Week 5 & 6

White Box Testing

In this chapter, we shall discuss

- Structural based testing (WhiteBox testing) techniques
- Line Coverage
- Code Flow Graph (CFG)
- Block coverage
- Branch/Decision coverage
- Condition (Basic and Condition + Branch) coverage
- Path coverage
- MC/DC coverage

Introduction

In previous chapters, we discussed how to test software using requirements as the main element to guide the testing. In this chapter, we will use the source code itself as a source of information to create tests. Techniques that use the structure of the source code as a way to guide the testing, are called structural testing or whitebox testing techniques.

> *"Program testing can be used to show the presence of bugs, but never to show their absence!"*
>
> --Edsgar W. Dijkstra,
> NATO SE Conference, 1969 [1].

Robert Merkel[1], in addition to what Dijkstra said, added:

> *"... and you can't show the presence of bugs in the code you haven't tested"*
>
> -- Robert Merkel,
> sitting in the attic with his cat, 2014.

In Whitebox testing, you select your test cases to ensure that the source code as it exists is systematically *"covered"* by test cases. That is, we focus on trying to systematically cover all the behaviour of the software as it exists, and not solely on the specification. These coverage criteria relate closely to test coverage, a concept that many developers know. By test coverage, we mean the amount (or percentage) of production code that is exercised by the tests [2].

[1] Robert Merkel was the lecturer for FIT2107 till 2018.

Blackbox vs Whitebox Testing

In a nutshell, the following table elaborates the differences between blackbox and whitebox testing techniques

	Blackbox Testing	Whitebox Testing
1	External - user view	Internal - developer view
2	Checks the output with the specifications to verify the results	Allows tester to be confident about the coverage.
3	Source code not needed	Source code needed helps in easier debugging
4	Different techniques such as equivalence testing, boundary value etc applied at different levels of granularity	Mostly techniques applied at unit and integration levels.
5	Unexpected functionality cannot be revealed	Missing functionalities cannot be revealed.

So, we should use both techniques as they both together can identify most of the issues in the software.

Why do we need Whitebox Testing?

In a nutshell, for two reasons:

1) to systematically derive tests from source code.
2) to know when to stop testing.

As a tester, when performing blackbox testing, your goal was clear: to derive classes out of the requirement specifications, and then to derive test cases for each of the classes. You were satisfied once all the classes and boundaries were systematically exercised.

The same idea applies to whitebox testing. First, it gives us a systematic way to devise tests. As we will see, a tester might focus on testing all the lines of a program; or focus on the branches and conditions of the program. Different criteria produce different test cases.

Second, to know when to stop. It is easy to imagine that the number of possible paths in a mildly complex piece of code is just too large, and exhaustive testing is impossible. Therefore, having clear criteria on when to stop helps testers in understanding the costs of their testing.

In blackbox testing, we cannot identify the unexpected behaviours of the programs because we are replying on the inputs and conforms the output through specifications. This can be resolved with whitebox testing but whitebox testing fails to reveal missing functionalities in the software as we look at the code and what it shows. If a developer fails to implement a functionality the white box testing will not capture it. Therefore, in practice, we use both techniques at the same time.

Control Flow Graphs

Control Flow Graphs (CFG), as the name says, represents the execution flow in a graph. It is essential to many computer optimisation and static analysis tasks. A control-flow graph (or CFG) is a representation of all paths that might be traversed during the execution of a piece of code. It consists of basic blocks, decision blocks, and arrows/edges that connect these blocks. Let us take an example to explain it.

Example 1: TicketPrice for pensioners

Given a python code below which provides concession to pensioners and senior citizens. The developer has written the code this way.

```python
def ticketprice(age, pensionstatus, seniorcard):
    if age <= 18:
        concession = True
    elif age <=55:
        if pensionstatus:
            concession = True
        else:
            concession = False
    else:
        if pensionstatus:
            concession = True
        elif seniorcard:
            concession = True
        else:
            concession =  False
    if concession:
        return 5.00
    else:
        return 10.00
```

the CFG for ticketprice looks like this:

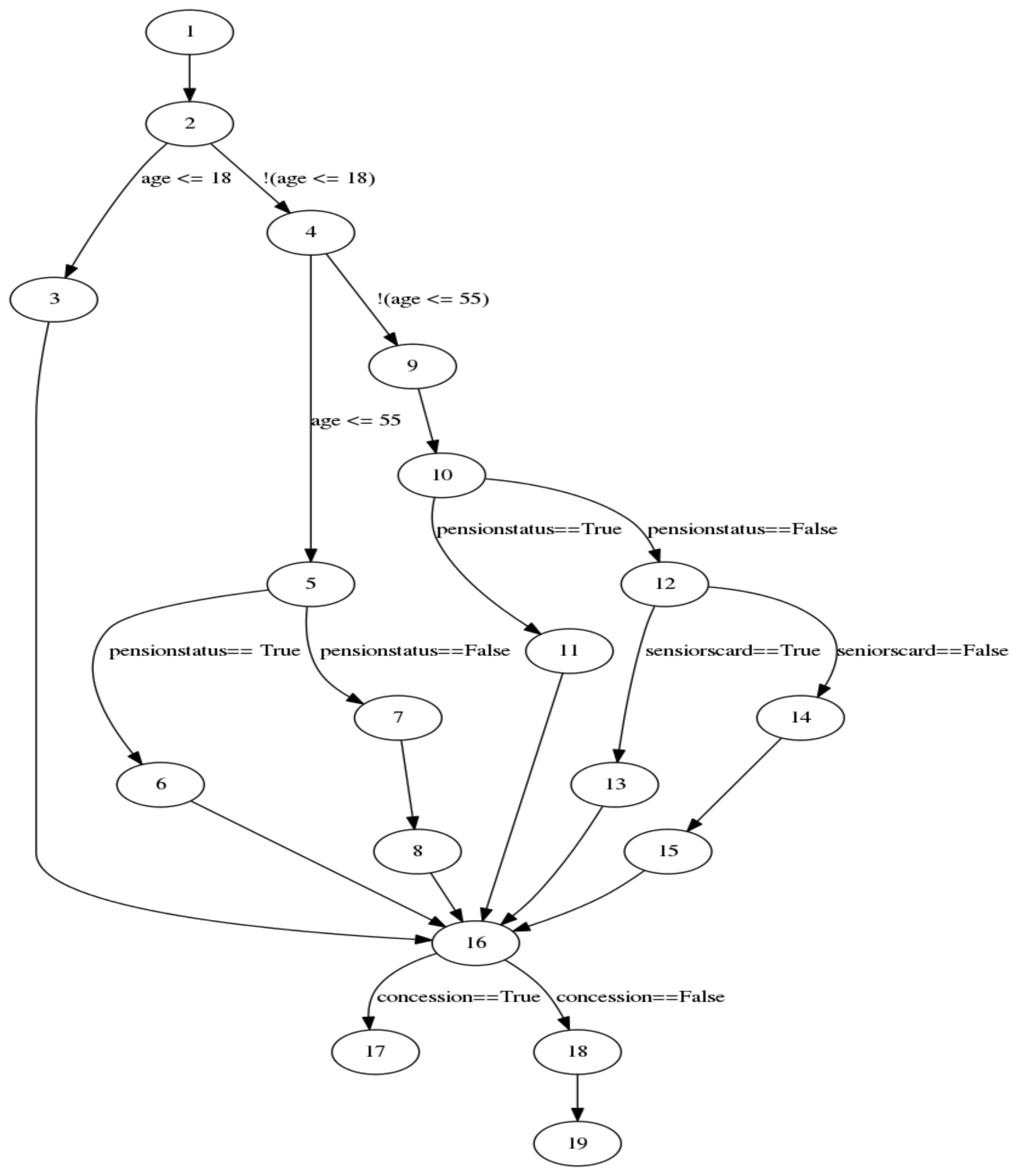


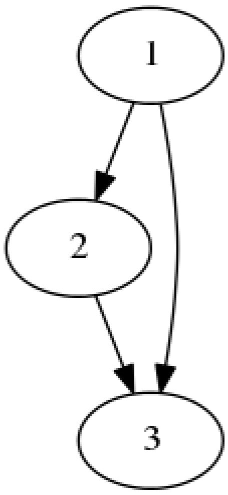
Figure 1: CFG for the ticketprice function.

That may look a bit big, ugly, and intimidating, but constructing a CFG is an easy, mechanical process. All you need to remember is how the control structures in the language come out in the CFG:

An if statement like this:

```
1   if CONDITION:
2     do_this()
3   continue_on_doing_stuff()
```

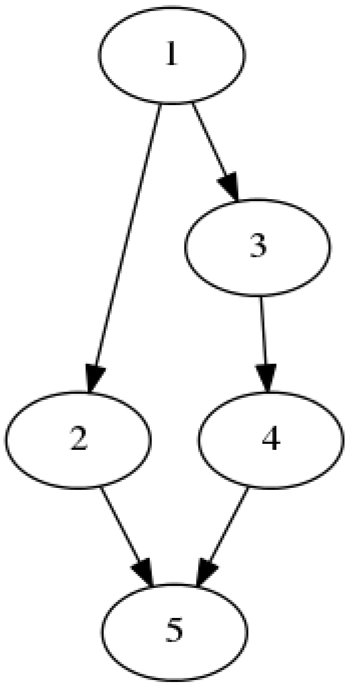
Results in a CFG section that looks like this:



An if-else statement like this:

```
1   if CONDITION:
2       do_this()
3   else:
4       do_that()
5   continue_doing_other_things()
```

Gives you a CFG section as follows:

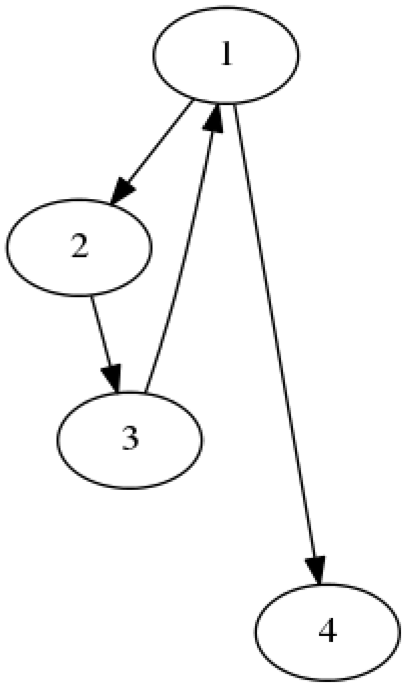


Note that for analysing structural coverage, only the decision nodes really matter, and that node 3 and 4 could be combined in many CFGs. I am just using separate nodes to make relating the CFG back to the source code easier.

A simple while loop like this:

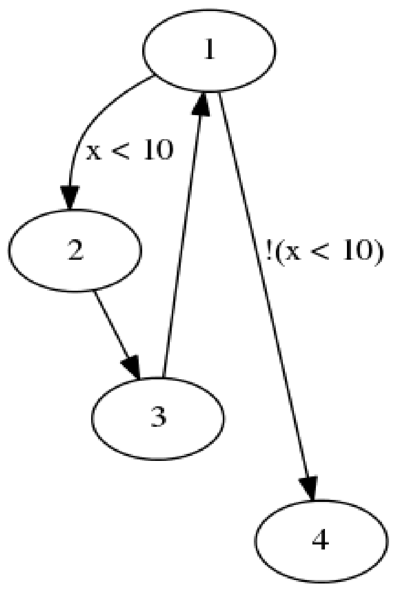
```
1   while x < 10:
2      print x
3      x+=1
4   print "done"
```

Results in a CFG like this:



$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dddr] & \\
2 \arrow[d] & & \\
3 \arrow[uur] & & \\
 & & 4
\end{tikzcd}$$

For some purposes, it can be useful to annotate the edges of decision nodes with the conditions that mean that particular edge is taken; for instance, the while loop CFG could look something like this:



Many different notations or styles are used for representing CFG. We stick with the line numbers as it makes it easier to understand the flow.

Whitebox testing techniques

We will cover the following coverage techniques in this chapter:

- Line coverage (and statement coverage)
- Block coverage
- Branch/Decision coverage
- Condition (Basic and Condition + Branch) coverage
- Path coverage
- MC/DC coverage

Let us discuss whitebox testing techniques (coverage) one by one.

Line (Statement) Coverage

It is defined as

All statements (or lines) in the program[2] should be executed at least once.

As the name suggests, when determining the line coverage, we look at the amount of lines of code that are covered by the tests (more specifically, by at least one test).

Let us look at the example:

[2] In the modern context, replace "program" with "software artifact being tested".

Example 2: Blackjack Game

In blackjack if a player gets 21 points, will win. **Given the points of two different players, the program must return the number of points the one who wins has.** So, if a player has 21 points it will lose. If both players have more than 21 points the programme must return 0. Let us look at the programme written by one of the developers as under:

```
 def blackjack_play(left, right):
1    ln = left
2    rn = right
3    if ln > 21:
4        ln = 0
5    if rn > 21:
6        rn = 0
7    if ln > rn:
8        return ln
9    else:
10        return rn
```

Let's say we use the following inputs for the `blackjack_play` method `blackjack_play(30,30)`. In this case, the programme will execute all lines except the line 8. This means 9 out of 10 lines are executed i.e. $\frac{9}{10}$ i.e. 90%. Let's say `blackjack_play(10,9)`. In this case the line no 9 will be executed, giving us 100% coverage. So two tests that are `blackjack_play(30,30)` and `blackjack_play(10,9)` gives us 100% line coverage.

More formally we can define line coverage as

$$linecoverage = \frac{no.\ of\ lines\ covered}{total\ no.\ of\ lines} * 100$$

Note: Defining what constitutes a line is up to the tester. One might count, for example, the method declaration as a code line. We prefer not to count the method declaration line. That is why I skipped it in my explanation above.

Why is line coverage a bit problematic?

Using lines of code as a way to determine line coverage is a simple and straightforward idea. However, counting the covered lines is not always a good way of calculating the coverage. The amount of lines in a piece of code is heavily dependent on the programmer that writes the code. In Java, for example, you can often write a whole method in just one line (for your future colleagues' sake, please do not). In that case, the line coverage would always be 100% if you test the method.

Example 3:

We are again looking at Blackjack. The play method can also be written in 6 lines, instead of 10 in Java programming language

```
def blackjack_play(left, right):
    ln = left
    rn = right
    if ln > 21: ln = 0
    if rn > 21: rn = 0
    if ln > rn: return ln
    else:return rn
```

Thus, we are getting 100% coverage with `blackjack_play(30,30),` which is not the ideal representation of the coverage.This urges for a better representation of source code. One that is independent of the developers' personal code styles.

Branch (Decision) Coverage

Complex programs often use a lot of conditions (e.g. if-statements). When testing these programs, aiming at 100% line or block coverage might not be enough to cover all the cases we want. We need stronger criteria.

Branch coverage works the same as line/statement coverage. This time, however, we do not count lines or blocks, but the number of possible decision outcomes our program has. Whenever you have a decision block, that decision block has two outcomes. We consider our test suite to achieve 100% branch coverage (or decision coverage, as both terms mean the same) whenever we have tests exercising all the possible outcomes.

$$branch\ coverage\ =\ \frac{decision\ covered}{decisions\ total} * 100$$

In practice, these decisions (or branches) are easy to find in a CFG. Each arrow with true or false (so each arrow going out of a decision) is a branch. Let’s look at another example.

Example 4: Count

The code is written as under:

```
 def count(word):
1    word = 0
2    last = ''
3    for i in word:
4        if not is_character() and (last == 's' or last == 'r'):
5            words = 1
6        last = character_at_location(i)
7    if last == 'r' or last == 's':
8        words + 1
9    return words
```

The CFG looks as under

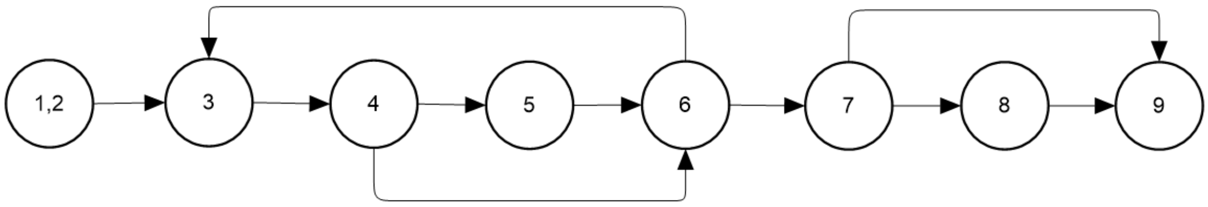


Figure 2: CFG

(Basic) Condition Coverage

Branch coverage gives two branches for each decision, no matter how complicated this decision is. When a decision gets complicated, i.e., it contains more than one condition like a `> 10 && b < 20 && c < 10`, branch coverage might not be enough to test all the possible outcomes of all these decisions. Look at this example: a test `T1 (a=20, b=10, c=5)` and a test `T2 (a=5, b=10, c=5)` already covers this decision block. However, look how many other possibilities we have for this branch to be evaluated to false (e.g., *`T3 (a=20, b=30, c=5)`*, …).

When using condition coverage as criteria, we split the decisions into single conditions. Instead of having one big decision block with the entire condition, we have multiple decision blocks, each one of one condition only. In practice, now we will exercise each condition separately, and not only the “big decision block”.

The formula is basically the same, but now we just have more decision outcomes to count:

$$conditioncoverage = \frac{conditions\ outcome\ covered}{conditions\ total\ outcome} * 100$$

We achieve 100% condition coverage whenever all the outcomes of all the conditions in our program have been exercised. In other words, whenever all the conditions have been true and false at least once.

Condition + Branch Coverage

Let us think carefully about condition coverage. If we only focus on exercising the individual conditions themselves, but do not think of the overall decision, we might end up in a situation like the one below.

```
def hello (a,b):
   if (a>10 and b>20):
      print("Hello")
   else:
      print("Hi")
```

Imagine if `(a > 10 && b > 20)` condition. A test `T1 = (20, 10)` makes the first condition `a > 10` to be true, and the second condition `b > 20` to be false. A test `T2 = (5, 30)` makes the first condition to be false, and the second condition to be true. Note that `T1` and `T2` together achieve 100% basic condition coverage; after all, both conditions have been exercised as true and false. However, the final outcome of the entire decision was also false! This means, we found a case where we achieved 100% basic condition coverage, but only 50% branch coverage! This is no good. Therefore, we called it basic condition coverage.

In practice, whenever we use condition coverage, we actually do branch + condition coverage. In other words, we make sure that we achieve 100% condition coverage (i.e., all the outcomes of all conditions are exercised) and 100% branch coverage (all the outcomes of the decisions are exercised).

From now on, whenever we mention condition coverage, we mean condition + branch coverage.

$$C/DC\,Coverage = \frac{Conditions\ outcome\ total + decisions\ outcome\ total}{conditions\ outcome\ overed + decision\ outcome\ covered} * 100$$

Path Coverage

Finally, with condition coverage, we looked at each condition individually. This gives us way more branches to generate tests. However, note that, although we are testing each condition to be evaluated to true and false, this does not ensure that we are testing all the paths that a program can have.

Path coverage does not consider the conditions individually; rather, it considers the (full) combination of the conditions in a decision. Each of these combinations is a path. You might see a path as a unique way to traverse the CFG. The calculation is the same as the other coverages:

$$pathcoverage = \frac{pathcovered}{pathtotal}*100$$

Example 11:

Let’s look at the following condition written in Java code

```java
if (!Character.isLetter(str.charAt(i)) & (last == 's' || last == 'r')) {
        words++;
}
```

The decision of this if-statement contains three conditions and can be generalized to `(A && ( B || C))`, with

- `A = !Character.isLetter(str.charAt(i)),`
- `B = last == 's' and`
- `C = last == 'r'.`

To get 100% path coverage, we would have to test all the possible combinations of these three conditions. We construct a truth table to find the combinations:

Test	**A**	**B**	**C**	**Outcome**
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

This means that, for full path coverage, we would need 8 tests just to cover this if-statement. That is quite a lot for just a single statement! The number of combinations might be too big! The number of tests needed for full path coverage will grow exponentially with the amount of conditions in a decision.

MC/DC (Modified Condition/Decision Coverage)

Modified condition/decision coverage (MC/DC) looks at the combinations of conditions like path coverage does. However, instead of aiming at testing all the possible combinations, we follow a process in order to identify the "important" combinations. The goal of focusing on these important combinations is to manage the large number of test cases that one needs to devise when aiming for 100% path coverage.

The idea of MC/DC is to exercise each condition in a way that it can, independently of the other conditions, affect the outcome of the entire decision. In short, this means that every possible condition of each parameter must have influenced the outcome at least once.

If we take the decision block from path coverage example, A && (B || C), MC/DC dictates that:

- For condition A:
  - There must be one test case where A=true (say T1).
  - There must be one test case where A=false (say T2).
  - T1 and T2 (which we call independence pairs) must have different outcomes (e.g., T1 makes the entire decision to evaluate to true, and T2 makes the entire decision to evaluate to false, or the other way around).
  - In both test cases T1 and T2, variables B and C should be the same.

- For condition B:
  - There must be one test case where B=true (say T3).
  - There must be one test case where B=false (say T4).
  - T3 and T4 have different outcomes.
  - In both test cases T3 and T4, variables A and C should be the same.

- For condition C:
  - There must be one test case where C=true (say T5).
  - There must be one test case where C=false (say T6).
  - T3 and T4 have different outcomes,
  - In both test cases T3 and T4, variables A and B should be the same.

Cost-wise, a relevant characteristic of MC/DC coverage is that, supposing that conditions only have binary outcomes (i.e., `true` or `false`), the number of tests required to achieve 100% MC/DC coverage is, on average, `N+1`, where `N` is the number of conditions in the decision. Note that `N+1` is definitely smaller than all the possible combinations (2^N)!

Again, to devise a test suite that achieves 100% MC/DC coverage, we should devise `N+1` test cases that, when combined, exercise all the combinations independently from the others.

The question is how to select such test cases. See the example below.

Imagine a program that decides whether an applicant should be admitted to the 'University of Character':

```
def admission (degree, experience, character):
    if character and (degree or experience):
        print("Admitted")
    else:
        print("Rejected")
```

The program takes three booleans as input (which, generically speaking, is the same as the `A && (B || C)` we just discussed):

- Whether the applicant has a good character (`true or false`),
- Whether the applicant has a degree (`true or false`),
- Whether the applicant has experience in a field of work (`true or false`).

If the applicant has good character and either a degree or experience in the field, he/she will be admitted. In any other case the applicant will be rejected.

To test this program, we first use the truth table for `A && (B || C)` to see all the combinations and their outcomes. In this case, we have `3` decisions and 2^3 is `8`, therefore we have tests that go from `1` to `8`:

Test	**A**	**B**	**C**	**Outcome**
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Our goal will be to apply the MC/DC criterion to these test cases, and select `N+1`, in this case `3+1=4`, tests. In this case, the 4 four tests that satisfy that MC/DC coverage is {2, 3, 4, 6}.

How did we find them? We go test by test, condition by condition.

We start with selecting the pairs of combinations (or tests) for the `Character` parameter.

- In test 1, we see that `Character`, `Degree`, and `Experience` are all `true` and the `Decision` (i.e., the outcome of the entire boolean expression) is also `true`. We now look for another test in this table, where only the value of Character is the opposite of the value in test 1, but the others (`Degree` and `Experience`) are still the same. This means we have to look for a test where `Character = false`, `Degree = true`, `Experience = true`, and `Decision = false`. This combination appears in test 5.

  Thus, we just found a pair of tests (again, called independence pairs), T_1 and T_5, where `Character` is the only parameter which changed and the outcome (`Decision`) changed as well. In other words, a pair of tests where the `Character` independently influences the outcome (`Decision`). Let's keep the pair {T_1, T_5} in our list of test cases.

- We could have stopped here and moved to the next variable. After all, we already found an independence pair for `Character`. However, finding them all might help us in reducing the number of test cases at the end, as you will see. So, let us continue and we look at the next test. In test 2, `Character = true`, `Degree = true`, `Experience = false`, and `Decision = true`. We repeat the process and search for a test where `Character` is the opposite of the value in test 2, but `Degree` and `Experience` remain the same (`Degree = true, Experience = false`). This set appears in test 6.

  This means we just found another pair of tests, T_2 and T_6, where `Character` is the only parameter which changed and the outcome (`Decision`) changed as well.

- Again, we repeat the process for test 3 (`Character = true`, `Degree = false`, `Experience = true`) and find that the Character parameter in test 7 (`Character = false`, `Degree = false`, `Experience = true`) is the opposite of the value in test 3 and changes the outcome (`Decision`).

- For test 4 (`Character = true`, `Degree = false`, `Experience = false`). Its pair is test 8 (`Character = false`, `Degree = false`, `Experience = false`). Now, the outcome of both tests is the same (`Decision = false`). This means that the pair {T_4, T_8} does not show how `Character` can independently affect the overall outcome; after all, `Character` is the only thing that changes in these two tests, but the outcome is still the same.

As we do not find another suitable pair when repeating the process for tests `5, 6, 7 and 8`, we move on from the `Character` parameter to the `Degree` parameter. We repeat the same process, but now we search for the opposite value of parameter `Degree` whilst `Character` and `Experience` stay the same.

- For test 1 (`Character = true, Degree = true, Experience = true`), we search for a test where (`Character = true, Degree = false, Experience = true`). This appears to be the case in test `3`. However, the outcome for both test cases stay the same. Therefore, {T_1, T_3} does not show how the `Degree` parameter can independently affect the outcome.

- After repeating all the steps for the other tests, we find only {T_2, T_4} to have different values for the `Degree` parameter where the outcome also changes.

- Finally, we move to the Experience parameter. As with the `Degree` parameter, there is only one pair of combinations that will work, which is {T_3, T_4}.

We highly recommend carrying out the entire process yourself to get a feeling of how the process works!

We now have all the pairs for each of the parameters:

- Character: {1, 5}, {2, 6}, {3, 7}
- Degree: {2, 4}
- Experience: {3, 4}

Having a single independence pair per variable (`Character`, `Degree` and `Experience`) is enough. After all, we want to minimise the total number of tests, and we know that we can achieve this with `N+1` test.

We do not have any choices with conditions `Degree` and `Experience`, as we found only one pair of tests for each parameter. This means that we have to test combinations `2`, `3` and `4`.

Lastly, we need to find the appropriate pair of `A`. Note that any of them would fit. However, we want to reduce the total amount of tests in the test suite (and again, we know we only need 4 in this case).

If we were to pick either test 1 or test 5, we would have to include either test 5 or test 1 as well, as they are their opposites, but therefore unnecessarily increasing our number of tests. In order to keep our test cases in accordance to `N+1` or in this case `3+1`, thus `4` test cases we can either add test `6` or test `7`, as their opposites (test 2 or 3) are already included in our test cases. Randomly, we pick test 6.

Therefore, the tests that we need for 100% MC/DC coverage are {2, 3, 4, 6}. These are the only 4 tests we need. This is indeed cheaper when compare to the 8 tests we would need for path coverage.

Let us now discuss some details about the MC/DC coverage:

- We have applied what we call unique-cause MC/DC criteria. We identify an independence pair `(T1, T2)`, where only a single condition changes between `T1` and `T2`, as well as the final outcome. That might not be possible in all cases. For example, `(A and B) or (A and C)`. Ideally, we would demonstrate the independence of the first A, B, the second A, and C. It is however impossible to change the first A and not change the second A. Thus, we cannot demonstrate the independence of each A in the expression. In such cases, we then allow A to vary, but we still fix all other variables (this is what is called masked MC/DC).

- It might not be possible to achieve MC/DC coverage in some expressions. See `(A and B) or (A and not B)`. While the independence pairs (TT, FT) would show the independence of A, there are no pairs that show the independence of B. While logically possible, in such cases, we recommend the developer to revisit the (degenerative) expression as it might had been poorly designed. In our example, the expression could be reformulated to simply A.

- Mathematically speaking, `N+1` is the minimum number of tests required for MC/DC coverage (and `2 * N` the theoretical upper bound). However, empirical studies indeed show that `N+1` is often the required number of tests.

Summary

- White box testing we investigate the internal structures and working of the system.
- Line coverage is all lines or statements should be executing at least once.
- Condition coverage splits conditions and use that for calculating the coverage

- Path coverage does not consider the conditions individually; rather, it considers the (full) combination of the conditions in a decision.
- Modified condition/decision coverage (MC/DC), looks at the combinations of conditions like path coverage does. However, instead of aiming at testing all the possible combinations, we take a certain selection process to identify the "*important ones*".

References

1. Accessible at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF (The context of the quote was interesting. Dijkstra was unfavourably contrasting testing to formal methods, which overall have turned out to be a whole lot less useful than he hoped).
2. Software Testing: From Theory to Practice accessible at https://sttp.site/
3. Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
4. Chapter 12 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
5. Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys (csur), 29(4), 366-427.
6. Cem Kaner on Code Coverage: http://www.badsoftware.com/coverage.htm