Unit 12 Testing Techniques

Structure:

12.1 Introduction

In the previous unit, we have discussed various automated tools available for developing software. Once the software is developed, the next step is to test it so that it satisfies customer needs. In this unit, we shall discuss various software testing techniques.

Objectives:

After studying this unit, you should be able to:

- explain various software testing concepts
- describe types of software testing
- discuss black box and white box testing
- list out various testing tools

12.2 Software Testing Concepts

Software Testing is the process of executing a program or system with the intent of finding errors or it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear – generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects – or bugs – will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable – and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32–bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on pre–error test cases that it passed before can no longer is guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

An interesting analogy parallels the difficulty in software testing with the pesticide, known as the Pesticide Paradox: *Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual*. But this alone will not guarantee to make the software better, because the Complexity Barrier principle states: *Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity*. By eliminating the (previous) easy bugs you allowed another escalation of features and complexity, but this time you have subtler bugs to face, just to retain the reliability you had before. Society seems to be unwilling to limit complexity because we all want that extra bell, whistle, and feature interaction. Thus, our users always push us to the complexity barrier and how close we can approach that barrier is largely determined by the strength of the techniques we can wield against ever more complex and subtle bugs.

Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle. Typically, more than 50% percent of the development time is spent in testing.

Testing is usually performed for the following purposes:

To improve quality

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugs can cause disasters. The so–called year 2000 (Y2K) bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed, is the purpose of debugging in programming phase.

For verification & validation (V&V)

Just as topic Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We cannot test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors – functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail.

Software testing is both a discipline and a process. Though software testing is part of the software development process, it should not be considered part of software development. It is a separate discipline from software development. Software development is the process of coding functionality to meet defined end–user needs. Software testing is an iterative process of both validating functionality, and, even more important, attempting to break the software. The iterative process of software testing consists of:

- Designing tests
- Executing tests
- Identifying problems
- Getting problems fixed

The objective of software testing is to find problems and fix them to improve quality. Software testing typically represents 40% of a software development budget.

While Software testing tends to be considered a part of development, it is really its own discipline and should be tracked as its own project. Software testing, while works very closely with development, should be independent enough to be able to hold–up or slow product delivery if quality objectives are not met.

12.3 Types of Software Testing

Software testing consists of several subcategories, each of which is done for different purposes, and often – using different techniques. Software testing categories include:

- *Functionality testing* to verify the proper functionality of the software, including validation of system and business requirements, validation of formulas and calculations, as well as testing of user interface functionality.
- *Forced error testing*, or attempting to break and fix the software during testing so that customers do not break it in production.
- *Compatibility testing* to ensure that software is compatible with various hardware platforms, operating systems, other software packages, and even previous releases of the same software.
- *Performance testing* to see how well software performs in terms of the speed of computations and responsiveness to the end–user.
- *Scalability testing* to ensure that the software will function well as the number of users and size of databases increase.
- *Stress testing* to see how the system performs under extreme conditions, such as a very large number of simultaneous users.
- *Usability testing* to ensure that the software is easy and intuitive to use.
- *Application security testing* to make sure that valuable and sensitive data cannot be accessed inappropriately or compromised under concerted attack.

In some cases, there may even have to be other types of testing such as regulatory – compliance testing, depending on the type of software and intended industry.

There are two basic methods of performing software testing:

1) Manual testing
2) Automated testing

Manual testing

As the name would imply, manual software testing is the process of an individual or individuals manually testing software. This can take the form of navigating user interfaces, submitting information, or even trying to hack the software or underlying database. As one might presume, manual software testing is labor–intensive and slow. There are some things for which manual software testing is appropriate, including:

- User interface or usability testing
- Exploratory/ad hoc testing (where testers do not follow a ‘script’, but rather testers ‘explore’ the application and use their instincts to find bugs)

- Testing areas of the application which experience a lot of change.
- User acceptance testing (often, this can also be automated)

The time commitment involved with manual software testing is one of its most significant drawbacks. The time needed to fully test the system will typically range from weeks to months. Variability of results depending on who is performing the tests can also be a problem. For these reasons, many companies look to automation as a means of accelerating the software testing process while minimizing the variability of results.

Automated testing

Automated software testing is the process of creating test scripts that can then be run automatically, repetitively, and through much iteration. Done properly, automated software testing can help to minimize the variability of results, speed up the testing process, increase test coverage (the number of different things tested), and ultimately provide greater confidence in the quality of the software being tested.

There are, however, some things for which automated software testing is not appropriate. These include:

- End user usability testing is not typically a good candidate for automated testing.
- Tests which will not be run more than a couple of times are typically not a good candidate for automated testing, since the payoff of in test automation comes after many test executions.
- Tests for areas of the application which experience a lot of change are also not a good candidate for automation since this can lead to substantial maintenance of test automation scripts. Such areas of the application may be more effectively tested manually.

It is important to note that test automation is software, and just like the software you are building for internal or external customers, it must be *well–architected*. A good test automation architecture, such as a keyword-driven testing framework, will reduce the overall cost of ownership of your test automation by minimizing maintenance expense and increasing the number of automated tests, allowing your organization to run more tests (and achieve higher quality) for the same investment of time and money.

Self Assessment Questions

1. Software Testing is the process of executing a program or system with the intent of finding errors. (True / False)
2. _______________ is done to ensure that the software will function well as the number of users and size of databases increase.
3. Software testing typically represents _______________ of a software development budget. (Pick right option)
   a) 40%
   b) 20%
   c) 70%
   d) 90%

12.4 Black Box Testing

Black Box Testing attempts to derive sets of inputs that will fully exercise all the functional requirements of a system. This type of testing attempts to find errors in the following categories:

1) Incorrect or missing functions
2) Interface errors
3) Errors in data structures or external database access
4) Performance errors
5) Initialization and termination errors.

Tests are designed to answer the following questions:

1) How is the function's validity tested?
2) What classes of input will make good test cases?
3) Is the system particularly sensitive to certain input values?
4) How are the boundaries of a data class isolated?
5) What data rates and data volume can the system tolerate?
6) What effect will specific combinations of data have on system operation?

White box testing should be performed early in the testing process, while black box testing tends to be applied during later stages. Test cases should be derived which

1) Reduce the number of additional test cases that must be designed to achieve reasonable testing.
2) Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Now let's see various methods of black box testing.

Equivalence partitioning

This method divides the input domain of a program into classes of data from which test cases can be derived. Equivalence partitioning strives to define a test case that uncovers classes of errors and thereby reduces the number of test cases needed. It is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions.

Equivalence classes may be defined according to the following guidelines:

1) If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2) If an input condition requires a specific value, then one valid and two invalid equivalence classes are defined.
3) If an input condition specifies a member of a set, then one valid and one invalid equivalence classes are defined.
4) If an input condition is Boolean, then one valid and one invalid equivalence class are defined.

Boundary value analysis

This method leads to a selection of test cases that exercise boundary values. It complements equivalence partitioning since it selects test cases at the edges of a class. Rather than focusing on input conditions solely, BVA derives test cases from the output domain also. BVA guidelines include:

1) For input ranges bounded by *a* and *b*, test cases should include values *a* and *b* and just above and just below *a* and *b* respectively.
2) If an input condition specifies a number of values, test cases should be developed to exercise the minimum and maximum numbers and values just above and below these limits.
3) Apply guidelines 1 and 2 to the output.
4) If internal data structures have prescribed boundaries, a test case should be designed to exercise the data structure at its boundary.

Cause – effect graphing techniques

Cause-effect graphing is a technique that provides a concise representation of logical conditions and corresponding actions. There are four steps:

1) Causes (input conditions) and effects (actions) are listed for a module and an identifier is assigned to each.

2) A cause – effect graph is developed.
3) The graph is converted to a decision table.
4) Decision table rules are converted to test cases.

Self Assessment Questions

4. Black Box Testing tests the internal structure of the system. (True / False)
5. _______________ method leads to a selection of test cases that exercise boundary values.
6. _______________ is a technique that provides a concise representation of logical conditions and corresponding actions. (Pick right option)
   a) Boundary Value Analysis
   b) Cause-effect graphing
   c) Equivalence Partitioning
   d) Basis Path Testing

12.5 White Box Testing Techniques

White box testing is a test case design method that uses the control structure of the procedural design to derive test cases.

The white box testing technique does the following,

1) Guarantee that all independent paths within a module have been exercised at least once,
2) Exercise all logical decisions on their true and false sides,
3) Execute all loops at their boundaries and within their operational bounds, and
4) Exercise internal data structures to ensure their validity

Now let's discuss various white box testing techniques:

Basis path testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing. This aims to derive a logical complexity measure of a procedural design and use this as a guide for defining a basic set of execution paths. Test cases which exercise basic set will execute every statement at least once.

Flow graph notation

Flow graphs can be used to represent control flow in a program and can help in the derivation of the basis set. Each flow graph node represents one or more procedural statements. The edges between nodes represent flow of control. An edge must terminate at a node, even if the node does not represent any useful procedural statements. A region in a flow graph is an area bounded by edges and nodes. Each node that contains a condition is called a predicate node. Cyclomatic complexity is a metric that provides a quantitative measure of the logical complexity of a program. It defines the number of independent paths in the basis set and thus provides an upper bound for the number of tests that must be performed.

Notation for representing control flow is shown in figure 12.1.

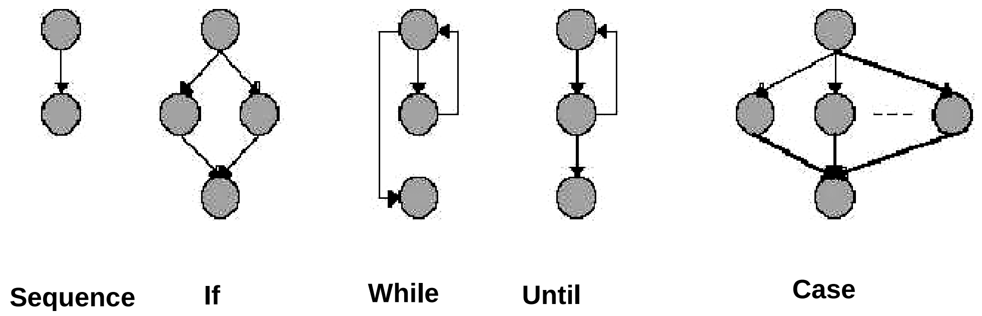


Fig. 12.1: Flow Graph Notations

On a flow graph:

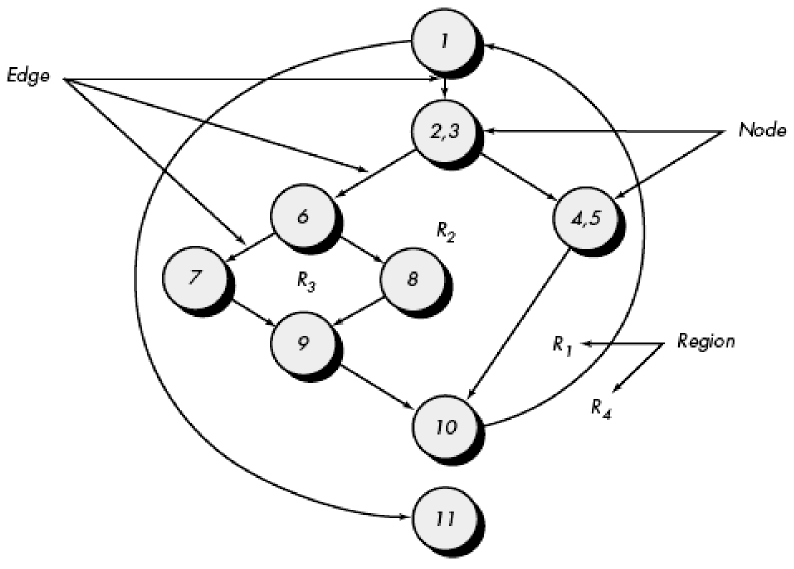
- Arrows called *edges* represent flow of control
- Circles called *nodes* represent one or more actions.
- Areas bounded by edges and nodes are called *regions*.
- A *predicate node* is a node containing a condition

Any procedural design can be translated into a flow graph. Note that compound Boolean expressions at tests generate at least two predicate node and additional arcs.

Cyclomatic complexity

Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once. An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in figure 12.2 is given below:

Path 1: 1-11.
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11



(B)

Fig. 12.2: An Example Flow Graph

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges. Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in figure 12.2. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design. How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer. Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, $V(G)$, for a flow graph, G, is defined as $V(G) = E - N + 2$. Where E is the number of flow graph edges, N is the number of flow graph nodes.
3. 3. Cyclomatic complexity, $V(G)$, for a flow graph, G, is also defined as $V(G) = P + 1$. Where P is the number of predicate nodes contained in the flow graph G.

Referring again to the flow graph in figure 12.2, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. $V(G)$ = 11 edges – 9 nodes + 2 = 4.

3) $V(G)$ = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in figure 12.2 is 4.

Control structure testing

Control structure testing is a group of white-box testing methods using loops. Let's see different types of loop testing.

Loop testing:

Loops are fundamental to many algorithms and need thorough testing. There are four different classes of loops: simple, concatenated, nested, and unstructured (See Figure 12.3).

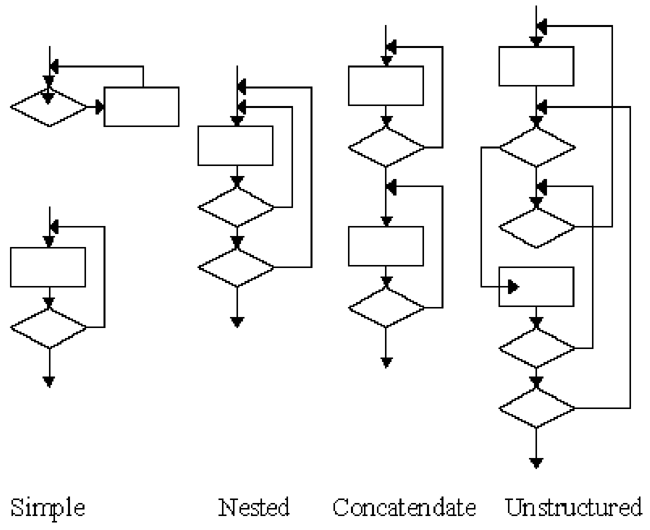


Fig. 12.3: Loop structures

- ***Simple loops,*** where *n* is the maximum number of allowable passes through the loop.
  - Skip loop entirely
  - Only one pass through loop
  - Two passes through loop
  - m passes through loop where m<n.
  - (n-1), n, and (n+1) passes through the loop.
- ***Nested loops***
  - Start with inner loop. Set all other loops to minimum values.
  - Conduct simple loop testing on inner loop.
  - Work outwards
  - Continue until all loops tested.
- ***Concatenated loops***
  - If independent loops, use simple loop testing.
  - If dependent, treat as nested loops.
- ***Unstructured loops***
  - Don't test – redesign.

Advantages of White Box testing

i) As the knowledge of internal coding structure is prerequisite, it becomes very easy to find out which type of input/data can help in testing the application effectively.

ii) The other advantage of white box testing is that it helps in optimizing the code

iii) It helps in removing the extra lines of code, which can bring in hidden defects.

Disadvantages of White Box testing:

i) As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, which increases the cost.

ii) And it is nearly impossible to look into every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.

Testing tools

Following are some of the popular Software Testing Tools:

- Win Runner
- Load Runner
- Test Director
- Silk Test
- Test Partner

Self Assessment Questions

7. White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. (True / False)
8. _______________ is software metric that provides a quantitative measure of the logical complexity of a program.
9. Which one of the following is a Software Testing tool? (Pick right option)
   a) Rational Rose
   b) Oracle
   c) Win Runner
   d) Linux

12.6 Summary

Let's recapitulate important points discussed in this unit:

- The primary objective for test case design is to derive a set of tests that have the highest likely hood for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: White–Box testing and Black–Box testing.
- White–box testing focuses on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised.
- Basis path testing, a white–box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white–box techniques by providing a procedure for exercising loops of varying degrees of complexity.
- Black–box testing, on the other hand, broadens our focus and might be called "testing in large". Black–box tests are designed to validate functional requirements without regard to the internal workings of a program.
- Black–box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage.

12.7 Terminal Questions

1. Explain the need for Software Testing.
2. What are the two broad types of Software Testing?
3. What do you mean by Black Box Testing? Why it is called so?
4. Briefly explain the significance of White Box Testing.

12.8 Answers

Self Assessment Questions

1. True
2. Scalability Testing
3. a) 40%
4. False

5. Boundary Value Analysis
6. b) Cause-effect graphing
7. True
8. Cyclomatic complexity
9. c) Win Runner

Terminal Questions

1. Software Testing is the process of executing a program or system with the intent of finding errors or it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. (Refer Section 12.2 for detail)
2. Software testing consists of several subcategories, each of which is done for different purposes, and often – using different techniques. Software testing categories include: Functionality Testing, Forced Error Testing, Compatibility Testing, Performance Testing, Scalability Testing, Stress Testing, Usability Testing, Application Security Testing etc. (Refer Section 12.3)
3. Black Box Testing attempts to derive sets of inputs that will fully exercise all the functional requirements of a system. (Refer Section 12.4)
4. White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. (Refer Section 12.5)