SOFTWARE TESTING

Principles and Practices

NARESH CHAUHAN
Professor and Chairman
Department of Computer Engineering
YMCA University of Science and Technology
Faridabad

OXFORD
UNIVERSITY PRESS

Contents

Chapter

1

Introduction to Software Testing

1.1 Introduction

OBJECTIVES

After reading this chapter, you should be able to understand:

- How software testing has evolved over the years
- Myths and facts of software testing
- Software testing is a separate discipline
- Testing is a complete process
- Goals of software testing
- Testing is based on a negative/destructive view
- Model for testing process
- Complete testing is not possible
- Various schools of software testing

Software has pervaded our society, from modern households to spacecrafts. It has become an essential component of any electronic device or system. This is why software development has turned out to be an exciting career for computer engineers in the last 10–15 years. However, software development faces many challenges. Software is becoming complex, but the demand for quality in software products has increased. This rise in customer awareness for quality increases the workload and responsibility of the software development team. That is why software testing has gained so much popularity in the last decade. Job trends have shifted from development to software testing. Today, software quality assurance and software testing courses are offered by many institutions. Organizations have separate testing groups with proper hierarchy. Software development is driven with testing outputs. If the testing team claims the presence of bugs in the software, then the development team cannot release the product.

However, there still is a gap between academia and the demand of industries. The practical demand is that passing graduates must be aware of testing terminologies, standards, and techniques. But students are not aware in most cases, as our universities and colleges do not offer separate software quality and testing courses. They study only software engineering. It can be said that software engineering is a mature discipline today in industry as well as in academia. On the other hand, software testing is mature in industry but not in academia. Thus, this gap must be bridged with separate courses on software quality and testing so that students do not face problems when they go for testing in industries. Today, the ideas and techniques of software testing have become essential knowledge for software developers, testers, and students as well. This book is a step forward to bridge this gap.

We cannot say that the industry is working smoothly, as far as software testing is concerned. While many industries have adopted effective software testing techniques and the development is driven by testing efforts, there are still some loopholes. Industries are dependent on automation of test execution. Therefore, testers also rely on efficient tools. But there may be an instance where automation will not help, which is why they also need to design test cases and execute them manually. Are the testers prepared for this case? This requires testing teams to have a knowledge of testing tactics and procedures of how to design test cases. This book discusses various techniques and demonstrates how to design test cases.

How do industries measure their testing process? Since software testing is a complete process today, it must be measured to check whether the process is suitable for projects. CMM (Capability Maturity Model) has measured the development process on a scale of 1–5 and companies are running for the highest scale. On the same pattern, there should be a measurement program for testing processes. Fortunately, the measurement technique for testing processes has also been developed. But how many managers, developers, testers, and of course students, know that we have a Testing Maturity Model (TMM) for measuring the maturity status of a testing process? This book gives an overview of various test process maturity models and emphasizes the need for these.

Summarizing the above discussion, it is evident that industry and academia should go parallel. Industries constantly aspire for high standards. Our university courses will have no value if their syllabi are not revised vis-à-vis industry requirements. Therefore, software testing should be included as a separate course in our curricula. On the other side, organizations cannot run with the development team looking after every stage, right from requirement gathering to implementation. Testing is an important segment of software development and it has to be thoroughly done. Therefore, there should be a separate testing group with divided responsibilities among the members.

In this chapter, we will trace the evolution of software testing. Once considered as a debugging process, it has now evolved into a complete process. Now we have software testing goals in place to have a clear picture as to why we want to study testing and execute test cases. There has been a misconception right from the evolution of software testing that it can be performed completely. But with time, we have grown out of this view and started focusing on effective testing rather than exhaustive testing. The psychology of a tester plays an important role in software testing. It matters whether one wants to show the absence of errors or their presence in the software. All these issues along with the model of testing, testing process, development of schools of testing, etc. have been discussed. This chapter presents an overview of effective software testing and its related concepts.

1.2 Evolution of Software Testing

In the early days of software development, software testing was considered only a debugging process for removing errors after the development of software. By 1970, the term 'software engineering' was in common use. But software testing was just a beginning at that time. In 1978, G. J. Myers realized the need to discuss the techniques of software testing in a separate subject. He wrote the book *The Art of Software Testing* [2] which is a classic work on software testing. He emphasized that there is a requirement that undergraduate students must learn software testing techniques so that they pass out with the basic knowledge of software testing and do not face problems in the industry. Moreover, Myers discussed the psychology of testing and emphasized that testing should be done with a mindset of finding errors and not to demonstrate that errors are not present.

By 1980, software professionals and organizations started emphasizing on quality. Organizations realized the importance of having quality assurance teams to take care of all testing activities for the project right from the beginning. In the 1990s, testing tools finally came into their own. There was a flood of various tools, which are absolutely vital to adequate testing of software systems. However, they do not solve all problems and cannot replace a testing process.

Gelperin and Hetzel [79] have characterized the growth of software testing with time. Based on this, we can divide the evolution of software testing into the following phases [80] (see Fig. 1.1).

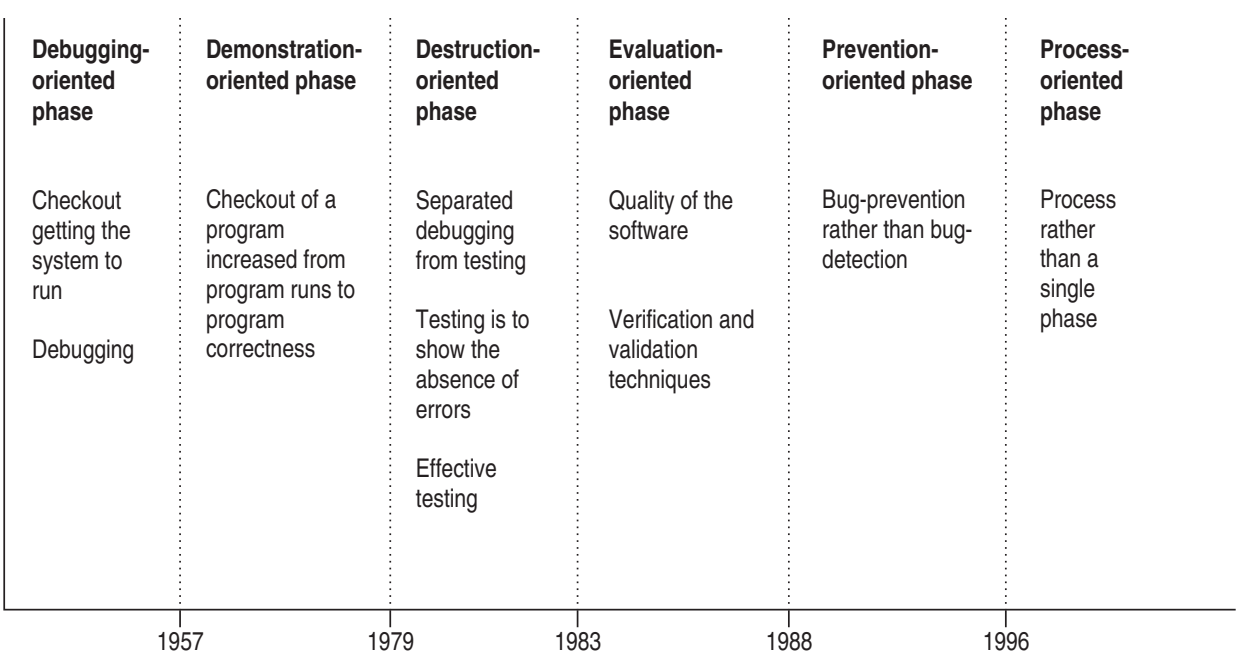

Figure 1.1 Evolution phases of software testing

Debugging-oriented Phase (Before 1957)

This phase is the early period of testing. At that time, testing basics were unknown. Programs were written and then tested by the programmers until they were sure that all the bugs were removed. The term used for testing was *checkout*, focused on getting the system to run. Debugging was a more general term at that time and it was not distinguishable from software testing. Till 1956, there was no clear distinction between software development, testing, and debugging.

Demonstration-oriented Phase (1957–78)

The term 'debugging' continued in this phase. However, in 1957, Charles Baker pointed out that the purpose of checkout is not only to run the software but also to demonstrate the correctness according to the mentioned requirements. Thus, the scope of checkout of a program increased from program runs to program correctness. Moreover, the purpose of checkout was to show the absence of errors. There was no stress on the test case design. In this phase, there was a misconception that the software could be tested exhaustively.

Destruction-oriented Phase (1979–82)

This phase can be described as the revolutionary turning point in the history of software testing. Myers changed the view of testing from 'testing is to show the absence of errors' to 'testing is to find more and more errors.' He separated debugging from testing and stressed on the valuable test cases if they explore more bugs. This phase has given importance to effective testing in comparison to exhaustive testing. The importance of early testing was also realized in this phase.

Evaluation-oriented Phase (1983–87)

With the concept of early testing, it was realized that if the bugs were identified at an early stage of development, it was cheaper to debug them as compared to the bugs found in implementation or post-implementation phases. This phase stresses on the quality of software products such that it can be evaluated at every stage of development. In fact, the early testing concept was established in the form of verification and validation activities which help in producing better quality software. In 1983, guidelines by the National Bureau of Standards were released to choose a set of verification and validation techniques and evaluate the software at each step of software development.

Prevention-oriented Phase (1988–95)

The evaluation model stressed on the concept of bug-prevention as compared to the earlier concept of bug-detection. With the idea of early detection of bugs in earlier phases, we can prevent the bugs in implementation or further phases. Beyond this, bugs can also be prevented in other projects with the experience gained in similar software projects. The prevention model includes test planning, test analysis, and test design activities playing a major role, while the evaluation model mainly relies on analysis and reviewing techniques other than testing.

Process-oriented Phase (1996 onwards)

In this phase, testing was established as a complete process rather than a single phase (performed after coding) in the software development life cycle (SDLC). The testing process starts as soon as the requirements for a project are specified and it runs parallel to SDLC. Moreover, the model for measuring the performance of a testing process has also been developed like CMM. The model for measuring the testing process is known as Testing Maturity Model (TMM). Thus, the emphasis in this phase is also on quantification of various parameters which decide the performance of a testing process.

The evolution of software testing was also discussed by Hung Q. Nguyen and Rob Pirozzi in a white paper [81], in three phases, namely Software Testing 1.0, Software Testing 2.0, and Software Testing 3.0. These three phases discuss the evolution in the earlier phases that we described. According to this classification, the current state-of-practice is Software Testing 3.0. These phases are discussed below.

Software Testing 1.0

In this phase, software testing was just considered a single phase to be performed after coding of the software in SDLC. No test organization was there. A few testing tools were present but their use was limited due to high cost. Management was not concerned with testing, as there was no quality goal.

Software Testing 2.0

In this phase, software testing gained importance in SDLC and the concept of early testing also started. Testing was evolving in the direction of planning the test resources. Many testing tools were also available in this phase.

Software Testing 3.0

In this phase, software testing is being evolved in the form of a process which is based on strategic effort. It means that there should be a process which gives us a roadmap of the overall testing process. Moreover, it should be driven by quality goals so that all controlling and monitoring activities can be performed by the managers. Thus, the management is actively involved in this phase.

1.3 Software Testing—Myths and Facts

Before getting into the details of software testing, let us discuss some myths surrounding it. These myths are there, as this field is in its growing phase.

Myth *Testing is a single phase in SDLC .*

Truth It is a myth, at least in the academia, that software testing is just a phase in SDLC and we perform testing only when the running code of the module is ready. But in reality, testing starts as soon as we get the requirement specifications for the software. And the testing work continues throughout the SDLC, even post-implementation of the software.

Myth *Testing is easy.*

Truth This myth is more in the minds of students who have just passed out or are going to pass out of college and want to start a career in testing. So the general perception is that, software testing is an easy job, wherein test cases are executed with testing tools only. But in reality, tools are there to automate the tasks and not to carry out all testing activities. Testers' job is not easy, as they have to plan and develop the test cases manually and it requires a thorough understanding of the project being developed with its overall design. Overall, testers have to shoulder a lot of responsibility which sometimes make their job even harder than that of a developer.

Myth *Software development is worth more than testing.*

Truth This myth prevails in the minds of every team member and even in freshers who are seeking jobs. As a fresher, we dream of a job as a developer. We get into the organization as a developer and feel superior to other team members. At the managerial level also, we feel happy about the achievements of the developers but not of the testers who work towards the quality of the product being developed. Thus, we have this myth right from the beginning of our career, and testing is considered a secondary job. But testing has now

become an established path for job-seekers. Testing is a complete process like development, so the testing team enjoys equal status and importance as the development team.

Myth *Complete testing is possible.*

Truth This myth also exists at various levels of the development team. Almost every person who has not experienced the process of designing and executing the test cases manually feels that complete testing is possible. Complete testing at the surface level assumes that if we are giving all the inputs to the software, then it must be tested for all of them. But in reality, it is not possible to provide all the possible inputs to test the software, as the input domain of even a small program is too large to test. Moreover, there are many things which cannot be tested completely, as it may take years to do so. This will be demonstrated soon in this chapter. This is the reason why the term 'complete testing' has been replaced with 'effective testing.' Effective testing is to select and run some select test cases such that severe bugs are uncovered first.

Myth *Testing starts after program development.*

Truth Most of the team members, who are not aware of testing as a process, still feel that testing cannot commence before coding. But this is not true. As mentioned earlier, the work of a tester begins as soon as we get the specifications. The tester performs testing at the end of every phase of SDLC in the form of *verification* (discussed later) and plans for the *validation testing* (discussed later). He writes detailed test cases, executes the test cases, reports the test results, etc. Testing after coding is just a part of all the testing activities.

Myth *The purpose of testing is to check the functionality of the software.*

Truth Today, all the testing activities are driven by quality goals. Ultimately, the goal of testing is also to ensure quality of the software. But quality does not imply checking only the functionalities of all the modules. There are various things related to quality of the software, for which test cases must be executed.

Myth *Anyone can be a tester.*

Truth This is the extension of the myth that 'testing is easy.' Most of us think that testing is an intuitive process and it can be performed easily without any training. And therefore, anyone can be a tester. As an established process, software testing as a career also needs training for various purposes, such as to understand (i) various phases of software testing life cycle, (ii) recent techniques to design test cases, (iii) various tools and how to work on them, etc. This is the reason that various testing courses for certified testers are being run.

After having discussed the myths, we will now identify the requirements for software testing. Owing to the importance of software testing, let us first identify the concerns related to it. The next section discusses the goals of software testing.

1.4 Goals of Software Testing

To understand the new concepts of software testing and to define it thoroughly, let us first discuss the goals that we want to achieve from testing. The goals of software testing may be classified into three major categories, as shown in Fig. 1.2.

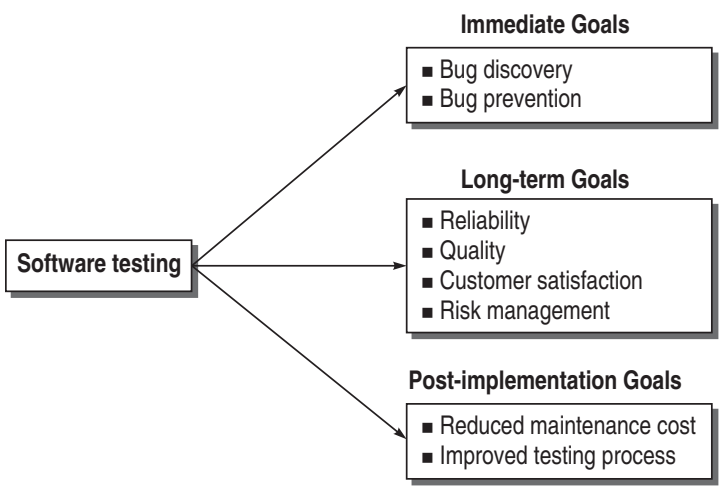

Figure 1.2 Software testing goals

Short-term or immediate goals These goals are the immediate results after performing testing. These goals may be set in the individual phases of SDLC. Some of them are discussed below.

Bug discovery The immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.

Bug prevention It is the consequent action of bug discovery. From the behaviour and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely such that the bugs discovered should not be repeated in later stages or future projects. Though errors cannot be prevented to zero, they can be minimized. In this sense, bug prevention is a superior goal of testing.

Long-term goals These goals affect the product quality in the long run, when one cycle of the SDLC is over. Some of them are discussed here.

Quality Since software is also a product, its quality is primary from the users' point of view. Thorough testing ensures superior quality. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. Though quality depends on various factors, such as correctness, integrity, efficiency, etc., reliability is the major factor to achieve quality. The software should be passed through a rigorous reliability analysis to attain high quality standards. Reliability is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in reliability, in turn, increases the quality, as shown in Fig. 1.3.

Figure 1.3 Testing produces reliability and quality

Customer satisfaction From the users' perspective, the prime concern of testing is customer satisfaction only. If we want the customer to be satisfied with the software product, then testing should be complete and thorough. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements which are otherwise understood. A complete testing process achieves reliability, reliability enhances the quality, and quality in turn, increases the customer satisfaction, as shown in Fig. 1.4.

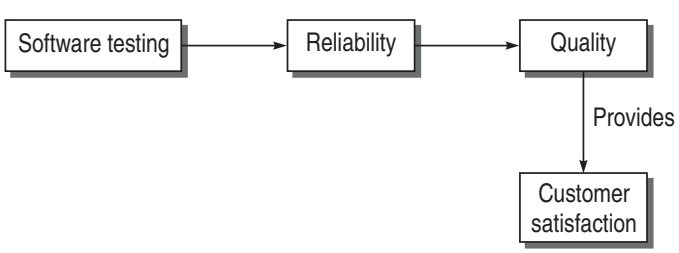

Figure 1.4 Quality leads to customer satisfaction

Risk management Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing its business initiatives. Thus, risk is basically concerned with the business perspective of an organization.

Risks must be controlled to manage them with ease. Software testing may act as a control, which can help in eliminating or minimizing risks (see Fig. 1.5).

Thus, managers depend on software testing to assist them in controlling their business goals. The purpose of software testing as a control is to provide information to management so that they can better react to risk situations [4]. For example, testing may indicate that the software being developed cannot be delivered on time, or there is a probability that high priority bugs will not be resolved by the specified time. With this advance information, decisions can be made to minimize risk situation.

Hence, it is the testers' responsibility to evaluate business risks (such as cost, time, resources, and critical features of the system being developed) and make the same a basis for testing choices. Testers should also categorize the levels of risks after their assessment (like high-risk, moderate-risk, low-risk) and this analysis becomes the basis for testing activities. Thus, risk management becomes the long-term goal for software testing.

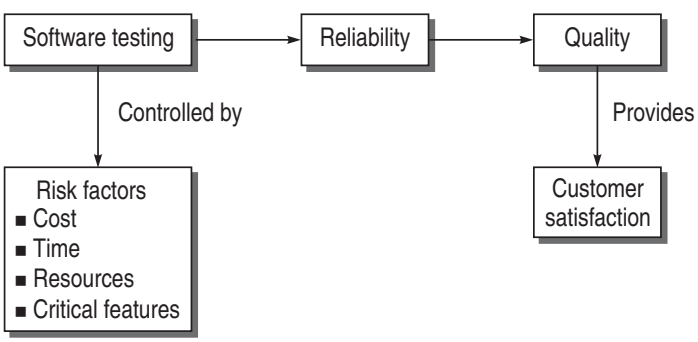

Figure 1.5 Testing controlled by risk factors

Post-implementation goals These goals are important after the product is released. Some of them are discussed here.

Reduced maintenance cost The maintenance cost of any software product is not its physical cost, as the software does not wear out. The only maintenance cost in a software product is its failure due to errors. Post-release errors are costlier to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and in turn, the maintenance cost is reduced.

Improved software testing process A testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analysed to find out snags in the present testing process, which can be rectified in future projects. Thus, the long-term post-implementation goal is to improve the testing process for future projects.

1.5 Psychology for Software Testing

Software testing is directly related to human psychology. Though software testing has not been defined till now, but most frequently, it is defined as,

Testing is the process of demonstrating that there are no errors.

The purpose of testing is to show that the software performs its intended functions correctly. This definition is correct, but partially. If testing is performed keeping this goal in mind, then we cannot achieve the desired goals (described above in the previous section), as we will not be able to test the software as a whole. Myers first identified this approach of testing the software. This approach is based on the human psychology that human beings tend to work according to the goals fixed in their minds. If we have a preconceived assumption that the software is error-free, then consequently, we will design the test cases to show that all the modules run smoothly. But it may hide some bugs. On the other hand, if our goal is to demonstrate that a program has errors, then we will design test cases having a higher probability to uncover bugs.

Thus, if the process of testing is reversed, such that we always presume the presence of bugs in the software, then this psychology of being always suspicious of bugs widens the domain of testing. It means, now we don't think of testing only those features or specifications which have been mentioned in documents like SRS (software requirement specification), but we also think in terms of finding bugs in the domain or features which are understood but not specified. You can argue that, being suspicious about bugs in the software is a negative approach. But, this negative approach is for the benefit of constructive and effective testing. Thus, software testing may be defined as,

Testing is the process of executing a program with the intent of finding errors.

This definition has implications on the psychology of developers. It is very common that they feel embarrassed or guilty when someone finds errors in their software. However, we should not forget that humans are prone to error. We should not feel guilty for our errors. This psychology factor brings the concept that we should concentrate on discovering and preventing the errors and not feel guilt about them. Therefore, testing cannot be a joyous event unless you cast out your guilt.

According to this psychology of testing, a successful test is that which finds errors. This can be understood with the analogy of medical diagnostics of a patient. If the laboratory tests do not locate the problem, then it cannot be regarded as a successful test. On the other hand, if the laboratory test determines the disease, then the doctor can start an appropriate treatment. Thus, in

the destructive approach of software testing, the definitions of successful and unsuccessful testing should also be modified.

1.6 Software Testing Definitions

Many practitioners and researchers have defined software testing in their own way. Some are given below.

Testing is the process of executing a program with the intent of finding errors.

Myers [2]

A successful test is one that uncovers an as-yet-undiscovered error.

Myers [2]

Testing can show the presence of bugs but never their absence.

W. Dijkstra [125]

Program testing is a rapidly maturing area within software engineering that is receiving increasing notice both by computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.

E. Miller[84]

Testing is a support function that helps developers look good by finding their mistakes before anyone else does.

James Bach [83]

Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate.

Cem Kaner [85]

The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems.

Miller [126]

Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e. testing artifacts) in order to measure and improve the quality of the software being tested.

Craig [117]

Since quality is the prime goal of testing and it is necessary to meet the defined quality standards, software testing should be defined keeping in view the quality assurance terms. Here, it should not be misunderstood that the testing team is responsible for quality assurance. But the testing team must

be well aware of the quality goals of the software so that they work towards achieving them.

Moreover, testers these days are aware of the definition that testing is to find more and more bugs. But the problem is that there are too many bugs to fix. Therefore, the recent emphasis is on categorizing the more important bugs first. Thus, software testing can be defined as,

Software testing is a process that detects important bugs with the objective of having better quality software.

1.7 Model for Software Testing

Testing is not an intuitive activity, rather it should be learnt as a process. Therefore, testing should be performed in a planned way. For the planned execution of a testing process, we need to consider every element and every aspect related to software testing. Thus, in the testing model, we consider the related elements and team members involved (see Fig. 1.6).

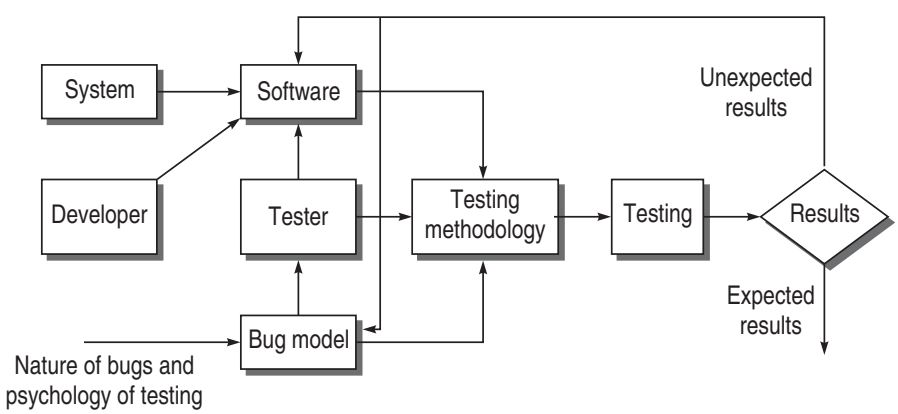

Figure 1.6 Software testing model

The software is basically a part of a system for which it is being developed. Systems consist of hardware and software to make the product run. The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, as a badly written software may be difficult to test. Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed. With suitable testing techniques decided in the testing methodology, testing is performed on the software with a particular goal. If the testing results are

in line with the desired goals, then the testing is successful; otherwise, the software or the bug model or the testing methodology has to be modified so that the desired results are achieved. The following describe the testing model.

Software and Software Model

Software is built after analysing the system in the environment. It is a complex entity which deals with environment, logic, programmer psychology, etc. But a complex software makes it very difficult to test. Since in this model of testing, our aim is to concentrate on the testing process, therefore the software under consideration should not be so complex such that it would not be tested. In fact, this is the point of consideration for developers who design the software. They should design and code the software such that it is testable at every point. Thus, the software to be tested may be modeled such that it is testable, avoiding unnecessary complexities.

Bug Model

Bug model provides a perception of the kind of bugs expected. Considering the nature of all types of bugs, a bug model can be prepared that may help in deciding a testing strategy. However, every type of bug cannot be predicted. Therefore, if we get incorrect results, the bug model needs to be modified.

Testing methodology and Testing

Based on the inputs from the software model and the bug model, testers can develop a testing methodology that incorporates both testing strategy and testing tactics. Testing strategy is the roadmap that gives us well-defined steps for the overall testing process. It prepares the planned steps based on the risk factors and the testing phase. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps. Thus, testing is performed on this methodology. However, if we don't get the required results, the testing plans must be checked and modified accordingly.

All the components described above will be discussed in detail in subsequent chapters.

1.8 Effective Software Testing vs. Exhaustive Software Testing

Exhaustive or complete software testing means that every statement in the program and every possible path combination with every possible combination of data must be executed. But soon, we will realize that exhaustive testing is out of scope. That is why the questions arise: (i) When are we done with testing? or (ii) How do we know that we have tested enough? There may be

many answers for these questions with respect to time, cost, customer, quality, etc. This section will explore that exhaustive or complete testing is not possible. Therefore, we should concentrate on effective testing which emphasizes efficient techniques to test the software so that important features will be tested within the constrained resources.

The testing process should be understood as a domain of possible tests (see Fig. 1.7). There are subsets of these possible tests. But the domain of possible tests becomes infinite, as we cannot test every possible combination.

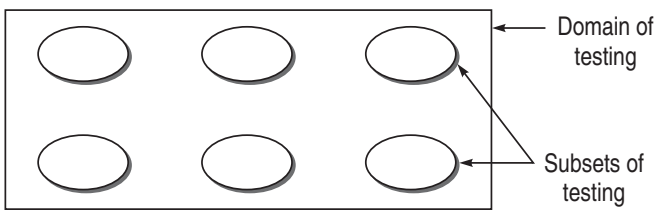

Figure 1.7 Testing domain

This combination of possible tests is infinite in the sense that the processing resources and time are not sufficient for performing these tests. Computer speed and time constraints limit the possibility of performing all the tests. Complete testing requires the organization to invest a long time which is not cost-effective. Therefore, testing must be performed on selected subsets that can be performed within the constrained resources. This selected group of subsets, but not the whole domain of testing, makes effective software testing. Effective testing can be enhanced if subsets are selected based on the factors which are required in a particular environment.

Now, let us see in detail why complete testing is not possible.

The Domain of Possible Inputs to the Software is too Large to Test

If we consider the input data as the only part of the domain of testing, even then, we are not able to test the complete input data combination. The domain of input data has four sub-parts: (a) valid inputs, (b) invalid inputs, (c) edited inputs, and (d) race condition inputs (See Fig. 1.8)

Valid inputs It seems that we can test every valid input on the software. But look at a very simple example of adding two-digit two numbers. Their range is from –99 to 99 (total 199). So the total number of test case combinations will be $199 \times 199 = 39601$. Further, if we increase the range from two digits to four-digits, then the number of test cases will be 399,960,001. Most addition programs accept 8 or 10 digit numbers or more. How can we test all these combinations of valid inputs?

Invalid inputs Testing the software with valid inputs is only one part of the input sub-domain. There is another part, invalid inputs, which must be

tested for testing the software effectively. The important thing in this case is the behaviour of the program as to how it responds when a user feeds invalid inputs. The set of invalid inputs is also too large to test. If we consider again the example of adding two numbers, then the following possibilities may occur from invalid inputs:

(i) Numbers out of range
(ii) Combination of alphabets and digits
(iii) Combination of all alphabets
(iv) Combination of control characters
(v) Combination of any other key on the keyboard

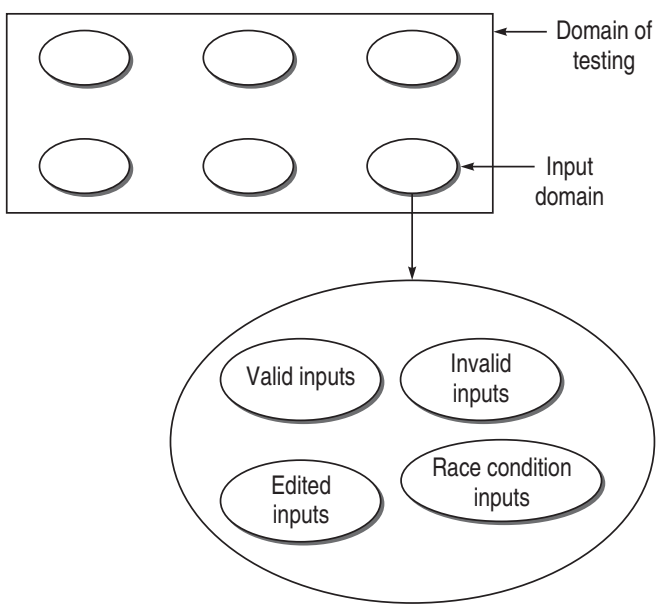

Figure 1.8 Input domain for testing

Edited inputs If we can edit inputs at the time of providing inputs to the program, then many unexpected input events may occur. For example, you can add many spaces in the input, which are not visible to the user. It can be a reason for non-functioning of the program. In another example, it may be possible that a user is pressing a number key, then Backspace key continuously and finally after sometime, he presses another number key and Enter. Its input buffer overflows and the system crashes.

The behaviour of users cannot be judged. They can behave in a number of ways, causing defect in testing a program. That is why edited inputs are also not tested completely.

Race condition inputs The timing variation between two or more inputs is also one of the issues that limit the testing. For example, there are two input events,

A and B. According to the design, A precedes B in most of the cases. But, B can also come first in rare and restricted conditions. This is the race condition, whenever B precedes A. Usually the program fails due to race conditions, as the possibility of preceding B in restricted condition has not been taken care, resulting in a race condition bug. In this way, there may be many race conditions in the system, especially in multiprocessing systems and interactive systems. Race conditions are among the least tested.

There are too Many Possible Paths Through the Program to Test

A program path can be traced through the code from the start of a program to its termination. Two paths differ if the program executes different statements in each, or executes the same statements but in different order. A testing person thinks that if all the possible paths of control flow through the program are executed, then possibly the program can be said to be completely tested. However, there are two flaws in this statement.

(i) The number of unique logic paths through a program is too large. This was demonstrated by Myers[2] with an example shown in Fig. 1.9. It depicts a 10–20 statements program consisting of a DO loop that iterates up to 20 times. Within the body of the DO loop is a set of nested IF statements. The number of all the paths from point A to B is approximately 10^{14}. Thus, all these paths cannot be tested, as it may take years of time.

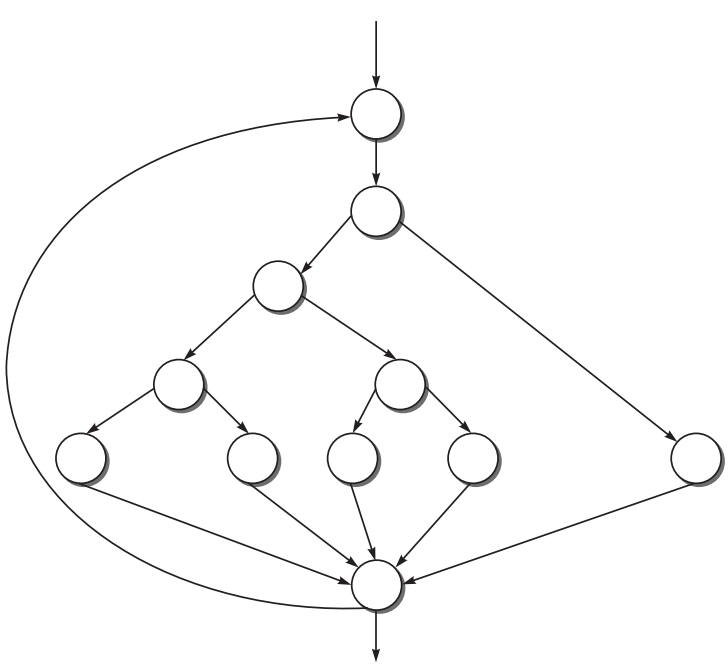

Figure 1.9 Sample flow graph 1

See another example for the code fragment shown in Fig. 1.10 and its corresponding flow graph in Fig. 1.11 (We will learn how to convert the program into a flow graph in Chapter 5).

```
for (int i = 0; i < n; ++i)
{
if (m >=0)
     x[i] = x[i] + 10;
else
     x[i] = x[i] - 2;
}
```

Figure 1.10 Sample code fragment

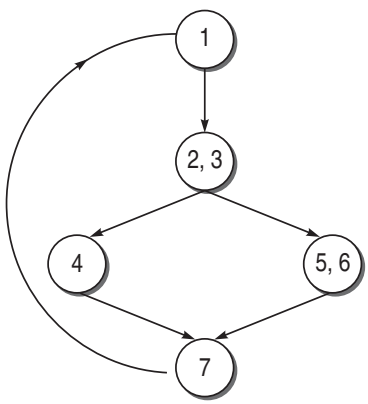

Figure 1.11 Example flow graph 2

Now calculate the number of paths in this fragment. For calculating the number of paths, we must know how many paths are possible in one iteration. Here in our example, there are two paths in one iteration. Now the total number of paths will be $2^n + 1$, where n is the number of times the loop will be carried out, and 1 is added, as the `for` loop will exit after its looping ends and it terminates. Thus, if n is 20, then the number of paths will be $2^{20} + 1$, i.e. 1048577. Therefore, all these paths cannot be tested, as it may take years.

(ii) The complete path testing, if performed somehow, does not guarantee that there will *not* be errors. For example, it does not claim that a program matches its specification. If one were asked to write an ascending order sorting program but the developer mistakenly produces a descending order program, then exhaustive path testing will be of little value. In another case, a program may be incorrect because of missing paths. In this case, exhaustive path testing would not detect the missing path.

Every Design Error Cannot be Found

Manna and Waldinger [15] have mentioned the following fact: 'We can never be sure that the specifications are correct.' How do we know that the specifications are achievable? Its consistency and completeness must be proved, and in general, that is a provably unsolvable problem [9]. Therefore, specification errors are one of the major reasons that make the design of the software faulty. If the user requirement is to have measurement units in inches and the specification says that these are in meters, then the design will also be in meters. Secondly, many user interface failures are also design errors.

The study of these limitations of testing shows that the domain of testing is infinite and testing the whole domain is just impractical. When we leave a single test case, the concept of complete testing is abandoned. But it does not mean that we should not focus on testing. Rather, we should shift our attention from exhaustive testing to effective testing. Effective testing provides the flexibility to select only the subsets of the domain of testing based on project priority such that the chances of failure in a particular environment is minimized.

1.9 Effective Testing is Hard

We have seen the limitations of exhaustive software testing which makes it nearly impossible to achieve. Effective testing, though not impossible, is hard to implement. But if there is careful planning, keeping in view all the factors which can affect it, then it is implementable as well as effective. To achieve that planning, we must understand the factors which make effective testing difficult. At the same time, these factors must be resolved. These are described as follows.

Defects are hard to find The major factor in implementing effective software testing is that many defects go undetected due to many reasons, e.g. certain test conditions are never tested. Secondly, developers become so familiar with their developed system that they overlook details and leave some parts untested. So a proper planning for testing all the conditions should be done and independent testing, other than that done by developers, should be encouraged.

When are we done with testing This factor actually searches for the definition of effective software testing. Since exhaustive testing is not possible, we don't know what should be the criteria to stop the testing process. A software engineer needs more rigorous criteria for determining when sufficient testing has been performed. Moreover, effective testing has the limiting factor of cost, time, and personnel. In a nutshell, the criteria should be developed for enough

testing. For example, features can be prioritized which must be tested within the boundary of cost, time, and personnel of the project.

1.10 Software Testing as a Process

Since software development is an engineering activity for a quality product, it consists of many processes. As it was seen in testing goals, software quality is the major driving force behind testing. Software testing has also emerged as a complete process in software engineering (see Fig. 1.12). Therefore, our major concern in this text is to show that testing is not just a phase in SDLC normally performed after coding, rather software testing is a process which runs parallel to SDLC. In Fig. 1.13, you can see that software testing starts as soon as the requirements are specified. Once the SRS document is prepared, testing process starts. Some examples of test processes, such as test plan, test design, etc. are given. All the phases of testing life cycle will be discussed in detail in the next chapter.

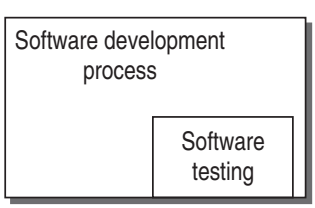

Figure 1.12 Testing process emerged out of development process

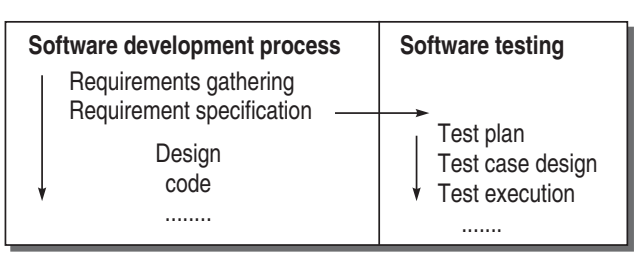

Figure 1.13 Testing process runs parallel to software process

Software testing process must be planned, specified, designed, implemented, and quantified. Testing must be governed by the quality attributes of the software product. Thus, testing is a dual-purpose process, as it is used to detect bugs as well as to establish confidence in the quality of software.

An organization, to ensure better quality software, must adopt a testing process and consider the following points:

- Testing process should be organized such that there is enough time for important and critical features of the software.

- Testing techniques should be adopted such that these techniques detect maximum bugs.
- Quality factors should be quantified so that there is a clear understanding in running the testing process. In other words, the process should be driven by quantified quality goals. In this way, the process can be monitored and measured.
- Testing procedures and steps must be defined and documented.
- There must be scope for continuous process improvement.

All the issues related to testing process will be discussed in succeeding chapters.

1.11 Schools of Software Testing

Software testing has also been classified into some views according to some practitioners. They call these views or ideas as *schools of testing*. The idea of schools of testing was given by Bret Pettichord [82]. He has proposed the following schools:

Analytical School of Testing

In this school of testing, software is considered as a logical artifact. Therefore, software testing techniques must have a logico-mathematical form. This school requires that there must be precise and detailed specifications for testing the software. Moreover, it provides an objective measure of testing. After this, testers are able to verify whether the software conforms to its specifications. Structural testing is one example for this school of testing. Thus, the emphasis is on testing techniques which should be adopted.

This school defines software testing as a branch of computer science and mathematics.

Standard School of Testing

The core beliefs of this school of testing are:

1. Testing must be managed (for example, through traceability matrix. It will be discussed in detail in succeeding chapters). It means the testing process should be predictable, repeatable, and planned.
2. Testing must be cost-effective.
3. Low-skilled workers require direction.
4. Testing validates the product.
5. Testing measures development progress.

Thus, the emphasis is on measurement of testing activities to track the development progress.

This school defines software testing as a managed process.

The implications of this school are:

1. There must be clear boundaries between testing and other activities.
2. Plans should not be changed as it complicates progress tracking.
3. Software testing is a complete process.
4. There must be some test standards, best practices, and certification.

Quality School of Testing

The core beliefs of this school of testing are:

1. Software quality requires discipline.
2. Testing determines whether development processes are being followed.
3. Testers may need to police developers to follow the rules.
4. Testers have to protect the users from bad software.

Thus, the emphasis is to follow a good process.

This school defines software testing as a branch of software quality assurance.

The implications of this school are:

1. It prefers the term 'quality assurance' over 'testing.'
2. Testing is a stepping stone to 'process improvement.'

Context-driven School of Testing

This school is based on the concept that testing should be performed according to the context of the environment and project. Testing solutions cannot be the same for every context. For example, if there is a high-cost real-time defense project, then its testing plans must be different as compared to any daily-life low-cost project. Test plan issues will be different for both projects. Therefore, testing activities should be planned, designed, and executed keeping in view the context of environment in which testing is to be performed. The emphasis is to select a testing type that is valuable. Thus, context-driven testing can be defined as the testing driven by environment, type of project, and the intended use of software.

The implications of this school are:

1. Expect changes. Adapt testing plans based on test results.
2. Effectiveness of test strategies can only be determined with field research.

3. Testing research requires empirical and psychological study.
4. Focus on skill over practice.

Agile School of Testing

This type of school is based on testing the software which is being developed by iterative method of development and delivery. In this type of process model, the software is delivered in a short span of time; and based on the feedback, more features and capabilities are added. The focus is on satisfying the customer by delivering a working software quickly with minimum features and then improvising on it based on the feedback. The customer is closely related to the design and development of the software. Since the delivery timelines are short and new versions are built by modifying the previous one, chances of introducing bugs are high during the changes done to one version. Thus, regression testing becomes important for this software. Moreover, test automation also assumes importance to ensure the coverage of testing in a short span of time.

It can be seen that agile software development faces various challenges. This school emphasizes on all the issues related to agile testing.

1.12 Software Failure Case Studies

At the end of this chapter, let us discuss a few case studies that highlight the failures of some expensive and critical software projects. These case studies show the importance of software testing. Many big projects have failed in the past due to lack of proper software testing. In some instances, the product was replaced without question. The concerned parties had to bear huge losses in every case. It goes on to establish the fact that the project cost increases manifold if a product is launched without proper tests being performed on it. These case studies emphasize the importance of planning the tests, designing, and executing the test cases in a highly prioritized way, which is the central theme of this book.

Air Traffic Control System Failure (September 2004)

In September 2004, air traffic controllers in the Los Angeles area lost voice contact with 800 planes allowing 10 to fly too close together, after a radio system shut down. The planes were supposed to be separated by five nautical miles laterally, or 2,000 feet in altitude. But the system shut down while 800 planes were in the air, and forced delays for 400 flights and the cancellations of 600 more. The system had voice switching and control system, which gives controllers a touch-screen to connect with planes in flight and with controllers across the room or in distant cities.

The reason for failure was partly due to a 'design anomaly' in the way Microsoft Windows servers were integrated into the system. The servers were timed to shut down after 49.7 days of use in order to prevent a data overload. To avoid this automatic shutdown, technicians are required to restart the system manually every 30 days. An improperly trained employee failed to reset the system, leading it to shut down without warning.

Welfare Management System Failure (July 2004)

It was a new government system in Canada costing several hundred million dollars. It failed due to the inability to handle a simple benefits rate increase after being put into live operation. The system was not given adequate time for system and acceptance testing and never tested for its ability to handle a rate increase.

Northeast Blackout (August 2003)

It was the worst power system failure in North American history. The failure involved loss of electrical power to 50 million customers, forced shutdown of 100 power plants and economic losses estimated at $6 billion. The bug was reportedly in one utility company's vendor-supplied power monitoring and management system. The failures occurred when multiple systems trying to access the same information at once got the equivalent of busy signals. The software should have given one system precedent. The error was found and corrected after examining millions of lines of code.

Tax System Failure (March 2002)

This system was Britain's national tax system which failed in 2002 and resulted in more than 100,000 erroneous tax overcharges. It was suggested in the error report that the integration testing of multiple parts could not be done.

Mars Polar Lander Failure (December 1999)

NASA's Mars Polar Lander was to explore a unique region of the red planet; the main focus was on climate and water. The spacecraft was outfitted with a robot arm which was capable of digging into Mars in search for near-surface ice. It was supposed to gently set itself down near the border of Mars' southern polar cap. But it couldn't be successful to touch the surface of Mars. The communication was lost when it was 1800 meters away from the surface of Mars.

When the Lander's legs started opening for landing on Martian surface, there were vibrations which were identified by the software. This resulted in the vehicle's descent engines being cut off while it was still 40 meters above the surface, rather than on touchdown as planned. The software design failed to take into account that a touchdown signal could be detected before the

Lander actually touched down. The error was in design. It should have been configured to disregard touchdown signals during the deployment of the Lander's legs.

Mars Climate Orbiter Failure (September 1999)

Mars Climate Orbiter was one of a series of missions in a long-term program of Mars exploration managed by the Jet Propulsion Laboratory for NASA's Office of Space Science, Washington, DC. Mars Climate Orbiter was to serve as a communications relay for the Mars Polar Lander mission. But it disappeared as it began to orbit Mars. Its cost was about $125 million. The failure was due to an error in transfer of information between a team in Colorado and a team in California. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit. One team used English units (e.g. inches, feet, and pounds), while the other team used metric units for a key spacecraft operation.

Stock Trading Service Failure (February 1999)

This was an online US stock trading service which failed during trading hours several times over a period of days in February 1999. The problem found was due to bugs in a software upgrade intended to speed online trade confirmations.

Intel Pentium Bug (April 1997)

Intel Pentium was also observed with a bug that is known as Dan-0411 or Flag Erratum. The bug is related to the operation where conversion of floating point numbers is done into integer numbers. All floating-point numbers are stored inside the microprocessor in an 80-bit format. Integer numbers are stored externally in two different sizes, i.e. 16 bits for short integers and 32 bits for long integers. It is often desirable to store the floating-point numbers as integer numbers. When the converted numbers won't fit the integer size range, a specific error flag is supposed to be set in a floating point status register. But the Pentium II and Pentium Pro fail to set this error flag in many cases.

The Explosion of Ariane 5 (June 1996)

Ariane 5 was a rocket launched by the European Space Agency. On 4 June 1996, it exploded at an altitude of about 3700 meters just 40 seconds after its lift-off from Kourou, French Guiana. The launcher turned off its flight path, broke up and exploded. The rocket took a decade of development time with a cost of $7 billion. The destroyed rocket and its cargo were valued at $500 million. The failure of Ariane was caused due to the complete loss of guidance and altitude information, 37 seconds after the start of main engine ignition sequence (30 seconds after lift-off).

A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It was found that the cause of the failure was a software error in the inertial reference system (SRI). The internal SRI software exception was caused during the execution of a data conversion from 64-bit floating point to 16-bit signed integer value. A 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. The number was larger than 32,767, the largest integer stored in a 16-bit signed integer; and thus the conversion failed. The error was due to specification and design errors in the software of the inertial reference system.

SUMMARY

This chapter emphasizes that software testing has emerged as a separate discipline. Software testing is now an established process. It is driven largely by the quality goals of the software. Thus, testing is the critical element of software quality. This chapter shows that testing cannot be performed with an optimistic view that the software does not contain errors. Rather, testing should be performed keeping in mind that the software always contains errors.

A misconception has prevailed through the evolution of software testing that complete testing is possible, but it is not true. Here, it has been demonstrated that complete testing is not possible. Thus, the term 'effective software testing' is becoming more popular as compared to 'exhaustive' or 'complete testing'. The chapter gives an overview of software testing discipline along with definitions of testing, model for testing, and different schools of testing. To realize the importance of effective software testing as a separate discipline, some case studies showing the software failures in systems have also been discussed.

Let us quickly review the important concepts described in this chapter.

- Software testing has evolved through many phases, namely (i) debugging-oriented phase, (ii) demonstration-oriented phase, (iii) destruction-oriented phase, (iv) evaluation-oriented phase, (v) prevention-oriented phase, and (vi) process-oriented phase.
- There is another classification for evolution of software testing, namely Software testing 1.0, Software testing 2.0, and Software testing 3.0.
- Software testing goals can be partitioned into following categories:

  1. Immediate goals
     - Bug discovery
     - Bug prevention
  2. Long-term goals
     - Reliability
     - Quality
     - Customer satisfaction
     - Risk management

3. Post-implementation goals
    - Reduced maintenance cost
    - Improved testing process

- Testing should be performed with a mindset of finding bugs. This suspicious strategy (destructive approach) helps in finding more and more bugs.
- Software testing is a process that detects important bugs with the objective of having better quality software.
- Exhaustive testing is not possible due to the following reasons:
    - It is not possible to test every possible input, as the input domain is too large.
    - There are too many possible paths through the program to test.
    - It is difficult to locate every design error.
- Effective software testing, instead of complete or exhaustive testing, is adopted such that critical test cases are covered first.
- There are different views on how to perform testing which have been categorized as schools of software testing, namely (i) analytical school, (ii) standard school, (iii) quality school, (iv) context school, and (v) agile school.
- Software testing is a complete process like software development.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Bug discovery is a _____ goal of software testing.
    (a) Long-term
    (b) Short-term
    (c) Post-implementation
    (d) All
2. Customer satisfaction and risk management are _____ goals of software testing.
    (a) Long-term
    (b) Short-term
    (c) Post-implementation
    (d) All
3. Reduced maintenance is a _____ goal of software testing.
    (a) Long-term
    (b) Short-term
    (c) Post-implementation
    (d) All

4. Software testing produces _____.
   (a) Reliability
   (b) Quality
   (c) Customer Satisfaction
   (d) All
5. Testing is the process of _____ errors.
   (a) Hiding
   (b) Finding
   (c) Removing
   (d) None
6. Complete testing is _____.
   (a) Possible
   (b) Impossible
   (c) None
7. The domain of possible inputs to the software is too _____ to test.
   (a) Large
   (b) Short
   (c) none
8. The set of invalid inputs is too _____ to test.
   (a) Large
   (b) Short
   (c) none
9. Race conditions are among the _____ tested.
   (a) Most
   (b) Least
   (c) None
10. Every design error _____ be found.
    (a) Can
    (b) Can definitely
    (c) Cannot
    (d) None

Review Questions

1. How does testing help in producing quality software?
2. 'Testing is the process of executing a program with the intent of finding errors.' Comment on this statement.
3. Differentiate between effective and exhaustive software testing.

4. Find out some myths related to software testing, other than those described in this chapter.
5. 'Every design error cannot be found.' Discuss this problem in reference to some project.
6. 'The domain of possible inputs to the software is too large to test.' Demonstrate using some example programs.
7. 'There are too many possible paths through the program to test.' Demonstrate using some example programs.
8. What are the factors for determining the limit of testing?
9. Explore some more software failure case studies other than those discussed in this chapter.