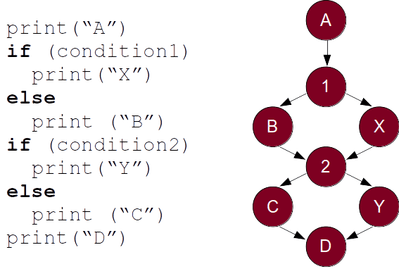
**Cyclomatic Complexity**

Cyclomatic complexity is a software measure, developed in 1976 by Thomas McCabe [[1]](http://www.literateprogramming.com/mccabe.pdf) to measure the complexity of a program (class, method, routine, etc.). In more current terms, it can be said that it indicates the difficulty of building unit tests on a given code since it measures **the number of linearly independent paths** in this code. More current measures such as [cognitive complexity](http://artesoftware.com.br/complexidade-cognitiva/) may better indicate the difficulty of reading and understanding a code, but cyclomatic complexity still has a strong relationship with testability.

**Linearly Independent Path**

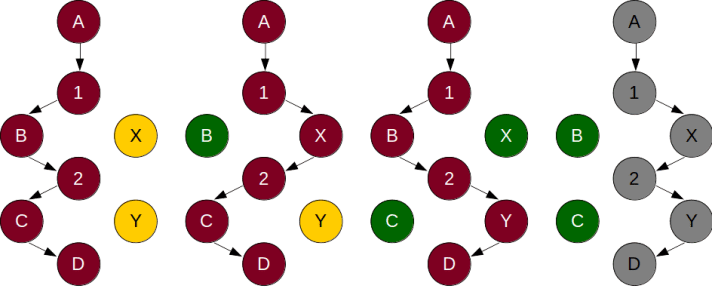
*Linearly Independent Path is any program path that introduces at least one new set of processing commands or a* ***new condition****. When defined in terms of flow graph, an independent path must include* ***at least one edge*** *that was not traversed before the path was defined (Pressman) [2].*

In simpler terms, when writing a unit test to go through all linearly independent paths, you will probably have covered all the lines of this code, establishing 100% coverage. This is not the same thing as covering all the possibilities of execution (NPATH), but going through all the lines at least once. The example in figure 1 shows, graphically, an algorithm where the letters represent a simple instruction and the numbers represent an “IF” (decision). Note that from a simple statement there is only one arrow (edge) while from a decision statement there are two.



*Example algorithm*

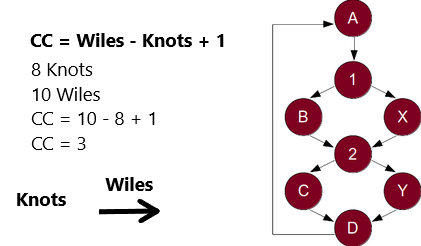
In Figure 2, the same algorithm is broken down into its possible paths (NPATH) which are 4, but only the first three are linearly independent. Note that the last path does not pass any new instructions that have not been passed before in one of the previous three paths.

*Figure 2: Three independent paths and a fourth repeating instructions*

In the example above, although there are 4 possible paths (NPATH), we can say that the cyclomatic complexity of the graphically represented algorithm is 3, that is, it is equal to the number of linearly independent paths. If we put one more “IF” in this algorithm we will see that the number of possible paths will jump to 8 in an exponential progression. However, the cyclomatic complexity will go to 4, as another node (ball) and two edges (arrows) will be added.

**Control Flow Graph Calculation**

For “simple programs” (routines, methods, functions, etc.) with only one output, the calculation of linearly independent paths is done considering that the algorithm is strongly connected and a return line must be made until the beginning of the algorithm as in example in Figure 3. With the drawing done in this way, just subtract the number of edges by the number of nodes in the resulting graph and add one. Thus, the cyclomatic complexity increases as decision instructions are inserted into the code. Likewise, there is a growing need for test cases to cover all of this code.



*Graph with knots and edges*

Based on this form of calculation, any new decision introduced in the algorithm will increase the cyclomatic complexity by 1 point, as it adds a node with two edges. Following this reasoning, the calculation can be expanded to programs with more than one output by adding an extra point for each output. In this way, the formula for calculating cyclomatic complexity can be simplified to “**π - s + 2**” where “π” is the number of decision points and “s” is the number of exit points. Through this formula it is possible to calculate values ​​of cyclomatic complexity without the need to draw the control flow graph, just identifying which reserved words of the languages ​​correspond to decisions or returns before the end of the program, method, routine or function.

**Calculation by Reserved Word**

As you may have noticed, it will not be possible to draw a graph of the graph represented by the algorithm every time we need to calculate complexity cyclomatic of some code. It is actually much simpler to apply the formula “**π - s + 2**” and count the reserved words that represent a decision in the code together with the program's early exit points. This simplified calculation can be described as follows:

1. The value starts at “1” for the method, function or routine (with or without return)

1. Add one more point for each element below:

2.1Selection – if, case

2.2 Loops – for, while, do-while, break e continue

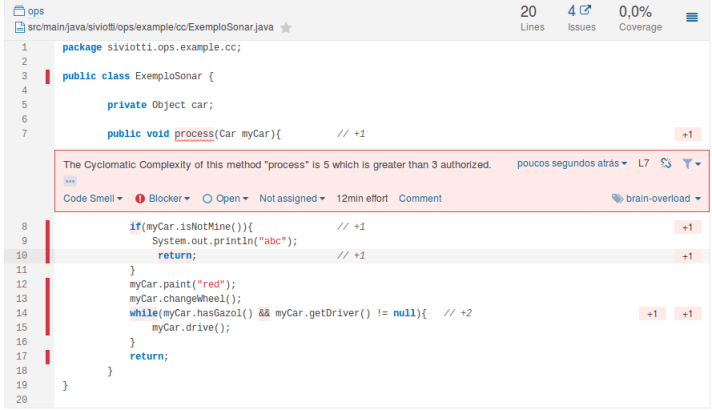
Operators – “&&”, “||”, “?”

Exceptions – catch, throw e throws

2.5 Flow - return that is not the last

Note: else, default, finally, “:” and the last return do not increase the count

Figure 4 shows a Java example of this count made by SonarQube. The SonarQube is one of the most used tools for measuring this type of measurement. Note that it indicates with a "+1" the points that fall under one of the increment conditions presented previously in the calculation formula. In the example in question, the complexity limit cyclomatic was lowered to 3 so that Sonar created a Issue and visualization was possible this way, but it is calibrated with a limit of 10 for this measure.

*Figure 4: Example of Java code counting done by Sonar*

The calculation of the example gives a total value of 5 , since one is added to the method (line 7), plus one by the first “IF” (line 8), plus one by “return ”which is not the last, plus one for“ while ”and“ && ”(line 14). Note that the last “return” does not count.

Once the form of calculation based on reserved words is known, it is possible to calculate cyclomaticcomplexity in several languages ​​in the same way, just identify the reserved words that represent a decision or early exit from the program.

**Ice Cream Shop Example - Requirement and Implementation 1**

Imagine an ice cream shop that sets the price of your ice cream as follows:

1. There are two types of ice cream: Common, whose price is R $ 15 and Premium whose price is R$ 20

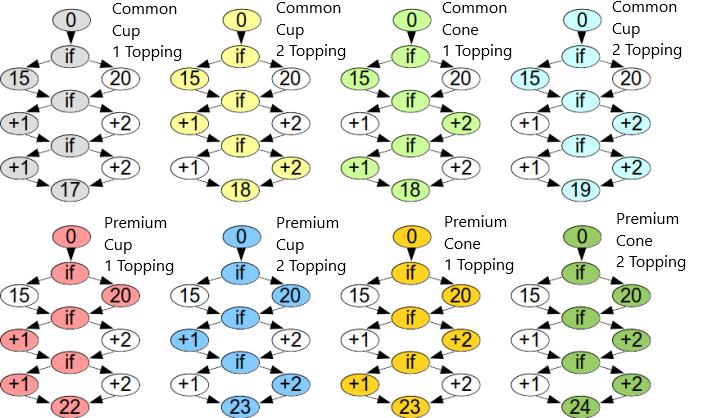
1. Ice cream can be sold in a cup or cone. The cup adds R $ 1 to the price while the cone adds R $ 2.

3. The coverage can be simple (just one) costing R $ 1 or special (two coverages or more) costing R$ 2.

The calculation of the ice cream price can be represented by the following code written in java (implementation 1):

*public int priceIce cream (boolean premium, boolean cone, int coverings) {  
  int price = 0;  
  if (premium) {  
    price = 20;  
  } else {  
    } else {  
  }  
  if (coverings) {  
    price = price +2;  
  } else {  
    price = price +1;  
  }  
  if (coverings > 1){  
    price = price +2;  
  } else {  
    price = price +1;  
  }  
  return prices;  
 }*

The method expects three parameters based on the price definition of the ice cream where the customer must inform three things: type of ice cream (whether it is premium or not), container (whether in the cone or not) and cover (one or more). From these three pieces of information, the algorithm calculates the price of the ice cream. We have eight possible combinations of these parameters, generating eight options for purchasing the ice cream as can be seen in Figure 5.

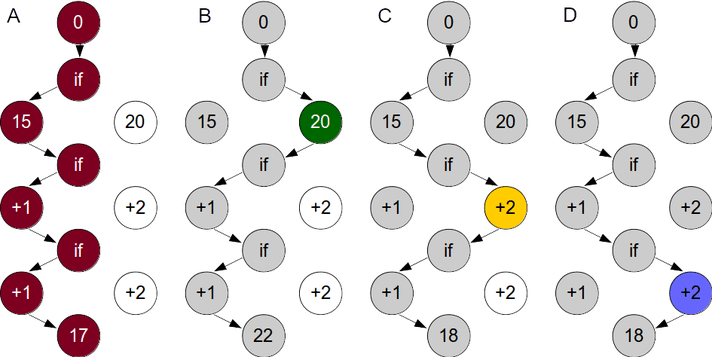
*Figure 5: Eight possible combinations of ice cream and its final price*

The eight combinations correspond to the eight possible paths of the price calculation algorithm. One way to get 100% test coverage would be to make a test case for each path as shown in the following code:

*assertEquals (17, sorbeteria.precoSorvete (false, false, 1)); // Common-Cup-1Cob  
 assertEquals(18, sorveteria.precoSorvete(false, false, 2)); // Comum-Copinho-2Cob>  
 assertEquals (18, sorbeteria.precoSorvete>(false, true, 1)); // Common-Cup-1Cob  
 assertEquals (19, sorbeteria.precoSorvete>(false, true, 2)); // Common-Cup-2Co  
 assertEquals (22, sorbeteria.precoSorvete>(true, false, 1)); // Common-Cup-1Co  
 assertEquals (23, sorbeteria.precoSorvete>(true, false, 2)); // Common-Cup-2Co  
 assertEquals (23, sorbeteria.precoSorvete>(true, true, 1)); // Common-Cup-1Co  
 assertEquals (24, sorbeteria.precoSorvete>(true, true, 2)); // Common-Cup-2Co*

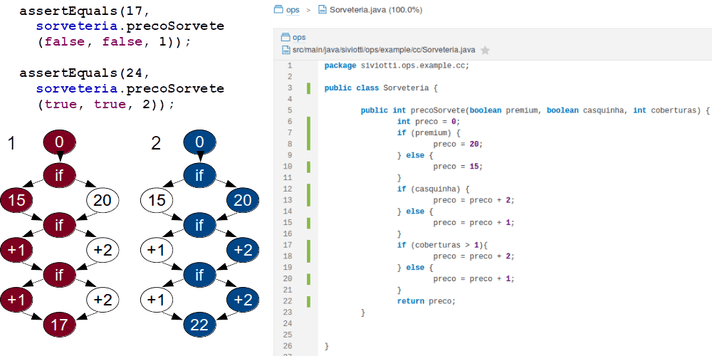
However, it is not difficult to see what will happen with this test if more “IFs” are added. The growth in the number of test cases will be exponential and soon it will be impossible to maintain high coverage for this algorithm. Later on we will seethat the cyclomatic complexity is a much better indicator for suggesting the number of test cases needed in a unit test.

Despite eight combinations, we already know that the total number of possible paths (NPATH) is different from the cyclomatic complexity. When applying the calculation by keyword on the algorithm we arrive at a value of cyclomatic complexity equal to4 (one point for the method and one more for each of the three “Ifs”). This value can be viewed graphically in Figure 6.

*Figure 6: Four paths linearly independent of the ice cream shop algorithm*

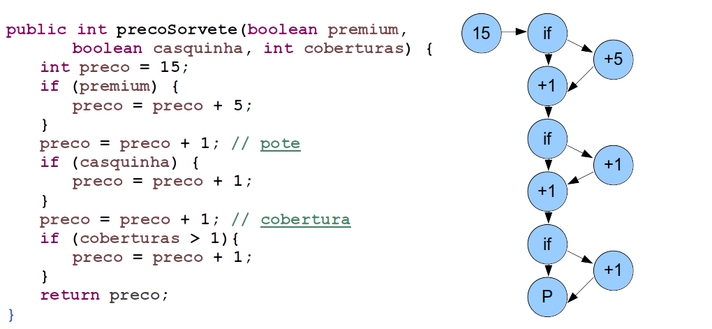
Path A highlights in red all instructions where the algorithm passed. Path B highlights in green the price instruction premium where path A had not passed and gray where it passed again. In path C, the instruction where the execution goes through the instruction of the cone price was highlighted in yellow and path D shows the passage (in blue) for the instruction of the most expensive price of the double roof.
<segment 0104> Thus, all nodes have been traversed at least once and we verify graphically because the cyclomatic complexity is 4.

Notice now, in Figure 7, that only two test cases are sufficient to cover 100% of the lines of code. This is due to the particularity of how the requirement was implemented. The algorithm in question has aspecies of main path and an alternative path. Thus, only two scenarios ended up passing through all its lines. Later on, the algorithm will be changed without changing its result and we will see that the number of test cases needed will change.

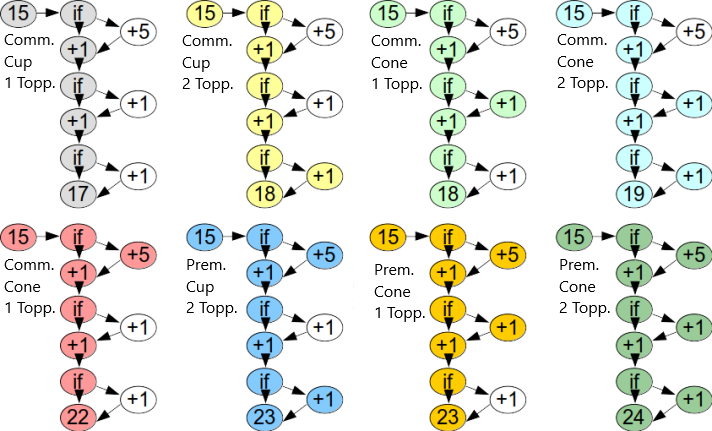
*Figure 7: Two test scenarios covering 100% of the lines of code*

**Exemplo da Sorveteria – Implementação 2**

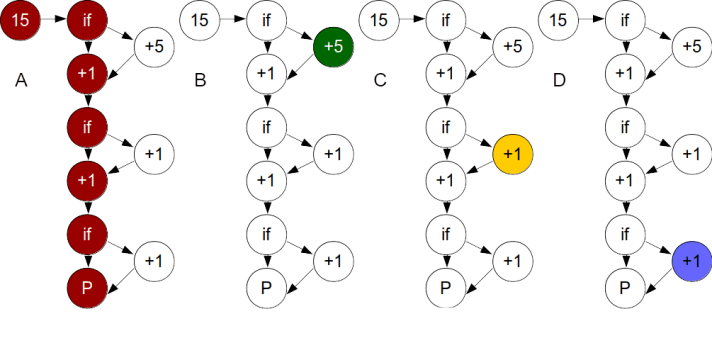
The implementation of the ice cream price requirement with If Else generated the need for two test cases. Figure 8 shows an even simplerimplementation where the main path becomes more explicit. Notice that the graph also changes.

*Figure 8: Alternative implementation with main path and without “elses”  
(implementation 2)*

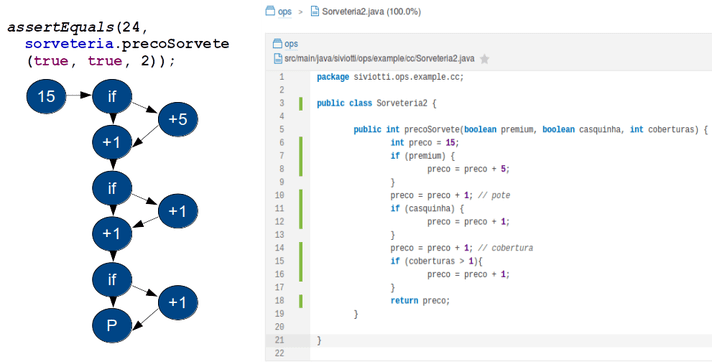
In Figure 9 it is possible to see that in this implementation the eight ice cream possibilities still exist and in Figure 10 that the cyclomatic complexity continues4 (one of the method plus one for each “IF” - the “elses” do not interfere).

*Figure 9: Eight possible paths for implementation 2*

See, graphically, the cyclomatic 4 complexity:

*Figura 10: Quatro caminhos linearmente independentes da implementação 2*

This new implementation generated a situation where the number of test scenarios for covering all lines dropped to one. With a single test scenario (the most expensive ice cream) all lines are covered. This does not mean that this implementation nly needs one scenario in its unit test, only that with just one (the right one) all lines will be covered.

*Figure 11: Test scenario for testing implementation 2  
(only one scenario covers 100%)*

**Ice Cream Shop Example - New Requirement and Implementation 3**

In a third implementation where the requirement has changed slightly the “IFs ” will be nested, making the code a little more difficult to read. In this scenario, only the premium ice creams have a cone and only the cones have more than one icing. The possible behavior and paths (now 4) are different from the previous two, however the cyclomatic complexity is still 4. Another change in this implementation is the need for test cases so that all lines are covered. In Figure 12, this new implementation and its corresponding graph can be seen.

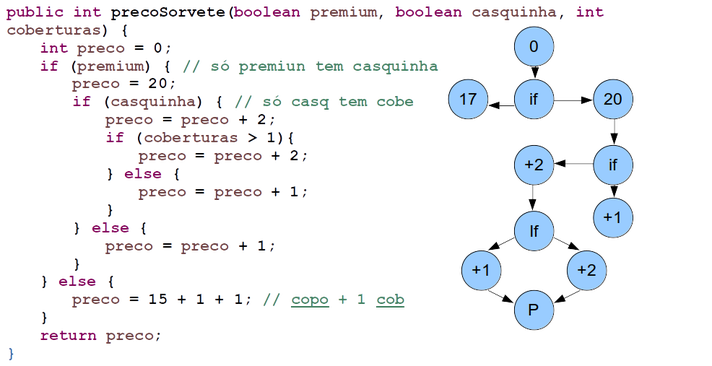
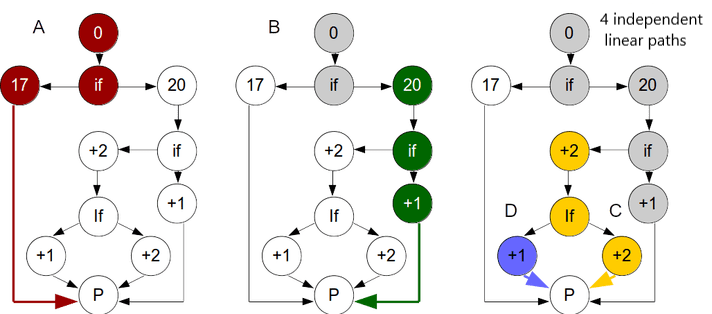
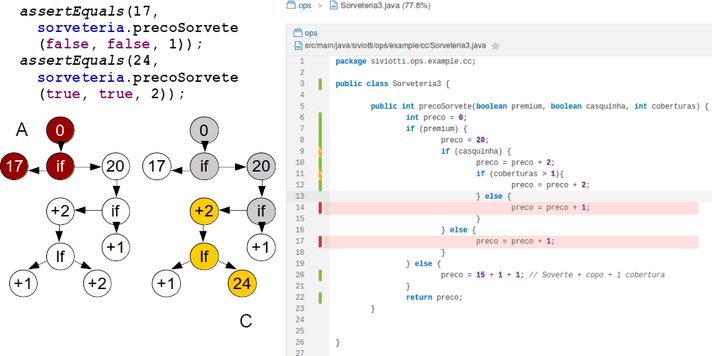
*Figure 12: Implementation 3 with nested IFs*

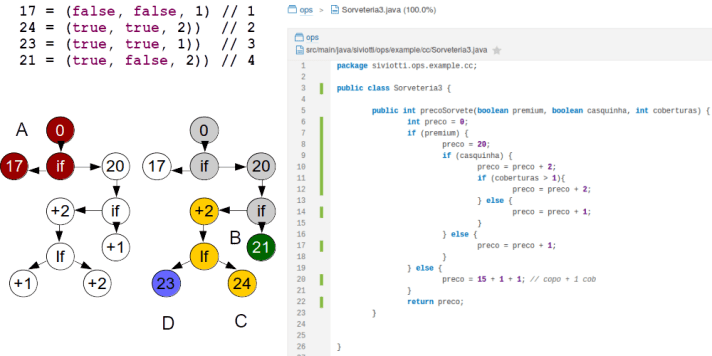
Figure 13 shows the cyclomatic complexity overlaid with the four possible paths (A: red, B: green, C: yellow and D: blue).

*Figure 13: Paths and complexity cyclomatic of the implementation 3*

Testing coverage now requires a little more scenarios. In Figure 14, it can be seen that the same two scenarios (cheapest 17 and most expensive 24) that covered 100% of the lines in the 1 implementation now cover only 77.6% of the lines.

  
*Figure 14: Two test scenarios covering 77.6% of the implementation lines 3*

Two other test scenarios are needed so that 100% of the lines can be covered as can be seen in Figure 15.

*Figure 15: Four test scenarios covering 100% of the implementation lines 3*

**Conclusion**

As shown, the cyclomatic complexity, whose initial objective was to measure the complexity of programs, has a strong relationship with testability. As seen in the three different implementations of the ice cream parlor problem, it is not feasible to develop a test case for each possibility of executing an algorithm, because even simple implementations will generate unit tests that are much more complex than they are. and will end up being abandoned over time. On the other hand, the complexity value cyclomatic seems to be a much more reasonable goal for the elaboration of test scenarios for a given source code. It can be said that the cyclomatic complexity is the **upper limit on the number of test scenarios needed to cover 100% of the lines of code** . So, as seen in the examples, it may be necessary a little more or a little less, depending on the implementation, but the number of linearly independent paths is a good guess for the quantity and the points of passing of the test scenarios. From a practical and software quality point of view, the cyclomatic complexity value can be used as a measure of the quality of a code, as well as other measures, indicating its complexity, but especially its difficulty of implement unit tests.