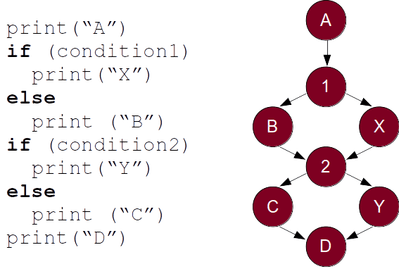
**Complexidade Ciclomática**

Complexidade ciclomática é uma medida de software, elaborada em 1976 por Thomas McCabe [[1]](http://www.literateprogramming.com/mccabe.pdf) para medir a complexidade de um programa (classe, método, rotina etc). Em termos mais atuais, pode-se dizer que ela indica a dificuldade de se construir testes de unidade em um determinado código uma vez que ela mede a **quantidade de caminhos linearmente independentes** neste código. Medidas mais atuais como a [complexidade cognitiva](http://artesoftware.com.br/complexidade-cognitiva/) podem indicar melhor a dificuldade de leitura e entendimento de um código, mas a complexidade ciclomática ainda mantém forte relação com a testabilidade.

**Caminho Linearmente Independente**

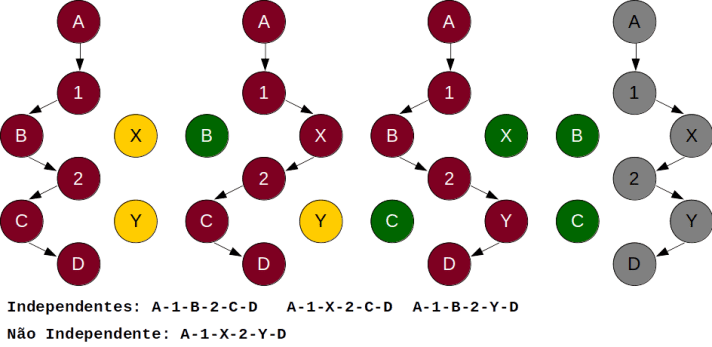
*Caminho Linearmente Independente é qualquer caminho do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma****nova condição****. Quando definido em termos de grafo de fluxo, um caminho independente deve incluir****pelo menos uma aresta****que não tenha sido atravessada antes de o caminho ser definido (Pressman)[2].*

Em termos mais simples, ao escrever um teste de unidade para passar por todos os caminhos linearmente independentes você provavelmente terá coberto todas as linhas deste código, estabelecendo uma cobertura de 100%. Isso não é a mesma coisa que cobrir todas as possibilidades de execução (NPATH) e sim passar por todas as linhas ao menos uma vez.O exemplo da figura 1 mostra, graficamente, um algoritmo onde as letras representam uma instrução simples e os números representam um “IF” (decisão). Repare que de uma instrução simples parte apenas uma seta (aresta) enquanto de uma instrução de decisão partem duas.



*Figura 1: Algoritmo de exemplo*

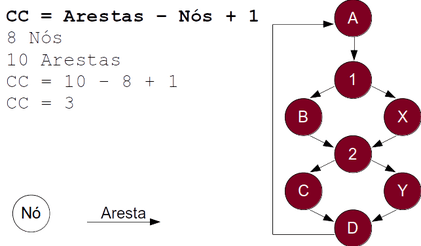
Na Figura 2, o mesmo algoritmo é desmembrado em seus caminhos possíveis (NPATH) que são 4, porém somente os três primeiros são linearmente independentes. Repare que o último caminho não passa em nenhuma instrução nova pela qual já não tenha sido passada antes em um dos três caminhos anteriores.

*Figura 2: Três caminhos independentes e um quarto repetindo instruções*

No exemplo acima, apesar de haver 4 caminhos possíveis (NPATH), podemos dizer que a complexidade ciclomática do algoritmo representado graficamente é 3, ou seja, é igual ao número de caminhos linearmente independentes. Se colocarmos mais um “IF” neste algoritmo veremos que o número de caminhos possíveis vai saltar para 8 numa progressão exponencial. No entanto, a complexidade ciclomática irá para 4, pois será adicionada mais um nó (bolinha) e duas arestas (setas).

**Cálculo pelo Grafo de Fluxo de Controle**

Para “programas simples” (rotinas, métodos, funções etc) com apenas uma saída, o cálculo de caminhos linearmente independentes é feito considerando-se que o algoritmo é fortemente conectado e uma linha de retorno deve ser feita até o início do algoritmo como no exemplo da Figura 3. Com o desenho feito desta forma, basta subtrair o número de arestas pelo número de nós do grafo resultante e somar um. Assim, a complexidade ciclomática cresce na medida em que instruções de decisão são inseridas no código. Da mesma forma, cresce a necessidade de casos de teste para cobrir todo este código.



*Figura 3: Grafo com nós e arestas*

Com base nessa forma de cálculo, qualquer nova decisão introduzida no algoritmo vai fazer a complexidade ciclomática aumentar em 1 ponto, pois adiciona um nó com duas arestas. Seguindo este raciocínio, o cálculo pode ser expandido para programas com mais de uma saída adicionando um ponto extra para cada saída. Dessa forma, a fórmula de cálculo da complexidade ciclomática pode ser simplificada para “**π – s + 2**” onde “π” é a quantidade de pontos de decisão e “s” é a quantidade de pontos de saída. Através desta fórmula é possível que sejam calculados valores de complexidade ciclomática sem a necessidade de desenhar o grafo de fluxo de controle, apenas identificando quais palavras reservadas das linguagens correspondem a decisões ou retornos antes do fim do programa, método, rotina ou função.

**Cálculo por Palavra Reservada**

Como você já deve ter percebido, não será possível fazer um desenho do grafo representado pelo algoritmo toda vez que precisarmos calcular complexidade ciclomática de algum código. É realmente bem mais simples aplicar a fórmula “**π – s + 2**” e contar as palavras reservadas que representam uma decisão no código junto com os pontos de saída antecipada do programa. Este cálculo simplificado pode ser descrito da seguinte forma:

1. O valor inicia em “1” para o método, função ou rotina (com ou sem retorno)

2. Adicione mais um ponto para cada elemento abaixo:

2.1 Seleção – if, case

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

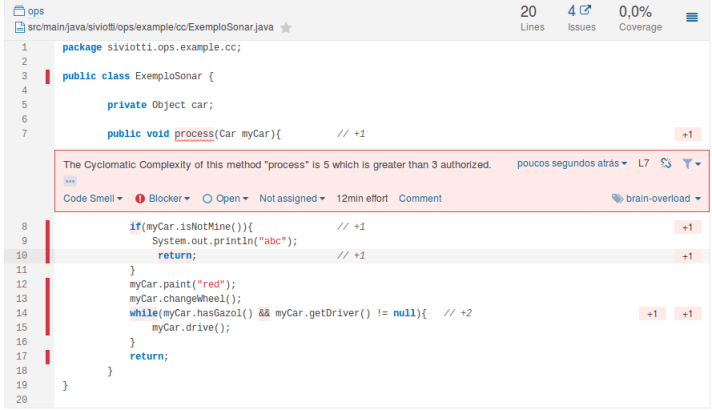
2.3 Operadores – “&&”, “||”, “?”

2.4 Exceções – catch, throw e throws

2.5 Fluxo – return que não seja o último

Obs: else, default, finally, “:” e o último return não incrementam a contagem

A Figura 4 mostra um exemplo em Java desta contagem feita pelo SonarQube. O SonarQube é uma das ferramentas mais usadas para aferição deste tipo de medição. Repare que ele indica com um “+1” os pontos que se enquadram em uma das condições de incremento apresentadas anteriormente na fórmula de cálculo. No exemplo em questão, o limite de complexidade ciclomática foi baixado para 3 para que o Sonar criasse uma Issue e a visualização fosse possível desta forma, mas ele vem calibrado com um limite de 10 para esta medida – veja em [Controle de Complexidade].

*Figura 4: Exemplo de contagem em código Java feito pelo Sonar*

O cálculo do exemplo dá um valor total de 5, pois soma-se um ao método (linha 7), mais um pelo primeiro “IF” (linha 8), mais um pelo “return” que não é o último, mais um pelo “while” e pelo “&&” (linha 14). Repare que o último “return” não conta.

Uma vez que se conheça a forma de cálculo baseado em palavras reservadas é possível calcular complexidade ciclomática em várias linguagens da mesma forma, basta identificar as palavras reservadas que representam uma decisão ou saída precoce do programa.

**Exemplo da Sorveteria – Requisito e Implementação 1**

Imagine uma sorveteria que define o preço de seus sorvetes da seguinte forma:

1. Há dois tipos de sorvete: Comum, cujo preço é R$15 e Premium cujo preço é R$20

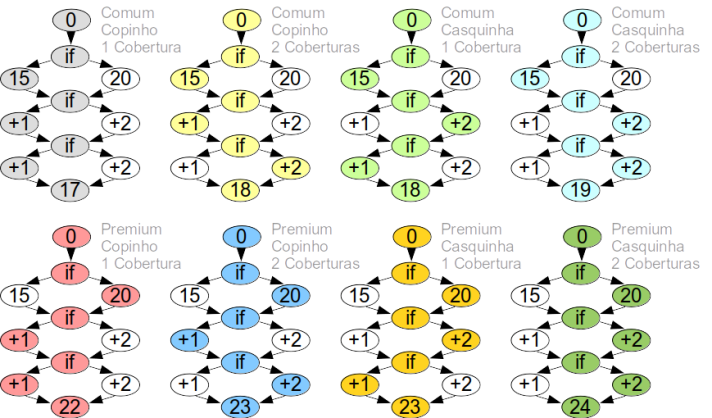
2. O sorvete pode ser vendido em copinho ou casquinha. O copinho adiciona R$1 ao preço enquanto a casquinha adiciona R$2.

3. A cobertura pode ser simples (apenas uma) custando R$1 ou especial (duas coberturas ou mais) custando R$2.

O cálculo do preço do sorvete pode ser representado pelo seguinte código escrito em java (implementação 1):

*public int precoSorvete(boolean premium, boolean casquinha, int coberturas) {  
  int preco = 0;  
  if (premium) {  
    preco = 20;  
  } else {  
    preco = 15;  
  }  
  if (casquinha) {  
    preco = preco + 2;  
  } else {  
    preco = preco + 1;  
  }  
  if (coberturas > 1){  
    preco = preco + 2;  
  } else {  
    preco = preco + 1;  
  }  
  return preco;  
 }*

O método espera três parâmetros baseados na definição do preço do sorvete onde o cliente deve informar três coisas: tipo do sorvete (se é premium ou não), recipiente (se quer na casquinha ou não) e cobertura (uma ou mais). A partir destas três informações o algoritmo calcula o preço do sorvete. Temos oito combinações possíveis destes parâmetros gerando oito opções de compra do sorvete como pode ser visto na Figura 5.

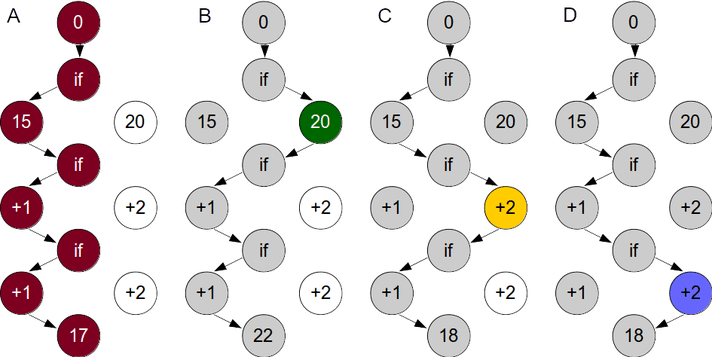
*Figura 5: Oito combinações possíveis do sorvete e seu preço final*

As oito combinações correspondem aos oito caminhos possíveis do algoritmo de cálculo de preço. Uma forma de obter uma cobertura de testes de 100% seria fazer um caso de teste para cada caminho como mostra o código a seguir:

*assertEquals(17, sorveteria.precoSorvete(false, false, 1)); // Comum-Copinho-1Cob  
 assertEquals(18, sorveteria.precoSorvete(false, false, 2)); // Comum-Copinho-2Cob  
 assertEquals(18, sorveteria.precoSorvete(false, true, 1)); // Comum-Casquinha-1Cob  
 assertEquals(19, sorveteria.precoSorvete(false, true, 2)); // Comum-Casquinha-2Cob  
 assertEquals(22, sorveteria.precoSorvete(true, false, 1)); // Premium-Copinho-1Cob  
 assertEquals(23, sorveteria.precoSorvete(true, false, 2)); // Premium-Copinho-2Cob  
 assertEquals(23, sorveteria.precoSorvete(true, true, 1)); // Premium-Casquinha-1Cob  
 assertEquals(24, sorveteria.precoSorvete(true, true, 2)); // Premium-Casquinha-2Cob*

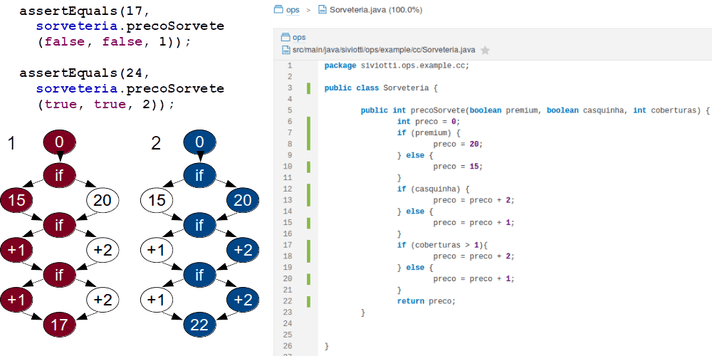
No entanto, não é difícil perceber o que acontecerá com este teste se mais “IFs” forem adicionados. O crescimento do número de casos de teste será exponencial e em breve será impossível manter uma cobertura alta para este algoritmo. Mais a frente veremos que a complexidade ciclomática é um indicador bem melhor para sugerir o número de casos de teste necessários em um teste de unidade.

Apesar de oito combinações, já sabemos que o total de caminhos possível (NPATH) é diferente da complexidade ciclomática. Ao aplicar o cálculo por palavra-chave sobre o algoritmo chegamos a um valor de complexidade ciclomática igual a 4 (um ponto pelo método e mais um para cada um dos três “Ifs”). Este valor pode ser visualizado graficamente na Figura 6.

*Figura 6: Quatro caminhos linearmente independentes do algoritmo da sorveteria*

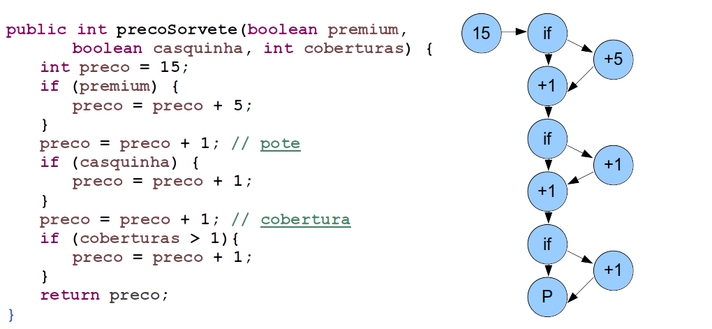
O caminho A destaca em vermelho todas as instruções por onde o algoritmo passou. O caminho B destaca em verde a instrução de preço premium por onde o caminho A não havia passado e de cinza onde passou de novo. No caminho C foi destacada em amarelo a instrução onde a execução passa pela instrução do preço da casquinha e o caminho D mostra a passagem (em azul) pela instrução do preço mais caro da cobertura dupla. Dessa forma, todas os nós foram percorridos ao menos uma vez e verificamos graficamente porque a complexidade ciclomática é 4.

Observe agora, na Figura 7, que apenas dois casos de teste são suficientes para cobrir 100% das linhas de código. Isso se deve a particularidade de como o requisito foi implementado. O algoritmo em questão tem uma espécia de caminho principal e um caminho alternativo. Dessa forma apenas dois cenários acabaram passando por todas as suas linhas. Mais a frente o algoritmo será alterado sem alterar seu resultado e veremos que a quantidade de casos de teste necessário vai mudar.

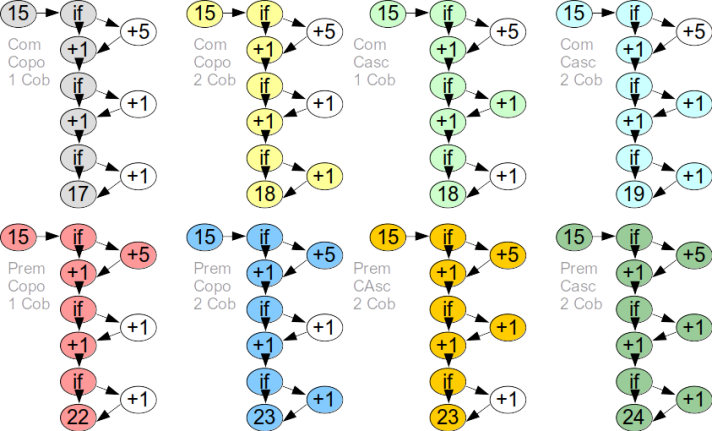
*Figura 7: Dois cenários de teste cobrindo 100% das linhas de código*

**Exemplo da Sorveteria – Implementação 2**

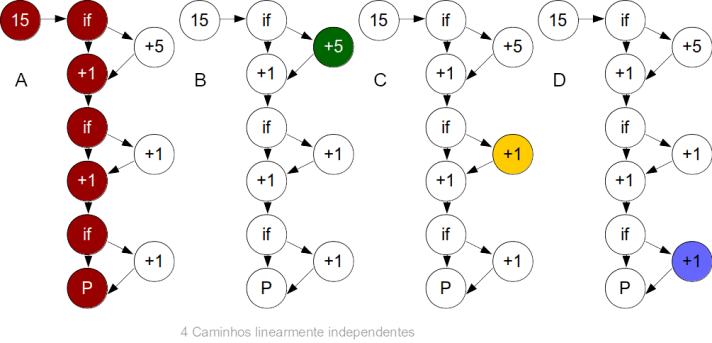
A implementação do requisito de cálculo de preço do sorvete com If/Else gerou a necessidade de dois casos de teste. A Figura 8 mostra uma implementação ainda mais simples onde o caminho principal fica mais explícito. Repare que o grafo também se altera.

*Figura 8: Implementação alternativa com caminho principal e sem “elses”  
(implementação 2)*

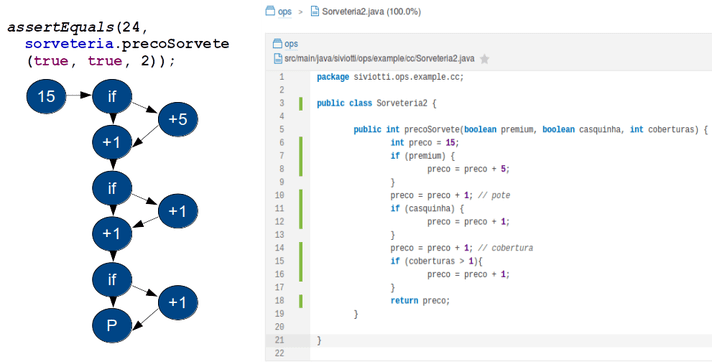
Na Figura 9 é possível ver que nesta implementação continuam existindo as oito possibilidades de sorvete e na Figura 10 que a complexidade ciclomática continua 4 (um do método mais um para cada “IF” – os “elses” não interferem).

*Figura 9: Oito caminhos possíveis para implementação 2*

Veja, graficamente, a complexidade ciclomática 4:

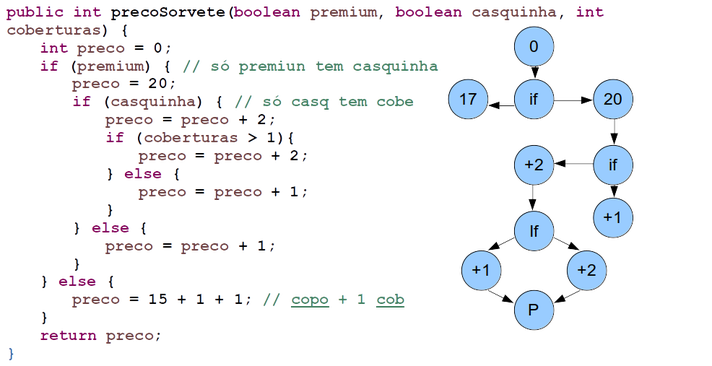
*Figura 10: Quatro caminhos linearmente independentes da implementação 2*

Esta nova implementação gerou uma situação onde a quantidade de cenários de teste para cobertura de todas as linhas baixou para um. Com um único cenário de teste (o sorvete mais caro) todas as linhas são percorridas. Isso não significa que esta implementação só precisa de um cenário em seu teste de unidade, apenas que com apenas um (o certo) todas as linhas serão cobertas.

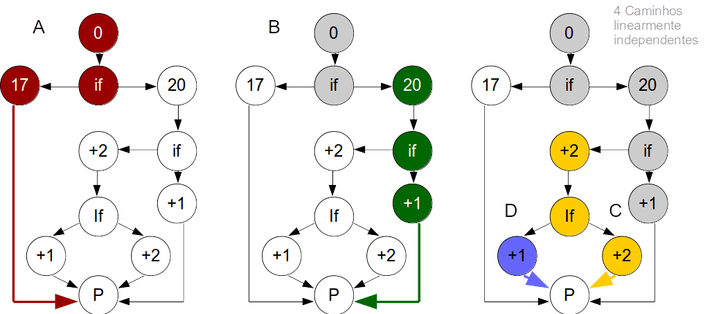
*Figura 11: Cenário de teste para testar a implementação 2  
(apenas um cenário cobre 100%)*

**Exemplo da Sorveteria – Requisito Novo e Implementação 3**

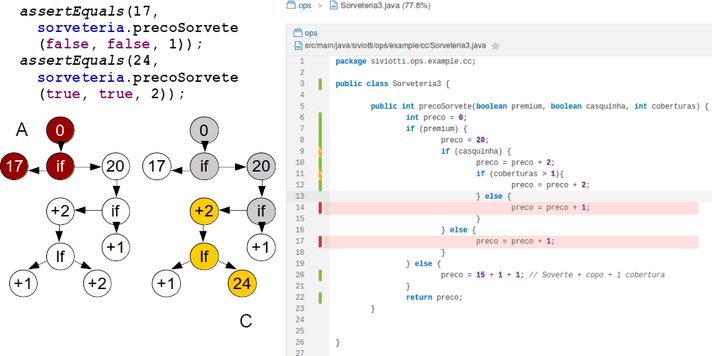
Em uma terceira implementação onde o requisito mudou um pouco os “IFs” estarão aninhados, tornando o código um pouco mais difícil de ler. Neste cenário somente os sorvetes premium têm casquinha e somente as casquinhas têm mais de uma cobertura. O comportamento e os caminhos possíveis (agora 4) são diferentes das duas anteriores, porém a complexidade ciclomática ainda é 4. Outra mudança nesta implementação é a necessidade de casos de teste para que todas as linhas sejam percorridas. Na Figura 12 pode ser vista esta nova implementação e seu grafo correspondente.

*Figura 12: Implementação 3 com IFs aninhados*

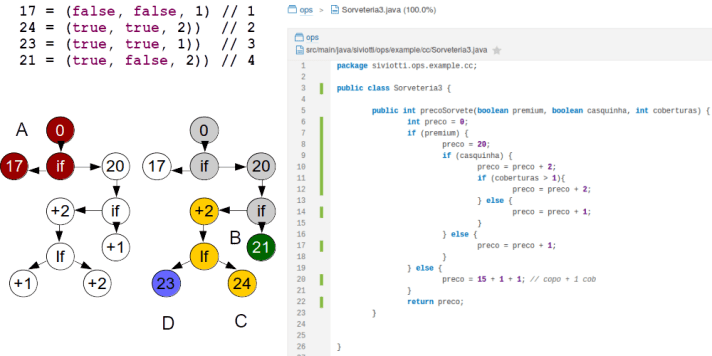
A Figura 13 mostra a complexidade ciclomática sobreposta com os quatro caminhos possíveis (A: vermelho, B: verde, C: amarelo e D: azul).

*Figura 13: Caminhos e complexidade ciclomática da implementação 3*

Agora a cobertura de testes requer um pouco mais de cenários. Na Figura 14 pode-se ver que os mesmos dois cenários (mais barato 17 e mais caro 24) que cobriam 100% das linhas na implementação 1 agora cobrem somente 77,6% das linhas.

  
*Figura 14: Dois cenários de teste cobrindo 77,6% das linhas da implementação 3*

São necessários outros dois cenários de teste para que 100% das linhas consigam ser cobertas conforme pode ser vista na Figura 15.

*Figura 15: Quatro cenários de teste cobrindo 100% das linhas da implementação 3*

**Conclusão**

Conforme apresentado, a complexidade ciclomática, cujo objetivo inicial era medir a complexidade de programas, apresenta forte relação com a testabilidade. Como foi visto nas três implementações diferentes do problema da sorveteria, não é factível elaborar um caso de testes para cada possibilidade de execução de um algoritmo, pois mesmo implementações simples gerarão testes de unidade muito mais complexos que elas mesmas e acabarão sendo abandonados com o passar do tempo. Por outro lado, o valor de complexidade ciclomática parece ser uma meta bem mais razoável para elaboração de cenários de teste para um determinado código fonte. Pode-se dizer que a complexidade ciclomática é o **limite superior do número de cenários de teste necessários para cobrir 100% das linhas de código**. Assim, como visto nos exemplos, eventualmente pode ser necessário um pouco mais ou um pouco menos, dependendo da implementação, mas a quantidade de caminhos linearmente independentes é um bom palpite para a quantidade e para os pontos de passagem dos cenários de teste. Do ponto de vista prático e da qualidade de software, o valor de complexidade ciclomática pode ser usado como medida de qualidade de um código, assim como outras medidas, indicando sua complexidade, mas especialmente sua dificuldade de implementar testes de unidade.