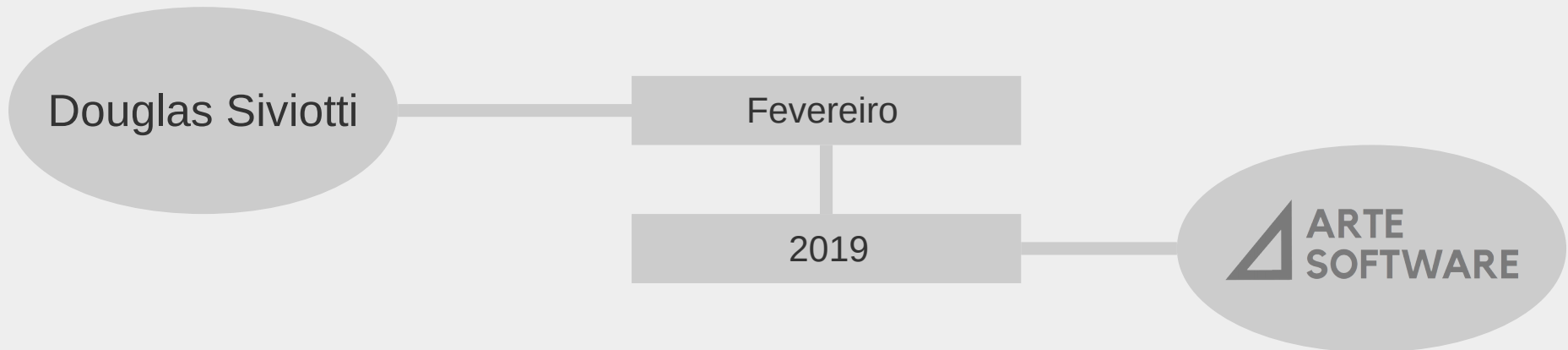
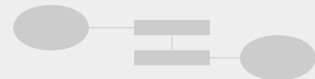


Complexidade Cognitiva

Uma Medida da Dificuldade sobre o Entendimento de um Código Fonte





1. Conteúdo: Complexidade Cognitiva (SonarSource)
2. Área/Foco: Qualidade de Software / Medida de Complexidade
3. Público alvo: Desenvolvedores
4. Conteúdo relacionado: Sonar, Complexidade Ciclomática

Organização: 34 Slides organizados em 4 partes (+- 30 minutos)

1. Definição, 2. Metodologia, 3. Exemplo Prático e 4. Implicações

definição

Complexidade Cognitiva
é uma medida de quão
difícil é entender uma
unidade de código

Complexidade Ciclomática (CC) mede a quantidade de **caminhos linearmente independentes** em um código fonte. Ou seja, é uma medida de quão difícil é **testar** uma determinada unidade de código. CC baseia-se em um modelo matemático de grafos de controle de fluxo.

Complexidade Cognitiva (C-Cog) mede a quantidade de **quebras do fluxo linear** de leitura ponderadas pelo **nível de aninhamento** dessas quebras. Ou seja, é uma medida de quão difícil é **entender** uma determinada unidade de código. C-Cog baseia-se em um modelo de percepção subjetiva sobre a dificuldade de entendimento (não matemático).

1. Diferença entre o valor medido e a real complexidade

- Códigos com complexidade bem diferente dão o mesmo valor de CC
- Totalmente baseada no número de caminhos linearmente independentes

2. Não leva em consideração aspectos de linguagens mais novas

- **Lambdas**, operadores “null safe” entre outros não são levados em conta

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

- Uma classe com 10 atributos tem CC = 20, apenas com “gets” e “sets”
- Logo, só serve para medição de métodos, rotinas ou funções.
- Não é possível usar CC para medir **classes, módulos ou sistemas**

Dois algoritmos com dificuldade de entendimento diferente, mas com complexidade ciclomática igual

```
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
```

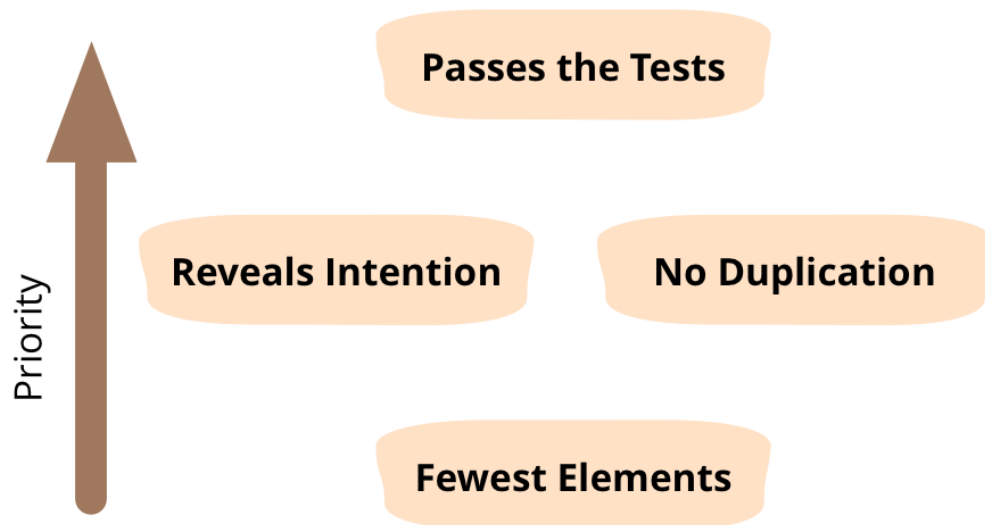
Método para cálculo da soma de todos os números primos até um máximo informado

```
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
```

Método que retorna um texto relativo ao parâmetro passado através de um “switch”

Filosofia

- Foco no **problema do programador** (dificuldade de entendimento, não caminhos)
- **Quebras no fluxo** linear aumentam o esforço de entendimento
- **Aninhamentos** aumentam ainda mais o esforço gerado por uma quebra
- Incentivo a **boas práticas**, beneficiando as boas escolhas (e.g. “.”, clausula de guarda)



- Os testes passam
- **Revela intenção**
- Nenhuma duplicação
- Mínimo de elementos

Sobre a **Complexidade Ciclométrica**:

- Aplicável a métodos, rotinas e funções, mas também a **classes, módulos e sistemas**
- Baseada na **percepção do programador** sobre o esforço de entendimento
- Considera recursos de linguagens mais recentes (e.g. **Lambdas**)

Sobre **Outras Medidas** em geral (cobertura, duplicação etc):

- A melhora ocorre com ou sem total compreensão. “Trate primeiro, entenda depois”
- Melhorias são mais rapidamente percebidas durante o tratamento do “vazamento”

Desvantagem: nativa do SonarQube, não é a medida padrão de complexidade



metodologia

1. Ignorar Abreviações
2. Incrementar por Quebra de Fluxo
3. Incrementar por Aninhamento

Regra 1: Ignorar Abreviações

Ignorar estruturas em que várias instruções podem ser abreviadas para apenas uma (**incentiva boas práticas**)

Regra 2: Incrementar por Quebra de Fluxo

Incrementar um ponto a cada quebra no fluxo linear do código

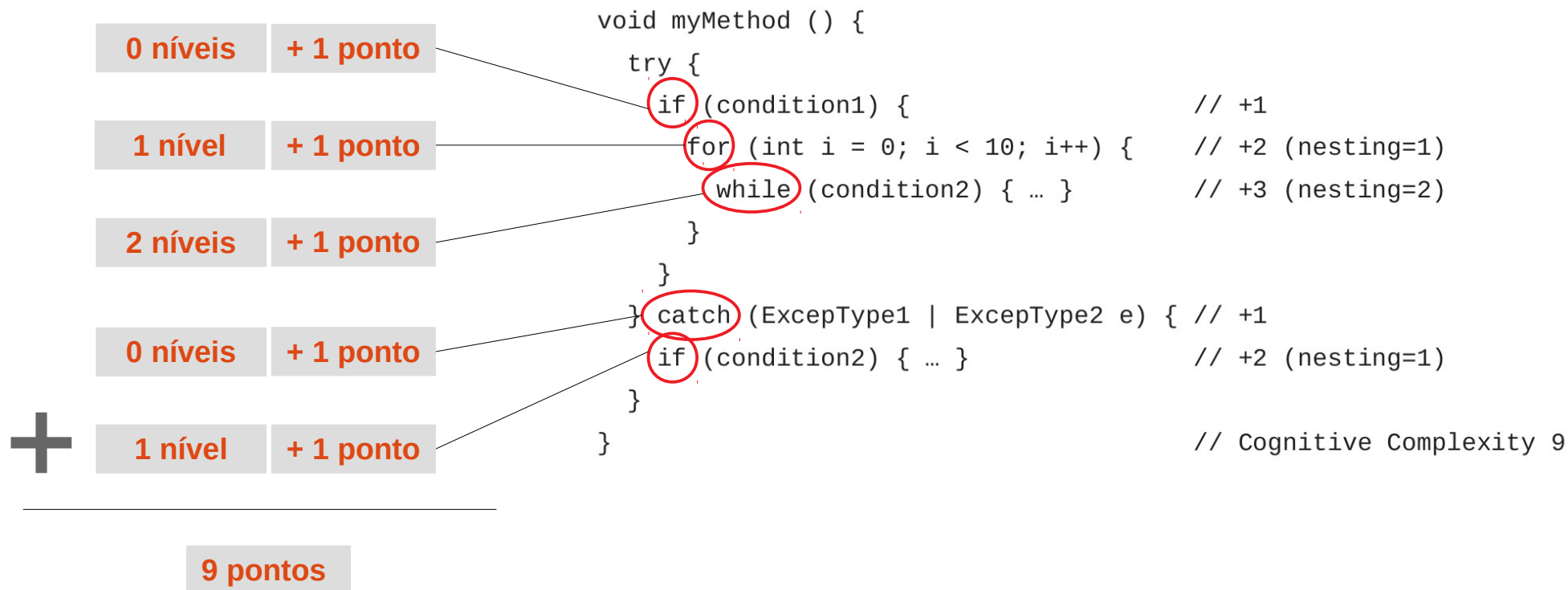
+ 1 ponto para: `if`, `else if`, `else`, operador ternário, `switch`, `for`, `foreach`, `while`, `do while`, `catch`, `goto LABEL`, `break LABEL`, `continue LABEL`, sequência de operadores lógicos (`&&`, `||` etc) e cada recorrência encontrada

Regra 3: Incrementar por Aninhamento

Incrementar um ponto a cada nível de aninhamento de uma quebra de fluxo (além do ponto por quebra - R2)

+ 1 nível para: `if`, `else if`, `else`, operador ternário, `switch`, `foreach`, `while`, `do while` `catch`, métodos aninhados e métodos ou estruturas tipo `lambda`

+ 1 ponto para: `if`, operador ternário, `switch`, `for`, `foreach`, `while`, `do while` e `catch`.



Lambda

1 nível

+ 1 ponto

```
void myMethod2 () {  
    Runnable r = () -> {  
        if (condition1) { ... }  
    };  
}
```

```
// +0 (but nesting level is now 1)  
// +2 (nesting=1)
```

```
// Cognitive Complexity 2
```

0 níveis

+ 1 ponto

```
#if DEBUG
```

```
// +1 for if
```

```
void myMethod2 () {
```

```
// +0 (nesting level is still 0)
```

```
    Runnable r = () -> {
```

```
// +0 (but nesting level is now 1)
```

2 níveis

+ 1 ponto

```
        if (condition1) { ... }  
    };
```

```
// +3 (nesting=2)
```

```
}
```

```
// Cognitive Complexity 4
```

```
#endif
```

Estrutural – avaliado em estruturas de controle de fluxo que estão sujeitas a incremento por aninhamento (1 ponto + aninhamento). Causam incremento do aninhamento

`if, #if, for, while` etc

Fundamental – avaliado em declarações não sujeitas a incremento por aninhamento (sempre conta 1 ponto). Não causam aninhamento.

`&&, ||, goto "label", recursão`

Híbrido – avaliado em estruturas de controle de fluxo que não estão sujeitas a incremento por aninhamento (sempre conta 1 ponto). Causam aninhamento.

`else, else if, elif` etc

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

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

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

Refatoração

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

“Safe Navigation Operator”

Ciclomática	2
Cognitiva	1

Ciclomática	2
Cognitiva	0

O cálculo privilegia boas práticas. O código à direita é mais fácil de entender

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. Cada quebra incrementa **um ponto**

Incremento estrutural para:

Estruturas de loop como `for`, `while`, `do while` etc

Condicionais como `if`, `#if`, `#ifdef` etc

Incremento híbrido para `else`, `elseif`, `elif` etc

Catch gera o incremento estrutural de **um ponto**

- Um “catch” representa um desvio no fluxo da mesma forma que um “if”
- Incrementa um ponto independentemente de quantas exceções são tratadas

```
catch (NullPointerException | IOException e) + 1 ponto
```

Switch gera o incremento estrutural de **um ponto**

- Apenas a variável de decisão precisa ser compreendida
- A quantidade de “cases” não aumenta a dificuldade de entendimento (é lido como “um desses”)
- Na C. Ciclomática cada “case” aumenta em um ponto!

+ 1 ponto

```
switch (number) {  
    case 1: “One”;  
    case 2: “Two”;  
    case 3: “A Few”;  
    default: “lots”;  
}
```

Sequências de **operadores iguais** geram incremento fundamental de **um ponto**

Operadores alternados contam um ponto cada um

“a && b” = “a && b && c && d” **+ 1 ponto**

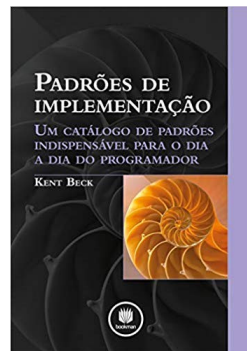
“a || b” = “a || b || c || d” **+ 1 ponto**

Uma sequência de “E” é lida como
“todos esses”

Uma sequência de “OU” é lida como
“algum desses”

Mesma lógica do “Switch”

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



Recursão gera um incremento fundamental de **um ponto**

- Recursão é uma espécie de “loop”
- Recursão dificulta a leitura e o entendimento

Desvios para Labels geram um incremento fundamental de **um ponto**

- Necessidade de entender para onde vai a execução (label)

“Return” antecipado (cláusula de guarda) **não incrementa nada**

- Cláusulas de guarda no início do método são um padrão de implementação [Beck]

```
if (parameter == null) return                // cláusula de guarda
println("Passou da cláusula de guarda")      // evita aumentar o nível de aninhamento
```

A **quebra do fluxo aninhada** dentro de outra (estrutural) é penalizada com **um ponto para cada nível de aninhamento** (além do ponto inicial)

```
if (condition1){ // +1
    statement1;
}
if (condition2){ // +1
    statement2;
}
if (condition3){ // +1
    statement3;
}
if (condition4){ // +1
    statement4;
}
if (condition5){ // +1
    statement5;
}
// complexidade cognitiva    = 5
// complexidade ciclomática = 6
```

```
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
// complexidade ciclomática = 6
```

A **quebra do fluxo aninhada** dentro de outra (estrutural) é penalizada com **um ponto para cada nível de aninhamento** (além do ponto inicial)

```
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
```

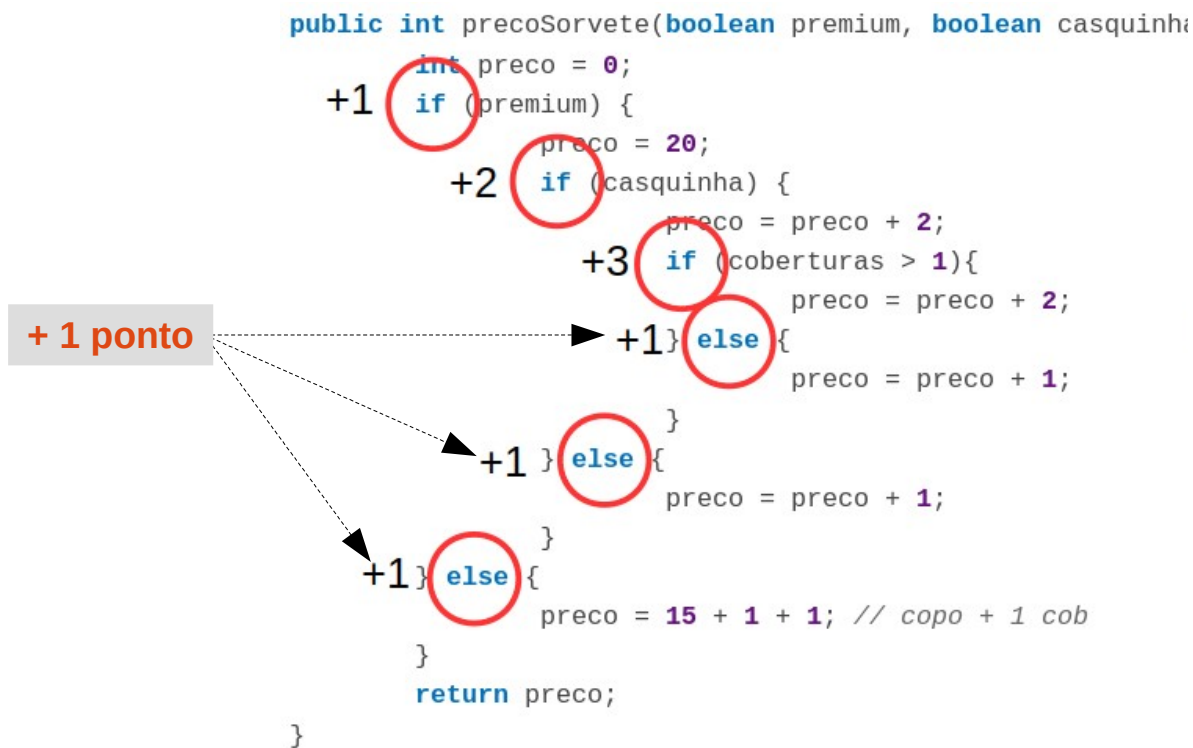
```
void myMethod2 () {
    Runnable r = () -> {
        if (condition1) { ... }
    };
}

// +0 (but nesting level is now 1)
// +2 (nesting=1)
// Cognitive Complexity 2

#ifdef DEBUG
void myMethod2 () {
    Runnable r = () -> {
        if (condition1) { ... }
    };
}

// +1 for if
// +0 (nesting level is still 0)
// +0 (but nesting level is now 1)
// +3 (nesting=2)
// Cognitive Complexity 4
#endif
```


Incrementos dos tipos **híbrido** e **fundamental** contam somente o ponto inicial. **Não geram incremento por aninhamento**



exemplo prático



**Três Algoritmos de
Preço do Sorvete
CC=4 C-Cog=3,6,9**

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.

```
public int precoSorvete(boolean premium,
                        boolean casquinha, int coberturas) {

    int preco = 0;

    if (premium) {
        preco = 20;
    } else {
        preco = 15;
    }

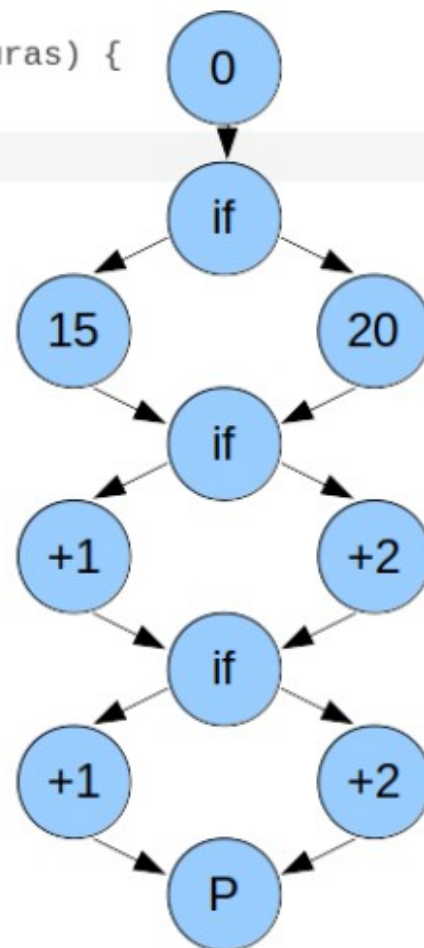
    if (casquinha) {
        preco = preco + 2;
    } else {
        preco = preco + 1;
    }

    if (coberturas > 1){
        preco = preco + 2;
    } else {
        preco = preco + 1;
    }

    return preco;
}
```

```
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

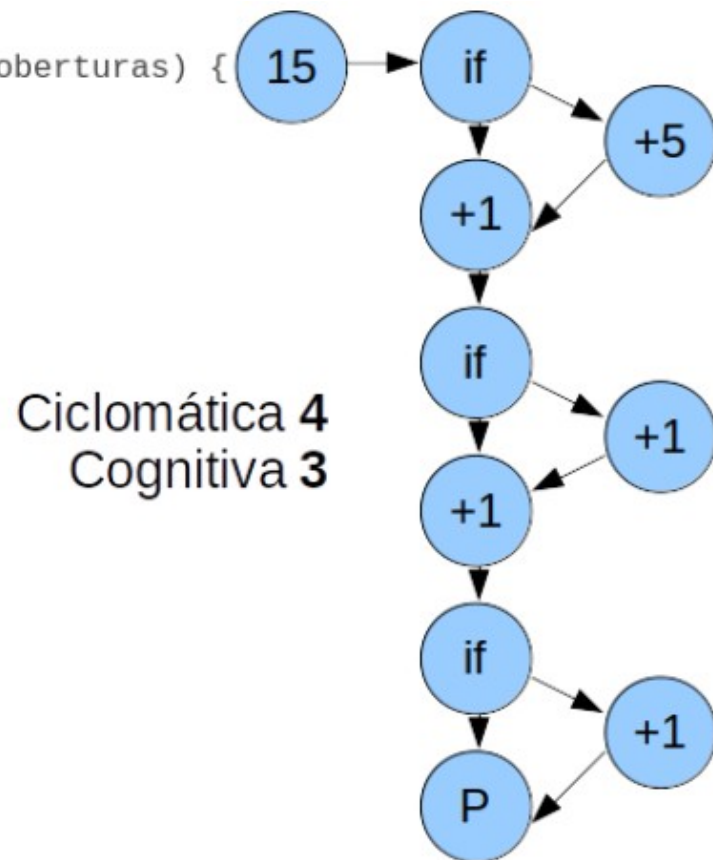


Sorveteria 2: Caminho Principal/Alternativo (Só IFs)

exemplo prático

```
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;
}
```



Agora, somente sorvete premium tem casquinha e coberturas extras

```
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 {
```

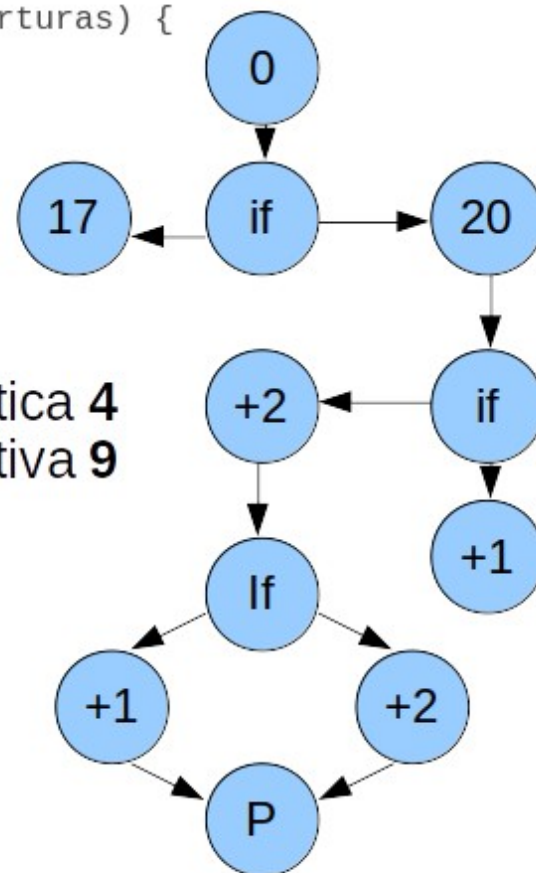
```
        preco = 15 + 1 + 1; // copo + 1 cob
```

```
    }
```

```
    return preco;
```

```
}
```

Ciclomática 4
Cognitiva 9



Como melhorar a pontuação de complexidade cognitiva mantendo o mesmo funcionamento (somente *premium* tem casquinha e cob. extra)?

+1 `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;

Refatoração

```
int preco = 15 + 1 + 1; // copo + 1 cob
+1 if (!premium) return preco;
    Preco = 20 + 1 + 1; // copo + 1 cob
+1 if (casquinha) preco = preco + 1;
+1 if (coberturas > 1) preco = preco + 1;
    return preco;
```

A complexidade cognitiva **desceu de 9 para 3**

- Inicia “preco” em 17 (simples + copo + 1 cobertura)
- Inverte a lógica do primeiro IF (not premium)
- Dois IFs simples de nível zero com “preco+1”



implicações

1. Score Mais Acurado
2. Contempla Estruturas Modernas
3. Mede Níveis Acima de Método

O score é mais sensível a mudanças de leitura/entendimento e mais **acurado** que a complexidade ciclomática quanto a isso

```
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
```

Ciclomática	4
-------------	---

Cognitiva	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
```

Ciclomática	4
-------------	---

Cognitiva	1
-----------	---

Estruturas de linguagens mais modernas são consideradas

Lambda

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

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



“Safe Navigation Operator”

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

Lambda

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

Complexidade cognitiva pode ser usada para **medir classes, módulos e aplicações** inteiras de forma realmente significativa

Cognitive Complexity 615 

/	–
src/main/java/br/net/buzu	25
src/main/java/br/net/buzu/context	22
src/main/java/br/net/buzu/data	7
src/main/java/br/net/buzu/metaclass	76
src/main/java/br/net/buzu/metadata	25
src/main/java/br/net/buzu/metadata/build	17
src/main/java/br/net/buzu/metadata/build/load	33
src/main/java/br/net/buzu/metadata/build/parse	108
src/main/java/br/net/buzu/metadata/code	27
src/main/java/br/net/buzu/parsing	39
src/main/java/br/net/buzu/parsing/complex	22
src/main/java/br/net/buzu/parsing/simple	22
src/main/java/br/net/buzu/parsing/simple/bool	4

O pacote “parse” pontuou 108 para complexidade cognitiva que é o somatório da pontuação de suas classes.

Cognitive Complexity 108

BasicMetadataParser.java	47
MetadataParseException.java	0
ParseNode.java	13
Splitter.java	48

4 of 4 shown

conclusão

**complexidade cognitiva apresenta-se
como uma **melhor opção** em relação
à complexidade ciclomática como
medida de **complexidade** de software**

referências

CAMPBELL, G. Ann (SonarSource S.A.), **Cognitive Complexity**,
<https://www.sonarsource.com/docs/CognitiveComplexity.pdf>,
acessado em Janeiro/2019

BECK, Kent, **Padrões de Implementação**, Bookman, 2008

FOWLER, Martin. **BeckDesignRules**,
<https://martinfowler.com/bliki/BeckDesignRules.html>, em Jan/2019

COGNITIVE COMPLEXITY


A new way of measuring understandability

sonarsource


Copyright SonarSource S.A., 2018, Switzerland. All content is copyright protected.

para encerrar...

Obrigado!



Complexidade
Cognitiva



Complexidade
Ciclomática

Douglas Siviotti

douglas.siviotti@gmail.com

github.com/siviotti