CHAPTER – 2
Software Testing

2.1 Introduction

Algorithm is a concrete representation of a program which is implemented by the developers. The developers of this program are naturally concerned with the correctness and performance of the implementation. Software engineers must ensure that their software systems achieve an appropriate level of quality. Software verification is the process of ensuring that a program meets its intended specification [KANE 93]. One technique that can assist during the specification, design, and implementation of a software system is software verification through correctness proof. Software testing, or the process of assessing the functionality and correctness of a program through execution or analysis, is another alternative for verifying a software system. Software testing can be appropriately used in conjunction with correctness proofs and other types of formal approaches in order to develop high quality software systems [BOWI: 95]. Yet, it is also possible to use software testing techniques in isolation from program correctness proofs or other formal methods.

Software testing is not a “silver bullet” that can guarantee the production of high quality software systems. While a “correct” correctness proof demonstrates that a software system (which exactly meets its specification) will always operate in a given manner, software testing that is not fully exhaustive can only suggest the presence of flaws and cannot prove their absence. Moreover, it is impossible to completely test an application because (1) the domain of program inputs is too large, (2) there are too many possible input paths and (3) design and specification issues are difficult to test [KANE 93].
The first and second points present obvious complications and the last point highlights the difficulty of determining if the specification of a problem solution and the design of its implementation are also correct.

Using a thought experiment developed by Beizer [BEIZ 90], it can explore the first assertion by assuming that a method that takes a String of ten characters as input and performs some arbitrary operation on the String. In order to test this simple function, exhaustively, it requires input $2^{10}$ strings and determines if they produce the appropriate output. The testing of our hypothetical method might also involved the usage of anomalous input, like strings consisting of more or less than ten characters, to determine the robustness of the operation. In this situation, the total number of inputs would be significantly greater than $2^{10}$. Therefore, we can conclude that exhaustive testing is an intractable problem since it is impossible to solve with a polynomial-time algorithm [BIND 99]. The difficulties alluded to by the second assertion are exacerbated by the fact that certain execution paths in a program could be infeasible. Finally, software testing is an algorithmically unsolvable problem since there may be input values for which the program does not halt [BIND 99, BEIZ 90].

While software testing is certainly faced with inherent limitations, there are also a number of practical considerations that can hinder the application of a testing technique. For example, some programming languages might not readily support a selected testing approach, a test automation framework might not easily facilitate the automatic execution of certain types of test suites, or there could be a lack of tool support to test with respect to a specific test adequacy criterion. Even though any testing effort will be faced with significant essential and accidental
limitations, the rigorous, consistent, and intelligent application of appropriate software testing techniques can improve the quality of the application under development.

2.2 Principles of Software Testing

2.2.1 Terminology

The IEEE standard defines a failure as the external, incorrect behavior of a program [IEEE 96]. Traditionally, the anomalous behavior of a program is observed when incorrect output is produced or a runtime failure occurs. Furthermore, it defines a fault as a collection of program source code statements that causes a failure. Finally, an error is a mistake made by a programmer during the implementation of a software system [IEEE 96]. We will use P to denote the program under test and F to represent the specification that describes the behavior of P. The purpose of software testing is to reveal software faults in order to ensure that they do not manifest themselves as runtime failures during program usage.

Figure 2.1 provides a useful hierarchical decomposition of different testing techniques and their relationship to different classes of test adequacy criteria. While our hierarchy generally follows the definitions provided by Binder and Zhu et. al, it is important to note that other decompositions of the testing process are possible [BIND 99]. This chapter focuses on execution-based software testing techniques. However, it is also possible to perform non-execution-based software testing through the usage of software inspections [FAGA 76]. During a software inspection, software engineers examine the source code of a system and any documentation that accompanies the system. A software inspector can be guided by a software inspection checklist that highlights some of the important questions.
that should be asked about the artifact under examination [BRYK 99]. While an inspection checklist is more sophisticated than an ad-hoc software inspection technique, it does not dictate how an inspector should locate the required information in the artifacts of a software system. Scenario based reading techniques, such as Perspective-Based Reading (PBR), enable a more focused review of software artifacts by requiring inspectors to assume the perspective of different classes of program users [LAIT 99].

Since the selected understanding of adequacy is central to any testing effort, the types of tests within Test Suite (T) will naturally vary based upon the chosen adequacy criterion C. As shown in Figure 1, all execution-based testing techniques are either program-based, specification-based, or combined [BIND 99]. A program-based testing approach relies upon the structure and attributes of P’s source code to create T. A specification-based testing technique simply uses F’s statements about the functional and/or non-functional requirements for P to create the desired test suite.

![Diagram: Hierarchy of Software Testing techniques](image)

Fig: 2.1 Hierarchy of Software Testing techniques

A combined testing technique creates a test suite T that is influenced by both program-based and specification-based testing approaches [BIND 99].
Moreover, the tests in T can be classified based upon whether they are white-box, black-box, or grey-box test cases. Specification-based test cases are black-box tests that were created without knowledge of P's source code. White-box (or, alternatively, glass-box) test cases consider the entire source code of P, while so-called grey-box tests only consider a portion of P's source code. Both white-box and grey-box approaches to testing would be considered program-based or combined techniques.

A complementary decomposition of the notion of software testing is useful to highlight the centrality of the chosen test adequacy criterion. The tests within T can be viewed based upon whether they are "good" with respect to a structurally-based, fault-based, or error-based adequacy criterion [BIND 99]. A structurally-based criterion requires the creation of a test suite T that solely requires the exercising of certain control structures and variables within P. Thus, it is clear that structurally-based test adequacy criterion require program-based testing. Fault-based test adequacy criterion attempt to ensure that P does not contain the types of faults that are commonly introduced into software systems by programmers [DEMI 91, BIND 99, MORE 90]. Traditionally, fault-based criterion are associated with program-based testing approaches. However, Richardson et al. have also described fault-based testing techniques that attempt to reveal faults in P or faults in P that are associated with misunderstandings of [RICH 89]. Therefore, a fault-based adequacy criterion C can require either program-based, specification-based, or combined testing techniques. Finally, error-based testing approaches rely upon a C that requires T to demonstrate that P does not deviate from F in any typical
fashion. Thus, error-based adequacy criteria necessitate specification-based testing approaches.

2.2.2 Model of Execution-Based Software Testing

Figure 2.2 provides a model of execution-based software testing. Since there are different understandings of the process of testing software, it is important to note that our model is only one valid and useful view of software testing. Using the notation established in Section 2.2.1, this model of software testing takes a system under test, \( P \), and a test adequacy criterion, \( C \), as input.

![Fig: 2.2 A Model of Software Testing Process](image)
This view of the software testing process is iterative in nature. That is, the initial creation and execution of T against P can be followed by multiple refinements of T and subsequent re-testings of P. Ideally, the testing process will stop iterating when the tests within test suite T have met the adequacy criterion C and the testing effort has established the desired level of confidence in the quality of P [BIND 99]. In practice, testing often stops when a release deadline is reached, monetary resources are exhausted, or the developers have an intuitive sense that P has reached an acceptable level of quality. Yet, it is important to note that even if the testing effort stops when T meets C, there is no guarantee that T has isolated all of the defects within P. Since C traditionally represents a single view of test case quality, it is often important to use multiple adequacy criterion and resist the temptation to allow the selected C(s) to exclusively guide the testing effort [MARI 99]. The formulation of a test adequacy criterion is a function of a chosen representation of P and a specific understanding of the "good" qualities that T should represent. The test specification stage analyzes a specific P in light of a chosen C in order to construct a listing of the tests that must be provided to create a completely adequate test suite. That is, the test case specification stage shown in Figure 2.2 is responsible for making C directly apply to the system under test. Once a test case specification tool has created test case descriptions for P, the test case generation phase can begin.

After the test cases have been generated, it is possible to perform test execution. Once again, the execution of the tests within T can be performed in a manual or automated fashion. Also, the results from the execution of the tests can be analyzed in either an automated or manual fashion to determine if each
individual test case passed or failed. The executable test cases that were constructed during the generation phase can be analyzed by a test adequacy evaluator that measures the quality of \( T \) with respect to the test case descriptions produced by the test specifier. Of course, the test results from the execution phase and the adequacy measurements produced by the evaluator can be used to change the chosen adequacy criteria and/or augment the listing of test case descriptions that will be used during subsequent testing.

The iterative process of testing can continue throughout the initial development of \( P \). However, it is also important to continue the testing of \( P \) after the software application has been released and it enters the maintenance phase of the software lifecycle [SOMM 00]. Regression testing is an important software maintenance activity that attempts to ensure that the addition of new functionality and/or the removal of program faults does not negatively impact the correctness of \( P \). The regression testing process can rely upon the existing test cases and the adequacy measurements for these tests to iteratively continue all of the previously mentioned stages [ONOM 98].

2.3 Data Flow-Based Criterion

For a standard program, the occurrence of a variable on the left hand side of an assignment statement is called a definition of this variable. Also, the occurrence of a variable on the right hand side of an assignment statement is called a computation-use (or c-use) of this variable. Finally, when a variable appears in the predicate of a conditional logic statement or an iteration construct, we call this a predicate-use (or p-use) of the variable.
In [SAND 82, SAND 85], the authors propose a family of test adequacy measures based upon data flow information in a program. Among their test adequacy measures, the all-uses data flow adequacy criteria requires a test suite to cover all of the def-c-use and def-p-use associations in a program. The all-uses criterion is commonly used as the basis for definition-use testing. Alternatively, the all-c-uses criterion requires the coverage of all the c-use associations for a given method under test and the all-p-uses adequacy criterion requires the coverage of all the p-use associations. Furthermore, the all-dupaths coverage criterion requires the coverage of all the paths from the definition to a usage of a program variable [SAND 82, SAND 85]. The all-c-uses/some-p-uses and the all-p-uses/some-c-uses test adequacy criteria are combinations of the all-c-uses and all-p-uses metrics. The all-DU test adequacy criterion is a modified version of all-uses that simply requires the coverage of def-use associations without distinguishing between a p-use and a c-use [IUTC 94].

2.4 Test Case Generation

The generation of test cases can be performed in a manual or automated fashion. Frequently, manual test generation involves the construction of test cases in a general purpose programming language or a test case specification language. Alternatively, test cases can be “recorded” or “captured” by simply using the program under test and monitoring the actions that were taken during usage [STEV 00].

An automated solution to the test data generation problem attempts to automatically create a T that will fulfill selected adequacy criterion C when it is used to test program P. While it is possible for C to be an error-based criterion,
automated test data generation is more frequently performed with fault-based and structurally-based test adequacy criteria. There are several different techniques that can be used to automatically generate test data. Random, Symbolic, and Dynamic Test Data Generation approaches are all alternatives that can be used to construct a T that adequately tests P. A random test data generation approach relies upon a random number generator to simply generate test input values. For complex (and sometimes quite simple) programs, it is often difficult for random test data generation techniques to produce adequate test suites [KORI: 96].

2.5 Test Execution

The execution of a test suite can occur either in a manual or automated fashion. For example, the test case descriptions that are the result of the test selection process could be manually executed against the program under test. However, we will focus on the automated execution of test cases and specifically examine the automated testing issues associated with the JUnit test automation framework [HIGH 01, JEFF 99].

2.6 Regression Testing

After a software system experiences changes in the form of bug fixes or additional functionality, a software maintenance activity known as regression testing can be used to determine if these changes introduced defects. As described in Section 2.2.1 and depicted in Figure 2.2, the regression testing process applies all of the other software testing stages whenever the program under test changes. The creation, maintenance, and execution of a regression test suite helps to ensure that the evolution of an application does not result in lower quality software. The industry experiences noted by Onoma et al. indicate that regression testing often
has a strong positive influence on software quality [ONOM 98]. Indeed, the importance of regression testing is well understood. However, as noted by Beizer, Leung, and White, many software development teams might choose to omit some or all of the regression testing tasks because they often account for as much as one-half the cost of software maintenance [BEIZ 90, LEUN 89].

Moreover, the high costs of regression testing are often directly associated with the execution of the test suite. Other industry reports from Rothermel et al. show that the complete regression testing of a 20,000 line software system required seven weeks of continuous execution [ROTH 99]. Since some of the most well-studied software failures, such as the Ariane-5 rocket and the 1990 AT&T outage, can be blamed on the failure to test changes in a software system [HAML 01], many techniques have been developed to support efficient regression testing.

Several different methods have been developed in an attempt to reduce the cost of regression testing. Regression test selection approaches attempt to reduce the cost of regression testing by selecting some appropriate subset of the existing test suite [BALI 98, ROTH 97, VOKO 97]. Test selection techniques normally use the source code of a program to determine which tests should be executed during the regression testing stage [ROTH 96]. Regression test prioritization techniques attempt to order a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with lower priority [ELBA 00, ROTH 99]. By prioritizing the execution of a regression test suite, these methods hope to reveal important defects in a software system earlier in the regression
testing process. Regression test distribution is another alternative that can make regression testing more practical by more fully utilizing the computing resources that are normally available to a testing team [KAP1 01].

2.7 Summary

Testing is an important technique for the improvement and measurement of a software system’s quality. Any approach to testing software faces essential and accidental difficulties. Indeed, as noted by Edsger Dijkstra the construction of the needed test programs is a “major intellectual effort” [DIJK 68]. While software testing is not a “silver bullet” that can guarantee the production of high quality applications, theoretical and empirical investigations have shown that the rigorous, consistent and intelligent application of testing techniques can improve software quality. Software testing normally involves the stages of test case specification, test case generation, test execution, test adequacy evaluation and regression testing. Each of these stages in our model of the software testing process plays an important role in the production of programs that meet their intended specification. The body of theoretical and practical knowledge about software testing continues to grow as research expands the applicability of existing techniques and proposes new testing techniques for an ever-widening range of programming languages and application domains.