

White-Box Testing

White-box testing is a verification technique software engineers can use to examine if their code works as expected. In this chapter, we will explain the following:

- a method for writing a set of white-box test cases that exercise the paths in the code
- the use of equivalence partitioning and boundary value analysis to manage the number of test cases that need to be written and to examine error-prone/extreme “corner” test cases
- how to measure how thoroughly the test cases exercise the code

White-box testing is testing that takes into account the internal mechanism of a system or component (IEEE, 1990). White-box testing is also known as *structural testing*, *clear box testing*, and *glass box testing* (Beizer, 1995). The connotations of “clear box” and “glass box” appropriately indicate that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code.

Using the white-box testing techniques outlined in this chapter, a software engineer can design test cases that (1) exercise independent paths within a module or unit; (2) exercise logical decisions on both their true and false side; (3) execute loops at their boundaries and within their operational bounds; and (4) exercise internal data structures to ensure their validity (Pressman, 2001).

There are six basic types of testing: unit, integration, function/system, acceptance, regression, and beta. White-box testing is used for three of these six types:

- *Unit testing*, which is *testing of individual hardware or software units or groups of related units* (IEEE, 1990). A *unit is a software component that cannot be subdivided into other components* (IEEE, 1990). Software engineers write white-box test cases to examine whether the unit is coded correctly. Unit testing is important for ensuring the code is solid before it is integrated with other code. Once the code is integrated into the code base, the cause of an observed failure is more difficult to find. Also, since the software engineer writes and runs unit tests him or herself, companies often do not track the unit test failures that are observed—making these types of defects the most “private” to the software engineer. We all prefer to find our own mistakes and to have the opportunity to fix them without others knowing. Approximately 65% of all bugs can be caught in unit testing (Beizer, 1990).
- *Integration testing*, which is *testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them* (IEEE, 1990). Test cases are written which explicitly examine the interfaces between the various units. These test cases can be black box test cases, whereby the tester understands that a test case requires multiple program units to interact. Alternatively, white-box test cases are written which explicitly exercise the interfaces that are known to the tester.
- *Regression testing*, which is *selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements* (IEEE, 1990). As with integration testing, regression testing can be done via black-box test cases, white-box test cases, or a combination of the two. White-box unit and integration test cases can be saved and re-run as part of regression testing.

1 White-Box Testing by Stubs and Drivers

With white-box testing, you must run the code with predetermined input and check to make sure that the code produces predetermined outputs. Often programmers write stubs and drivers for white-box testing. A *driver* is a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results (IEEE, 1990) or most simplistically a *line of code that calls a method and passes that method a value*. For example, if you wanted to move a Player instance, Player1, two spaces on the board, the driver code would be

```
movePlayer(Player1, 2);
```

This driver code would likely be called from the main method. A white-box test case would execute this driver line of code and check `Player.getPosition()` to make sure the player is now on the expected cell on the board.

A *stub* is a computer program statement substituting for the body of a software module that is or will be defined elsewhere (IEEE, 1990) or a dummy component or object used to simulate the behavior of a real component (Beizer, 1990) until that component has been developed. For example, if the `movePlayer` method has not been written yet, a stub such as the one below might be used temporarily – which moves any player to position 1.

```
public void movePlayer(Player player, int diceValue) {
    player.setPosition(1);
}
```

Ultimately, the dummy method would be completed with the proper program logic. However, developing the stub allows the programmer to call a method in the code being developed, even if the method does not yet have the desired behavior.

Stubs and drivers are often viewed as throwaway code (Kaner, Falk et al., 1999). However, they do not have to be thrown away: Stubs can be “filled in” to form the actual method. Drivers can become automated test cases.

2 Deriving Test Cases

In the following sections, we will discuss various methods for devising a thorough set of white-box test cases. We will refer to the Monopoly example to illustrate the methods under discussion. These methods can serve as guidelines for you as you design test cases. Even though it may seem like a lot of work to use these methods, statistics show [1] that the act of careful, complete, systematic test **design** will catch as many bugs as the act of testing. The test design process, at all levels, is at least as effective at catching bugs as is running the test case designed by that process.

Each time you write a code module, you should write test cases for it based on the guidelines. A possible exception to this recommendation is the accessor methods (i.e., getters and setters) of your projects. You should concentrate your testing effort on code that could easily be broken. Generally, accessor methods will be written error-free.

2.1 Basis Path Testing

Basis path testing (McCabe, 1976) is a means for ensuring that all independent paths through a code module have been tested. An *independent path* is any path through the code that introduces at least one new set of processing statements or a new condition. (Pressman, 2001) Basis path testing provides a minimum, lower-bound on the number of test cases that need to be written.

To introduce the basis path method, we will draw a flowgraph of a code segment. Once you understand basis path testing, it may not be necessary to draw the flowgraph – though you may always find a quick sketch helpful. If you test incrementally and the modules you test are small enough, you can consider having a mental picture of the flow graph. As you will see, the main objective is to identify the number of decision points in the module and you may be able to identify them without a written representation.

A flowgraph of purchasing property appears in Figure 1. The flowgraph is intended to depict the following requirement.

If a player lands on a property owned by other players, he or she needs to pay the rent. If the player does not have enough money, he or she is out of the game. If the property is not owned by any players, and the player has enough money buying the property, he or she may buy the property with the price associated with the property.

In the simple flowgraph in Figure 2, a rectangle shows a sequence of processing steps that are executed unconditionally. A diamond represents a *logic conditional* or *predicate*. Some examples of logical conditionals are if-then, if-then-else, selection, or loops. The head of the arrow indicates the flow of control. For a rectangle, there will be one arrow heading out. For a predicate, there will be two arrows heading out – one for a true/positive result and the other for a false/negative result.

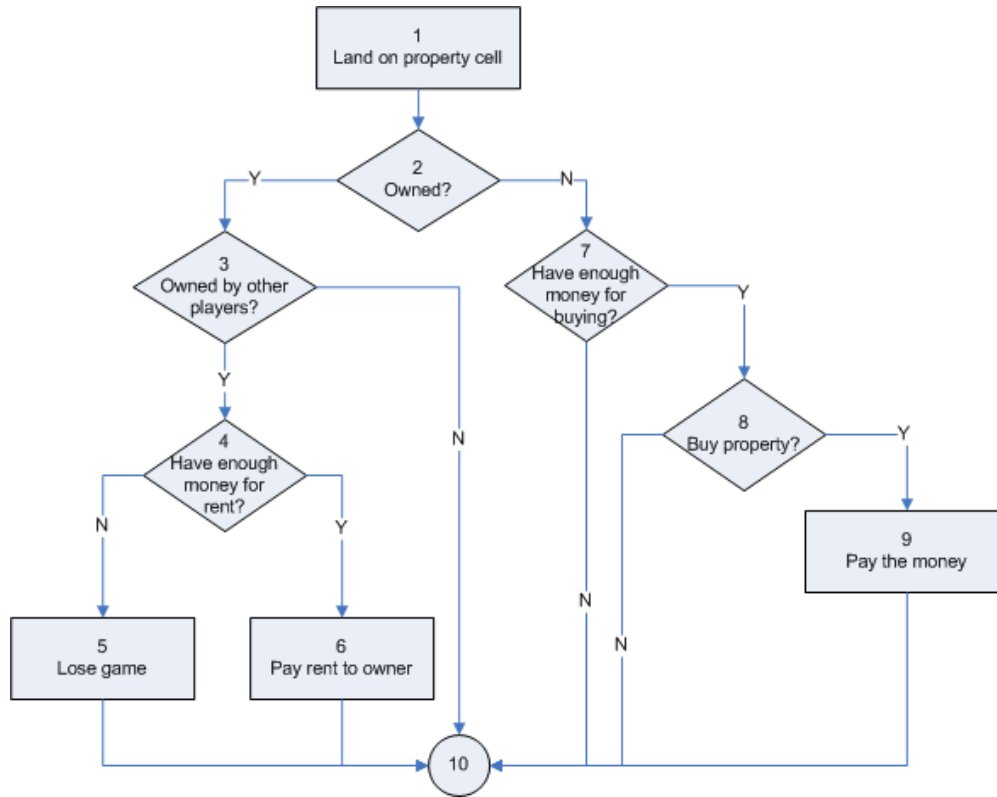


Figure 1: Flowgraph of purchasing property

Using this flow graph, we can compute the number of independent paths through the code. We do this using a metric called the *cyclomatic number* (McCabe, 1976), which is based on graph theory. The easiest way to compute the cyclomatic number is to count the number of conditionals/predicates (diamonds) and add 1. In our example above, there are five conditionals. Therefore, our cyclomatic number is 6, and we have six independent paths through the code. We can now enumerate them:

- | | |
|-----------------|--|
| 1. 1-2-3-4-5-10 | (property owned by others, no money for rent) |
| 2. 1-2-3-4-6-10 | (property owned by others, pay rent) |
| 3. 1-2-3-10 | (property owned by the player) |
| 4. 1-2-7-10 | (property available, don't have enough money) |
| 5. 1-2-7-8-10 | (property available, have money, don't want to buy it) |
| 6. 1-2-7-8-9-10 | (property available, have money, and buy it) |

We would want to write a test case to ensure that each of these paths is tested at least once. As said above, the cyclomatic number is the lower bound on the number of test cases we will write. The test cases that are determined this way are the ones we use in basis path testing. There are other things to consider, as we now discuss.

2.2 Equivalence Partitioning/Boundary Value Analysis

Equivalence partitioning (EP) and boundary value analysis (BVA) provide a strategy for writing white-box test cases. Undoubtedly, whenever you encounter any kind of number or limit in a requirement, you should be alert for EP/BVA issues. For example, a person might want to buy a house, but may or may not have enough money. Considering EP/BVA, we would want to ensure our test cases include the following:

1. property costs \$100, have \$200 (equivalence class “have enough money”)
2. property costs \$100, have \$50 (equivalence class, “don’t have enough money”)
3. property costs \$100, have \$100 (boundary value)
4. property costs \$100, have \$99 (boundary value)
5. property costs \$100, have \$101 (boundary value)

With programming loops (such as while loops), consider EP and execute the loops in the middle of their operational bound. For BVA, you will want to ensure that you execute loops right below, right at, and right above their boundary conditions.

3 Control-flow/Coverage Testing

Another way to devise a good set of white-box test cases is to consider the control flow of the program. The *control flow* of the program is represented in a flow graph, as shown in Figure 1. We consider various aspects of this flowgraph in order to ensure that we have an adequate set of test cases. The adequacy of the test cases is often measured with a metric called coverage. *Coverage* is a measure of the completeness of the set of test cases. To demonstrate the various kinds of coverage, we will use the simple code example shown in Figure 2 as a basis of discussion as we take up the next five topics.

```

1  int foo (int a, int b, int c, int d, float e) {
2      float e;
3      if (a == 0) {
4          return 0;
5      }
6      int x = 0;
7      if ((a==b) OR ((c == d) AND bug(a) )) {
8          x=1;
9      }
10     e = 1/x;
11     return e;
12 }
```

Figure 2: Sample Code for Coverage Analysis

Keeping with a proper testing technique, we write methods to ensure they are testable – most simply by having the method return a value. Additionally, we predetermine the “answer” that is returned when the method is called with certain parameters so that our testing returns that predetermined value. Another good testing technique is to use the simplest set of input that could possibly test your situation – it’s better not to input values that cause complex,

error-prone calculations when you are predetermining the values. We'll illustrate this principle as we go through the next items.

3.1 Method Coverage

Method coverage is a measure of the percentage of methods that have been executed by test cases. Undoubtedly, your tests should call 100% of your methods. It seems irresponsible to deliver methods in your product when your testing never used these methods. As a result, you need to ensure you have 100% method coverage.

In the code shown in Figure 3, we attain 100% method coverage by calling the foo method. Consider Test Case 1: the method call **foo(0, 0, 0, 0, 0.)**, expected return value of 0. If you look at the code, you see that if a has a value of 0, it doesn't matter what the values of the other parameters are – so we'll make it really easy and make them all 0. Through this one call we attain 100% method coverage.

3.2 Statement Coverage

Statement coverage is a measure of the percentage of statements that have been executed by test cases. Your objective should be to achieve 100% statement coverage through your testing. Identifying your cyclomatic number and executing this minimum set of test cases will make this statement coverage achievable.

In Test Case 1, we executed the program statements on lines 1-5 out of 12 lines of code. As a result, we had 42% (5/12) statement coverage from Test Case 1. We can attain 100% statement coverage by one additional test case, Test Case 2: the method call **foo(1, 1, 1, 1, 1.)**, expected return value of 1. With this method call, we have achieved 100% statement coverage because we have now executed the program statements on lines 6-12.

3.3 Branch Coverage

Branch coverage is a measure of the percentage of the decision points (Boolean expressions) of the program have been evaluated as both true and false in test cases. The small program in Figure 3 has two decision points – one on line 3 and the other on line 7.

3	if (a == 0) {
7	if ((a==b) OR ((c == d) AND bug(a))) {

For decision/branch coverage, we evaluate an entire Boolean expression as one true-or-false predicate even if it contains multiple logical-and or logical-or operators – as in line 7. We need to ensure that each of these predicates (compound or single) is tested as both true and false. Table 1 shows our progress so far:

Table 1: Decision Coverage

Line #	Predicate	True	False
3	(a == 0)	Test Case 1 foo(0, 0, 0, 0, 0) return 0	Test Case 2 foo(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2 foo(1, 1, 1, 1, 1) return 1	

Therefore, we currently have executed three of the four necessary conditions; we have achieved 75% branch coverage thus far. We add Test Case 3 to bring us to 100% branch coverage: **foo(1, 2, 1, 2, 1)**. When we look at the code to calculate an expected return value, we realize that this test case uncovers a previously undetected division-by-zero problem on line 10! We can then immediately go to the code and protect from such an error. This illustrates the value of test planning. Through the test case, we achieve 100% branch coverage.

In many cases, the objective is to achieve 100% branch coverage in your testing, though in large systems only 75%-85% is practical. Only 50% branch coverage is practical in very large systems of 10 million source lines of code or more (Beizer, 1990).

3.4 Condition Coverage

We will go one step deeper and examine condition coverage. *Condition coverage is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome [applies to compound predicate] in test cases.* Notice that in line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a). Condition coverage measures the outcome of each of these sub-expressions independently of each other. With condition coverage, you ensure that each of these sub-expressions has independently been tested as both true and false. We consider our progress thus far in Table 2.

Table 2: Condition coverage

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, x, x, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)		Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)		

At this point, our condition coverage is only 50%. The true condition (c==d) has never been tested. Additionally, short-circuit Boolean has prevented the method bug(int) from ever being executed. We examine our available information on the bug method and determine that it should return a value of true when passed a value of a=1. We write Test Case 4 to

address test (c==d) as true: **foo(1, 2, 1, 1, 1)**, expected return value 1. However, when we actually run the test case, the function bug(a) actually returns false, which causes our actual return value (division by zero) to not match our expected return value. This allows us to detect an error in the bug method. Without the addition of condition coverage, this error would not have been revealed.

To finalize our condition coverage, we must force bug(a) to be false. We again examine our bug() information, which informs us that the bug method should return a false value if fed any integer greater than 1. So we create Test Case 5, **foo(3, 2, 1, 1, 1)**, expected return value “division by error”. The condition coverage thus far is shown in Table 15.3.

Table 3: Condition Coverage Continued

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, 1, 1, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 5 foo(3, 2, 1, 1, 1) division by zero!

There are no industry standard objectives for condition coverage, but we suggest that you keep condition coverage in mind as you develop your test cases. You have seen that our condition coverage revealed that some additional test cases were needed.

There are commercial tools available, called *coverage monitors*, that can report the coverage metrics for your test case execution. Often these tools only report method and statement coverage. Some tools report decision/branch and/or condition coverage. These tools often also will color code the lines of code that have not been executed during your test efforts. It is recommended that coverage analysis is automated using such a tool because manual coverage analysis is unreliable and uneconomical (IEEE, 1987).

4 Data Flow Testing

In data flow-based testing, the control flowgraph is annotated with information about how the program variables are defined and used. Different criteria exercise with varying degrees of precision how a value assigned to a variable is used along different control flow paths. A reference notation is a definition-use pair, which is a triple of (d, u, V) such that V is a variable, d is a node in which V is defined, and u is a node in which V is used. There exists a path between d and u in which the definition of V in d is used in u.

5 Failure (“Dirty”) Test Cases

As with black-box test cases, you must think diabolically about the kinds of things users might do with your program. Look at the structure of your code and think about every possible way a user might break it. These devious ways may not be uncovered by the previously mentioned methods for forming test cases. You need to be smart enough to think of your particular code and how people might outsmart it (accidentally or intentionally). Augment your test cases to handle these cases. Some suggestions follow:

- Look at every input into the code you are testing. Do you handle each input if it is incorrect, the wrong font, or too large (or too small)?
- Look at code from a security point of view. Can a user overflow a buffer, causing a security problem?
- Look at every calculation. Could it possible create an overflow? Have you protected from possible division by zero?

6 Flow Graphs Revisited

The flowgraph of Figure 1 was fairly straightforward because there were no compound Boolean predicates. Let’s go back and look at what a flowgraph of the code in Figure 2 would look like. When you encounter a compound predicate, such as in line 7, you must break the expression up so that each Boolean sub-expression is evaluated on its own, as shown below in Figure 3.

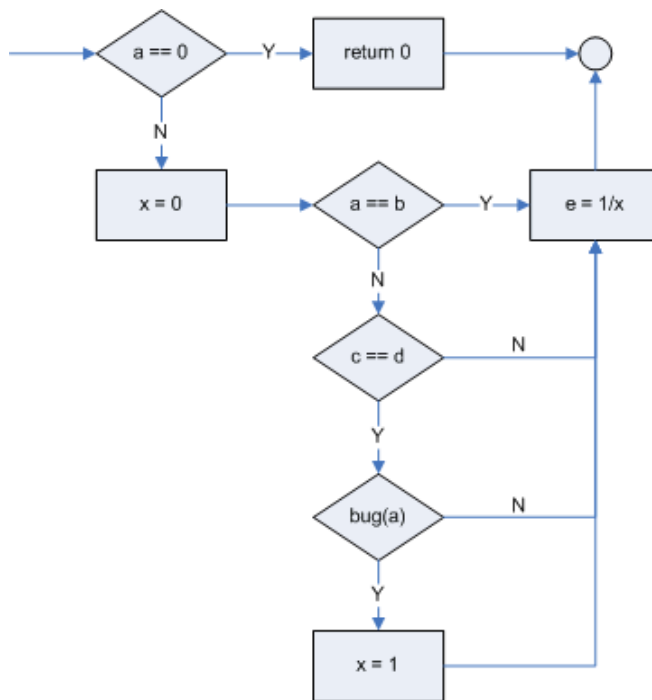


Figure 3: Compound Predicate Flow Graph

If you look back at the previous section on deriving test cases, you see that as we strove to get 100% method, statement, decision/branch and condition coverage, we wrote five test cases. Examining Figure 3, you can see we have four predicates (diamonds). Therefore, our cyclomatic number is $4 + 1 = 5$ – which is the number of test cases we wrote.


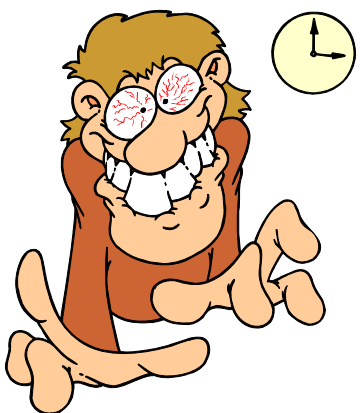
As code becomes larger and more complex, devising the flowgraph and calculating the cyclomatic complexity can become difficult or impossible. However, if you write methods that are not overly long (which is a good practice anyway), the methods we have discussed in this chapter are quite helpful in your quest for high quality.

7 Summary

Properly planned with explicit input/output combinations, white-box testing is a controlled V&V technique. You run a test case, you know what lines of code you execute, and you know what the “answer” should be. If you don’t get the right answer, the test case reveals a *problem* (a fault). Fortunately, you know which lines of code to look at based upon the test case that fails. Because of this control, removing defects in unit test is more economical than later phases in the development cycle. Later testing phases that involve block-box testing can be more chaotic. In those phases, a test case no longer reveals a problem (and an approximate location of where the problem needs to be fixed). Instead, a failed black-box test case reveals a *symptom of a problem* (a failure). It can be difficult, time consuming, and take an unpredictable amount of time to find the root cause of the symptom (the fault that caused the failure) so that the software engineer knows what to change in the code.

Therefore, unit testing is a more economical defect removal technique when compared with black box testing. Therefore, as much as possible should be tested at the unit level (IEEE, 1987). A comparison between white-box testing and black box testing can be found in Table 5.

Table 5. A comparison of white-box testing and black-box testing





Type of Testing	White-box Testing	Black-box Testing
Tester visibility	have visibility to the code and write test cases based upon the code	have no visibility to the code and write test cases based on possible inputs and outputs for functionality documented in specifications and/or requirements
A failed test case reveals	a problem (fault)	a symptom of a problem (a failure)
Controlled?	Yes – the test case helps to identify the specific lines of code involved	No – it can be hard to find the cause of the failure
		

Both white-box and black-box testing techniques are important and are intended to find different types of faults. Simple unit faults might need to be found in black-box testing if adequate white-box testing is not done adequately). You should strive to remove as many defects as possible using white-box testing techniques when the identification of the faults is more controllable.

Several practical tips for risk management were presented throughout this chapter. The keys for successful risk management are summarized in Table 6.

Table 6. Key Ideas for White-Box Testing

White-Box Testing

	Use an automated coverage monitor for the analysis of control flow-based unit testing.
	Compute the cyclomatic complexity to determine the least number of test cases that should be written. This number does not consider equivalence class partitioning or boundary value analysis – which should be done for most decision points.
	Draw the flowgraph for a code segment – at least until you get more used to computing cyclomatic complexity.
	At a minimum, write enough white box test cases to cover 100% of your statements. Get as high a coverage as possible with your decision/branch and condition coverage.

Glossary of Chapter Terms

Word	Definition	Source
branch coverage	a measure of the percentage of the decision points (Boolean expressions) of the program have been evaluated as both true and false in test cases	
condition coverage	a measure of the percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome [applies to compound predicate] in test cases	
driver	software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results	(IEEE, 1990)
integration testing	testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them	(IEEE, 1990)
method coverage	a measure of the percentage of methods that have been executed by test cases.	
regression testing	selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements	(IEEE, 1990)
statement coverage	a measure of the percentage of statements that have been executed by test cases	
stub	computer program statement substituting for the body of a software module that is or will be defined elsewhere	(IEEE, 1990)
unit	a separable, testable element specified in the design of a computer software component; a software component that cannot be subdivided into other components	(IEEE, 1990)
unit testing	testing of individual hardware or software units or groups of related units	(IEEE, 1990)
white-box testing	testing that takes into account the internal mechanism of a system or component	(IEEE, 1990)

References

- Beizer, B. (1990). Software Testing Techniques. Boston, International Thompson Computer Press.
- Beizer, B. (1995). Black Box Testing. New York, John Wiley & Sons, Inc.
- IEEE (1987). "ANSI/IEEE Standard 1008-1987, IEEE Standard for Software Unit Testing."
- IEEE (1990). IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.
- Kaner, C., J. Falk, et al. (1999). Testing Computer Software. New York, Wiley Computer Publishing.
- McCabe, T. (1976). "A Software Complexity Measure." IEEE Transactions on Software Engineering **SE-2**: 308-320.
- Pressman, R. (2001). Software Engineering: A Practitioner's Approach. Boston, McGraw Hill.

Chapter Questions:

1. If we have a program which has 10 *independent* if...then...else... statements, there are totally 2^{10} execution paths. Suppose that, on average, each test case needs 50 microseconds to exercise one execution path and the program itself takes 100 microseconds. If we write a test case for each possible execution path, how much time does it take to run all the test cases?
2. If a program passes all the black box tests, it means that this program should work properly. Then, in addition to black-box testing, why do we need white-box testing?
3. Consider the following Java code snippet:

```

Class ProductDB{
:
/**
 * returns an instance of product database
 */
public static ProductDB getInstance(){
:
}
/**
 * returns the price of a product.
 * throws Exception if the product is not found
 */
public float getProductPrice(String productID)
throws Exception{
:
}
}
Class Cashier{
ProductDB db;

public Cashier(ProductDB db){
this.db = db;
}
}

```

```

}
:
/**
 * Calculate the total of the prices of several products
 * param productIDs a String array that contains all the
 * product IDs.
 * return The total price of the products.
 */
public float calculateTotal (String[] productIDs)
    throws Exception{
    float total = 0;

    if(productIDs == null)
        return 0;
    for(int x=0; x<productIDs.length; x++){
        float price =
            db.getProductPrice(productIDs[x]);
        total += price;
    }
    return total;
}
}

```

The getInstance method of ProductDB returns an instance of the product database. Assume that ProductDB is a tested component. Suppose we are going to write a unit test to test this calculateTotal method. Write suitable test drivers. Make proper assumptions.

4. Consider the calculateTotal method in question 3 and the following test case:

```

public void testCalculateTotal (){
    Cashier cashier = new Cashier(new MockProductDB());
    String[] products = new String[0];
    assertEquals(0, cashier.calculateTotal (products));
}

```

- A. Compute the statement coverage of the test for the calculateTotal method.
- B. Can we say this test achieves 100% branch coverage for the method?

5. Read the following pseudo code:

```

if (input is in AllowedCharacterSet)
    if (input is a number)
        if (input >= 0)
            put input into positiveNumberList
        else
            put input into negativeNumberList
    else
        if (input is an alphabet)
            put input into alphabetList
        else
            put input into symbolList
else
    exception("Illegal character")

```

- A. Draw a flow diagram that depicts the pseudo code. Label each node in the diagram with a unique alphabet.
- B. What is the cyclomatic number of the program?
- C. Identify each independent execution path in this program.

6. Following is the code from the information system of Video Buster video rental company. The purpose of the following program is to calculate the fee of the rental.

```

Float calcRentalFee(Tape[] tapes, Customer customer){
    float total = 0;
    for(int i = 0; i < tapes.length; i++){

```

```

        total += tapes[l]. price;
    }
    if (tapes.length > 10){
        total *= .8;
    } else if(tapes.length > 5){
        total *= .9;
    }
    if(customer.isPremium()){
        total *= .9;
    }
    return total;
}

```

- A. Using EP/BVA techniques, how many test cases are needed?
 - B. How many test cases are needed to achieve 100% branch coverage?
7. Read the program snippet in question 6.
- A. Derive the test cases that achieve 100% statement coverage and branch coverage.
 - B. This program will throw a null pointer exception if we use null as the either of the two arguments. Do any of your test cases catch this bug?
 - C. From this experience, we can find that it is wise to add test cases to test the null values. This is a good rule for dirty tests. Write this finding in your notebook.
8. From question 7, we know that even if the test cases have 100% test coverage, it is still possible for the program to go wrong. Find some rules that can help software developers discover more test cases (or dirty test cases) that are useful.
9. Discuss the meaning of cyclomatic number, and why it is useful.
10. Consider the following Java code segment:

```

public Hashtabl e countAl phabet(Stri ng aStri ng){
    Hashtabl e tabl e = new Hashtabl e();
    If (aStri ng.length > 4000) return tabl e;
    Stri ngBuffer buffer = new Stri ngBuffer(aStri ng);
    Whi l e (buffer.length() > 0){
        Stri ng fi rstChar = buffer.substri ng(0, 1);
        I nteger count = (I nteger)tabl e.get(fi rstChar);
        if (count == null){
            count = new I nteger(1);
        } el se{
            count = new I nteger(count.i ntVal ue() + 1);
        }
        tabl e.put(fi rstChar, count);
        buffer.del ete(0, 1);
    }
    return tabl e;
}

```

The program counts the numbers of each alphabet in a string, and put the result in a hashtable. Develop a minimum set of test cases that:

- 1. Guarantees that all independent execution path is exercised at least once;
- 2. Guarantees that both the true and false side of all logical decisions are exercised;
- 3. Executes the loop at the boundary values and within the boundaries.