Complexidade Cognitiva

Uma Medida da Dificuldade sobre o Entendimento de um Código Fonte

Douglas Siviotti

V-1.0 - Janeiro/2019

Índice

Versão 1.0 de 20 de Janeiro de 2019

Complexidade Cognitiva

Complexidade cognitiva é uma medida de quão difícil é entender uma unidade de código. Assim como a complexidade ciclomática, ela está associada à complexidade de um software e consequentemente a sua dificuldade de manutenção. Contudo, enquanto a complexidade ciclomática indica a dificuldade ou esforço para se fazer testes de unidade, a complexidade cognitiva indica a dificuldade de ler e entender um determinado código. Estas duas medidas, criadas com 40 anos de diferença, têm uma forte relação. O próprio *white paper* da SonarSource [1] sobre o assunto aborda os problemas encontrados no uso de complexidade ciclomática como ponto de partida, especialmente em linguagens mais novas. Assim, este artigo seguirá a mesma linha, baseando-se neste *white paper*, e apresentará a complexidade cognitiva em função da complexidade ciclomática que é uma medida estabelecida e consagrada como padrão de fato para este tipo de avaliação. Se você não conhece complexidade ciclomática é fortemente recomendado que você leia [este] artigo antes para acompanhar o que será apresentado aqui. Adicionalmente, recomendamos que você leia o próprio *white paper* original [1] disponibilizado pela SonarSource.

Críticas sobre a Complexidade Ciclomática

Crítica 1: Diferença entre o valor medido e a real complexidade do código

A complexidade ciclomática foi criada inicialmente para ser uma medida da dificuldade de manutenção de um código. Entretanto, apesar de indicar o limite máximo de casos de teste para cobertura de todo o código, métodos com o mesmo valor de complexidade ciclomática apresentam diferença significativa na dificuldade de leitura e entendimento. Ou seja, apesar de ser razoavelmente precisa para indicar dificuldade de testabilidade através da indicação do número de cenários de teste, a complexidade ciclomática não apresenta esta mesma regularidade quando o assunto é o entendimento deste código por uma pessoa. O exemplo abaixo (extraído do white paper da SonarSource [1]) exemplifica esta diferença:

```
int sumOfPrimes(int max) {                  // +1
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) {     // +1
    for (int j = 2; j < i; ++j) {           // +1
      if (i % j == 0) {                     // +1
        continue OUT;
      }
    }
    total += i;
  }
  return total;
}                  // Cyclomatic Complexity 4
```

```
String getWords(int number) { // +1
    switch (number) {
      case 1:                 // +1
        return "one";
      case 2:                 // +1
        return "a couple";
      case 3:                 // +1
        return “a few”;
      default:
        return "lots";
    }
}      // Cyclomatic Complexity 4
```

Figura 1: Exemplo de dois métodos com mesma complexidade ciclomática (fonte: SonarSource)

Observe que apesar de terem complexidade ciclomática 4, os métodos apresentam dificuldade de leitura e entendimento bem diferentes. O método á esquerda é claramente mais complexo (somatório de números primos) do que o da direita que á basicamente um ”switch” sobre uma variável.

Crítica 2: Não levar em consideração aspectos das linguagens mais novas

Além da diferença entre métodos com valor similar, paira sobre a complexidade ciclomática o peso de já ter mais de 40 anos e ter sido pensada inicialmente para o ambiente Fortran. De lá para cá bastante coisa mudou inclusive as próprias linguagens de programação. Dessa forma, não contempla estruturas modernas das linguagens mais novas como os lambdas, por exemplo.

Crítica 3: Valor mínimo de 1 para cada método, mesmo quando são simples

Para completar, a complexidade ciclomática atribui o valor 1 para qualquer método e dessa forma o número total de uma classe ou mesmo de um sistema cresce indefinidamente junto com seu tamanho e não relacionado à inclusão de complexidade de fato. Devido a esta característica, a complexidade ciclomática só pode ser usada adequadamente sobre métodos ou funções e não sobre classes e módulos. Um bom exemplo desta discrepância é uma classe simples em Java que tem somente atributos e seus métodos de acesso (get/set). Neste caso, uma classe com 10 atributos terá complexidade ciclomática 20 apesar de, de fato, não apresentar nenhuma complexidade. Pior que isso, se esta classe ganhar mais um atributo sem nenhuma complexidade, o valor da medida pulará para 22 por causa do “get” e do “set” que serão criados. Este exemplo ilustra bem que uma boa medida de complexidade deve iniciar em zero para evitar estas distorções.

Uma Medida Alternativa

A complexidade cognitiva apresenta-se como uma alternativa à complexidade ciclomática a partir dos elementos apresentados nas três críticas vistas anteriormente. Ao contrário de ser aplicada somente para métodos, ela pode ser calculada como medida de uma classe, módulo ou aplicação por causa de sua abordagem de iniciar em zero para um método simples. Em vez de partir de modelos matemáticos (grafos de fluxo de controle) ela baseia-se na percepção dos programadores sobre o esforço necessário para entender um código, indicando assim uma medida de complexidade. Por fim, ela leva em consideração recursos de linguagens mais recentes (como lambdas, por exemplo) na sua forma de cálculo.

Filosofia

Mais do que atacar os três problemas citados anteriormente, a complexidade cognitiva funciona de uma forma diferente de outras medidas de software em relação à motivação, capacidade e percepção de melhoria por parte do desenvolvedor para fazer seu código adequar-se ao que se propõe como qualidade (meta). Diferentemente de medidas como "cobertura linhas de código" ou "duplicação de código" que, ao serem perseguidas sem maior preparo ou conhecimento, acabarão gerando mais problemas do que benefícios, a busca por melhorar a complexidade cognitiva quase sempre gera bons resultados percebidos de imediato. Mesmo desenvolvedores inexperientes ou alguém simplesmente atuando para bater uma meta pode beneficiar-se ao tentar melhorar os números de complexidade cognitiva. Isso não significa, de forma alguma, que medidas de software devam ser tratadas dessa forma. Pelo contrário, conforme proposto pela própria SonarSource [2] (Fix the Leak [conserte o vazamento]), medidas e ferramentas de aferição devem ser usadas continuamente como forma de evitar que problemas sejam criados e não para encontrá-los e sair resolvendo afobadamente. No entanto, sabemos que muitas vezes essa é a forma como esse tipo de ferramenta acaba sendo introduzida em uma equipe ou organização.

A boa relação benefício/custo do investimento em medição e melhoria de complexidade cognitiva está relacionada aos princípios que norteiam sua forma de cálculo. Entre estes princípios podemos destacar:

- Cálculo baseado em **percepção subjetiva sobre a dificuldade de entendimento** focando em um problema do programador e não em um modelo matemático de caminhos percorridos. Isto significa que o cálculo leva em consideração o programador lendo o código e não o código em execução e suas possibilidades. A melhora do indicador é percebida de forma rápida e positiva pois ocorre diretamente no ponto de atuação.

- **Incentivo a boas práticas de codificação**. Ao introduzir boas práticas (que antes não existiam) em um sistema, naturalmente os números irão melhorar. A tendência é que bons desenvolvedores gerando código com boas práticas provavelmente criarão um código com complexidade cognitiva baixa. O uso da medida como indicador ajuda na motivação de se fazer um código melhor em contraponto às várias motivações para entregá-lo mais rápido.

- **A quebra de um fluxo linear aumenta o esforço de entendimento**. Os incrementos no valor acontecem por desvios de cima para baixo e da esquerda para a direita. Pontos de **decisões** e **aninhamentos** dificultam o entendimento e consequentemente a manutenção. Assim, este tipo de desvio é onerado no cálculo final, criando a motivação certa para que sejam minimizados e/ou otimizados.

Metodologia

A pontuação de complexidade cognitiva é avaliada de acordo com três regras básicas:

Regra 1: Ignorar Abreviações

Ignorar estruturas em que várias instruções podem ser abreviadas para apenas uma.

Regra 2: Incrementar por Quebra de Fluxo

Incrementar um ponto a cada quebra no fluxo linear do código.

O incremento de 1 ponto ocorre para cada um destes elementos encontrados: `if`, `eles if`, `else`, operador ternário, `switch`, `for`, `foreach`, `while`, `do while`, `catch`, `goto LABEL`, `break LABEL`, `continue LABEL`, sequência de operadores lógicos (`&&`, `||` etc) e cada recurção encontrada.

Regra 3: Incrementar por Aninhamento

Incrementar um ponto a cada nível de aninhamento de uma quebra de fluxo encontrada além do ponto já incrementado pela própria quebra.

O incremento do nível de aninhamento ocorre para cada um destes elementos encontrados: `if`, `eles if`, `eles`, operador ternário, `switch`, `foreach`, `while`, `do while catch`, métodos aninhados e métodos ou estruturas tipo lambda.

O incremento de 1 ponto para cada nível de aninhamento ocorre para um destes elementos encontrados dentro de outros: `if`, operador ternário, `switch`, `for`, `foreach`, `while`, `do while` e `catch`.

Os incrementos são classificados em quatro tipos:

A. Aninhamento – avaliado em estruturas de controle de fluxo dentro de outras

B. Estrutural – avaliado em estruturas de controle de fluxo que estão sujeitas a incremento por aninhamento

C. Fundamental – avaliado em declarações não sujeitas a incremento por aninhamento

D. Híbrido – avaliado em estruturas de controle de fluxo que não estão sujeitas a incremento por aninhamento, mas que aumentam a contagem de aninhamento

Ignorando Abreviações

Um dos princípios norteadores da formulação da medida de complexidade cognitiva é o incentivo a boas práticas de codificação. Dessa forma, deve-se ignorar (não penalizar) recursos da linguagem que fazem o código mais legível.

Um bom exemplo disso é a divisão de um método em outros métodos. No caso da complexidade ciclomática, ao quebrar um método em vários ocorre um aumento de um ponto para cada novo método criado, enquanto na complexidade cognitiva o valor não aumenta somente pela existência de novos métodos. A decomposição de métodos grandes é uma boa prática e deve ser incentivada.

Outro exemplo de incentivo a boas práticas é ignorar os operadores de checagens implícitas de nulidade. Ou seja, aquelas situações onde o “if x! =null” é trocado por um operador que faz o mesmo papel dentro da própria instrução. O “IF” causará incremento da pontuação, mas o uso do operador não. Assim, como mostra a Figura 2, o desenvolvedor pode reduzir a complexidade cognitiva trocando a primeira opção (esquerda) pela segunda (direita).

```
MyObj myObj = null;
if (a != null) {
  myObj = a.myObj;
}
```

```
MyObj myObj = a?.myObj;
```

Figura 2: Exemplo de equivalência entre um "if" e um operador de teste de nulidade (fonte: SonarSource)

Em muitos casos, alterar o código para baixar a complexidade ciclomática pode significar fazer alterações sem sentido. Por outro lado, ações deste tipo para redução de complexidade cognitiva costumam incentivar uma versão realmente melhor do código, especialmente do ponto de vista da dificuldade de leitura e entendimento como o exemplo da Figura 2.

Incrementos por Quebra de Fluxo

Outro princípio norteador da formulação da complexidade cognitiva é que a quebra do fluxo linear de cima para baixo ou da esquerda para a direita aumenta o esforço necessário de quem precisa ler e manter o software. Reconhecendo isto como um esforço extra, este tipo de quebra gera um incremento estrutural de um ponto para:

- Estruturas de loop: `for`, `while`, `do while` etc
- Condicionais: `if`, `#if`, `#ifdef` etc

Também é gerado um incremento híbrido para:
- `eles if`, `elif`, `else` etc

Este tipo de estrutura, contudo, não gera incremento de aninhamento (apesar de gerar uma indentação no código) uma vez que já é contabilizado o custo cognitivo de entender a quebra em si.

Catches

Um “catch” representa um desvio no fluxo de execução da mesma forma que um “if”. Por este motivo, cada “catch” causa um incremento estrutural no valor da complexidade cognitiva de um ponto independente da quantidade de exceções tratadas.

Switches

Diferentemente da complexidade ciclomática onde cada “case” representa um caminho dentro do algoritmo e, consequentemente o incremento de um ponto no valor final, na complexidade cognitiva um “switch” e todos os seus “cases” geram um incremento estrutural de apenas um ponto. Isto porque a leitura e entendimento de um “switch” não são proporcionais ao tamanho de suas possibilidades/caminhos e sim ao fato de haver uma múltipla escolha onde somente uma será escolhida.

Apesar de um “switch” com “cases” fazer o mesmo papel de um conjunto de “if-else” do ponto de vista do fluxo de execução, é claramente mais simples entender um switch que toma decisão em um único ponto sobre uma única variável. Mesmo com muitos “cases” o que de fato precisa ser compreendido é a variável sendo analisada.

Sequência de Operadores Lógicos

Pela lógica aplicada aos “switches”, uma sequência de operadores de um mesmo tipo é tratada como se fosse apenas um operador. Ou seja, “a && b” gera o mesmo incremento fundamental que “a && b && c && d”. O mesmo ocorre se fosse “a || b” comparado a “a || b || c || d”. Uma sequência contínua de “E” é lida como “todos estes” e uma sequência contínua de “OU” é lida como “algum destes” independente de quantos operadores e possibilidades existam. Ou seja, ao encontrar uma sequência de operadores lógicos iguais, o

desenvolvedor forma somente uma frase em sua cabeça para entender o que o código faz e, sendo assim, esta frase conta somente um ponto.

Por outro lado, quando em uma expressão com operadores booleanos os operadores são alternados, a leitura torna-se bem mais difícil gerando uma combinação mental maior de frases como “todos” e “algum”. Por isso, a quebra de sequência de um operador por outro operador gera um incremento de mais um ponto à contagem. Alguns exemplos de sequência contínua (contando somente um) e sequência alternada (contando um a cada quebra) podem ser vistos na Figura 3 retirada do *white paper* da SonarSource[1].

```
if (a                                   // +1 for `if`
    && b && c                           // +1
    || d || e                           // +1
    && f)                               // +1

if (a                                   // +1 for `if`
    &&                                  // +1
    !(b && c))                          // +1
```

Figura 3: Exemplo de sequência quebrada por outro operador (fonte: SonarSource)

Se por um lado a complexidade cognitiva pode dar um “desconto” em operadores sequenciais contínuos dentro de condições (IF) que na complexidade ciclomática gerariam um ponto cada um, por outro lado, em situações onde a complexidade ciclomática não contaria nenhum ponto (como em atribuições de variáveis e retornos de método), a complexidade cognitiva gerará incremento se encontrar sequências alternadas de operadores. Ou seja, para a complexidade cognitiva qualquer sequência alternada de operadores será avaliada do ponto de vista do cálculo.

Recursão

Ao contrário de complexidade ciclomática, a complexidade cognitiva adiciona um incremento fundamental para cada método que utilize recursão, de forma direta ou indireta. Existem duas razões para a definição deste incremento. A primeira é que uma recursão é uma espécie de “loop” e os “loops” sempre geram um incremento na complexidade cognitiva. A segunda é a abordagem baseada no entendimento. A presença de recursão normalmente dificulta o entendimento de como um método funciona, incluindo dificuldades adicionais de debug quando a recursão ocorre muitas vezes.

Desvios para Rótulos (goto e similares)

Comandos do tipo “goto” ou “break”/”continue” que desviam para um ponto específico do método (rótulos) geram um incremento fundamental de um ponto na complexidade cognitiva. Este tipo de desvio leva a execução para uma parte do método anotada com um rótulo para definir onde é o ponto de entrada após o desvio. Tal tipo de desvio requer um esforço de entendimento porque o ponto de reentrada pode fazer diferença no algoritmo. Por outro lado, um “return” antecipado (cláusula de guarda[3]) ou qualquer outra saída inicial de um método não geram um novo incremento por serem consideradas boas práticas para a leitura e entendimento do código. Uma cláusula de guarda, por exemplo, evita que todo o corpo do método fique dentro de um “if” caso uma condição seja atendida. Com a cláusula de guarda ocorre o contrário, se uma condição não é atendida ocorre uma saída antecipada do método. Dessa forma, o resto do método passa a estar aninhado à raiz.

Incremento por Quebra de Fluxo Aninhada

Imagine cinco “Ifs” em sequencia, um após o outro. O desenvolvedor que precisar ler este código vai fazer cinco vezes o esforço de entender o desvio gerado por cada “if”, mas não vai precisar entender a relação entre eles uma vez que ao final do “if” o fluxo volta para o nível raiz do método.

```
if (condition1){  // +1
  statement1;
}
if (condition2){  // +1
  statement2;
}
if (condition3){  // +1
  statement3;
}
if (condition4){  // +1
  statement4;
}
if (condition5){  // +1
  statement5;
}
                  // complexidade cognitiva = 5
```

Agora imagine os mesmos cinco “Ifs” um dentro do outro. Intuitivamente já sabemos que não será tão fácil quanto quando estavam separados, mas podemos tentar tornar claro o motivo. Ao ler o segundo “if” vai haver um certo empilhamento de consequências em relação ao primeiro, ou seja, não é possível esquecer que já se está “dentro” de um desvio quando da análise do que ocorre em um segundo desvio. Para situações de sobrecarga de entendimento e atenção como esta, o cálculo de complexidade cognitiva incrementa em um ponto para cada nível de aninhamento de uma certa quebra de fluxo (que já teria um ponto pela quebra). Assim, nosso exemplo de 5 “ifs” aninhados teria uma complexidade de 1 (só quebra) + 2 (quebra + 1 nível de aninhamento) + 3 (quebra + 2 níveis) + 4 (quebra + 3 níveis) + 5 (quebra + 4 níveis) resultando em 15 no total deste método como mostra o código de exemplo a seguir.

```
if (condition1){           // +1
  statement1;
  if (condition2){         // +1 (adiciona 1 por aninhamento) = 2
    statement2;
    if (condition3){       // +1 (adiciona 2 por aninhamento) = 3
      statement3;
      if (condition4){     // +1 (adiciona 3 por aninhamento) = 4
        statement4;
        if (condition5){   // +1 (adiciona 4 por aninhamento) = 5
          statement5;
        }
      }
    }
  }
}
                           // complexidade cognitiva = 15
```

Apesar do exemplo anterior mostrar “Ifs” aninhados, este ponto adicional para cada nível de aninhamento vale para qualquer combinação de estrutura de quebra dentro de outra como mostra a Figura 4 presente no *white paper* [1] original.

```
void myMethod () {
  try {
    if (condition1) {                    // +1
      for (int i = 0; i < 10; i++) {     // +2 (nesting=1)
        while (condition2) { … }         // +3 (nesting=2)
      }
    }
  } catch (ExcepType1 | ExcepType2 e) { // +1
    if (condition2) { … }               // +2 (nesting=1)
  }
}                                       // Cognitive Complexity 9
```

Figura 4: Exemplo de adição de ponto por aninhamento (fonte: SonarSource)

Repare que o “if” (linha 3) tem um incremento normal de quebra de fluxo enquanto que o “for” tem um incremento por quebra e um por aninhamento. O “while” tem um incremento por quebra e dois por aninhamento (está dentro do “for” que está dentro do “if”). No segundo “if” também há um incremento por aninhamento já que está dentro de um “catch” que, por sua vez, gera um incremento por quebra.

```
  void myMethod2 () {
    Runnable r = () -> {          // +0 (but nesting level is now 1)
      if (condition1) { … }       // +2 (nesting=1)
    };
  }                               // Cognitive Complexity 2


  #if DEBUG                       // +1 for if
  void myMethod2 () {             // +0 (nesting level is still 0)
    Runnable r = () -> {          // +0 (but nesting level is now 1)
      if (condition1) { … }       // +3 (nesting=2)
    };
  }                               // Cognitive Complexity 4
  #endif
```

Figura 5: Exemplo de aumento do nível de aninhamento dentro de um lambda (fonte: SonarSource)

Adicionalmente, enquanto métodos de nível mais alto são ignorados na contagem e não há nenhum incremento para lambdas, métodos aninhados e funcionalidades similares incrementam o nível de aninhamento para possíveis incrementos de alguma estrutura de quebra que esteja dentro destes métodos. Em outras palavras, coisas como lambdas não incrementam a contagem, mas aumentam a contagem de coisas que estarão dentro delas. Veja na Figura 5 o exemplo presente no *white paper* [1] original da SonarSource.

Repare que o lambda criado para “Runnable” não conta, mas ele incrementa o nível de aninhamento. Dessa forma, o “if” dentro dele conta 2 como se estivesse dentro de outra estrutura de quebra (outro “if”, por exemplo). Em seguida o mesmo exemplo foi “circundado” por um “#if” e as pontuações de aninhamento foram incrementadas em 1 além da contagem do ponto de quebra do próprio “#if”.

As Implicações

A complexidade cognitiva foi formulada com o objetivo inicial de calcular um **score mais acurado** possível sobre dificuldade de entendimento de um método, e como segundo objetivo, **contemplar estruturas mais modernas**, produzindo medidas significativas sobre os **níveis acima do de método** (classe, módulo, aplicação etc). Como foi demonstrado, as questões de estruturas de linguagens mais modernas, como lambdas, estão devidamente contempladas. Os outros dois objetivos serão examinados a seguir. [1]

Score de Complexidade Intuitivamente “Correto”

Na Figura 1 foi apresentado um exemplo de dois métodos com o mesmo valor de complexidade ciclomática, porém níveis claramente diferentes de dificuldade de entendimento. Esta era, inclusive, uma das críticas à complexidade ciclomática que pontua com o mesmo valor métodos com grande diferença na complexidade do ponto de vista do entendimento. Observe que, neste mesmo exemplo com a pontuação calculada de complexidade cognitiva, a diferença de entendimento é refletida na pontuação.

```
int sumOfPrimes(int max) {
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) { // +1
    for (int j = 2; j < i; ++j) {       // +2
      if (i % j == 0) {                 // +3
        continue OUT;                   // +1
      }
    }
    total += i;
  }
  return total;
}                 // Cognitive Complexity 7
```

```
String getWords(int number) {
    switch (number) {         // +1
      case 1:
        return "one";
      case 2:
        return "a couple";
      case 3:
        return “a few”;
      default:
        return "lots";
    }
  }      // Cognitive Complexity 1
```

Figura 6: Exemplo de métodos com c. ciclomática igual, mas c. cognitiva diferente (fonte: SonarSource)

Talvez o método a esquerda não seja realmente sete vezes mais complexo ou mais difícil de entender, mas o resultado para complexidade cognitiva é bem mais fiel à realidade que o resultado obtido com a complexidade ciclomática.

Medidas Significativas para os Níveis Acima dos de Método

Em função da complexidade cognitiva não gerar incremento para a simples estrutura do método (como na complexidade ciclomática) os números totais de uma classe, módulo ou aplicação são realmente significativos em relação à complexidade e a dificuldade de entendimento destes elementos. Agora fica clara a diferença entre uma classe contendo basicamente estrutura de dados com seus “*getters*” e “*setters*” de uma classe onde estão implementadas regras de negócio. Desas forma, a complexidade cognitiva credencia-se como ferramenta para medir a compreensibilidade relativa de classes e módulos maiores.

Exemplo da Sorveteria

No artigo sobre [Complexidade Ciclomática] foi apresentado um exemplo de algoritmo para cálculo de preço de uma sorveteria. Vamos usá-lo novamente para fazer o mesmo tipo de comparação feita na Figura 6 em relação à Figura 1. Os três exemplos mostram o resultado calculado de complexidade ciclomática bem como o de complexidade cognitiva. Há indicação de onde houve incremento e quantos foram incrementados.

O primeiro algoritmo usava 3 conjuntos de “if/else”(Figura 7).

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

Ciclomática **4**
Cognitiva **6**

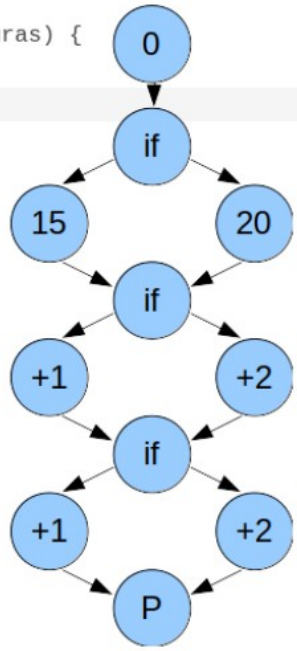

Figura 7: Código e grafo do algoritmo da Sorveteria 1

O segundo algoritmo usava apenas 3 “Ifs” (Figura 8).

```
public int precoSorvete(boolean premium, boolean casquinha, int coberturas) {
        int preco = 15;
   +1   if (premium) {
                preco = preco + 5;
        }
        preco = preco + 1; // pote
   +1   if (casquinha) {
                preco = preco + 1;
        }
        preco = preco + 1; // cobertura
   +1   if (coberturas > 1){
                preco = preco + 1;
        }
        return preco;
}
```

Ciclomática **4**
Cognitiva **3**

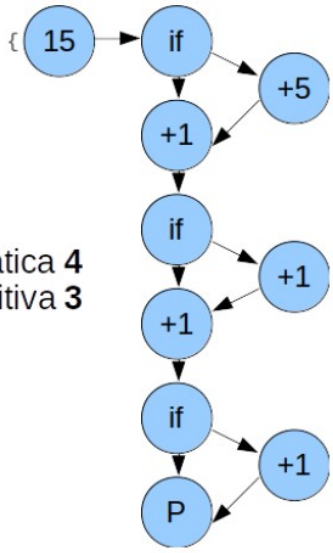

Figura 8: Código e grafo do algoritmo da Sorveteria 2

O terceiro algoritmo fazia uma pequena alteração no requisito e usava “Ifs/elses” aninhados (Figura 9).

```
public int precoSorvete(boolean premium, boolean casquinha, int coberturas) {
        int preco = 0;
+1      if (premium) {
            preco = 20;
+2          if (casquinha) {
                preco = preco + 2;
+3              if (coberturas > 1){
                    preco = preco + 2;
+1              } else {
                    preco = preco + 1;
                }
+1          } else {
                preco = preco + 1;
            }
+1      } else {
            preco = 15 + 1 + 1; // copo + 1 cob
        }
        return preco;
}
```

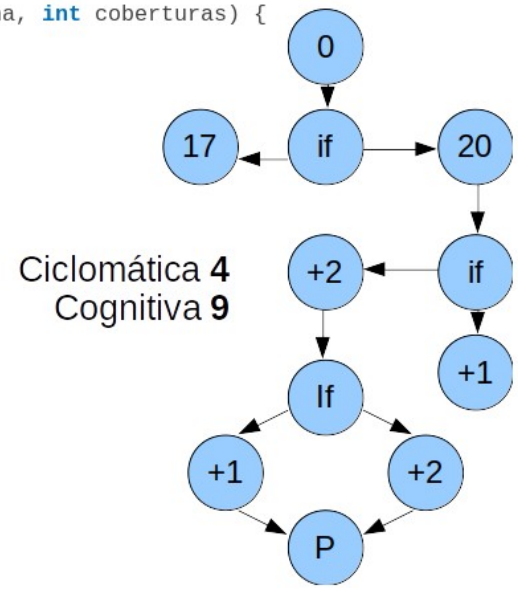

Figura 9: Código e grafo do algoritmo da Sorveteria 3

A primeira coisa que salta os olhos nestes exemplos é que a complexidade ciclomática é a mesma (4) para os três algoritmos com complexidades diferentes. Enquanto isso, a complexidade cognitiva capta essa diferença com três valores proporcionalmente diferentes (6, 3 e 9). No artigo original deste exemplo já havia sido apresentado o número de casos de teste necessários para o teste de unidade cobrir 100% das linhas. O primeiro precisava de 2, o segundo de apenas um e o terceiro de quatro cenários. Isto também já sugeria uma diferença de complexidade entre os algoritmos que não era representado pelo valor da complexidade ciclomática calculada. Na ocasião, o artigo conclui que apesar de ter sido pensada para medir complexidade, a complexidade ciclomática exerce melhor o papel de indicar o "limite superior do número de cenários de teste para cobrir 100% do código".

A complexidade cognitiva, por outro lado, indica de forma mais precisa a complexidade do ponto de vista da leitura e entendimento dos três exemplos. Os dois primeiros, inclusive, tem comportamento idêntico apesar de diferentes. O segundo pode ser considerado uma versão mais simples do primeiro, estabelecendo uma espécie de caminho principal quando nenhum "if" dá verdadeiro. Já o terceiro, apresenta condicionais dependentes e realmente exige um pouco mais para ser compreendido em detalhes. Observe que o segundo "if" ganhou um incremento extra por aninhamento bem como o terceiro "if" ganhou dois pontos extras pelo mesmo motivo.

Conclusão

Do ponto de vista da aferição de complexidade, baseada na dificuldade de leitura e/ou entendimento de um código fonte, **a complexidade cognitiva apresenta-se como uma melhor opção em relação à complexidade ciclomática**. Sua abordagem não matemática (subjetiva) para o problema da aferição de dificuldade de manutenção relaciona, de forma mais acurada, o número aferido com o esforço envolvido em sua melhoria. Ou seja, apesar de gerar números como qualquer outra medida, ela está intrinsecamente ligada a forma como um ser humano entende o código e consequentemente o que deve fazer para que este entendimento possa ser melhorado.

A complexidade cognitiva tem o incentivo a boas práticas como princípio norteador de sua formulação de cálculo. Em função disso, o trabalho de melhoria baseado em sua pontuação costuma causar melhoras mais rapidamente perceptíveis pelos desenvolvedores que, frequentemente, costumam ter uma relação de desconfiança com medidas e indicadores de software. O problema da dificuldade de entendimento e manutenção são mais compreensíveis e palatáveis do que questões de cobertura, coesão ou duplicação, por exemplo.

Adicionalmente, graças à atribuição de zero para métodos sem complexidade adicional, a medida pode ser usada em níveis mais altos que o de método, como o de classe, módulo ou aplicação. Assim, diferentemente da complexidade ciclomática, **a pontuação total de complexidade cognitiva para elementos de alto nível é realmente significativa** e pode ser relacionada a outras medidas destes mesmos elementos.

Referências

[1] G. Ann Campbell (SonarSource S.A.), Cognitive Complexity, https://www.sonarsource.com/docs/CognitiveComplexity.pdf, acessado em Janeiro/2019

[2] SonarSource S.A., Fix the Leak, https://www.sonarsource.com/why-us/unique-approach/water-leak/ , acessado em Janeiro/2019

[3] Kent Beck, Padrões de Implementação, Bookman, 2008