less proficient, less effective testing process state then a

higher-level score. The usefulness of the TMM level number as a measurement of testing process strength, proficiency, and effectiveness is derived not only from its relative value on the TMM maturity scale, but also from the process profile that accompanies the level number showing strong and weak testing areas. In addition, the maturity goals hierarchy give structure and direction to improvement efforts so that the test process can become more effective.

5.3 Status meetings -Reports and control issues

Roughly forty measurements have been listed here that are useful for monitoring testing efforts. Organizations should decide which are of the most value in terms of their current TMM level, and the monitoring and controlling goals they want to achieve. The measurement selection process should begin with these goals, and compilation of a set of questions most likely to be asked by management relating to monitoring and controlling of the test process. The measurements that are selected should help to answer the questions (see brief discussion of the Goal/Question/Metric paradigm in Section 9.1). A sample set of questions is provided at the beginning of this chapter. Measurement-related data, and other useful test-related information such as test documents and problem reports, should be collected and organized by the testing staff. The test manager can then use these items for presentation and discussion at the periodic meetings used for project monitoring and controlling. These are called project status meetings. Test-specific status meetings can also serve to monitor testing efforts, to report test progress, and to identify any test-related problems. Testers can meet separately and use test measurement data and related documents to specifically discuss test status. Following this meeting they can then participate in the overall project status meeting, or they can attend the project meetings as an integral part of the project team and present and discuss test-oriented status data at that time. Each organization should decide how to organize and partition the meetings. Some deciding factors may be the size of the test and development teams, the nature of the project, and the scope of the testing effort. Another type of project-monitoring meeting is the milestone meeting that occurs when a milestone has been met. A milestone meeting is an important event; it is a mechanism for the project team to communicate with upper management and in some cases user/client groups. Major testing milestones should also precipitate such meetings to discuss accomplishments and problems that have occurred in meeting each test milestone, and to review activities for the next milestone phase. Testing staff, project managers, SQA staff, and upper managers should attend. In some cases process improvement group and client attendance is also useful.

Milestone meetings have a definite order of occurrence; they are held when each milestone is completed. How often the regular statues meetings are held depends on the type of project and the urgency to discuss issues. Rakos recommends a weekly schedule as best for small- to medium-sized projects . Typical test milestone meeting attendees are shown in Figure 9.4. It is important that all test-related information be available at the meeting, for example, measurement data, test designs, test logs, test incident reports, and the test plan itself.

Status meetings usually result in some type of status report published by the project manager that is distributed to upper management. Testmanagers should produce similar reports to inform

management of test progress. Rakos recommends that the reports be brief and contain the following items :

• *Activities and accomplishments during the reporting period.* All tasks that were attended to should be listed, as well as which are complete. The latter can be credited with earned value amounts. Progress made since the last reporting period should also be described.

• *Problems encountered since the last meeting period.* The report should include a discussion of the types of new problems that have occurred, their probable causes, and how they impact on the project.Problem solutions should be described.

FIG. 9.4
Test milestone meetings, participants, inputs, and outputs.

• *Problems solved.* At previous reporting periods problems were reported that have now been solved. Those should be listed, as well as the solutions and the impact on the project.

• *Outstanding problems.* These have been reported previously, but have not been solved to date. Report on any progress.

• *Current project (testing) state versus plan.* This is where graphs using process measurement data play an important role. Examples will be described below. These plots show the current state of the project (testing) and trends over time.

• *Expenses versus budget.* Plots and graphs are used to show budgeted versus actual expenses. Earned value charts and plots are especially useful here.

• *Plans for the next time period.* List all the activities planned for the next time period as well as the milestones.

Preparing and examining graphs and plots using the measurement data we have discussed helps managers to see trends over time as the test effort progresses. They can be prepared for presentation at meetings and included in the status report. An example bar graph for monitoring purposes is shown in Figure 9.1. The bar graph shows the numbers for tests that were planned, available, executed, and passed during the first 6 weeks of the testing effort. Note the trends. The number of tests executed and the number passed has gone up over the 6 weeks, The number passed is approaching the number executed. The graph indicates to the manager that the number of executed tests is approaching the number of tests available, and that the number of tests passed is also approaching the number available, but not quite as quickly. All are approaching the number planned. If one extrapolates, the numbers should eventually converge at some point in time. The bar graph, or a plot, allows the manager to identify the time frame in which this will

occur. Managers can also compare the number of test cases executed each week with the amount that were planned for execution.

Figure 9.5 shows another graph based on defect data. The total number of faults found is plotted against weeks of testing effort. In this plot the number tapers off after several weeks of testing. The number of defects repaired is also plotted. It lags behind defect detection since the code must be returned to the developers who locate the defects and repair the code. In many cases this be a very time-consuming process. Managers can also include on a plot such as Figure 9.5 the expected rate of defect detection using data from similar past projects. However, even if the past data are typical there is no guarantee that the current software will behave in a similar way. Other ways of estimating the number of potential defects use rules of thumb (heuristics) such as "0.5-1% of the total lines of code" [8]. These are at best guesses, and give managers a way to estimate the number of defects remaining in the code, and as a consequence how long the testing effort needs to continue. However, this heuristic gives no indication of the severity level of the defects. Hetzel gives additional examples of the types of plots that are useful for monitoring testing efforts [9]. These include plots of number of requirements tested versus weeks of effort and the number of statements not yet exercised over time. Other graphs especially useful for monitoring testing costs are those that plot staff hours versus time, both actual and planned. Earned value tables and graphs are also useful. Table 9.1 is an example [4].

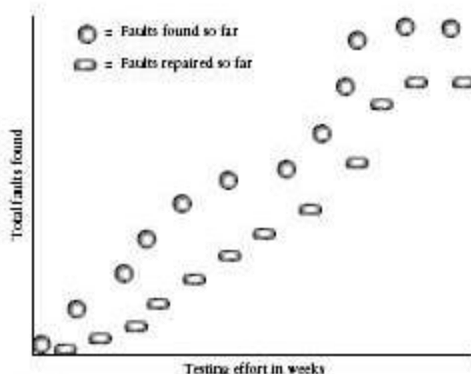


FIG. 9.5
Sample plot for monitoring fault detection durings test.

Note that the earned value table shown in Table 9.1 has two parti tions, one for planned values and one for actual values. Each testing task should be listed, as well as its estimated hours for completion. The total hours for all the tasks is determined and the estimated earned value for each task is then calculated based on its estimated percentage of the total time as described previously. This gives a relative value to each testing task with respect to the entire testing effort. The estimated earned values are accumulated in the next column. When the testing effort is in progress, the date and actual earned value for each task is listed, as well as the actual accumulated earned values. In status report graphs, earned value is usually plotted against time, and on the same graph budgeted expenses and actual expenses may also be plotted against time for comparison. Although actual expenses may be more than budget, if earned value is higher than expected, then progress may be considered satisfactory [4,5]. The agenda for a status

meeting on testing includes a discussion of the work in progress since the last meeting period. Measurement data is presented, graphs are produced, and progress is evaluated. Test logs and incident reports may be examined to get a handle on the problems occurring.

If there are problem areas that need attention, they are discussed

Testing task	Planned			Actual		
	Estimated hours	Estimated earned value	Cummulative earned value	Date	Actual earned value	Cummulative earned value

TABLE 9.1
Sample earned value table [4].

and solutions are suggested to get the testing effort back on track (control it). Problems currently occurring may be closely associated with risks identified by the test manager through the risk analysis done in test planning. Recall that part of the planner's job is identify and prioritize risks, and to develop contingency plans to handle the risk-prone events if they occur. If the test manager has done a careful job, these contingency plans may be applied to the problem at hand. Suggested and agreed-upon solutions should appear in the status report. The corrective actions should be put in place, their effect on testing monitored, and their success/failure discussed at the next status meeting. As testing progresses, status meeting attendees have to make decisions about whether to stop testing or to continue on with the testing efforts, perhaps developing additional tests as part of the continuation process. They need to evaluate the status of the current testing efforts as compared to the expected state specified in the test plan. In order to make a decision about whether testing is complete the test team should refer to the stoptest criteria included in the test plan (see the next section for a discussion on stop-test criteria). If they decide that the stop-test criteria have been met, then the final status report for testing, the test summary report, should be prepared. This is a summary of the testing efforts, and becomes a part of the project's historical database. At project postmortems the test summary report can be used to discuss successes and failures that occurred during testing. It is a good source for test lessons learned for each project.

5.3 Criteria for test completion

In the test plan the test manager describes the items to be tested, test cases, tools needed, scheduled activities, and assigned responsibilities. As the testing effort progresses many factors impact on planned testing schedules and tasks in both positive and negative ways. For example, although a certain number of test cases were specified, additional tests may be required. This may be due to changes in requirements, failure to achieve coverage goals, and unexpected high numbers of defects in critical modules. Other unplanned events that impact on test schedules are, for example, laboratories that were supposed to be available are not (perhaps because of equipment failures) or testers who were assigned responsibilities are absent (perhaps because of illness or assignments to other higherpriority projects). Given these events and uncertainties, test progress does not often follow plan. Tester managers and staff should do their best to take

actions to get the testing effort on track. In any event, whether progress is smooth or bumpy, at some point every project and test manager has to make the decision on when to stop testing. Since it is not possible to determine with certainty that all defects have been identified, the decision to stop testing always carries risks. If we stop testing now, we do save resources and are able to deliver the software to our clients. However, there may be remaining defects that will cause catastrophic failures, so if we stop now we will not find them. As a consequence, clients may be unhappy with our software and may not want to do business with us in the future. Even worse there is the risk that they may take legal action against us for damages. On the other hand, if we continue to test, perhaps there are no defects that cause failures of a high severity level. Therefore, we are wasting resources and risking our position in the market place. Part of the task of monitoring and controlling the testing effort is making this decision about when testing is complete under conditions of uncertainly and risk. Managers should not have to use guesswork to make this critical decision. The test plan should have a set of quantifiable stop-test criteria to support decision making. The weakest stop test decision criterion is to stop testing when the project runs out of time and resources. TMM level 1 organizations often operate this way and risk client dissatisfaction for many projects. TMM level 2 organizations plan for testing and include stop-test criteria in the test plan. They have very basic measurements in place to support management when they need to make this decision. Shown in Figure 9.6 and described below are five stop-test criteria that are based on a more quantitative approach. No one criteria is recommended. In fact, managers should use a combination of criteria and cross-checking for better results. The stop-test criteria are as follows.

1 . A l l the Planned Tests That Were Developed Have Been Executed and Passed.

This may be the weakest criterion. It does not take into account the actual dynamics of the testing effort, for example, the types of defects found and their level of severity. Clues from analysis of the test cases and defects found may indicate that there are more defects in the code that the planned test cases have not uncovered. These may be ignored by the testers if this stop-test criteria is used in isolation.

2 . A l l Specified Coverage Goals Have Been Met.

An organization can stop testing when it meets its coverage goals as specified in the test plan. For example, using white box coverage goals we can say that we have completed unit test when we have reached 100% branch coverage for all units. Using another coverage category, we can say we have completed system testing when all the requirements have been covered by our tests. The graphs prepared for the weekly status meetings can be applied here to show progress and to extrapolate to a completion date. The graphs will show the growth of degree of coverage over the time.

3 . The Detection of a Specific Number of Defects Has Been Accomplished.

This approach requires defect data from past releases or similar projects. The defect distribution and total defects is known for these projects, and is applied to make estimates of the number and types of defects for the current project. Using this type of data is very risky, since it assumes the current software will be built, tested, and behave like the past projects. This is not always true. Many projects and their development environments are not as similar as believed, and making

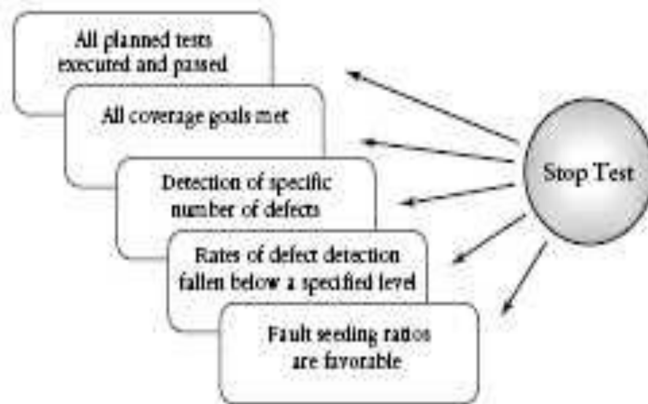


FIG. 9.6

Some possible stop-test criteria.

this assumption could be disastrous. Therefore, using this stop-criterion on its own carries high risks.

4 . The Rates of Defect Detection for a Certain Time Period Have Fallen Below a Specified Level.

The manager can use graphs that plot the number of defects detected per unit time. A graph such as Figure 9.5, augmented with the severity level of the defects found, is useful. When the rate of detection of defects of a severity rating under some specified threshold value falls below that rate threshold, testing can be stopped. For example, a stop-test criterion could be stated as: "We stop testing when we find 5 defects or less, with impact equal to, or below severity level 3, per week." Selecting a defect detection rate threshold can be based on data from past projects.

5 . Fault Seeding Ratios Are Favorable.

Fault (defect) seeding is an interesting technique first proposed by Mills [10]. The technique is based on intentionally inserting a known set of defects into a program. This provides support for a stop-test decision. It is assumed that the inserted set of defects are typical defects; that is, they are of the same type, occur at the same frequency, and have the same impact as the actual defects in the code. One way of selecting such a set of defects is to use historical defect data from past releases or similar projects.

The technique works as follow. Several members of the test team insert (or seed) the code under test with a known set of defects. The other members of the team test the code to try to reveal as many of the defects as possible. The number of undetected seeded defects gives an indication of the number of total defects remaining in the code (seeded plus actual). A ratio can be set up as follows:

$$\frac{\text{Detected seeded defects}}{\text{Total seeded defects}} = \frac{\text{Detected actual defects}}{\text{Total actual defects}}$$

Using this ratio we can say, for example, if the code was seeded with 100 defects and 50 have been found by the test team, it is likely that 50% of the actual defects still remain and the testing effort should continue.When all the seeded defects are found the manager has some confidence that the test efforts have been completed.

5.4 SCM

Software systems are constantly undergoing change during development and maintenance. By software systems we include all software artifacts such as requirements and design documents, test plans, user manuals, code, and test cases. Different versions, variations, builds, and releases exist for these artifacts. Organizations need staff, tools, and techniques to help them track and manage these artifacts and changes to the artifacts that occur during development and maintenance. The Capability Maturity Model includes configuration management as a Key Process Area at level 2. This is an indication of its fundamental role in support of repeatable, controlled, and managed processes. To control and monitor the testing process, testers and test mangers also need access to configuration management tools and staff.

There are four major activities associated with configuration management. These are:

1 . Identification of the Configuration Items

The items that will be under configuration control must be selected, and the relationships between them must be formalized. An example relationship is "part-of" which is relevant to composite items. Relationships are often expressed in a module interconnection language (MIL). Figure 9.7 shows four configuration items, a design specification, a test specification, an object code module, and source code module as they could exist in a configuration management system (CMS) repository (see item 2 below for a brief description of a CMS). The arrows indicate links or relationships between them. Note in this example that the configuration management system is aware that these four items are related only to one another and not to other versions of these items in the repository.

In addition to identification of configuration items, procedures for establishment of baseline versions for each item must be in place.

Baselines are formally reviewed and agreed upon versions of software artifacts, from which all changes are measured. They serve as the basis for further development and can be changed only through formal change procedures. Baselines plus approved changes from those baselines constitute the correct configuration identification for the item. [11].

2 . Change Control

There are two aspects of change control—one is tool-based, the other team-based. The team involved is called a configuration control board. This group oversees changes in the software system. The members of the board should be selected from SQA staff, test specialists, developers, and analysts. It is this team that oversees, gives approval for, and follows up on changes. They develop change procedures and the formats for change request forms. To make a change, a change request form must be prepared by the requester and submitted to the board. It then reviews and approves/ disapproves. Only approved changes can take place. The board also participates in configuration reporting and audits as described further on in this section.

In addition to the configuration control board, control of configuration items requires a configuration management system (CMS) that will

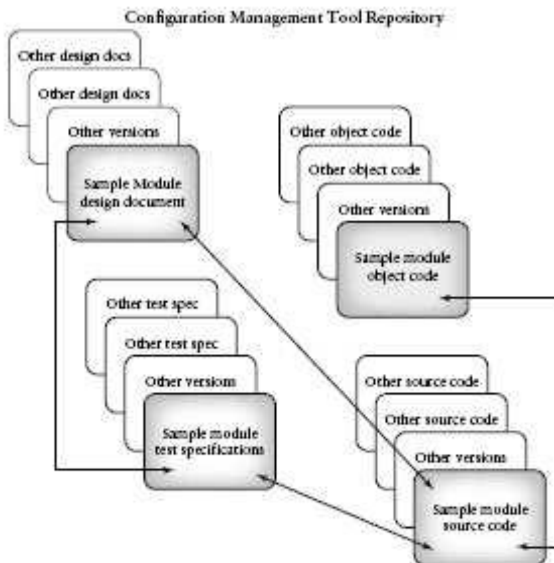


FIG. 9.7
Sample configuration items.

store the configuration items in a repository (or project database) and maintain control and access to those items. The CMS will manage the versions and variations of the items. It will keep track of the items and their relationships with one another. For example, developers and testers need to know which set of test cases is associated with which design item, and which version of object code is associated with which version of source code? The CMS will provide the information needed to answer these questions by supporting relationships as shown in Figure 9.7. It also supports baseline versions for each configuration item, and it only allows designated engineers to make changes to a configuration item after formal approval by the change control board. The software engineer must check- out the item undergoing change from the CMS. A copy of it is made in her work station. When the changes are complete, and they are reviewed, the new version is "checked in" to the CMS, and the version control mechanism in the CMS creates the newest version in its repository. Relationships to existing configuration items are updated. TheCMScontrols change-making by ensuring that an engineer has the proper access rights to the configuration item. It also synchronizes the change-making process so that parallel changes made by different software engineers do not overwrite each other. The CMS also allows software engineers to create builds of the system consisting of different versions and variations of object and source code.

3. Configuration status reporting

These reports help to monitor changes made to configuration items. They contain a history of all the changes and change information for each configuration item. Each time an approved change is made to a configuration item, a configuration status report entry is made. These reports are kept in the CMS database and can be accessed by project personnel so that all can be aware of changes that are made. The reports can answer questions such as:

• who made the change;
• what was the reason for the change;
• what is the date of the change;
• what is affected by the change.
Reports for configuration items can be disturbed to project members and discussed at status meetings.

4. Configuration audits
After changes are made to a configuration item, how do software engineers follow up to ensure the changes have been done properly? One way to do this through a technical review, another through a configuration audit. The audit is usually conducted by the SQA group or members of the configuration control board. They focuses on issues that are not covered in a technical review. A checklist of items to cover can serve as the agenda for the audit. For each configuration item the audit should cover the following:

(i) *Compliance with software engineering standards.* For example, for the source code modules, have the standards for indentation, white space, and comments been followed?
(ii) *The configuration change procedure.* Has it been followed correctly?
(iii) *Related configuration items.* Have they been updated?
(iv) *Reviews.* Has the configuration item been reviewed?

Why is configuration management of interest to testers? Configuration management will ensure that test plans and other test-related documents are being prepared, updated, and maintained properly. To support these objectives, Ayer has suggested a test documentation checklist to be used for configuration audits to verify the accuracy and completeness of test documentation [12]. Configuration management also allows the tester to determine if the proper tests are associated with the proper source code, requirements, and design document versions, and that the correct version of the item is being tested. It also tells testers who is responsible for a given item, if any changes have been made to it, and if it has been reviewed before it is scheduled for test.

5.5 Review program

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.
The general goals for the reviewers are to:
• identify problem components or components in the software artifact that need improvement;
• identify components of the software artifact that do not need improvement;
• identify specific errors or defects in the software artifact (defect detection);
• ensure that the artifact conforms to organizational standards.

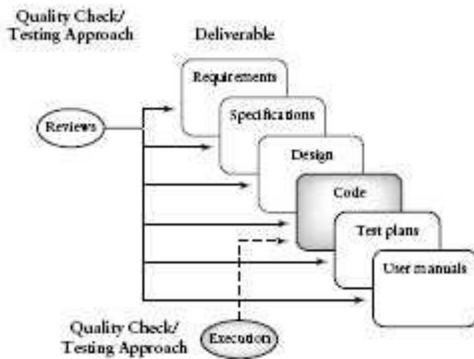


FIG. 10.1
Role of reviews in testing software deliverables.

Other review goals are informational, communicational, and educational, whereby review participants learn about the contents of the developing software artifacts to help them understand the role of their own work and to plan for future stages of development. Reviews often represent project milestones and support the establishment of a baseline for a software artifact. Thus, they also have a role in project management, project monitoring, and control. Review data can also have an influence on test planning. The types and quantity of defects found during review can help test planners select effective classes of tests, and may also have an influence testing goals. In some cases clients/users attend the review meetings and give feedback to the development team, so reviews are also a means for client communication. To summarize, the many benefits of a review program are:

- higher-quality software;
- increased productivity (shorter rework time);
- closer adherence to project schedules (improved process control);
- increased awareness of quality issues; teaching tool for junior staff;
- opportunity to identify reusable software artifacts;
- reduced maintenance costs;
- higher customer satisfaction;
- more effective test planning;
- a more professional attitude on the part of the development staff.

Not all test educators, practitioners, and researchers consider technical reviews to be a testing activity. Some prefer to consider them in a special category called verification testing; others believe they should be associated with software quality assurance activities. The author, as well as many others, for example, Hetzel [2], hold the position that testing activities should cover both validation and verification, and include both static and dynamic analyses. The TMM structure supports this view. If one adheres to this broader view of testing, then the author argues the following:

(i) Reviews as a verification and static analysis technique should be considered a testing activity. **(ii)** Testers should be involved in review activities.

Also, if you consider the following:

(i) a software system is more than the code; it is a set of related artifacts;

(ii) these artifacts may contain defects or problem areas that should be reworked, or removed; and
(iii) quality-related attributes of these artifacts should be evaluated;

then the technical review is one of the most important tools we can use to accomplish these goals. In addition, reviews are the means for testing these artifacts early in the software life cycle. It gives us an early focus on quality issues, helps us to build quality into the system from the beginning, and, allows us to detect and eliminate errors/defects *early* in the software life cycle as close as possible to their point of origin. If we detect defects early in the life cycle, then:

- they are easier to detect;
- they are less costly to repair;
- overall rework time is reduced;
- productivity is improved;
- they have less impact on the customer.

Use of the review as a tool for increasing software quality and developer productivity began in the 1970s. Fagen and Myers wrote pioneering papers that described the review process and its benefits. This chapter will discuss two types of technical reviews, inspections, and walkthroughs.

It will show you how they are run, who should attend, what the typical activities and outputs are, and what are the benefits. Having a review program requires a commitment of organizational time and resources. It is the author's goal to convince you of the benefits of reviews, their important role in the testing process, their cost effectiveness as a quality tool, and why you as a tester should be involved in the review process.

5.6 Types of Reviews

Reviews can be formal or informal. They can be technical or managerial. Managerial reviews usually focus on project management and project status. The role of project status meetings is discussed in Chapter 9. In this chapter we will focus on technical reviews. These are used to:

- verify that a software artifact meets its specification;
- to detect defects; and
- check for compliance to standards.

Readers may not realize that informal technical reviews take place very frequently. For example, each time one software engineer asks another to evaluate a piece of work whether in the office, at lunch, or over a beer, a review takes place. By review it is meant that one or more peers have inspected/evaluated a software artifact. The colleague requesting the review receives feedback about one or more attributes of the reviewed software artifact. Informal reviews are an important way for colleagues to communicate and get peer input with respect to their work. There are two major types of technical reviews—inspections and walkthroughs— which are more formal in nature and occur in a meeting-like setting. Formal reviews require written reports that summarize findings, and in the case of one type of review called an inspection, a statement of responsibility for the results by the reviewers is also required. The two most widely used types of reviews will be described in the next several paragraphs.

I n s p e c t i o n s a s a T y p e o f T e c h n i c a l R e v i e w

Inspections are a type of review that is formal in nature and requires prereview preparation on the part of the review team. Several steps are involved in the inspection process as outlined in Figure 10.2. The responsibility for initiating and carrying through the steps belongs to the inspection leader (or moderator) who is usually a member of the technical staff or the software quality assurance team. Myers suggests that the inspection leader be a member of a group from an unrelated project to preserve objectivity [4]. The inspection leader plans for the inspection, sets the date, invites the participants, distributes the required documents, runs the inspection meeting, appoints a recorder to record results, and monitors the followup period after the review. The key item that the inspection leader prepares is the checklist of items that serves as the agenda for the review. The checklist varies with the software artifact being inspected (examples are provided later in this chapter). It contains items that inspection participants should focus their attention on, check, and evaluate. The inspection participants address each item on the checklist. The recorder records any discrepancies, misunderstandings, errors, and ambiguities; in general, any problems associated with an item. The completed checklist is part of the review summary document. The inspection process begins when inspection preconditions are met as specified in

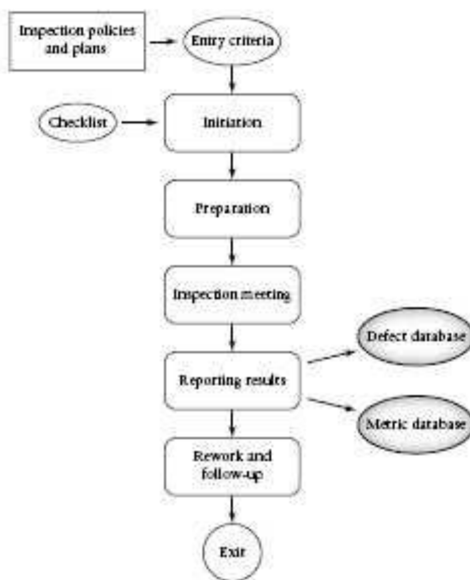


FIG. 10.2
Steps in the inspection process.

the inspection policies, procedures, and plans. The inspection leader announces the inspection meeting and distributes the items to be inspected, the checklist, and any other auxiliary material to the participants usually a day or two before the scheduled meeting. Participants must do their homework and study the items and the checklist. Apersonal preinspection should be performed carefully by each team member [3,5].

Errors, problems, and items for discussion should be noted by each individual for each item on the list. When the actual meeting takes place the document under review is presented by a reader, and is discussed as it read. Attention is paid to issues related to quality, adherence to standards, testability, traceability, and satisfaction of the users/clients requirements. All the items on the checklist are addressed by the group as a whole, and the problems are recorded. Inspection

metrics are also recorded (see Section 10.7). The recorder documents all the findings and the measurements.
When the inspection meeting has been completed (all agenda items covered) the inspectors are usually asked to sign a written document that is sometimes called a summary report that will be described in Section 10.4.6. The inspection process requires a formal follow-up process. Rework sessions should be scheduled as needed and monitored to ensure that all problems identified at the inspection meeting have been addressed and resolved. This is the responsibility of the inspection leader. Only when all problems have been resolved and the item is either reinspected by the group or the moderator (this is specified in the summary report) is the inspection process completed.

WalkthroughsasaTypeofTechnicalReview

Walkthroughs are a type of technical review where the producer of the reviewed material serves as the review leader and actually guides the progression of the review [6]. Walkthroughs have traditionally been applied to design and code. In the case of detailed design or code walkthroughs, test inputs may be selected and review participants then literally walk through the design or code with the set of inputs in a line-by-line manner. The reader can compare this process to a manual execution of the code. The whole group "plays computer" to step through an execution lead by a reader or presenter. This is a good opportunity to "pretest" the design or code. If the presenter gives a skilled presentation of the material, the walkthrough participants are able to build a comprehensive mental (internal) model of the detailed design or code and are able to both evaluate its quality and detect defects. Walkthroughs may be used for material other than code, for example, data descriptions, reference manuals, or even specifications [6].

Some researchers and practitioners believe walkthroughs are efficient because the preparer leads the meeting and is very familiar with the item under review. Because of these conditions a larger amount of material can be processed by the group. However, many of the steps that are mandatory for an inspection are not mandatory for a walkthrough. Comparing inspections and walkthroughs one can eliminate the checklist and the preparation step (this may prove to be a disadvantage to the review team) for the walkthrough. In addition, for the walkthrough there usually no mandatory requirement for a formal review report and a defect list. There is also no formal requirement for a follow-up step. In some cases the walkthrough is used as a preinspection tool to familiarize the team with the code or any other item to be reviewed.

There are other types of technical reviews, for example, the roundrobin review where there is a cycling through the review team members so that everyone gets to participate in an equal manner. For example, in some forms of the round-robin review everyone would have the opportunity to play the role of leader. In another instance, every reviewer in a code walkthrough would lead the group in inspecting a specific line or a section of the code [6]. In this way inexperienced or more reluctant reviewers have a chance to learn more about the review process. In subsequent sections of this chapter the general term review will be used in the main to represent the inspection process, which is the review type most formal in nature. Where specific details are relevant for other types of reviews, such as round-robin or walkthroughs, these will be mentioned in the discussion.

5.7 Components of review plans

Reviews are development and maintenance activities that require time and resources. They should be planned so that there is a place for them in the project schedule. An organization should develop a review plan template that can be applied to all software projects. The template should specify the following items for inclusion in the review plan.
• review goals;
• items being reviewed;
• preconditions for the review;
• roles, team size, participants;
• training requirements;
• review steps;
checklists and other related documents to be disturbed to participants;
• time requirements;
• the nature of the review log and summary report;
• rework and follow-up.
We will now explore each of these items in more detail.

R e v i e w G o a l s
As in the test plan or any other type of plan, the review planner should specify the goals to be accomplished by the review. Some general review goals have been stated in Section 9.0 and include (i) identification of problem components or components in the software artifact that need improvement, (ii) identification of specific errors or defects in the software artifact, (iii) ensuring that the artifact conforms to organizational standards, and (iv) communication to the staff about the nature of the product being developed. Additional goals might be to establish traceability with other project documents, and familiarization with the item being reviewed. Goals for inspections and walkthroughs are usually different; those of walkthroughs are more limited in scope and are usually confined to identification of defects.

P r e c o n d i t i o n s a n d I t e m s t o B e R e v i e w e d
Given the principal goals of a technical review—early defect detection, identification of problem areas, and familiarization with software artifacts— many software items are candidates for review. In many organizations the items selected for review include:
• requirements documents;
• design documents;
• code;
• test plans (for the multiple levels);
• user manuals;training manuals;
• standards documents.
Note that many of these items represent a deliverable of a major life cycle phase. In fact, many represent project milestones and the review serves as a progress marker for project progress. Before each of these items are reviewed certain preconditions usually have to be met. For

example, before a code review is held, the code may have to undergo a successful compile. The preconditions need to be described in the review policy statement and specified in the review plan for an item. General preconditions for a review are:

(i) the review of an item(s) is a required activity in the project plan. (Unplanned reviews are also possible at the request of management, SQA or software engineers. Review policy statements should include the conditions for holding an unplanned review.)
(ii) a statement of objectives for the review has been developed;
(iii) the individuals responsible for developing the reviewed item indicate readiness for the review;
(iv) the review leader believes that the item to be reviewed is sufficiently complete for the review to be useful [8].

The review planner must also keep in mind that a given item to be reviewed may be too large and complex for a single review meeting. The smart planner partitions the review item into components that are of a size and complexity that allows them to be reviewed in 1-2 hours. This is the time range in which most reviewers have maximum effectiveness. For example, the design document for a procedure-oriented system may be reviewed in parts that encompass:
(i) the overall architectural design;
(ii) data items and module interface design;
(iii) component design.
If the architectural design is complex and/or the number of components is large, then multiple design review sessions should be scheduled for each. The project plan should have time allocated for this.

Roles, Participants, Team Size, and Time Requirements

Two major roles that need filling for a successful review are (i) a leader or moderator, and (ii) a recorder. These are shown in Figure 10.3. Some of the responsibilities of the moderator have been described. These include planning the reviews, managing the review meeting, and issuing the review report. Because of these responsibilities the moderator plays an important role; the success of the review depends on the experience and expertise of the moderator. Reviewing a software item is a tedious process and requires great attention to details. The moderator needs to be sure that all are prepared for the review and that the review meeting stays on track. Reviewers often tire and become less effective at detecting errors if the review time period is too long and the item is too complex for a single review meeting. The moderator/planner must ensure that a time period is selected that is appropriate for the size and complexity of the item under review. There is no set value for a review time period, but a rule of thumb advises that a review session should not be longer than 2 hours [3]. Review sessions can be scheduled over 2-hour time periods separated by breaks. The time allocated for a review should be adequate enough to ensure that the material under review can be adequately covered.

The review recorder has the responsibility for documenting defects, and recording review findings and recommendations, Other roles may include a reader who reads or presents the item under review. Readers are usually the authors or preparers of the item under review. The author(

s) is responsible for per forming any rework on the reviewed item. In a walkthrough type of review, the author may serve as the moderator, but this is not true for an inspection. All reviewers should be trained in the review process. The size of the review team will vary depending type, size, and complexity of the item under review. Again, as with time, there is no fixed size for a review team. In most cases a size between 3 and 7 is a rule of thumb, but that depends on the items under review and the experience level of the review team. Of special importance is the experience of the review moderator who is responsible for ensuring the material is covered, the review meeting stays on track, and review outputs are produced. The minimal team size of 3 ensures that the review will be public [6].

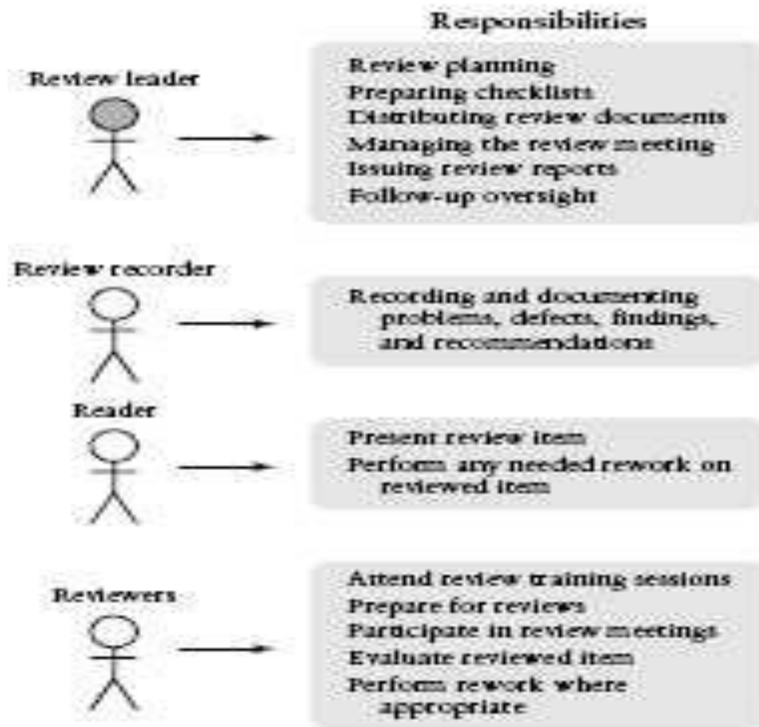


FIG. 10.3
Review roles.

Organizational policies guide selection of review team members. Membership may vary with the type of review. As shown in Figure 10.4 the review team can consist of software quality assurance staff members, testers, and developers (analysts, designers, programmers). In some cases the size of the review team will be increased to include a specialist in a particular area related to the reviewed item; in other cases "outsiders" may be invited to a review to get a more unbiased evaluation of the item. These outside members may include users/clients. Users/clients should certainly be present at requirements, user manual, and acceptance test plan reviews. Some recommend that users also be present at design and even code reviews. Organizational policy should refer to this issue, keeping in mind the limited technical knowledge of most users/clients.

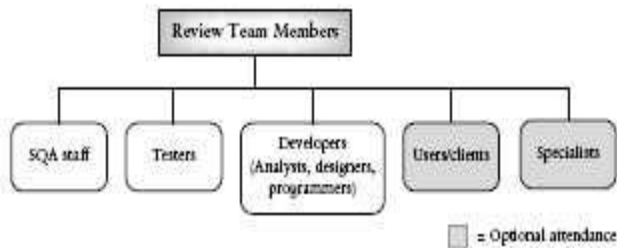


FIG. 10.4
Review team membership constituency.

In many cases it is wise to invite review team members from groups that were involved in the preceding and succeeding phases of the life cycle document being reviewed. These participants could be considered to be outsiders. For example, if a design document is under review, it would be useful to invite a requirements team representative and a coding team member to be a review participant since correctness, consistency, implementability, and traceability are important issues for this review. In addition, these attendees can offer insights and perspectives that differ from the group members that were involved in preparing the current document under review. It is the author's option that testers take part in all major milestone reviews to ensure:

- effective test planning;
- traceability between tests, requirements, design and code elements;
- discussion, and support of testability issues;
- support for software product quality issues;
- the collection and storage of review defect data;
- support for adequate testing of "trouble-prone" areas.

Testers need to especially interact with designers on the issue of testability. A more testable design is the goal. For example, in an object-oriented system a tester may request during a design review that additional methods be included in a class to display its state variables. In this case and others, it may appear on the surface that this type of design is more expensive to develop and implement. However, consider that in the long run if the software is more testable there will be two major positive effects:

(i) the testing effort is likely to be decreased, thus lowering expenses, and
(ii) the software is likely to be of higher quality, thus increasing customer satisfaction.

Review Procedures

For each type of review that an organization wishes to implement, there should be a set of standardized steps that define the given review procedure. For example, the steps for an inspection are shown in Figure 10.2. These are initiation, preparation, inspection meeting, reporting results, and rework and follow-up. For each step in the procedure the activities and tasks for all the reviewer participants should be defined. The review plan should refer to the standardized procedures where applicable.

Review Training

Review participants need training to be effective. Responsibility for reviewer training classes usually belongs to the internal technical training staff. Alternatively, an organization may decide to send its review trainees to external training courses run by commercial institutions. Review participants, and especially those who will be review leaders, need the training. Test specialists

should also receive review training. Suggested topics for a training program are shown in Figure 10.5 and described below. Some of the topics can be covered very briefly since it is assumed that the reviewers (expect for possible users/clients) are all technically proficient.

1 . Review of Process Concepts.
Reviewers should understand basic process concepts, the value of process improvement, and the role of reviews as a product and process improvement tool.

Review Training Topics

Topic1.
Basic concepts
Topic 2.
Review of quality issues
Topic 3.
Review of standards
Topic 4.
Understanding the material to be reviewed
Topic 5.
Defect and problem types
Topic 6.
Communication and meeting management skills
Topic 7.
Review documentation and record keeping
Topic 8.
Special instructions
Topic 9.
Practice review sessions

FIG. 10.5
Topics for review training sessions.

2 . Review of Quality Issues.
Reviewers should be made familiar with quality attributes such as correctness, testability, maintainability, usability, security, portability, and so on, and how can these be evaluated in a review.

3 . Review of Organizational Standards for Software A r t i f a c t s .
Reviewers should be familiar with organizational standards for software artifacts. For example, what items must be included in a software document; what is the correct order and degree of coverage of topics expected; what types of notations are permitted. Good sources for this material are IEEE standards and guides [1,9,10].

4 . Understanding the Material to Be Reviewed.
Concepts of understanding and how to build mental models during comprehension of code and software-related documents should be covered. A critical issue is how fast a reviewed document should be read/checked by an individual and by the group as a whole. This applies to requirements,design, test plans and other documents, as well as source code. A rate of 5-10 pages/hour or 125-150 LOC/hour for a review group has been quoted as favorable [7]. Reading rates that are too slow will make review meetings ineffective with respect to the number of defects found per unit time. Readings that are too fast will allow defects and problems to go undetected.

5 . Defect and Problem Types.

Review trainees need to become aware of the most frequently occurring types of problems or errors that are likely to occur during development. They need to be aware what their causes are, how they are transformed into defects, and where they are likely to show up in the individual deliverables. The trainees should become familiar with the defect type categories, severity levels, and numbers and types of defects found in past deliverables of similar systems. Review trainees should also be made aware of certain indicators or clues that a certain type of defect or problem has occurred [3]. The definitions of defects categories, and maintenance of a defect data base are the responsibilities of the testers and SQA staff.

6 . Communication and Meeting Management S k i l l s .

These topics are especially important for review leaders. It is their responsibility to communicate with the review team, the preparers of the reviewed document, management, and in some cases clients/user group members. Review leaders need to have strong oral and written communication skills and also learn how to conduct a review meeting. During a review meeting there are interactions and expression of opinion from a group of technically qualified people who often want to be heard. The review leader must ensure that all are prepared, that the meeting stays on track, that all get a chance to express their opinions, that the proper page/code document checking rate is achieved, and that results are recorded. Review leaders also must trained so that they can ensure that authors of the document or artifact being reviewed are not under the impression that they themselves are being evaluated. The review leader needs to uphold the organizational view that the purpose of the review is to support the authors in improving the quality of the item they have developed. Policy statements to this effect need to be written and explained to review trainees, especially those who will be review leaders.

Skills in conflict resolution are very useful, since very often reviewers will have strong opinions and arguments can dominate a review session unless there is intervention by the leader. There are also issues of power and control over deliverables and aspects of deliverables and other hidden agenda that surface during a review meeting that must be handled by the review leader. In this case people and management skills are necessary, and sometime these cannot be taught. They come through experience.

7 . Review Documentation and Record Keeping.

Review leaders need to learn how to prepare checklists, agendas, and logs for review meetings. Examples will be provided for some of these documents later in this chapter. Other examples can be found in Freedman and Weinberg [6], Myers [11], and Kit [12]. Checklists for inspections should be appropriate for the item being inspected. Checklists in general should focus on the following issues:

- most frequent errors;
- completeness of the document;
- correctness of the document;
- adherence to standards.

8 . Special Instructions.

During review training there may be some topics that need to be covered with the review participants. For example, there may be interfaces with hardware that involve the reviewed item,

and reviewers may need some additional background discussion to be able to evaluate those interfaces.

9 . Practice Review Sessions.
Review trainees should participate in practice review sessions. There are very instructive and essential. One option is for instructors to use existing documents that have been reviewed in the past and have the trainees do a practice review of these documents. Results can be compared to those of experienced reviewers, and useful lessons can be learned from problems identified by the trainees and those that were not. Instructors can discuss so-called "false positives" which are not true defects but are identified as such. Trainees can also attend review sessions with experienced reviewers as observers, to learn review lessons.

In general, training material for review trainees should have adequate examples, graphics, and homework exercises. Instructors should be provided with the media equipment needed to properly carry out instruction. Material can be of the self-paced type, or for group course work.

Review Checklists
Inspections formally require the use of a checklist of items that serves as the focal point for review examinations and discussions on both the individual and group levels. As a precondition for checklist development an organization should identify the typical types of defects made in past projects, develop a classification scheme for those defects, and decide on impact or severity categories for the defects. If no such defect data is available, staff members need to search the literature, industrial reports, or the organizational archives to retrieve this type of information.
Checklists are very important for inspectors. They provide structure and an agenda for the review meeting. They guide the review activities, identify focus areas for discussion and evaluation, ensure all relevant items are covered, and help to frame review record keeping and measurement. Reviews are really a two-step process: (i) reviews by individuals, and (ii) reviews by the group. The checklist plays its important role in both steps. The first step involves the individual reviewer and the review material. Prior to the review meeting each individual must be provided with the materials to review and the checklist of items. It is his responsibility to do his homework and individually inspect that document using the checklist as a guide, and to document any problems he encounters.

When they attend the group meeting which is the second review step, each reviewer should bring his or her individual list of defect/problems, and as each item on the checklist is discussed they should comment. Finally, the reviewers need to come to a consensus on what needs to be fixed and what remains unchanged. Each item that undergoes a review requires a different checklist that addresses the special issues associated with quality evaluation for that item. However each checklist should have components similar to those shown in Table 10.1. The first column lists all the defect types or potential problem areas that may occur in the item under review. Sources for these defect types are usually data from past projects. Abbreviations for detect/ problem types can be developed to simplify the checklist forms. Status refers to coverage during the review meeting—has the item been discussed? If so, a check mark is placed in the column. Major or minor are the two severity or impact levels shown here. Each organization needs to decide on the severity levels that work for them. Using this simple severity scale, a defect or problem that is

classified as major has a large impact on product quality; it can cause failure or deviation from specification. A minor problem has a small impact on these; in general, it would affect a nonfunctional aspect of the software. The letters M, I, and S indicate whether a checklist item is missing (M), incorrect (I), or superfluous (S).

In this section we will look at several sample checklists. These are shown in Tables 10.2-10.5. One example is the general checklist shown in Table 10.2, which is applicable to almost all software documents. The checklist is used is to ensure that all documents are complete, correct, consistent, clear, and concise. Table 10.2 only shows the problem/defect types component (column) for simplicity's sake. All the components as found in Table 10.1 should be present on each checklist form. That also holds true for the checklists illustrated in Tables 10.3-10.5. The recorder is responsible for completing the group copy of the checklist form during the review meeting (as opposed to the individual checklist form completed during review preparation by each individual reviewer). The recorder should also keep track of each defect and where in the document it occurs (line, page, etc.). The group checklist can appear on a wallboard so that all can see what has been entered. Each individual should bring to the review meeting his or her own version of the checklist completed prior to the review meeting.In addition to using the widely applicable problem/defect types shown in Table 10.2 each item undergoing review has

Problem/defect type	Status	Major	Minor	M	I	S

TABLE 10.1
Example components for an inspection checklist.

specific attributes that should be addressed on a checklist form. Some examples will be given in the following pages of checklist items appropriate for reviewing different types of software artifacts.

Requirements Reviews

In addition to covering the items on the general document checklist as shown in Table 10.2, the following items should be included in the checklist for a requirements review.

• completeness (have all functional and quality requirements described in the problem statement been included?);
• correctness (do the requirements reflect the user's needs? are they stated without error?);
• consistency (do any requirements contradict each other?);
• clarity (it is very important to identify and clarify any ambiguous requirements);
• relevance (is the requirement pertinent to the problem area? Requirements should not be superfluous);
• redundancy (a requirement may be repeated; if it is a duplicate it should be combined with an equivalent one);

• testability (can each requirement be covered successfully with one or more test cases? can tests determine if the requirement has been satisfied?);feasibility (are requirements implementable given the conditions underwhich the project will progress?).

Users/clients or their representatives should be present at a requirements review to ensure that the requirements truly reflect their needs, and that the requirements are expressed clearly and completely. It is also very important for testers to be present at the requirements review. One of their major responsibilities it to ensure that the requirements are testable. Very often the master or early versions of the system and acceptance test plans are included in the requirements review. Here the reviewers/testers can use a traceability matrix to ensure that each requirement can be covered by one or more tests. If requirements are not clear, proposing test cases can be of help in focusing attention on these areas, quantifying imprecise requirements, and providing general information to help resolve problems.
Although not on the list above, requirements reviews should also ensure that the requirements are free of design detail. Requirements focus on what the system should do, not on how to implement it.

Design Reviews

Designs are often reviewed in one or more stages. It is useful to review the high level architectural design at first and later review the detailed design. At each level of design it is important to check that the design is consistent with the requirements and that it covers all the requirements. Again the general checklist is applicable with respect to clarity, completeness, correctness and so on. Some specific items that should be checked for at a design review are:

• a description of the design technique used;
• an explanation of the design notation used;
• evaluation of design alternatives (it is important to establish that design alternatives have been evaluated, and to determine why this particular approach was selected);
• quality of the high-level architectural model (all modules and their relationships should be defined; this includes newly developed modules, revised modules, COTS components, and any other reused modules; module coupling and cohesion should be evaluated.);
• description of module interfaces;
• quality of the user interface;
• quality of the user help facilities;
• identification of execution criteria and operational sequences;
• clear description of interfaces between this system and other software and hardware systems;
• coverage of all functional requirements by design elements; coverage of all quality requirements, for example, ease of use, portability, maintainability, security, readability, adaptability, performance requirements (storage, response time) by design elements;
• reusability of design components;
• testability (how will the modules, and their interfaces be tested? How will they be integrated and tested as a complete system?).

For reviewing detailed design the following focus areas should also be revisited:

• encapsulation, information hiding and inheritance;
• module cohesion and coupling;
• quality of module interface description;
• module reuse.

Both levels of design reviews should cover testability issues as described above. In addition, measures that are now available such as module complexity, which gives an indication of testing effort, can be used to estimate the extent of the testing effort. Reviewers should also check traceability from tests to design elements and to requirements. Some organizations may re-examine system and integration test plans in the context of the design elements under review. Preliminary unit test plans can also be examined along with the design documents to ensure traceability, consistency, and complete coverage. Other issues to be discussed include language issues and the appropriateness of the proposed language to implement the design.

Code Reviews

Code reviews are useful tools for detecting defects and for evaluating code quality. Some organizations require a clean compile as a precondition for a code review. The argument is that it is more effective to use an automated tool to identify syntax errors than to use human experts to perform this task. Other organizations will argue that a clean compile makes rediligent in checking for defects since they will assume the compiler has detected many of them.

Code review checklists can have both general and language-specific components. The general code review checklist can be used to review code written in any programming language. There are common quality features that should be checked no matter what implementation language is selected. Table 10.3 shows a list of items that should be included in a general code checklist. The general checklist is followed by a sample checklist that can be used for a code review for programs written in the C programming language. The problem/defect types are shown in Table
10.4. When developing your own checklist documents be sure to include the other columns as shown in Table 10.1. The reader should note that since the languagespecific checklist addresses programming-language-specific issues, a different checklist is required for each language used in the organization.

Test Plan Reviews

Test plans are also items that can be reviewed. Some organizations will review them along with other related documents. For example, a master test plan and an acceptance test plan could be reviewed with the requirements document, the integration and system test plans reviewed with the design documents, and unit test plans reviewed with detailed design documents [2]. Other organizations, for example, those that use the Extended/ Modified V-model, may have separate review meetings for each of the test plans. In Chapter 7 the components of a test plan were discussed, and the review should insure that all these components are present and that they are correct, clear, and complete. The general document checklist can be applied to test plans, and a more specific checklist can be developed for test-specific issues. An example test plan checklist is shown in Table 10.4. The test plan checklist is applicable to all levels of test plans.

Other testing products such as test design specifications, test procedures, and test cases can also be reviewed. These reviews can be held in conjunction with reviews of other test-related items or other software items.

5.8 Reporting review results.

Several information-rich items result from technical reviews. These items are listed below. The items can be bundled together in a single report or distributed over several distinct reports. Review polices should indicate the formats of the reports required. The review reports should contain the following information.

1. *For inspections*—the group checklist with all items covered and comments relating to each item.

2. *For inspections*—a status, or summary, report (described below) signed by all participants.

3. A list of defects found, and classified by type and frequency. Each defect should be crossreferenced to the line, pages, or figure in the reviewed document where it occurs.

4. Review metric data (see Section 10.7 for a discussion).

The inspection report on the reviewed item is a document signed by all the reviewers. It may contain a summary of defects and problems found and a list of review attendees, and some review measures such as the time period for the review and the total number of major/minor defects.

The reviewers are responsible for the quality of the information in the written report [6]. There are several status options available to the review participants on this report. These are:

1. *Accept:* The reviewed item is accepted in its present form or with minor rework required that does not need further verification.

2. *Conditional accept:* The reviewed item needs rework and will be accepted after the moderator has checked and verified the rework.

3. *Reinspect:* Considerable rework must be done to the reviewed item.

The inspection needs to be repeated when the rework is done. Before signing their name to such a inspection report reviewers need to be sure that all checklist items have been addressed, all defects recorded, and all quality issues discussed. This is important for several reasons. Very often when a document has passed an inspection it is viewed as a baseline item for configuration management, and any changes from this baseline item need approval from the configuration management board. In addition, the successful passing of a review usually indicates a project milestone has been passed, a certain level of quality has been achieved, and the project has made progress toward meeting its objectives. A milestone meeting is usually held, and clients are notified of the completion of the milestone.

If the software item is given a conditional accept or a reinspect, a follow-up period occurs where the authors must address all the items on the problem/defect list. The moderator reviews the rework in the case of a conditional accept. Another inspection meeting is required to reverify the items in the case of a "reinspect" decision. For an inspection type of review, one completeness or exit criterion requires that all identified problems be resolved. Other criteria may be required by the organization. In addition to the summary report, other outputs of an inspection include a defect report and an inspection report. These reports are vital for collecting and organizing review measurement data. The defect report contains a description of the defects, the defect type, severity level, and the location of each defect. On the report the defects can be organized so that their type and occurrence rate is easy to determine. IEEE standards suggest that the inspection report contain vital data such as [8]:

(i) number of participants in the review;
(ii) the duration of the meeting;
(iii) size of the item being reviewed (usually LOC or number of pages);
(iv) total preparation time for the inspection team;
(v) status of the reviewed item;
(vi) estimate of rework effort and the estimated date for completion of the rework.
This data will help an organization to evaluate the effectiveness of the review process and to make improvements.The IEEE has recommendations for defect classes [8]. The classes are based on the reviewed software items' conformance to:

• standards;
• capability;
• procedures;
• interface;
• description.

A defect class may describe an item as missing, incorrect, or superfluous as shown in Table 10.1. Other defect classes could describe an item as ambiguous or inconsistent [8]. Defects should also be ranked in severity, for example:
(i) major (these would cause the software to fail or deviate from its specification); **(ii)** minor (affects nonfunctional aspects of the software).
A ranking scale for defects can be developed in conjunction with a failure severity scale as described in Section 9.1.4.

A walkthrough review is considered complete when the entire document has been covered or walked through, all defects and suggestions for improvement have been recorded, and the walkthrough report has been completed. The walkthrough report lists all the defects and deficiencies, and contains data such as [8]:

• the walkthrough team members;
• the name of the item being examined;
• the walkthrough objectives;
• list of defects and deficiencies;
• recommendations on how to dispose of, or resolve the deficiencies.

Note that the walkthrough report/completion criteria are not as formal as those for an inspection. There is no requirement for a signed status report, and no required follow-up for resolution of deficiencies, although that could be recommended in the walkthrough report.A final important item to note: The purpose of a review is to evaluate a software artifact, *not* the developer or author of the artifact. Reviews should not be used to evaluate the performance of a software analyst, developer, designer, or tester [3]. This important point should be well established in the review policy. It is essential to adhere to this policy for the review process to work. If authors of software artifacts believe they are being evaluated as individuals, the objective and impartial nature of the review will change, and its effectiveness in revealing problems will be minimized .

Unit V

Part-A Questions

1. What is project monitoring?
2. List the benefits of review program.
3. List the function of conducting status meeting.
4. Define Monitoring.
5. List the four major activities associated with configuration management.

Part-B Questions

1. Write a summary about the following types of reviews.
2. Write a note on five stop test criteria based on quantitative approach.
3. What is software configuration management ? Explain the four major activities associated with configuration management.
4. Explain the functions of monitoring and controlling management.
5. Give a note on: Components of review plans & Reporting review results.

N.P.R. COLLEGE OF ENGINEERING & TECHNOLOGY

N.P.R Nagar, Natham, Dindigul – 624 401, Tamil Nadu

Phone No. : 04544 – 305500, 501, Fax No. : 04544 – 305562

Website: www.nprcolleges.org E-Mail: nprgc@nprcolleges.org

AN ISO 9001:2008 Certified Institution

DEPARTMENT OF INFORMATION TECHNOLOGY

QUESTION BANK

Sub. Code/ Name : CS1016 – SOFTWARE TESTING
Year / Sem : IV / VIII

UNIT- I
TESTING BASICS

PART – A (2 MARKS)

1. Define Software Engineering.
2. Define software Testing.
3. List the elements of the engineering disciplines.
4. Differentiate between verification and validation?
5. Define the term Testing.
6. Differentiate between testing and debugging.
7. Define process in the context of software quality.
8. Define the term Debugging or fault localization.
9. List the levels of TMM.
10. List the members of the critical groups in a testing process.
11. Define Error.
12. Define Faults (Defects).
13. Define failures.
14. Distinguish between fault and failure.
15. Define Test Cases.
16. Write short notes on Test, Test Set, and Test Suite.
17. Define Test Oracle.
18. Define Test Bed.
19. Define Software Quality.
20. List the Quality Attributes.
21. Define SQA group.
22. Explain the work of SQA group.
23. Define reviews.
24. List the sources of Defects or Origins of defects. Or list the classification of defect.
25. Programmer A and Programmer B are working on a group of interfacing modules. Programmer A tends to be a poor communicator and does not get along well with

    Programmer B. Due to this situation, what types of defects are likely to surface in these interfacing modules?

PART –B (16 MARKS)

1. Explain The Role of process in Software quality. (16)
2. Explain Testing as a Process. (16)
3. Overview of the Testing Maturity Model (TMM) & the test related activities that should be done for V-model architecture. (16)
4. Explain Software Testing Principles. (16)
5. Explain Origins of defects. (16)
6. Explain Defect Classes, Defect Repository, and Test Design. (16)
7. Explain Defect Examples: The Coin Problem. (16)
8. Explain the tester's role in a Software Development Organization. (16)
9. Explain Developer / Tester support for developing a defect repository. (16)

UNIT- II
TESTCASE DESIGN

PART – A (2 MARKS)

1. Define Smart Tester.
2. Compare black box and white box testing.
3. Draw the tester's view of black box and white box testing.
4. Write short notes on Random testing and Equivalence class portioning.
5. List the Knowledge Sources & Methods of black box and white box testing.
6. Define State.
7. Define Finite-State machine.
8. Define Error Guessing.
9. Define COTS Components.
10. Define usage profiles and Certification.
11. Write the application scope of adequacy criteria?
12. What are the factors affecting less than 100% degree of coverage?
13. What are the basic primes for all structured program.
14. Define path.
15. Write the formula for cyclomatic complexity?
16. List the various iterations of Loop testing.
17. Define test set.
18. What are the errors uncovered by black box testing?

PART –B (16 MARKS)

1. Explain in detail about the Smart Tester. (16)
2. Explain in Test case design strategies. (16)
3. Explain the Types of black box testing. (16)

4. Explain Other Black box test design Approaches. (16) 5. Explain Black Box Testing and COTS (Commercial Off-the-shelf) components.

(16)

6. Explain Types of white box testing. (16)
7. Explain Additional white box test design approaches. (16)
8. Evaluating Test adequacy Criteria. (16)

UNIT-III

LEVELS OF TESTING

PART – A (2 MARKS)

1. List the levels of Testing or Phases of testing.
2. Define Unit Test and characterized the unit test.
3. List the phases of unit test planning.
4. List the work of test planner.
5. Define integration Test.
6. Define System test.
7. Define Alpha and Beta Test.
8. What are the approaches are used to develop the software?
9. List the issues of class testing.
10. Define test Harness.
11. Define Test incident report.
12. Define Summary report.
13. Goals of Integration test.
14. What are the Integration strategies?
15. What is Cluster?
16. List the different types of system testing.
17. Define load generator and Load.
18. Define functional Testing.
19. What are the two major requirements in the Performance testing?
20. Define stress Testing.
21. Define Breaking the System.
22. What are the steps for top down integration?
23. What is meant by regression testing?

PART –B (16 MARKS)

1. Explain the Need for levels testing. (16)
2. Explain Levels of testing and software development paradigm. (16)
3. Explain Unit Test. (16)
4. Explain Unit Test Planning. (16)
5. Explain the class as testable unit. (16)
6. Explain in detail about the Test harness. (16)
7. Explain Integration Test. (16)
8. Explain System test: Different Types. (16)

UNIT- IV
TEST MANAGEMENT

PART – A (2 MARKS)

1. Write the different types of goals.
2. Define Goal and Policy.
3. Define Plan.
4. Define Milestones.
5. List the Test plan components.
6. Draw a hierarchy of test plans.
7. Define a Work Breakdown Structure.(WBS)
8. Write the approaches to test cost Estimation?
9. Write short notes on Cost driver.
10. Write the WBS elements for testing.
11. What is the function of Test Item Transmittal Report or Locating Test Items?
12. What is the information present in the Test Item Transmittal Report or Locating Test Items?
13. Define Test incident Report.
14. Define Test Log.
15. What are the three critical groups in testing planning and test plan policy?
16. Define Procedure.
17. What are the skills needed by a test specialist?
18. Write the test term hierarchy?

PART –B (16 MARKS)

1. Explain Testing and Debugging goals and Policy. (16)
2. Explain Test planning. (16)
3. Explain Test Plan Components. (16)
4. Explain Test Plan Attachments. (16)
5. Explain Reporting Test Results. (16)
6. Explain the role of the 3 critical groups. (16)

UNIT- V
CONTROLLING AND MONITORING

PART – A (2 MARKS)

1. Define Project monitoring or tracking.
2. Define Project Controlling.
3. Define Milestone.
4. Define SCM (Software Configuration management).
5. Define Base line.
6. Differentiate version control and change control.
7. What is testing?
8. Define Review.
9. What are the goals of Reviewers?
10. What are the benefits of a Review program?
11. What are the various types of Reviews?
12. What is Inspections?
13. What is WalkThroughs?
14. List out the members present in the Review Team.
15. List the components of review plans.

PART –B (16 MARKS)

1. Explain Measurements and milestones for monitoring and controlling. (16)
2. Explain Criteria for test completion. (16)
3. Explain Software configuration management. (16)
4. Explain in detail about the Types of reviews. (16)
5. Explain in Components of review plans. (16)

B.E/B.Tech. DEGREE EXAMINATIONS, APRIL/MAY 2011
EIGTH SEMESTER
CS1016 SOFTWARE TESTING
(REGULATION 2007)

Time: Three hours ***Maximum:100 marks***

Answer ALL questions.

PART A-(10*2=20 MARKS)

1.Define Validation.
2. What is Data Defect?
3. What is Random Testing?
4. Define Test Data Set.
5. What is Integration Testing?
6. What is Alpha Testing?
7. List any four components of Test Plan.
8. What is Test Log?
9. List four items for Controlling and Monitoring the test efforts for a Project.
10. Define Review.

PART B-(5*16=80 Marks)

11. Explain "Testing as a Process" with suitable example.
or
12. Briefly discuss the testers role in a Software Development Organization.

13. Explain Black Box Testing and Commercial Off the Shelf (COTS)Components.
or
14. How to evaluate a Test Adequacy Criteria of an application.

15. Describe "The Class as a Testable Unit" in detail.
or
16. Explain the types of System Tests in detail.

17. Discuss in detail the Test Plan Components.
or
18. Explain the skills needed by a Test Specialist.

19. Discuss the Criteria for Test Completion.
or
20. Explain the various types of Reviews.