CHAPTER 1

Basic Concepts and Preliminaries

> Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics, i.e., it always increases.
> — *Norman Ralph Augustine*

1.1 QUALITY REVOLUTION

People seek quality in every man-made artifact. Certainly, the concept of quality did not originate with software systems. Rather, the quality concept is likely to be as old as human endeavor to mass produce artifacts and objects of large size. In the past couple of decades a quality revolution, has been spreading fast throughout the world with the explosion of the Internet. Global competition, outsourcing, off-shoring, and increasing customer expectations have brought the concept of quality to the forefront. Developing quality products on tighter schedules is critical for a company to be successful in the new global economy. Traditionally, efforts to improve quality have centered around the end of the product development cycle by emphasizing the detection and correction of defects. On the contrary, the new approach to enhancing quality encompasses all phases of a product development process—from a requirements analysis to the final delivery of the product to the customer. Every step in the development process must be performed to the highest possible standard. An effective quality process must focus on [1]:

- Paying much attention to customer's requirements
- Making efforts to continuously improve quality
- Integrating measurement processes with product design and development
- Pushing the quality concept down to the lowest level of the organization
- Developing a system-level perspective with an emphasis on methodology and process
- Eliminating waste through continuous improvement

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy

A quality movement started in Japan during the 1940s and the 1950s by William Edwards Deming, Joseph M. Juran, and Kaoru Ishikawa. In circa 1947, W. Edwards Deming "visited India as well, then continued on to Japan, where he had been asked to join a statistical mission responsible for planning the 1951 Japanese census" [2], p. 8. During his said visit to Japan, Deming invited statisticians for a dinner meeting and told them how important they were and what they could do for Japan [3]. In March 1950, he returned to Japan at the invitation of Managing Director Kenichi Koyanagi of the Union of Japanese Scientists and Engineers (JUSE) to teach a course to Japanese researchers, workers, executives, and engineers on statistical quality control (SQC) methods. Statistical quality control is a discipline based on measurements and statistics. Decisions are made and plans are developed based on the collection and evaluation of actual data in the form of metrics, rather than intuition and experience. The SQC methods use seven basic quality management tools: Pareto analysis, cause-and-effect diagram, flow chart, trend chart, histogram, scatter diagram, and control chart [2].

In July 1950, Deming gave an eight-day seminar based on the Shewhart methods of statistical quality control [4, 5] for Japanese engineers and executives. He introduced the *plan–do–check–act* (PDCA) cycle in the seminar, which he called the Shewhart cycle (Figure 1.1). The Shewhart cycle illustrates the following activity sequence: setting goals, assigning them to measurable milestones, and assessing the progress against those milestones. Deming's 1950 lecture notes formed the basis for a series of seminars on SQC methods sponsored by the JUSE and provided the criteria for Japan's famed Deming Prize. Deming's work has stimulated several different kinds of industries, such as those for radios, transistors, cameras, binoculars, sewing machines, and automobiles.

Between circa 1950 and circa 1970, automobile industries in Japan, in particular Toyota Motor Corporation, came up with an innovative principle to compress the time period from customer order to banking payment, known as the "lean principle." The objective was to minimize the consumption of resources that added no value to a product. The lean principle has been defined by the National Institute of Standards and Technology (NIST) Manufacturing Extension Partnership program [61] as "a systematic approach to identifying and eliminating waste through continuous improvement, flowing the product at the pull of the customer in pursuit of perfection," p.1. It is commonly believed that lean principles were started in Japan by Taiichi Ohno of Toyota [7], but Henry Ford

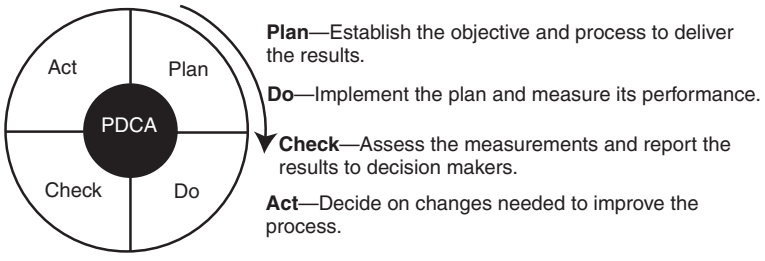

Figure 1.1 Shewhart cycle.

had been using parts of lean as early as circa 1920, as evidenced by the following quote (Henry Ford, 1926) [61], p.1:

> One of the noteworthy accomplishments in keeping the price of Ford products low is the gradual shortening of the production cycle. The longer an article is in the process of manufacture and the more it is moved about, the greater is its ultimate cost.

This concept was popularized in the United States by a Massachusetts Institute of Technology (MIT) study of the movement from mass production toward production, as described in *The Machine That Changed the World*, by James P. Womack, Daniel T. Jones, and Daniel Roos, New York: Rawson and Associates, 1990. Lean thinking continues to spread to every country in the world, and leaders are adapting the principles beyond automobile manufacturing, to logistics and distribution, services, retail, health care, construction, maintenance, and software development [8].

Remark: Walter Andrew Shewhart was an American physicist, engineer, and statistician and is known as the father of statistical quality control. Shewhart worked at Bell Telephone Laboratories from its foundation in 1925 until his retirement in 1956 [9]. His work was summarized in his book *Economic Control of Quality of Manufactured Product*, published by McGraw-Hill in 1931. In 1938, his work came to the attention of physicist W. Edwards Deming, who developed some of Shewhart's methodological proposals in Japan from 1950 onward and named his synthesis the Shewhart cycle.

In 1954, Joseph M. Juran of the United States proposed raising the level of quality management from the manufacturing units to the entire organization. He stressed the importance of systems thinking that begins with product requirement, design, prototype testing, proper equipment operations, and accurate process feedback. Juran's seminar also became a part of the JUSE's educational programs [10]. Juran spurred the move from SQC to TQC (total quality control) in Japan. This included companywide activities and education in quality control (QC), audits, quality circle, and promotion of quality management principles. The term TQC was coined by an American, Armand V. Feigenbaum, in his 1951 book *Quality Control Principles, Practice and Administration*. It was republished in 2004 [11]. By 1968, Kaoru Ishikawa, one of the fathers of TQC in Japan, had outlined, as shown in the following, the key elements of TQC management [12]:

- Quality comes first, not short-term profits.
- The customer comes first, not the producer.
- Decisions are based on facts and data.
- Management is participatory and respectful of all employees.
- Management is driven by cross-functional committees covering product planning, product design, purchasing, manufacturing, sales, marketing, and distribution.

Remark: A quality circle is a volunteer group of workers, usually members of the same department, who meet regularly to discuss the problems and make presentations to management with their ideas to overcome them. Quality circles were started in Japan in 1962 by Kaoru Ishikawa as another method of improving quality. The movement in Japan was coordinated by the JUSE.

One of the innovative TQC methodologies developed in Japan is referred to as the *Ishikawa* or *cause-and-effect* diagram. Kaoru Ishikawa found from statistical data that dispersion in product quality came from four common causes, namely *materials*, *machines*, *methods*, and *measurements*, known as the 4 Ms (Figure 1.2). The bold horizontal arrow points to quality, whereas the diagonal arrows in Figure 1.2 are probable causes having an effect on the quality. Materials often differ when sources of supply or size requirements vary. Machines, or equipment, also function differently depending on variations in their parts, and they operate optimally for only part of the time. Methods, or processes, cause even greater variations due to lack of training and poor handwritten instructions. Finally, measurements also vary due to outdated equipment and improper calibration. Variations in the 4 Ms parameters have an effect on the quality of a product. The Ishikawa diagram has influenced Japanese firms to focus their quality control attention on the improvement of materials, machines, methods, and measurements.

The total-quality movement in Japan has led to pervasive top-management involvement. Many companies in Japan have extensive documentation of their quality activities. Senior executives in the United States either did not believe quality mattered or did not know where to begin until the National Broadcasting Corporation (NBC), an America television network, broadcast the documentary "If Japan Can ... Why Can't We?" at 9:30 P.M. on June 24, 1980 [2]. The documentary was produced by Clare Crawford-Mason and was narrated by Lloyd Dobyns. Fifteen minutes of the broadcast was devoted to Dr. Deming and his work. After the

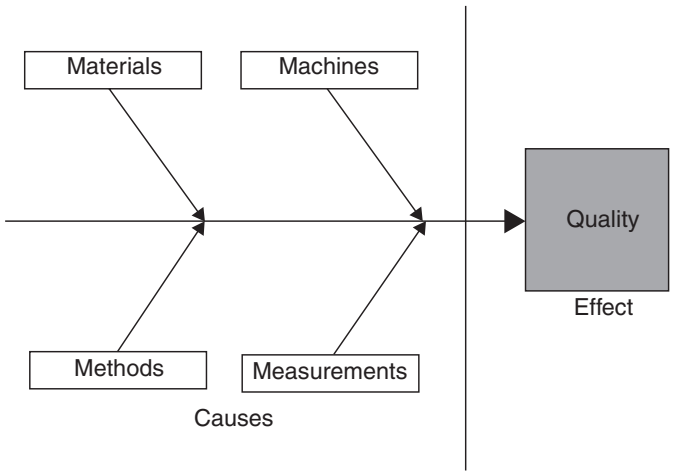

Figure 1.2 Ishikawa diagram.

broadcast, many executives and government leaders realized that a renewed emphasis on quality was no longer an option for American companies but a necessity for doing business in an ever-expanding and more demanding competitive world market. Ford Motor Company and General Motors immediately adopted Deming's SQC methodology into their manufacturing process. Other companies such as Dow Chemical and the Hughes Aircraft followed suit. Ishikawa's TQC management philosophy gained popularity in the United States. Further, the spurred emphasis on quality in American manufacturing companies led the U.S. Congress to establish the Malcolm Baldrige National Quality Award—similar to the Deming Prize in Japan—in 1987 to recognize organizations for their achievements in quality and to raise awareness about the importance of quality excellence as a competitive edge [6]. In the Baldrige National Award, quality is viewed as something defined by the customer and thus the focus is on *customer-driven quality*. On the other hand, in the Deming Prize, quality is viewed as something defined by the producers by conforming to specifications and thus the focus is on *conformance to specifications*.

Remark: Malcolm Baldrige was U.S. Secretary of Commerce from 1981 until his death in a rodeo accident in July 1987. Baldrige was a proponent of quality management as a key to his country's prosperity and long-term strength. He took a personal interest in the quality improvement act, which was eventually named after him, and helped draft one of its early versions. In recognition of his contributions, Congress named the award in his honor.

Traditionally, the TQC and lean concepts are applied in the manufacturing process. The software development process uses these concepts as another tool to guide the production of quality software [13]. These concepts provides a framework to discuss software production issues. The software capability maturity model (CMM) [14] architecture developed at the Software Engineering Institute is based on the principles of product quality that have been developed by W. Edwards Deming [15], Joseph M. Juran [16], Kaoru Ishikawa [12], and Philip Crosby [17].

1.2 SOFTWARE QUALITY

The question "What is software quality?" evokes many different answers. Quality is a complex concept—it means different things to different people, and it is highly context dependent. Garvin [18] has analyzed how software quality is perceived in different ways in different domains, such as philosophy, economics, marketing, and management. Kitchenham and Pfleeger's article [60] on software quality gives a succinct exposition of software quality. They discuss five views of quality in a comprehensive manner as follows:

1. *Transcendental View*: It envisages quality as something that can be recognized but is difficult to define. The transcendental view is not specific to software quality alone but has been applied in other complex areas

of everyday life. For example, In 1964, Justice Potter Stewart of the U.S. Supreme Court, while ruling on the case *Jacobellis v. Ohio*, 378 U.S. 184 (1964), which involved the state of Ohio banning the French film *Les Amants* ("The Lovers") on the ground of pornography, wrote "I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. But *I know it when I see it*, and the motion picture involved in this case is not that" (emphasis added).

2. *User View*: It perceives quality as fitness for purpose. According to this view, while evaluating the quality of a product, one must ask the key question: "Does the product satisfy user needs and expectations?"
3. *Manufacturing View*: Here quality is understood as conformance to the specification. The quality level of a product is determined by the extent to which the product meets its specifications.
4. *Product View*: In this case, quality is viewed as tied to the inherent characteristics of the product. A product's inherent characteristics, that is, internal qualities, determine its external qualities.
5. *Value-Based View*: Quality, in this perspective, depends on the amount a customer is willing to pay for it.

The concept of software quality and the efforts to understand it in terms of measurable quantities date back to the mid-1970s. McCall, Richards, and Walters [19] were the first to study the concept of software quality in terms of *quality factors* and *quality criteria*. A quality factor represents a behavioral characteristic of a system. Some examples of high-level quality factors are *correctness*, *reliability*, *efficiency*, *testability*, *maintainability*, and *reusability*. A quality criterion is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. A highly modular software allows designers to put cohesive components in one module, thereby improving the maintainability of the system.

Various software quality models have been proposed to define quality and its related attributes. The most influential ones are the ISO 9126 [20–22] and the CMM [14]. The ISO 9126 quality model was developed by an expert group under the aegis of the International Organization for Standardization (ISO). The document ISO 9126 defines six broad, independent categories of quality characteristics: *functionality, reliability, usability, efficiency, maintainability*, and *portability*. The CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. In the CMM framework, a development process is evaluated on a scale of 1–5, commonly known as level 1 through level 5. For example, level 1 is called the initial level, whereas level 5—optimized—is the highest level of process maturity.

In the field of software testing, there are two well-known process models, namely, the test process improvement (TPI) model [23] and the test maturity Model (TMM) [24]. These two models allow an organization to assess the current state

of their software testing processes, identify the next logical area for improvement, and recommend an action plan for test process improvement.

1.3 ROLE OF TESTING

Testing plays an important role in achieving and assessing the quality of a software product [25]. On the one hand, we improve the quality of the products as we repeat a *test–find defects–fix* cycle during development. On the other hand, we assess how good our system is when we perform system-level tests before releasing a product. Thus, as Friedman and Voas [26] have succinctly described, software testing is a verification process for software quality assessment and improvement. Generally speaking, the activities for software quality assessment can be divided into two broad categories, namely, *static analysis* and *dynamic analysis*.

- **Static Analysis:** As the term "static" suggests, it is based on the examination of a number of documents, namely requirements documents, software models, design documents, and source code. Traditional static analysis includes code review, inspection, walk-through, algorithm analysis, and proof of correctness. It does not involve actual execution of the code under development. Instead, it examines code and reasons over all possible behaviors that might arise during run time. Compiler optimizations are standard static analysis.
- **Dynamic Analysis:** Dynamic analysis of a software system involves actual program execution in order to expose possible program failures. The behavioral and performance properties of the program are also observed. Programs are executed with both typical and carefully chosen input values. Often, the input set of a program can be impractically large. However, for practical considerations, a finite subset of the input set can be selected. Therefore, in testing, we observe some representative program behaviors and reach a conclusion about the quality of the system. Careful selection of a finite test set is crucial to reaching a reliable conclusion.

By performing static and dynamic analyses, practitioners want to identify as many faults as possible so that those faults are fixed at an early stage of the software development. Static analysis and dynamic analysis are complementary in nature, and for better effectiveness, both must be performed repeatedly and alternated. Practitioners and researchers need to remove the boundaries between static and dynamic analysis and create a hybrid analysis that combines the strengths of both approaches [27].

1.4 VERIFICATION AND VALIDATION

Two similar concepts related to software testing frequently used by practitioners are *verification* and *validation*. Both concepts are abstract in nature, and each can be

realized by a set of concrete, executable activities. The two concepts are explained as follows:

- **Verification:** This kind of activity helps us in evaluating a software system by determining whether the product of a given development phase satisfies the requirements established before the start of that phase. One may note that a product can be an intermediate product, such as requirement specification, design specification, code, user manual, or even the final product. Activities that check the correctness of a development phase are called *verification activities*.
- **Validation:** Activities of this kind help us in confirming that a product meets its intended *use*. Validation activities aim at confirming that a product meets its customer's expectations. In other words, validation activities focus on the final product, which is extensively tested from the customer point of view. Validation establishes whether the product meets overall expectations of the users.

  Late execution of validation activities is often risky by leading to higher development cost. Validation activities may be executed at early stages of the software development cycle [28]. An example of early execution of validation activities can be found in the eXtreme Programming (XP) software development methodology. In the XP methodology, the customer closely interacts with the software development group and conducts acceptance tests during each development iteration [29].

The verification process establishes the correspondence of an implementation phase of the software development process with its specification, whereas validation establishes the correspondence between a system and users' expectations. One can compare verification and validation as follows:

- Verification activities aim at confirming that one is *building the product correctly*, whereas validation activities aim at confirming that one is *building the correct product* [30].
- Verification activities review interim work products, such as requirements specification, design, code, and user manual, during a project life cycle to ensure their quality. The quality attributes sought by verification activities are consistency, completeness, and correctness at each major stage of system development. On the other hand, validation is performed toward the end of system development to determine if the entire system meets the customer's needs and expectations.
- Verification activities are performed on interim products by applying mostly static analysis techniques, such as inspection, walkthrough, and reviews, and using standards and checklists. Verification can also include dynamic analysis, such as actual program execution. On the other hand, validation is performed on the entire system by actually running the system in its real environment and using a variety of tests.

1.5 FAILURE, ERROR, FAULT, AND DEFECT

In the literature on software testing, one can find references to the terms *failure*, *error*, *fault*, and *defect*. Although their meanings are related, there are important distinctions between these four concepts. In the following, we present first three terms as they are understood in the fault-tolerant computing community:

- **Failure:** A failure is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification.
- **Error:** An error is a *state* of the system. In the absence of any corrective action by the system, an error state could lead to a failure which would not be attributed to any event subsequent to the error.
- **Fault:** A fault is the adjudged cause of an error.

A fault may remain undetected for a long time, until some event activates it. When an event activates a fault, it first brings the program into an intermediate error state. If computation is allowed to proceed from an error state without any corrective action, the program eventually causes a failure. As an aside, in fault-tolerant computing, corrective actions can be taken to take a program out of an error state into a desirable state such that subsequent computation does not eventually lead to a failure. The process of failure manifestation can therefore be succinctly represented as a behavior chain [31] as follows: fault $\rightarrow$ error $\rightarrow$ failure. The behavior chain can iterate for a while, that is, failure of one component can lead to a failure of another interacting component.

The above definition of failure assumes that the given specification is acceptable to the customer. However, if the specification does not meet the expectations of the customer, then, of course, even a fault-free implementation fails to satisfy the customer. It is a difficult task to give a precise definition of fault, error, or failure of software, because of the "human factor" involved in the overall acceptance of a system. In an article titled "What Is Software Failure" [32], Ram Chillarege commented that in modern software business software failure means "the customer's expectation has not been met and/or the customer is unable to do useful work with product," p. 354.

Roderick Rees [33] extended Chillarege's comments of software failure by pointing out that "failure is a matter of function only [and is thus] related to purpose, not to whether an item is physically intact or not" (p. 163). To substantiate this, Behrooz Parhami [34] provided three interesting examples to show the relevance of such a view point in wider context. One of the examples is quoted here (p. 451):

> Consider a small organization. *Defects* in the organization's staff promotion policies can cause improper promotions, viewed as *faults*. The resulting ineptitudes & dissatisfactions are *errors* in the organization's state. The organization's personnel or departments probably begin to *malfunction* as result of the errors, in turn causing an overall *degradation* of performance. The end result can be the organization's *failure* to achieve its goal.

There is a fine difference between defects and faults in the above example, that is, execution of a defective policy may lead to a faulty promotion. In a software

context, a software system may be defective due to design issues; certain system states will expose a defect, resulting in the development of faults defined as incorrect signal values or decisions within the system. In industry, the term defect is widely used, whereas among researchers the term fault is more prevalent. For all practical purpose, the two terms are synonymous. In this book, we use the two terms interchangeably as required.

1.6 NOTION OF SOFTWARE RELIABILITY

No matter how many times we run the test–find faults–fix cycle during software development, some faults are likely to escape our attention, and these will eventually surface at the customer site. Therefore, a quantitative measure that is useful in assessing the quality of a software is its *reliability* [35]. *Software reliability* is defined as the probability of failure-free operation of a software system for a specified time in a specified environment. The level of reliability of a system depends on those inputs that cause failures to be observed by the end users. Software reliability can be estimated via *random testing*, as suggested by Hamlet [36]. Since the notion of reliability is specific to a "specified environment," test data must be drawn from the input distribution to closely resemble the future usage of the system. Capturing the future usage pattern of a system in a general sense is described in a form called the *operational profile*. The concept of operational profile of a system was pioneered by John D. Musa at AT&T Bell Laboratories between the 1970s and the 1990s [37, 38].

1.7 OBJECTIVES OF TESTING

The stakeholders in a test process are the programmers, the test engineers, the project managers, and the customers. A stakeholder is a person or an organization who influences a system's behaviors or who is impacted by that system [39]. Different stakeholders view a test process from different perspectives as explained below:

- **It does work:** While implementing a program unit, the programmer may want to test whether or not the unit works in normal circumstances. The programmer gets much confidence if the unit works to his or her satisfaction. The same idea applies to an entire system as well—once a system has been integrated, the developers may want to test whether or not the system performs the basic functions. Here, for the psychological reason, the objective of testing is to show that the system works, rather than it does not work.
- **It does not work:** Once the programmer (or the development team) is satisfied that a unit (or the system) works to a certain degree, more tests are conducted with the objective of finding faults in the unit (or the system). Here, the idea is to try to make the unit (or the system) fail.

- **Reduce the risk of failure:** Most of the complex software systems contain faults, which cause the system to fail from time to time. This concept of "failing from time to time" gives rise to the notion of *failure rate*. As faults are discovered and fixed while performing more and more tests, the failure rate of a system generally decreases. Thus, a higher level objective of performing tests is to bring down the risk of failing to an acceptable level.
- **Reduce the cost of testing:** The different kinds of costs associated with a test process include

  the cost of designing, maintaining, and executing test cases,

  the cost of analyzing the result of executing each test case,

  the cost of documenting the test cases, and

  the cost of actually executing the system and documenting it.

  Therefore, the less the number of test cases designed, the less will be the associated cost of testing. However, producing a small number of arbitrary test cases is not a good way of saving cost. The highest level of objective of performing tests is to produce low-risk software with fewer number of test cases. This idea leads us to the concept of *effectiveness of test cases*. Test engineers must therefore judiciously select fewer, effective test cases.

1.8 WHAT IS A TEST CASE?

In its most basic form, a *test case* is a simple pair of $<$ input, expected outcome $>$. If a program under test is expected to compute the square root of nonnegative numbers, then four examples of test cases are as shown in Figure 1.3.

In stateless systems, where the outcome depends solely on the current input, test cases are very simple in structure, as shown in Figure 1.3. A program to compute the square root of nonnegative numbers is an example of a stateless system. A compiler for the C programming language is another example of a stateless system. A compiler is a stateless system because to compile a program it does not need to know about the programs it compiled previously.

In state-oriented systems, where the program outcome depends both on the current state of the system and the current input, a test case may consist of a

TB_1: $< 0, 0 >$,
TB_2: $< 25, 5 >$,
TB_3: $< 40, 6.3245553 >$,
TB_4: $< 100.5, 10.024968 >$.

Figure 1.3 Examples of basic test cases.

TS_1: < check balance, \$500.00 >, < withdraw, ''amount?'' >,
< \$200.00, ''\$200.00'' >, < check balance, \$300.00 > .

Figure 1.4 Example of a test case with a sequence of < input, expected outcome >.

sequence of < input, expected outcome > pairs. A telephone switching system and an automated teller machine (ATM) are examples of state-oriented systems. For an ATM machine, a test case for testing the *withdraw* function is shown in Figure 1.4. Here, we assume that the user has already entered validated inputs, such as the cash card and the personal identification number (PIN).

In the test case TS_1, "check balance" and "withdraw" in the first, second, and fourth tuples represent the pressing of the appropriate keys on the ATM keypad. It is assumed that the user account has \$500.00 on it, and the user wants to withdraw an amount of \$200.00. The expected outcome "\$200.00" in the third tuple represents the cash dispensed by the ATM. After the withdrawal operation, the user makes sure that the remaining balance is \$300.00.

For state-oriented systems, most of the test cases include some form of decision and timing in providing input to the system. A test case may include loops and timers, which we do not show at this moment.

1.9 EXPECTED OUTCOME

An *outcome* of program execution is a complex entity that may include the following:

- Values produced by the program:
  - Outputs for local observation (integer, text, audio, image)
  - Outputs (messages) for remote storage, manipulation, or observation
- State change:
  - State change of the program
  - State change of the database (due to add, delete, and update operations)
- A sequence or set of values which must be interpreted together for the outcome to be valid

An important concept in test design is the concept of an *oracle*. An oracle is any entity—program, process, human expert, or body of data—that tells us the expected outcome of a particular test or set of tests [40]. A test case is meaningful only if it is possible to decide on the acceptability of the result produced by the program under test.

Ideally, the expected outcome of a test should be computed while designing the test case. In other words, the test outcome is computed before the program is

executed with the selected test input. The idea here is that one should be able to compute the expected outcome from an *understanding* of the program's requirements. Precomputation of the expected outcome will eliminate any implementation bias in case the test case is designed by the developer.

In exceptional cases, where it is extremely difficult, impossible, or even undesirable to compute a single expected outcome, one should identify expected outcomes by examining the actual test outcomes, as explained in the following:

1. Execute the program with the selected input.
2. Observe the actual outcome of program execution.
3. Verify that the actual outcome is the expected outcome.
4. Use the verified actual outcome as the expected outcome in subsequent runs of the test case.

1.10 CONCEPT OF *COMPLETE* TESTING

It is not unusual to find people making claims such as "I have exhaustively tested the program." Complete, or exhaustive, testing means *there are no undiscovered faults at the end of the test phase*. All problems must be known at the end of complete testing. For most of the systems, complete testing is near impossible because of the following reasons:

- The domain of possible inputs of a program is too large to be completely used in testing a system. There are both valid inputs and invalid inputs. The program may have a large number of states. There may be timing constraints on the inputs, that is, an input may be valid at a certain time and invalid at other times. An input value which is valid but is not properly timed is called an *inopportune* input. The input domain of a system can be very large to be completely used in testing a program.
- The design issues may be too complex to completely test. The design may have included implicit design decisions and assumptions. For example, a programmer may use a global variable or a *static* variable to control program execution.
- It may not be possible to create all possible execution environments of the system. This becomes more significant when the behavior of the software system depends on the real, outside world, such as weather, temperature, altitude, pressure, and so on.

1.11 CENTRAL ISSUE IN TESTING

We must realize that though the outcome of complete testing, that is, discovering all faults, is highly desirable, it is a near-impossible task, and it may not be attempted. The next best thing is to select a subset of the input domain to test a program.

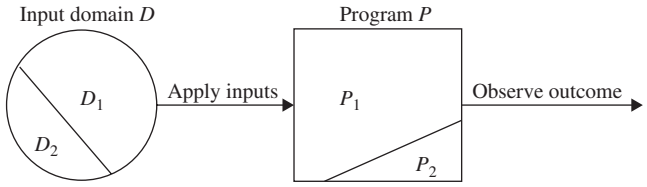

Figure 1.5 Subset of the input domain exercising a subset of the program behavior.

Referring to Figure 1.5, let D be the input domain of a program P. Suppose that we select a subset D_1 of D, that is, $D_1 \subset D$, to test program P. It is possible that D_1 exercises only a part P_1, that is, $P_1 \subset P$, of the *execution behavior* of P, in which case faults with the other part, P_2, will go undetected.

By selecting a subset of the input domain D_1, the test engineer attempts to deduce properties of an entire program P by observing the behavior of a part P_1 of the entire behavior of P on selected inputs D_1. Therefore, *selection* of the subset of the input domain must be done in a systematic and careful manner so that the deduction is as accurate and complete as possible. For example, the idea of *coverage* is considered while selecting test cases.

1.12 TESTING ACTIVITIES

In order to test a program, a test engineer must perform a sequence of testing activities. Most of these activities have been shown in Figure 1.6 and are explained in the following. These explanations focus on a single test case.

- **Identify an objective to be tested:** The first activity is to identify an *objective* to be tested. The objective defines the intention, or *purpose*, of designing one or more test cases to ensure that the program supports the objective. A clear purpose must be associated with every test case.

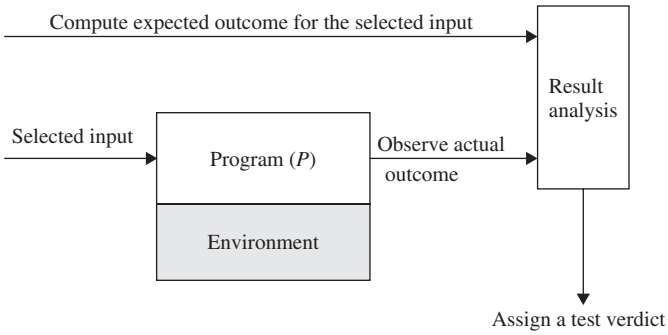

Figure 1.6 Different activities in program testing.

- **Select inputs:** The second activity is to select test inputs. Selection of test inputs can be based on the requirements specification, the source code, or our expectations. Test inputs are selected by keeping the test objective in mind.
- **Compute the expected outcome:** The third activity is to compute the expected outcome of the program with the selected inputs. In most cases, this can be done from an overall, high-level understanding of the test objective and the specification of the program under test.
- **Set up the execution environment of the program:** The fourth step is to prepare the right execution environment of the program. In this step all the assumptions external to the program must be satisfied. A few examples of assumptions external to a program are as follows:

  Initialize the local system, external to the program. This may include making a network connection available, making the right database system available, and so on.

  Initialize any remote, external system (e.g., remote partner process in a distributed application.) For example, to test the client code, we may need to start the server at a remote site.

- **Execute the program:** In the fifth step, the test engineer executes the program with the selected inputs and observes the actual outcome of the program. To execute a test case, inputs may be provided to the program at different physical locations at different times. The concept of *test coordination* is used in synchronizing different components of a test case.
- **Analyze the test result:** The final test activity is to analyze the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome. The complexity of comparison depends on the complexity of the data to be observed. The observed data type can be as simple as an integer or a string of characters or as complex as an image, a video, or an audio clip. At the end of the analysis step, a test verdict is assigned to the program. There are three major kinds of test verdicts, namely, *pass*, *fail*, and *inconclusive*, as explained below.

  If the program produces the expected outcome and the purpose of the test case is satisfied, then a pass verdict is assigned.

  If the program does not produce the expected outcome, then a fail verdict is assigned.

  However, in some cases it may not be possible to assign a clear pass or fail verdict. For example, if a timeout occurs while executing a test case on a distributed application, we may not be in a position to assign a clear pass or fail verdict. In those cases, an inconclusive test verdict is assigned. An inconclusive test verdict means that further tests are needed to be done to refine the inconclusive verdict into a clear pass or fail verdict.

A *test report* must be written after analyzing the test result. The motivation for writing a test report is to get the fault fixed if the test revealed a fault. A test report contains the following items to be informative:

Explain how to reproduce the failure.

Analyze the failure to be able to describe it.

A pointer to the actual outcome and the test case, complete with the input, the expected outcome, and the execution environment.

1.13 TEST LEVELS

Testing is performed at different levels involving the complete system or parts of it throughout the life cycle of a software product. A software system goes through four stages of testing before it is actually deployed. These four stages are known as *unit*, *integration*, *system*, and *acceptance* level testing. The first three levels of testing are performed by a number of different stakeholders in the development organization, where as acceptance testing is performed by the customers. The four stages of testing have been illustrated in the form of what is called the classical V model in Figure 1.7.

In unit testing, programmers test individual program units, such as a procedures, functions, methods, or classes, in isolation. After ensuring that individual units work to a satisfactory extent, modules are assembled to construct larger subsystems by following integration testing techniques. Integration testing is jointly performed by software developers and integration test engineers. The objective of

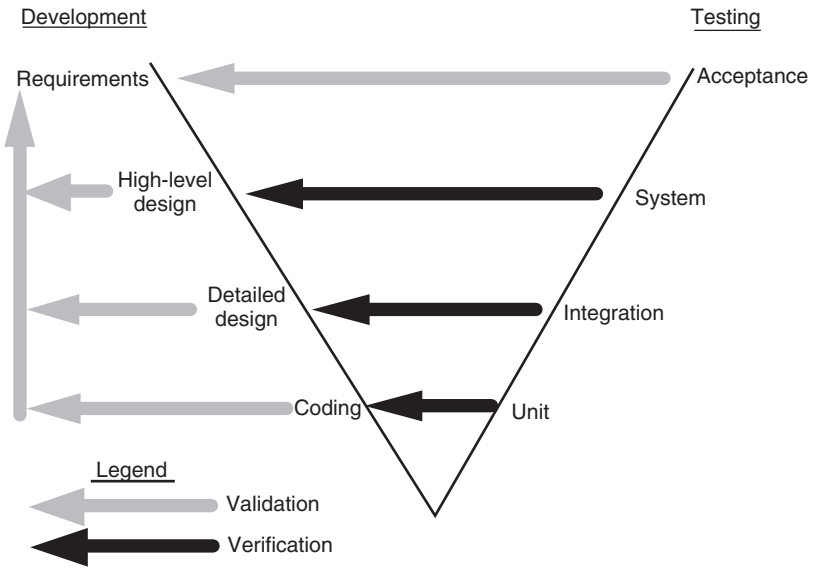

Figure 1.7 Development and testing phases in the V model.

integration testing is to construct a reasonably stable system that can withstand the rigor of system-level testing. System-level testing includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing. System testing is a critical phase in a software development process because of the need to meet a tight schedule close to delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

Regression testing is another level of testing that is performed throughout the life cycle of a system. Regression testing is performed whenever a component of the system is modified. The key idea in regression testing is to ascertain that the modification has not introduced any new faults in the portion that was not subject to modification. To be precise, regression testing is not a distinct level of testing. Rather, it is considered as a subphase of unit, integration, and system-level testing, as illustrated in Figure 1.8 [41].

In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software. Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry. It is desirable to select a subset of the test cases from the existing pool to reduce the cost. A key question is how many and which test cases should be selected so that the selected test cases are more likely to uncover new faults [42–44].

After the completion of system-level testing, the product is delivered to the customer. The customer performs their own series of tests, commonly known as *acceptance testing*. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing. A key notion in acceptance testing is the customer's *expectations* from the system. By the time of acceptance testing, the customer should have developed their acceptance criteria based on their own expectations from the system. There are two kinds of acceptance testing as explained in the following:

- User acceptance testing (UAT)
- Business acceptance testing (BAT)

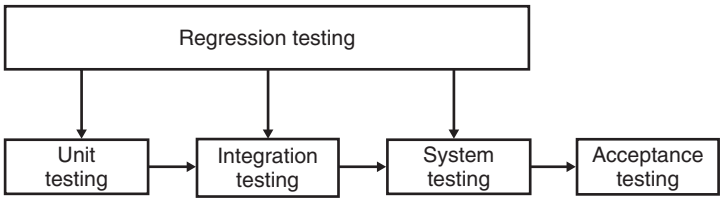

Figure 1.8 Regression testing at different software testing levels. (From ref. 41. © 2005 John Wiley & Sons.)

User acceptance testing is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed off as meeting user needs. On the other hand, BAT is undertaken within the supplier's development organization. The idea in having a BAT is to ensure that the system will eventually pass the user acceptance test. It is a rehearsal of UAT at the supplier's premises.

1.14 SOURCES OF INFORMATION FOR TEST CASE SELECTION

Designing test cases has continued to stay in the foci of the research community and the practitioners. A software development process generates a large body of information, such as requirements specification, design document, and source code. In order to generate effective tests at a lower cost, test designers analyze the following sources of information:

- Requirements and functional specifications
- Source code
- Input and output domains
- Operational profile
- Fault model

Requirements and Functional Specifications The process of software development begins by capturing user needs. The nature and amount of user needs identified at the beginning of system development will vary depending on the specific life-cycle model to be followed. Let us consider a few examples. In the Waterfall model [45] of software development, a requirements engineer tries to capture most of the requirements. On the other hand, in an agile software development model, such as XP [29] or the Scrum [46–48], only a few requirements are identified in the beginning. A test engineer considers all the requirements the program is expected to meet whichever life-cycle model is chosen to test a program.

The requirements might have been specified in an informal manner, such as a combination of plaintext, equations, figures, and flowcharts. Though this form of requirements specification may be ambiguous, it is easily understood by customers. For example, the Bluetooth specification consists of about 1100 pages of descriptions explaining how various subsystems of a Bluetooth interface is expected to work. The specification is written in plaintext form supplemented with mathematical equations, state diagrams, tables, and figures. For some systems, requirements may have been captured in the form of *use cases*, *entity–relationship diagrams*, and *class diagrams*. Sometimes the requirements of a system may have been specified in a formal language or notation, such as Z, SDL, Estelle, or finite-state machine. Both the informal and formal specifications are prime sources of test cases [49].

Source Code Whereas a requirements specification describes the *intended behavior* of a system, the source code describes the *actual behavior* of the system. High-level assumptions and constraints take concrete form in an implementation. Though a software designer may produce a detailed design, programmers may introduce additional details into the system. For example, a step in the detailed design can be "sort array A." To sort an array, there are many sorting algorithms with different characteristics, such as iteration, recursion, and temporarily using another array. Therefore, test cases must be designed based on the program [50].

Input and Output Domains Some values in the input domain of a program have special meanings, and hence must be treated separately [5]. To illustrate this point, let us consider the *factorial* function. The factorial of a nonnegative integer n is computed as follows:

```
factorial(0) = 1;
factorial(1) = 1;
factorial(n) = n * factorial(n-1);
```

A programmer may wrongly implement the factorial function as

```
factorial(n) = 1 * 2 * ... * n;
```

without considering the special case of $n = 0$. The above wrong implementation will produce the correct result for all positive values of n, but will fail for $n = 0$.

Sometimes even some output values have special meanings, and a program must be tested to ensure that it produces the special values for all possible causes. In the above example, the output value 1 has special significance: (i) it is the minimum value computed by the factorial function and (ii) it is the only value produced for two different inputs.

In the integer domain, the values 0 and 1 exhibit special characteristics if arithmetic operations are performed. These characteristics are $0 \times x = 0$ and $1 \times x = x$ for all values of x. Therefore, all the special values in the input and output domains of a program must be considered while testing the program.

Operational Profile As the term suggests, an *operational profile* is a quantitative characterization of how a system will be used. It was created to guide test engineers in selecting test cases (inputs) using samples of system usage. The notion of operational profiles, or *usage profiles*, was developed by Mills et al. [52] at IBM in the context of Cleanroom Software Engineering and by Musa [37] at AT&T Bell Laboratories to help develop software systems with better reliability. The idea is to infer, from the observed test results, the future reliability of the software when it is in actual use. To do this, test inputs are assigned a probability distribution, or profile, according to their occurrences in actual operation. The ways test engineers assign probability and select test cases to operate a system may significantly differ from the ways actual users operate a system. However, for accurate estimation of the reliability of a system it is important to test a system by considering the ways it will actually be used in the field. This concept is being used to test web

applications, where the user session data are collected from the web servers to select test cases [53, 54].

Fault Model Previously encountered faults are an excellent source of information in designing new test cases. The known faults are classified into different classes, such as initialization faults, logic faults, and interface faults, and stored in a repository [55, 56]. Test engineers can use these data in designing tests to ensure that a particular class of faults is not resident in the program.

There are three types of fault-based testing: error guessing, fault seeding, and mutation analysis. In error guessing, a test engineer applies his experience to (i) assess the situation and guess where and what kinds of faults might exist, and (ii) design tests to specifically expose those kinds of faults. In fault seeding, known faults are injected into a program, and the test suite is executed to assess the effectiveness of the test suite. Fault seeding makes an assumption that a test suite that finds seeded faults is also likely to find other faults. Mutation analysis is similar to fault seeding, except that mutations to program statements are made in order to determine the fault detection capability of the test suite. If the test cases are not capable of revealing such faults, the test engineer may specify additional test cases to reveal the faults. Mutation testing is based on the idea of fault simulation, whereas fault seeding is based on the idea of fault injection. In the fault injection approach, a fault is inserted into a program, and an oracle is available to assert that the inserted fault indeed made the program incorrect. On the other hand, in fault simulation, a program modification is not guaranteed to lead to a faulty program. In fault simulation, one may modify an incorrect program and turn it into a correct program.

1.15 WHITE-BOX AND BLACK-BOX TESTING

A key idea in Section 1.14 was that test cases need to be designed by considering information from several sources, such as the specification, source code, and special properties of the program's input and output domains. This is because all those sources provide complementary information to test designers. Two broad concepts in testing, based on the sources of information for test design, are *white-box* and *black-box* testing. White-box testing techniques are also called *structural testing* techniques, whereas black-box testing techniques are called *functional testing* techniques.

In structural testing, one primarily examines *source code* with a focus on control flow and data flow. Control flow refers to flow of control from one instruction to another. Control passes from one instruction to another instruction in a number of ways, such as one instruction appearing after another, function call, message passing, and interrupts. Conditional statements alter the normal, sequential flow of control in a program. Data flow refers to the propagation of values from one variable or constant to another variable. Definitions and uses of variables determine the data flow aspect in a program.

In functional testing, one does not have access to the internal details of a program and the program is treated as a black box. A test engineer is concerned only with the part that is accessible outside the program, that is, just the input and the externally visible outcome. A test engineer applies input to a program, observes the externally visible outcome of the program, and determines whether or not the program outcome is the expected outcome. Inputs are selected from the program's requirements specification and properties of the program's input and output domains. A test engineer is concerned only with the functionality and the features found in the program's specification.

At this point it is useful to identify a distinction between the scopes of structural testing and functional testing. One applies structural testing techniques to individual units of a program, whereas functional testing techniques can be applied to both an entire system and the individual program units. Since individual programmers know the details of the source code they write, they themselves perform structural testing on the individual program units they write. On the other hand, functional testing is performed at the external interface level of a system, and it is conducted by a separate software quality assurance group.

Let us consider a program unit U which is a part of a larger program P. A program unit is just a piece of source code with a well-defined objective and well-defined input and output domains. Now, if a programmer derives test cases for testing U from a knowledge of the internal details of U, then the programmer is said to be performing structural testing. On the other hand, if the programmer designs test cases from the stated objective of the unit U and from his or her knowledge of the special properties of the input and output domains of U, then he or she is said to be performing functional testing on the same unit U.

The ideas of structural testing and functional testing do not give programmers and test engineers a choice of whether to design test cases from the source code or from the requirements specification of a program. However, these strategies are used by different groups of people at different times during a software's life cycle. For example, individual programmers use both the structural and functional testing techniques to test their own code, whereas quality assurance engineers apply the idea of functional testing.

Neither structural testing nor functional testing is by itself good enough to detect most of the faults. Even if one selects all possible inputs, a structural testing technique cannot detect all faults if there are *missing paths* in a program. Intuitively, a path is said to be missing if there is no code to handle a possible condition. Similarly, without knowledge of the structural details of a program, many faults will go undetected. Therefore, a combination of both structural and functional testing techniques must be used in program testing.

1.16 TEST PLANNING AND DESIGN

The purpose of system test planning, or simply test planning, is to get ready and organized for test execution. A test plan provides a framework, scope, details of resource needed, effort required, schedule of activities, and a budget. A framework

is a set of ideas, facts, or circumstances within which the tests will be conducted. The stated scope outlines the domain, or extent, of the test activities. The scope covers the managerial aspects of testing, rather than the detailed techniques and specific test cases.

Test design is a critical phase of software testing. During the test design phase, the system requirements are critically studied, system features to be tested are thoroughly identified, and the objectives of test cases and the detailed behavior of test cases are defined. Test objectives are identified from different sources, namely, the requirement specification and the functional specification, and one or more test cases are designed for each test objective. Each test case is designed as a combination of modular test components called *test steps*. These test steps can be combined together to create more complex, multistep tests. A test case is clearly specified so that others can easily borrow, understand, and reuse it.

It is interesting to note that a new test-centric approach to system development is gradually emerging. This approach is called test-driven development (TDD) [57]. In test-driven development, programmers design and implement test cases before the production code is written. This approach is a key practice in modern agile software development processes such as XP. The main characteristics of agile software development processes are (i) incremental development, (ii) coding of unit and acceptance tests conducted by the programmers along with customers, (iii) frequent regression testing, and (iv) writing test code, one test case at a time, before the production code.

1.17 MONITORING AND MEASURING TEST EXECUTION

Monitoring and measurement are two key principles followed in every scientific and engineering endeavor. The same principles are also applicable to the testing phases of software development. It is important to monitor certain metrics which truly represent the progress of testing and reveal the quality level of the system. Based on those metrics, the management can trigger corrective and preventive actions. By putting a small but critical set of metrics in place the executive management will be able to know whether they are on the right track [58]. Test execution metrics can be broadly categorized into two classes as follows:

- Metrics for monitoring test execution
- Metrics for monitoring defects

The first class of metrics concerns the process of executing test cases, whereas the second class concerns the defects found as a result of test execution. These metrics need to be tracked and analyzed on a periodic basis, say, daily or weekly. In order to effectively control a test project, it is important to gather valid and accurate information about the project. One such example is to precisely know when to trigger revert criteria for a test cycle and initiate root cause analysis of

the problems before more tests can be performed. By triggering such a revert criteria, a test manager can effectively utilize the time of test engineers, and possibly money, by suspending a test cycle on a product with too many defects to carry out a meaningful system test. A management team must identify and monitor metrics while testing is in progress so that important decisions can be made [59]. It is important to analyze and understand the test metrics, rather than just collect data and make decisions based on those raw data. Metrics are meaningful only if they enable the management to make decisions which result in lower cost of production, reduced delay in delivery, and improved quality of software systems.

Quantitative evaluation is important in every scientific and engineering field. Quantitative evaluation is carried out through measurement. Measurement lets one evaluate parameters of interest in a quantitative manner as follows:

- Evaluate the effectiveness of a technique used in performing a task. One can evaluate the effectiveness of a test generation technique by counting the number of defects detected by test cases generated by following the technique and those detected by test cases generated by other means.
- Evaluate the productivity of the development activities. One can keep track of productivity by counting the number of test cases designed per day, the number of test cases executed per day, and so on.
- Evaluate the quality of the product. By monitoring the number of defects detected per week of testing, one can observe the quality level of the system.
- Evaluate the product testing. For evaluating a product testing process, the following two measurements are critical:

  Test case effectiveness metric: The objective of this metric is twofold as explained in what follows: (1) measure the "defect revealing ability" of the test suite and (2) use the metric to improve the test design process. During the unit, integration, and system testing phases, faults are revealed by executing the planned test cases. In addition to these faults, new faults are also found during a testing phase for which no test cases had been designed. For these new faults, new test cases are added to the test suite. Those new test cases are called test case escaped (TCE). Test escapes occur because of deficiencies in test design. The need for more testing occurs as test engineers get new ideas while executing the planned test cases.

  Test effort effectiveness metric: It is important to evaluate the effectiveness of the testing effort in the development of a product. After a product is deployed at the customer's site, one is interested to know the effectiveness of testing that was performed. A common measure of test effectiveness is the number of defects found by the customers that were not found by the test engineers prior to the release of the product. These defects had escaped our test effort.

1.18 TEST TOOLS AND AUTOMATION

In general, software testing is a highly labor intensive task. This is because test cases are to a great extent manually generated and often manually executed. Moreover, the results of test executions are manually analyzed. The durations of those tasks can be shortened by using appropriate tools. A test engineer can use a variety of tools, such as a *static code analyzer*, a *test data generator*, and a *network analyzer*, if a network-based application or protocol is under test. Those tools are useful in increasing the efficiency and effectiveness of testing.

Test automation is essential for any testing and quality assurance division of an organization to move forward to become more efficient. The benefits of test automation are as follows:

- Increased productivity of the testers
- Better coverage of regression testing
- Reduced durations of the testing phases
- Reduced cost of software maintenance
- Increased effectiveness of test cases

Test automation provides an opportunity to improve the skills of the test engineers by writing programs, and hence their morale. They will be more focused on developing automated test cases to avoid being a bottleneck in product delivery to the market. Consequently, software testing becomes less of a tedious job.

Test automation improves the coverage of regression testing because of accumulation of automated test cases over time. Automation allows an organization to create a rich library of reusable test cases and facilitates the execution of a consistent set of test cases. Here consistency means our ability to produce repeated results for the same set of tests. It may be very difficult to reproduce test results in manual testing, because exact conditions at the time and point of failure may not be precisely known. In automated testing it is easier to set up the initial conditions of a system, thereby making it easier to reproduce test results. Test automation simplifies the debugging work by providing a detailed, unambiguous log of activities and intermediate test steps. This leads to a more organized, structured, and reproducible testing approach.

Automated execution of test cases reduces the elapsed time for testing, and, thus, it leads to a shorter time to market. The same automated test cases can be executed in an unsupervised manner at night, thereby efficiently utilizing the different platforms, such as hardware and configuration. In short, automation increases test execution efficiency. However, at the end of test execution, it is important to analyze the test results to determine the number of test cases that passed or failed. And, if a test case failed, one analyzes the reasons for its failure.

In the long run, test automation is cost-effective. It drastically reduces the software maintenance cost. In the sustaining phase of a software system, the regression tests required after each change to the system are too many. As a result, regression testing becomes too time and labor intensive without automation.

A repetitive type of testing is very cumbersome and expensive to perform manually, but it can be automated easily using software tools. A simple repetitive type of application can reveal memory leaks in a software. However, the application has to be run for a significantly long duration, say, for weeks, to reveal memory leaks. Therefore, manual testing may not be justified, whereas with automation it is easy to reveal memory leaks. For example, stress testing is a prime candidate for automation. Stress testing requires a worst-case load for an extended period of time, which is very difficult to realize by manual means. Scalability testing is another area that can be automated. Instead of creating a large test bed with hundreds of equipment, one can develop a simulator to verify the scalability of the system.

Test automation is very attractive, but it comes with a price tag. Sufficient time and resources need to be allocated for the development of an automated test suite. Development of automated test cases need to be managed like a programming project. That is, it should be done in an organized manner; otherwise it is highly likely to fail. An automated test suite may take longer to develop because the test suite needs to be debugged before it can be used for testing. Sufficient time and resources need to be allocated for maintaining an automated test suite and setting up a test environment. Moreover, every time the system is modified, the modification must be reflected in the automated test suite. Therefore, an automated test suite should be designed as a modular system, coordinated into reusable libraries, and cross-referenced and traceable back to the feature being tested.

It is important to remember that test automation cannot replace manual testing. Human creativity, variability, and observability cannot be mimicked through automation. Automation cannot detect some problems that can be easily observed by a human being. Automated testing does not introduce minor variations the way a human can. Certain categories of tests, such as usability, interoperability, robustness, and compatibility, are often not suited for automation. It is too difficult to automate all the test cases; usually 50% of all the system-level test cases can be automated. There will always be a need for some manual testing, even if all the system-level test cases are automated.

The objective of test automation is not to reduce the head counts in the testing department of an organization, but to improve the productivity, quality, and efficiency of test execution. In fact, test automation requires a larger head count in the testing department in the first year, because the department needs to automate the test cases and simultaneously continue the execution of manual tests. Even after the completion of the development of a test automation framework and test case libraries, the head count in the testing department does not drop below its original level. The test organization needs to retain the original team members in order to improve the quality by adding more test cases to the automated test case repository.

Before a test automation project can proceed, the organization must assess and address a number of considerations. The following list of prerequisites must be considered for an assessment of whether the organization is ready for test automation:

- The test cases to be automated are well defined.
- Test tools and an infrastructure are in place.

- The test automation professionals have prior successful experience in automation.
- Adequate budget should have been allocated for the procurement of software tools.

1.19 TEST TEAM ORGANIZATION AND MANAGEMENT

Testing is a distributed activity conducted at different levels throughout the life cycle of a software. These different levels are unit testing, integration testing, system testing, and acceptance testing. It is logical to have different testing groups in an organization for each level of testing. However, it is more logical—and is the case in reality—that unit-level tests be developed and executed by the programmers themselves rather than an independent group of unit test engineers. The programmer who develops a software unit should take the ownership and responsibility of producing good-quality software to his or her satisfaction. System integration testing is performed by the system integration test engineers. The integration test engineers involved need to know the software modules very well. This means that all development engineers who collectively built all the units being integrated need to be involved in integration testing. Also, the integration test engineers should thoroughly know the build mechanism, which is key to integrating large systems.

A team for performing system-level testing is truly separated from the development team, and it usually has a separate head count and a separate budget. The mandate of this group is to ensure that the system requirements have been met and the system is acceptable. Members of the system test group conduct different categories of tests, such as functionality, robustness, stress, load, scalability, reliability, and performance. They also execute business acceptance tests identified in the user acceptance test plan to ensure that the system will eventually pass user acceptance testing at the customer site. However, the real user acceptance testing is executed by the client's special user group. The user group consists of people from different backgrounds, such as software quality assurance engineers, business associates, and customer support engineers. It is a common practice to create a temporary user acceptance test group consisting of people with different backgrounds, such as integration test engineers, system test engineers, customer support engineers, and marketing engineers. Once the user acceptance is completed, the group is dismantled. It is recommended to have at least two test groups in an organization: integration test group and system test group.

Hiring and retaining test engineers are challenging tasks. Interview is the primary mechanism for evaluating applicants. Interviewing is a skill that improves with practice. It is necessary to have a recruiting process in place in order to be effective in hiring excellent test engineers. In order to retain test engineers, the management must recognize the importance of testing efforts at par with development efforts. The management should treat the test engineers as professionals and as a part of the overall team that delivers quality products.

1.20 OUTLINE OF BOOK

With the above high-level introduction to quality and software testing, we are now in a position to outline the remaining chapters. Each chapter in the book covers technical, process, and/or managerial topics related to software testing. The topics have been designed and organized to facilitate the reader to become a software test specialist. In Chapter 2 we provide a self-contained introduction to the theory and limitations of software testing.

Chapters 3–6 treat unit testing techniques one by one, as quantitatively as possible. These chapters describe both static and dynamic unit testing. Static unit testing has been presented within a general framework called *code review*, rather than individual techniques called *inspection* and *walkthrough*. Dynamic unit testing, or execution-based unit testing, focuses on control flow, data flow, and domain testing. The JUnit framework, which is used to create and execute dynamic unit tests, is introduced. We discuss some tools for effectively performing unit testing.

Chapter 7 discusses the concept of integration testing. Specifically, five kinds of integration techniques, namely, top down, bottom up, sandwich, big bang, and incremental, are explained. Next, we discuss the integration of hardware and software components to form a complete system. We introduce a framework to develop a plan for system integration testing. The chapter is completed with a brief discussion of integration testing of off-the-shelf components.

Chapters 8–13 discuss various aspects of system-level testing. These six chapters introduce the reader to the technical details of system testing that is the practice in industry. These chapters promote both qualitative and quantitative evaluation of a system testing process. The chapters emphasize the need for having an independent system testing group. A process for monitoring and controlling system testing is clearly explained. Chapter 14 is devoted to acceptance testing, which includes acceptance testing criteria, planning for acceptance testing, and acceptance test execution.

Chapter 15 contains the fundamental concepts of software reliability and their application to software testing. We discuss the notion of *operation profile* and its application in system testing. We conclude the chapter with the description of an example and the time of releasing a system by determining the additional length of system testing. The additional testing time is calculated by using the idea of software reliability.

In Chapter 16, we present the structure of test groups and how these groups can be organized in a software company. Next, we discuss how to hire and retain test engineers by providing training, instituting a reward system, and establishing an attractive career path for them within the testing organization. We conclude this chapter with the description of how to build and manage a test team with a focus on teamwork rather than individual gain.

Chapters 17 and 18 explain the concepts of software quality and different maturity models. Chapter 17 focuses on quality factors and criteria and describes the ISO 9126 and ISO 9000:2000 standards. Chapter 18 covers the CMM, which

was developed by the SEI at Carnegie Mellon University. Two test-related models, namely the TPI model and the TMM, are explained at the end of Chapter 18.

We define the key words used in the book in a glossary at the end of the book. The reader will find about 10 practice exercises at the end of each chapter. A list of references is included at the end of each chapter for a reader who would like to find more detailed discussions of some of the topics. Finally, each chapter, except this one, contains a literature review section that, essentially, provides pointers to more advanced material related to the topics. The more advanced materials are based on current research and alternate viewpoints.

REFERENCES

1. B. Davis, C. Skube, L. Hellervik, S. Gebelein, and J. Sheard. *Successful Manager's Handbook*. Personnel Decisions International, Minneapolis, 1996.
2. M. Walton. *The Deming Management Method*. The Berkley Publishing Group, New York, 1986.
3. W. E. Deming. Transcript of Speech to GAO Roundtable on Product Quality—Japan vs. the United States. *Quality Progress*, March 1994, pp. 39–44.
4. W. A. Shewhart. *Economic Control of Quality of Manufactured Product*. Van Nostrand, New York, 1931.
5. W. A. Shewhart. The Application of Statistics as an Aid in Maintaining Quality of a Manufactured Product. *Journal of American Statistical Association*, December 1925, pp. 546–548.
6. National Institute of Standards and Technology, *Baldridge National Quality Program*, 2008. Available: http://www.quality.nist.gov/.
7. J. Liker and D. Meier. *The Toyota Way Fieldbook*. McGraw-Hill, New York, 2005.
8. M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley, Reading, MA, 2006.
9. A. B. Godfrey and A. I. C. Endres. The Evolution of Quality Management Within Telecommunications. *IEEE Communications Magazine*, October 1994, pp. 26–34.
10. M. Pecht and W. R. Boulton. *Quality Assurance and Reliability in the Japanese Electronics Industry*. Japanses Technology Evaluation Center (JTEC), Report on Electronic Manufacturing and Packaging in Japan, W. R. Boulton, Ed. International Technology Research Institute at Loyola College, February 1995, pp. 115–126.
11. A. V. Feigenbaum. *Total Quality Control*, 4th ed. McGraw-Hill, New York, 2004.
12. K. Ishikawa. *What Is Total Quality Control*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
13. A. Cockburn. What Engineering Has in Common With Manufacturing and Why It Matters. *Crosstalk, the Journal of Defense Software Engineering*, April 2007, pp. 4–7.
14. S. Land. *Jumpstart CMM/CMMI Software Process Improvement*. Wiley, Hoboken, NJ, 2005.
15. W. E. Deming. *Out of the Crisis*. MIT, Cambridge, MA, 1986.
16. J. M. Juran and A. B. Godfrey. *Juran's Quality Handbook*, 5th ed. McGraw-Hill, New York, 1998.
17. P. Crosby. *Quality Is Free*. New American Library, New York, 1979.
18. D. A. Garvin. What Does "Product Quality" Really Mean? *Sloan Management Review*, Fall 1984, pp. 25–43.
19. J. A. McCall, P. K. Richards, and G. F. Walters. Factors in Software Quality, Technical Report RADC-TR-77-369. U.S. Department of Commerce, Washington, DC, 1977.
20. International Organization for Standardization (ISO). Quality Management Systems—Fundamentals and Vocabulary, ISO 9000:2000. ISO, Geneva, December 2000.
21. International Organization for Standardization (ISO). Quality Management Systems—Guidelines for Performance Improvements, ISO 9004:2000. ISO, Geneva, December 2000.
22. International Organization for Standardization (ISO). Quality Management Systems—Requirements, ISO 9001:2000. ISO, Geneva, December 2000.
23. T. Koomen and M. Pol. *Test Process Improvement*. Addison-Wesley, Reading, MA, 1999.

24. I. Burnstein. *Practical Software Testing*. Springer, New York, 2003.
25. L. Osterweil et al. Strategic Directions in Software Quality. *ACM Computing Surveys*, December 1996, pp. 738–750.
26. M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability*. Wiley, New York, 1995.
27. Michael D. Ernst. Static and Dynamic Analysis: Synergy and Duality. Paper presented at ICSE Workshop on Dynamic Analysis, Portland, OR, May 2003, pp. 24–27.
28. L. Baresi and M. Pezzè. *An Introduction to Software Testing*, Electronic Notes in Theoretical Computer Science. Elsevier, Vol. 148, Feb. 2006, pp. 89–111.
29. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, MA, 2004.
30. B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
31. J. C. Laprie. Dependability—Its Attributes, Impairments and Means. In *Predictably Dependable Computing Systems*, B. Randall, J. C. Laprie, H. Kopetz, and B. Littlewood, Eds. Springer-Verlag, New York, 1995.
32. R. Chillarege. What Is Software Failure. *IEEE Transactions on Reliability*, September 1996, pp. 354–355.
33. R. Rees. What Is a Failure. *IEEE Transactions on Reliability*, June 1997, p. 163.
34. B. Parhami. Defect, Fault, Error, . . . , or Failure. *IEEE Transactions on Reliability*, December 1997, pp. 450–451.
35. M. R. Lyu. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1995.
36. R. Hamlet. Random Testing. In *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, New York, 1994, pp. 970–978.
37. J. D. Musa. Software Reliability Engineering. *IEEE Software*, March 1993, pp. 14–32.
38. J. D. Musa. A Theory of Software Reliability and Its Application. *IEEE Transactions on Software Engineering*, September 1975, pp. 312–327.
39. M. Glinz and R. J. Wieringa. Stakeholders in Requirements Engineering. *IEEE Software*, March–April 2007, pp. 18–20.
40. A. Bertolino and L. Strigini. On the Use of Testability Measures for Dependability Assessment. *IEEE Transactions on Software Engineering*, February 1996, pp. 97–108.
41. A. Bertolino and E Marchelli. *A Brief Essay on Software Testing*. In *Software Engineering, Vol. 1, The Development Process*, 3rd ed., R. H. Thayer and M. J. Christensen, Eds. Wiley–IEEE Computer Society Press, Hoboken, NJ, 2005.
42. D. Jeffrey and N. Gupta. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. *IEEE Transactions on Software Engineering*, February 2007, pp. 108–123.
43. Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, April 2007, pp. 225–237.
44. W. Masri, A. Podgurski, and D. Leon. An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. *IEEE Transactions on Software Engineering*, July 2007, pp. 454–477.
45. W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of IEEE WESCON*, August 1970, pp. 1–9. Republished in ICSE, Monterey, 1987, pp. 328–338.
46. L. Rising and N. S. Janoff. The Scrum Software Development Process for Small Teams. *IEEE Software*, July/August 2000, pp. 2–8.
47. K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, Redmond, WA, 2004.
48. H. Takeuchi and I. Nonaka. The New Product Development Game. *Harvard Business Review*, Boston, January-February 1986, pp. 1–11.
49. A. P. Mathur. *Foundation of Software Testing*. Pearson Education, New Delhi, 2007.
50. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
51. M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2007.
52. H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom Software Engineering. *IEEE Software*, September 1987, pp. 19–24.

53. S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging User Session Data to Support Web Application Testing. *IEEE Transactions on Software Engineering*, March 2005, pp. 187–202.
54. S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald. Applying Concept Analysis to User-Session-Based Testing of Web Applications. *IEEE Transactions on Software Engineering*, October 2007, pp. 643–657.
55. A. Endress. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering*, June 1975, pp. 140–149.
56. T. J. Ostrand and E. J. Weyuker. Collecting and Categorizing Software Error Data in an Industrial Environment. *Journal of Systems and Software*, November 1984, pp. 289–300.
57. K. Beck. *Test-Driven Development*. Addison-Wesley, Reading, MA, 2003.
58. D. Lemont. *CEO Discussion—From Start-up to Market Leader—Breakthrough Milestones*. Ernst and Young Milestones, Boston, May 2004, pp. 9–11.
59. G. Stark, R. C. Durst, and C. W. Vowell. Using Metrics in Management Decision Making. *IEEE Computer*, September 1994, pp. 42–48.
60. B. Kitchenham and S. L. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, January 1996, pp. 12–21.
61. J. Kilpatrick. Lean Principles. http://www.mep.org/textfiles/LeanPrinciples.pdf, 2003, pp. 1–5.

Exercises

1. Explain the principles of statistical quality control. What are the tools used for this purpose? Explain the principle of a control chart.
2. Explain the concept of lean principles.
3. What is an "Ishikawa" diagram? When should the Ishikawa diagram be used? Provide a procedure to construct an Ishikawa diagram.
4. What is total quality management (TQM)? What is the difference between TQM and TQC?
5. Explain the differences between *validation* and *verification*.
6. Explain the differences between *failure, error*, and *fault*.
7. What is a test case? What are the objectives of testing?
8. Explain the concepts of *unit, integration, system, acceptance*, and *regression* testing.
9. What are the different sources from which test cases can be selected?
10. What is the difference between *fault injection* and *fault simulation*?
11. Explain the differences between *structural* and *functional* testing.
12. What are the strengths and weaknesses of automated testing and manual testing?