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.

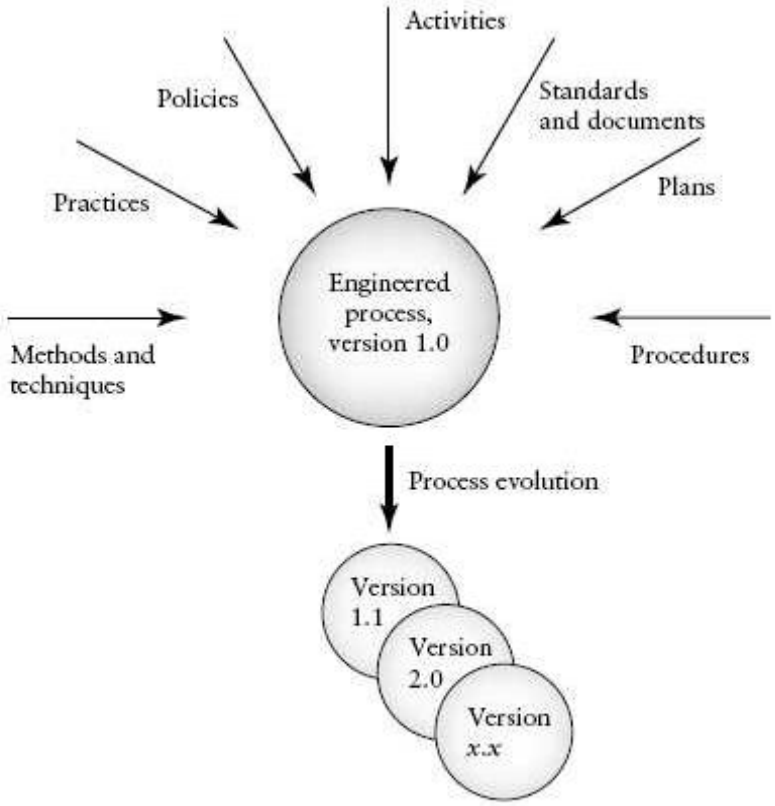

FIG. 1.2

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].

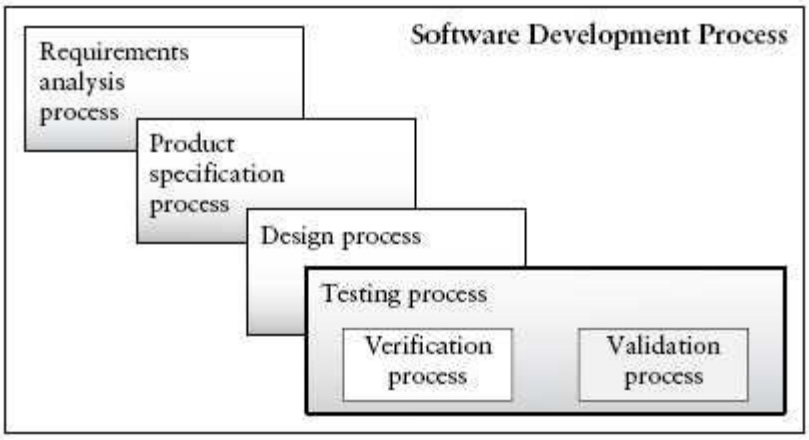

FIG. 1.3
Example processes embedded in the software development process.

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]:

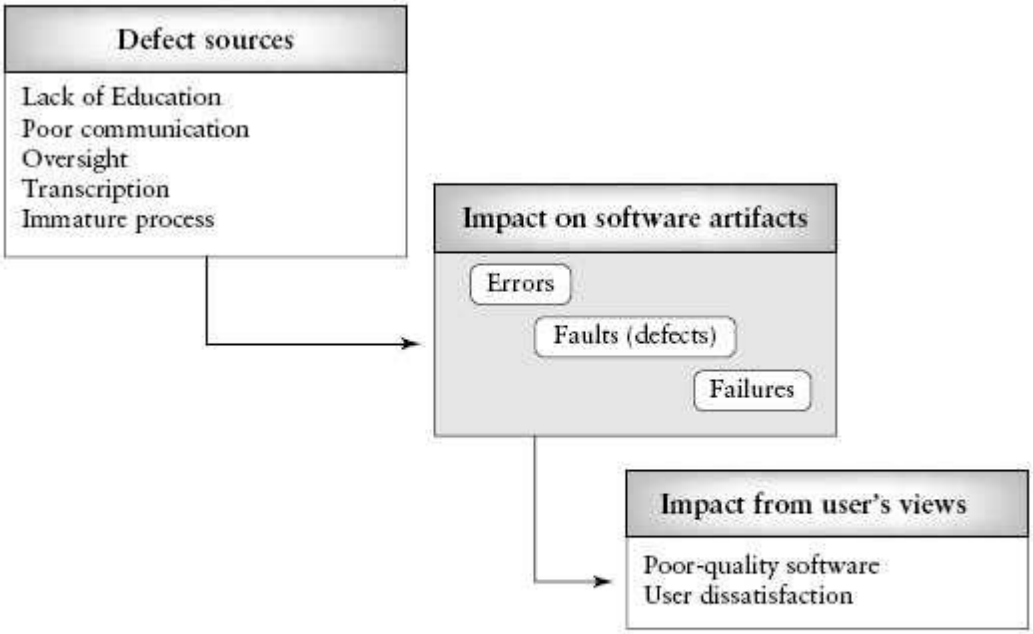

FIG. 3.1

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.

C o d i n g D e f e c t s

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 . I n i t i a l i z a t i o n 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.

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

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.

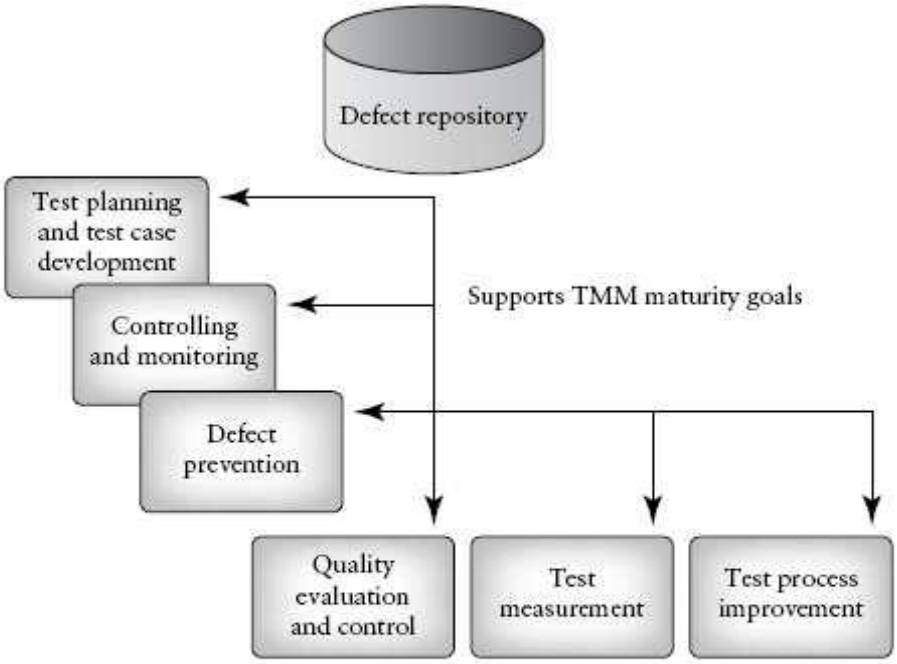

FIG. 3.6

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.