

# Cláusula de Guarda

Reduzindo a Complexidade com o  
Padrão de Implementação  
Cláusula de Guarda



Online, 10 de Junho de 2021

Douglas Siviotti



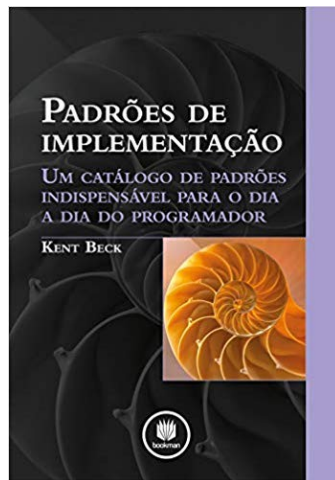
## Sobre a apresentação

Assunto: Padrão de Projeto Cláusula de Guarda

Público Alvo: Desenvolvedores

Organização: 46 Slides em 4 partes (+- 30 minutos)

1. Definição
2. Além do Estilo
3. Complexidade e Testabilidade
4. Conclusão



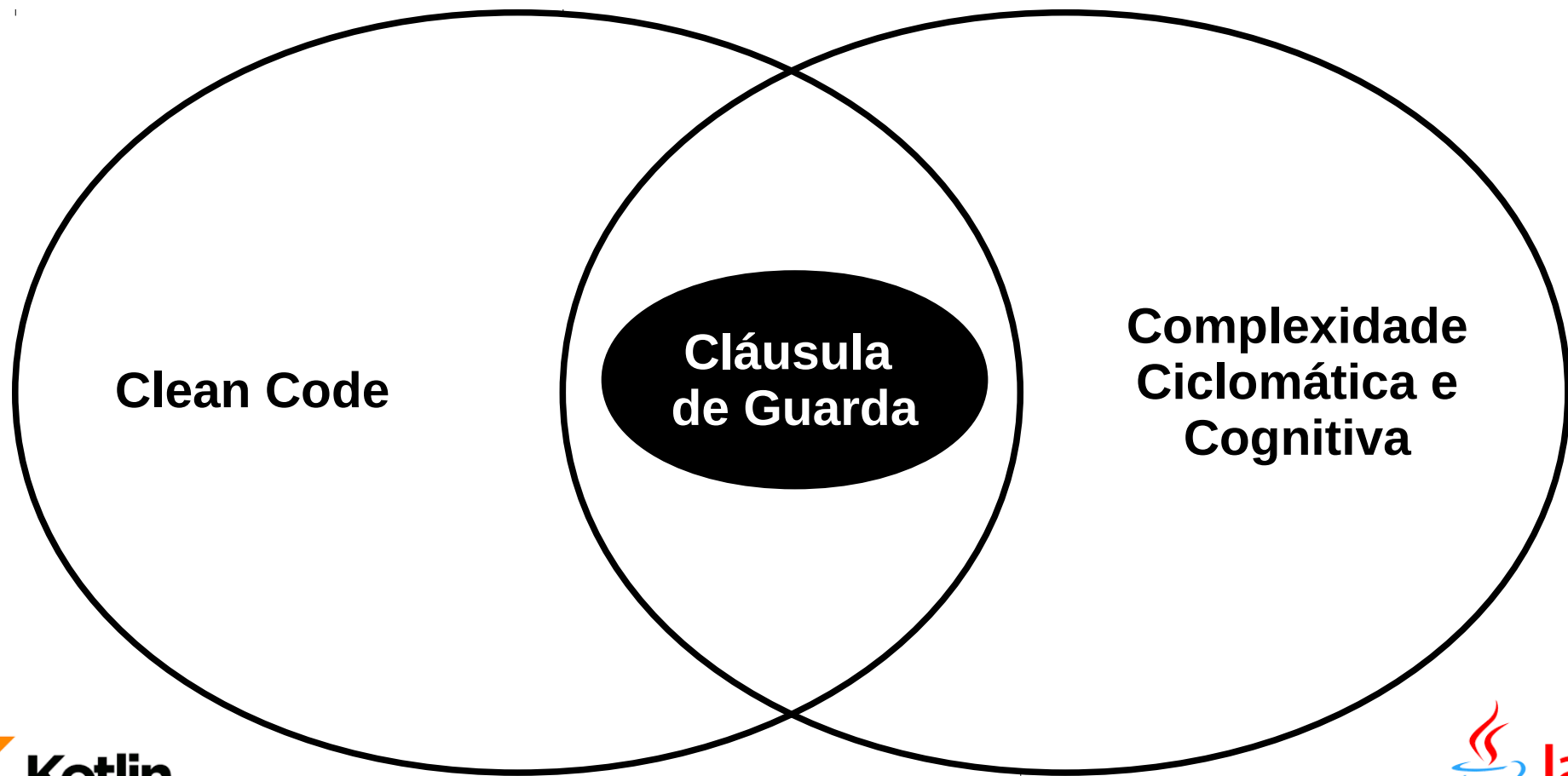
### Douglas Siviotti

Desenvolvedor há mais de 20 anos, analista de sistemas, especialista em engenharia de software pela UFRGS, pós graduando em direito do uso e proteção de dados pessoais pela PUC-MG, trabalha como arquiteto de software no SERPRO desde 2005 e com qualidade de software desde 2012. Atualmente atua como especialista em proteção de dados pessoais atuando no suporte, especificação de produtos, construção de processos e geração de cursos e conteúdos relacionados ao tema no blog [artessoftware.com.br](http://artessoftware.com.br) e na plataforma Udemy.



Material de apoio desta apresentação:  
<https://github.com/siviotti/clausula-de-guarda>





## definição



**cláusula de guarda é uma  
verificação de pré-condições  
de um escopo local (método)  
com uma possível saída  
antecipada deste escopo**

**cliente: Olá, como faço para chegar na sala 3?**

**porteiro: Deixe-me ver seu tíquete, por favor.**

**cliente: Ops, acho que eu perdi.**

**porteiro: Sinto muito, o senhor não pode entrar.**



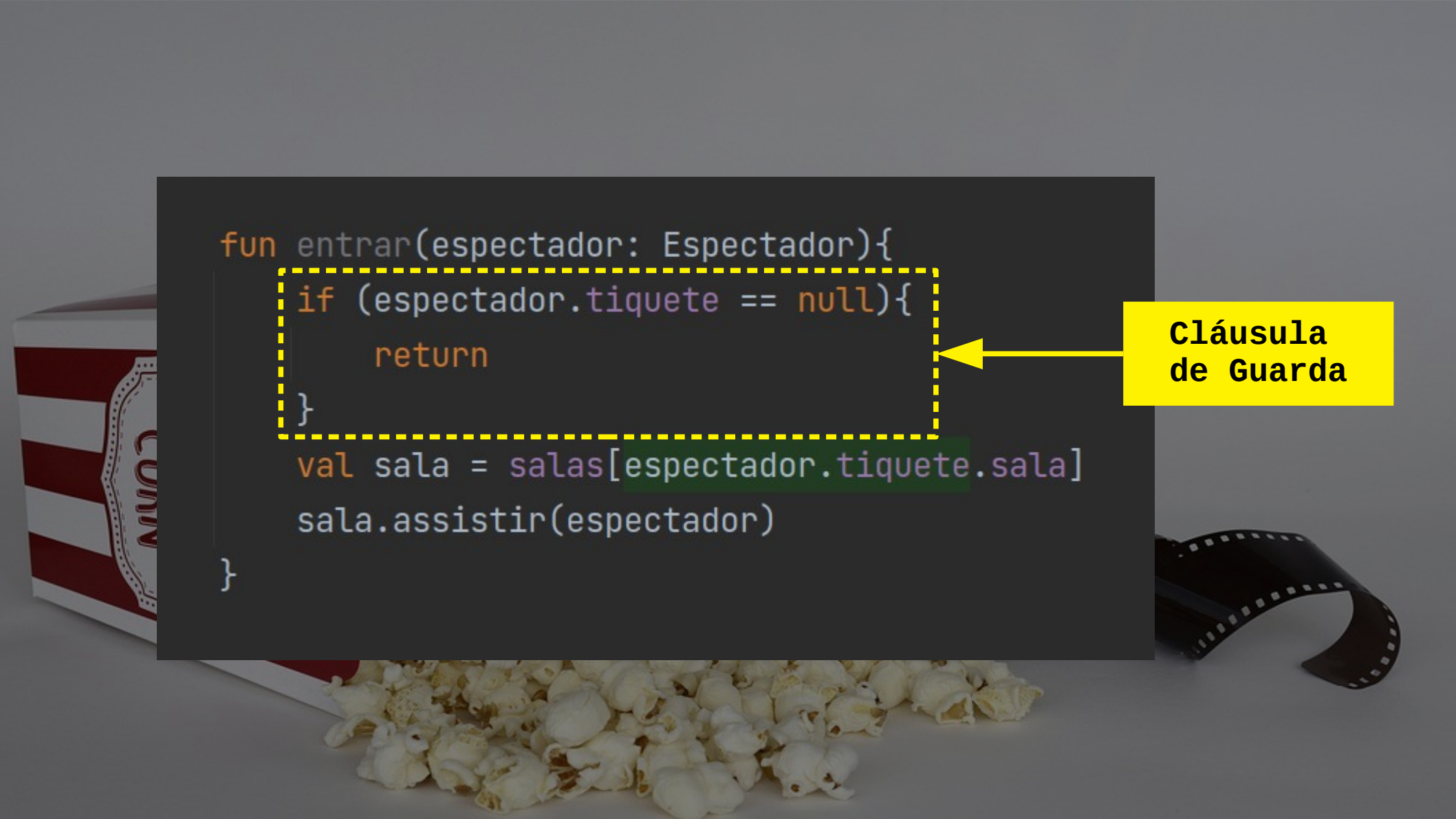


**objetivo:** Assistir o filme

**pré-condição:** Ter e mostrar o tíquete de entrada


se a pré-condição **não for atendida:** O objetivo principal que era assistir o filme não acontece





```
fun entrar(espectador: Espectador){  
    if (espectador.tiquete == null){  
        return  
    }  
    val sala = salas[espectador.tiquete.sala]  
    sala.assistir(espectador)  
}
```

Cláusula  
de Guarda



```
fun entrar(espectador: Espectador){  
    if (espectador.tiquete != null){  
        val sala = salas[espectador.tiquete.sala]  
        sala.asistir(espectador)  
    }  
}
```

Cláusula  
de Guarda

?



A **Cláusula de Guarda** é uma checagem prévia de uma ou mais **pré-condições** para execução de uma função ou método\* que, caso a pré-condição não seja atendida, provoca a **saída antecipada** do método ou função, separando as verificações de pré-condições do código principal deste método. Dessa forma, o o método torna-se **menos complexo** e mais **fácil de ser lido, mantido e testado**.

\* ou construtor, inicializador etc

apenas  
estilos  
diferentes

?

```
// "Estilo" 1: Se atende a pré-condição, faz  
if (espectador.tiquete != null){  
    // faz o que tem que fazer  
}
```

```
// "Estilo" 2: Se não atende a pré-condição, sai  
if (espectador.tiquete == null){  
    // Dispara uma exceção ou dá "return"  
}  
// faz o que tem que fazer
```

Cláusula  
de Guarda



**além do estilo**

**cláusula de guarda é um  
recurso de design de código e  
não um mero estilo de  
codificação**

## Aspectos de Estilo de Código

- Abertura de chaves
- Uso de espaços ou “tabs”
- “Code Style” da equipe
- Camel case, snake case etc
- Pulo de linha, espaços

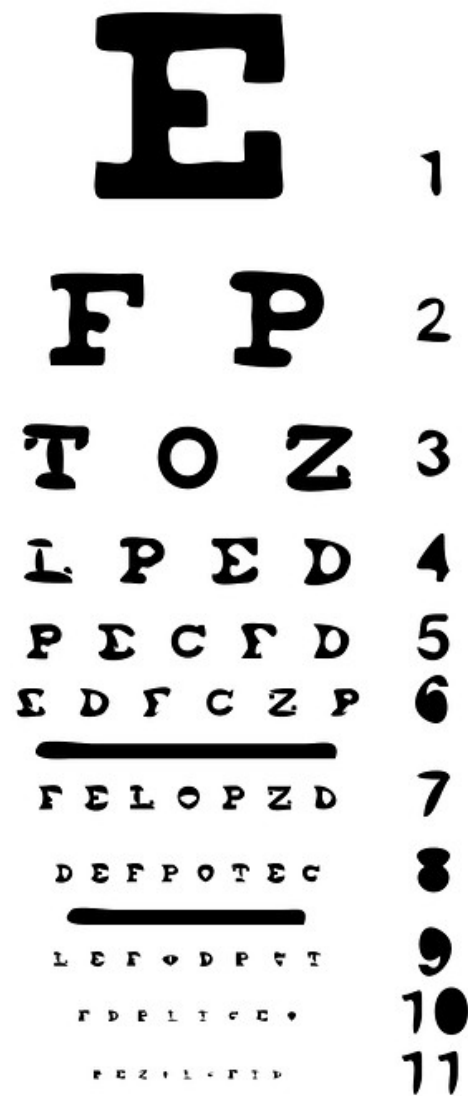


além do estilo

```
3      class Style1 {  
4          fun teste(){  
5              //conteudo  
6          }  
7      }  
8  
9      class Style2  
10     {  
11         fun teste()  
12         {  
13             //conteudo  
14         }  
15     }
```



- Legibilidade e Facilidade de Manutenção
- Impactos e Efeitos Colaterais em Código Cliente
- Complexidade e Testabilidade (parte 3)



```
public BigInteger bigDividir(BigInteger dividendo, BigInteger divisor) {  
    if (dividendo != null) { // pré-condição 1: dividendo não é nulo  
        if (divisor != null) { // pré-condição 2: divisor não é nulo  
            if (divisor.intValue() != 0) { // pré-condição 3: divisor não é zero  
                return dividendo.divide(divisor);  
            }  
        }  
    }  
    return null; // CLEAN CODE ALERT: Nunca passe ou retorne nulo  
}
```

**Código 1**

CLEAN CODE ALERT: Os comentários são desnecessários!

```
public BigInteger bigDivide(BigInteger dividendo, BigInteger divisor) {  
    if (dividendo == null) {// pré-condição 1: dividendo não é nulo  
        throw new NullPointerException("0 dividendo não pode ser nulo!");  
    }  
    if (divisor == null) {// pré-condição 2: divisor não é nulo  
        throw new NullPointerException("0 divisor não pode ser nulo!");  
    }  
    if (divisor.intValue() == 0) {// pré-condição 3: divisor não é zero  
        throw new ArithmeticException("0 divisor não pode ser zero!");  
    }  
    return dividendo.divide(divisor);  
}
```

**Código 2**



## Código 1

- Um pouco menor, mas com IFs aninhados
- Retorna “null” quando as pré-condições não são atendidas
- Quando retorna “null” não fica claro qual pré-condição não foi atendida
- Pré-condições e código principal estão misturados

## Código 2

- Um pouco maior, mas os IFs são independentes
- Ou dá erro ou roda o código principal, sem ambiguidade (null)
- Cada pré-condição não atendida gera uma exceção clara
- Separa as pré-condições do código principal (divisão)

# Impactos e Efeitos Colaterais em Código Cliente

além do estilo

**Código 1**

```
6 public static void main(String[] args) {  
7  
8     BigInteger dividendo = BigInteger.valueOf(6);  
9     BigInteger divisor = BigInteger.valueOf(0);  
10    Matematica matematica = new Matematica();  
11    BigInteger resultado = matematica.bigDividir(dividendo, divisor);  
12    System.out.println("Resultado = " + resultado.toString());  
13
```

Run: Impacto x

↑ [/home/douglas/bin/jdk11/bin/java](#) -javaagent:/home/douglas/bin/intelij2020

↓ Exception in thread "main" java.lang.NullPointerException Create breakpoint

at siviotti.clausuladeguarda.Impacto.main([Impacto.java:12](#))

Process finished with exit code 1

# Impactos e Efeitos Colaterais em Código Cliente

além do estilo

Código 2

```
6 public static void main(String[] args) {  
7  
8     BigInteger dividendo = BigInteger.valueOf(6);  
9     BigInteger divisor = BigInteger.valueOf(0);  
10    Matematica matematica = new Matematica();  
11    BigInteger resultado = matematica.bigDivide(dividendo, divisor);  
12    System.out.println("Resultado = " + resultado.toString());  
13
```

Run: Impacto x

```
me/douglas/bin/jdk11/bin/java -javaagent:/home/douglas/bin/intelij2020/lib/idea_rt.jar=347  
ption in thread "main" java.lang.ArithmeticException Create breakpoint : 0 divisor não pode s  
at siviotti.clausuladeguarda.Matematica.bigDivide(Matematica.java:86)  
at siviotti.clausuladeguarda.Impacto.main(Impacto.java:11)  
cess finished with exit code 1
```

```
public BigInteger bigDividir(BigInteger dividendo, BigInteger divisor) {  
    if (dividendo != null) { // pré-condição 1: dividendo não é nulo  
        if (divisor != null) { // pré-condição 2: divisor não é nulo  
            if (divisor.intValue() != 0) { // pré-condição 3: divisor não é zero  
                return dividendo.divide(divisor);  
            }  
        }  
    }  
    //return null; // CLEAN CODE ALERT: Nunca passe ou retorne nulo  
    throw new ArithmeticException("0 divisor não pode ser zero!");  
}
```

**Código 1**  
**Refatorado**

CLEAN CODE ALERT: Código morto deve ser removido!



# Efeitos Colaterais: Mensagem Errada

além do estilo

**Código 1  
Refatorado**

**Pré-condição 1  
não atendida**

```
public class Impacto {  
    public static void main(String[] args) {  
  
        BigInteger dividendo = BigInteger.valueOf(6);  
        BigInteger divisor = BigInteger.valueOf(0);  
        Matematica matematica = new Matematica();  
        BigInteger resultado = matematica.bigDividir(dividendo: null, divisor);  
        System.out.println("Resultado = " + resultado.toString());  
    }  
}
```

Impacto x

↑ [loulas/bin/jdk11/bin/java](#) -javaagent:/home/douglas/bin/intelij2020/lib/idea\_rt.jar=44395:

↓ on in thread "main" java.lang.ArithmeticException Create breakpoint : 0 divisor não pode ser z

↕ siviotti.clausuladeguarda.Matematica.bigDividir(Matematica.java:67)

↕ siviotti.clausuladeguarda.Impacto.main(Impacto.java:11)

⌵ ; finished with exit code 1

**Mensagem  
Equivocada  
Pré-cond. 3**

## Código 1

- Retornar “null” gerou uma **confusão** no código cliente
- Obriga o programador a ler o **código** “bigDividir()”
- Ao retornar uma exceção ao final melhorou, mas ainda gera confusão sobre **qual pré-condição não foi atendida**

## Código 2

- Executa divisão ou dispara exceção, **sem dubiedade**
- O programador só precisa conhecer o “**contrato**”
- Cada pré-condição não atendida gera **uma exceção clara**



**complexidade**

**complexidade é fator chave  
para aumentar a facilidade de  
entendimento/manutenção e a  
testabilidade**

**Complexidade Ciclométrica** (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).



**E o que a cláusula de guarda tem com isso?**



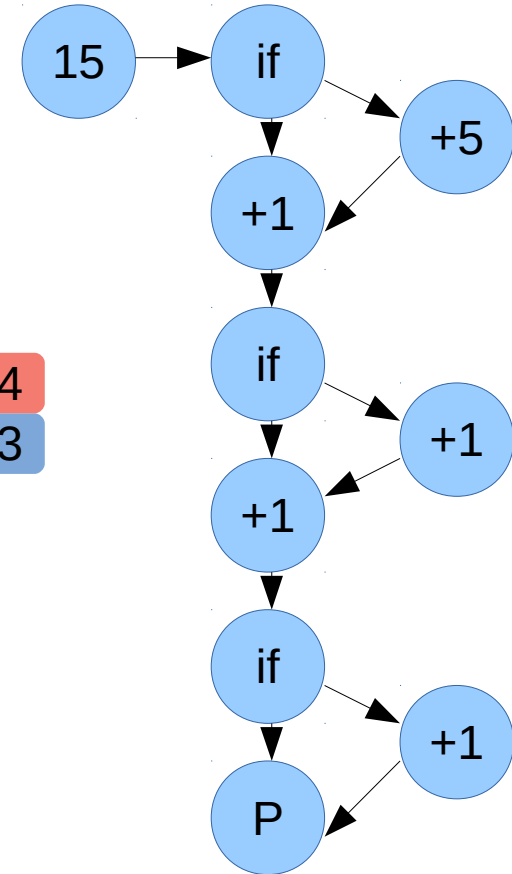
# Algoritmo Exemplo (Requisito 1)

complexidade

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

<sup>+1</sup>	
Ciclomática	4
Cognitiva	3

+1...

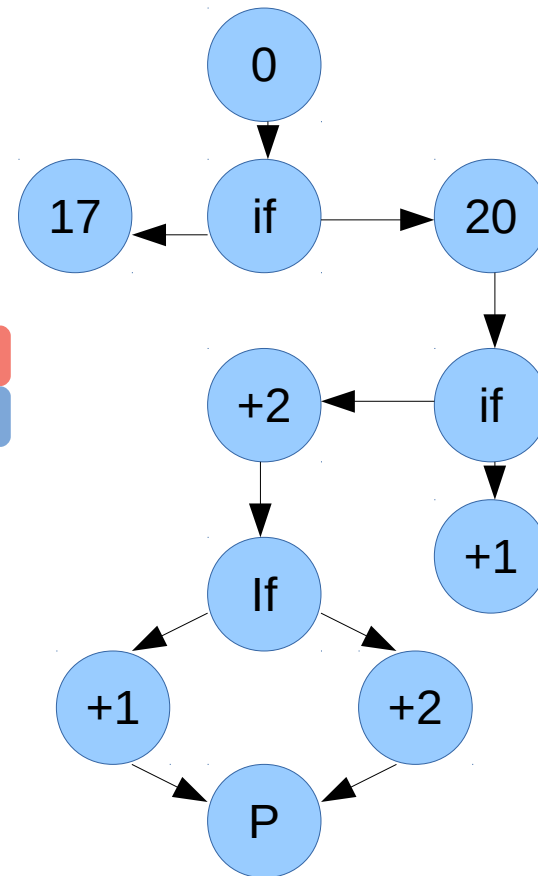


## Algoritmo Exemplo com Aninhamento (Requisito 2)

complexidade

```
    +1
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 = 15 + 1 + 1;
        }
    }
    return preco;
}
```

	+1
Ciclomática	4
Cognitiva	9
	+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;
}
```

## Requisito 1

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



@Test

## Teste do Requisito 1

```
public void testPrecoSorvete(){
```

```
    // Sorvete Comum
```

```
    assertEquals(17, sorvete.precoSorvete(premium: false, casquinha: false, coberturas: 1));
```

```
    // Sorvete com somente casquinha ou somente coberturas = 18
```

```
    assertEquals(18, sorvete.precoSorvete(premium: false, casquinha: false, coberturas: 3));
```

```
    assertEquals(18, sorvete.precoSorvete(premium: false, casquinha: true, coberturas: 1));
```

```
    // Sorvete comum com casquinha e coberturas
```

```
    assertEquals(19, sorvete.precoSorvete(premium: false, casquinha: true, coberturas: 3));
```

```
    // Sorvete Premium
```

```
    assertEquals(22, sorvete.precoSorvete(premium: true, casquinha: false, coberturas: 1));
```

```
    assertEquals(23, sorvete.precoSorvete(premium: true, casquinha: false, coberturas: 3));
```

```
    // Premium casquinha
```

```
    assertEquals(23, sorvete.precoSorvete(premium: true, casquinha: true, coberturas: 1));
```

```
    // Premium Completo
```

```
    assertEquals(24, sorvete.precoSorvete(premium: true, casquinha: true, coberturas: 3));
```

```
}
```



```
public int precoSorvete1(boolean premium, boolean casquinha, int coberturas)
{
    int preco = 0;
    if (premium) {
        preco = 20;
        if (casquinha) {
            preco = preco + 2;
            if (coberturas > 1) {
                preco = preco + 2;
            } else {
                preco = preco + 1;
            }
        } else {
            preco = preco + 2;
        }
    } else {
        preco = 15 + 1 + 1;
    }
    return preco;
}
```

Novo Requisito:

1. Somente sabores *premium* podem ser casquinha

2. Somente casquinha pode ter mais de uma cobertura

Requisito 2

Código 1

```
/**
```

- \* Este versão usa IFs de saída antecipada. **Não** é exatamente uma cláusula de
  - \* guarda já que não são pré-condições, mas sua adoção gera iguais vantagens.
  - \* O código fica menor e mais simples de ser lido.
  - \* A complexidade cognitiva passa de 9 para 3!
- ```
*/
```

**Requisito 2**

```
public int precoSorvete2(boolean premium, boolean casquinha, int coberturas) {  
    int preco = 15 + 1 + 1; // copo + 1 cob  
    if (!premium) return preco; IF de saída antecipada  
    preco = 20 + 1 + 1; // copo + 1 cob  
    if (!casquinha) return preco; IF de saída antecipada  
    return (coberturas > 1) ? preco + 2 : preco + 1;  
}
```

**Código 2**

## Teste do Requisito 2

@Test

```
public void testPrecoSorvete1() {
```

```
    // Sorvete Comum
```

```
    assertEquals(17, sorvete.precoSorvete1(premium: false, casquinha: false, coberturas: 1));
```

```
    assertEquals(17, sorvete.precoSorvete1(premium: false, casquinha: false, coberturas: 3));
```

```
    assertEquals(17, sorvete.precoSorvete1(premium: false, casquinha: true, coberturas: 1));
```

```
    assertEquals(17, sorvete.precoSorvete1(premium: false, casquinha: true, coberturas: 3));
```

```
    // Sorvete Premium
```

```
    assertEquals(22, sorvete.precoSorvete1(premium: true, casquinha: false, coberturas: 1));
```

```
    assertEquals(22, sorvete.precoSorvete1(premium: true, casquinha: false, coberturas: 3));
```

```
    // Premium casquinha
```

```
    assertEquals(23, sorvete.precoSorvete1(premium: true, casquinha: true, coberturas: 1));
```

```
    // Premium Completo
```

```
    assertEquals(24, sorvete.precoSorvete1(premium: true, casquinha: true, coberturas: 3));
```

```
}
```

Deveria dar erro nesses cenários impossíveis?

|          |             |   |                                           |             |   |          |
|----------|-------------|---|-------------------------------------------|-------------|---|----------|
| Código 1 | Ciclomática | 4 | Testabilidade (4)<br>Leitura e Manutenção | Ciclomática | 4 | Código 2 |
|          | Cognitiva   | 9 |                                           | Cognitiva   | 3 |          |

Refatoração

```

+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;
    
```

```

int preco = 15 + 1 + 1; // copo + 1 cob
+1 if (!premium) return preco;
    preco = 20 + 1 + 1; // copo + 1 cob
+1 if (!casquinha) return preco;
    return (coberturas > 1)
+1 ? preco + 2 : preco + 1;
    
```

- Os “IFs de saída antecipada” se comportam de forma semelhante à cláusula de guarda
- O código 2 ficou menor e mais simples

1. Se a complexidade ciclomática não se altera, será que o número de cenários necessários é o mesmo ao utilizar cláusulas de guarda?
2. Como a cláusula de guarda impacta testes de unidade?



1. Se a complexidade ciclomática não se altera, será que o número de cenários necessários é o mesmo ao utilizar cláusulas de guarda?

**Resposta: Depende do algoritmo, mas geralmente SIM**

2. Como a cláusula de guarda impacta testes de unidade?

**vejamos a seguir...**

```
public BigInteger bigDividir(BigInteger dividendo, BigInteger divisor) {  
+1  if (dividendo != null) { // pré-condição 1: dividendo não é nulo  
    +2  if (divisor != null) { // pré-condição 2: divisor não é nulo  
        +3  if (divisor.intValue() != 0) { // pré-condição 3: divisor não é zero  
            return dividendo.divide(divisor);  
        }  
    }  
}  
return null; // CLEAN CODE ALERT: Nunca passe ou retorne nulo  
}
```

Ciclomática

4

Cognitiva

6

**Código 1**

```
public BigInteger bigDivide(BigInteger dividendo, BigInteger divisor) {  
+1  if (dividendo == null) { // pré-condição 1: dividendo não é nulo  
    throw new NullPointerException("0 dividendo não pode ser nulo!");  
  }  
+1  if (divisor == null) { // pré-condição 2: divisor não é nulo  
    throw new NullPointerException("0 divisor não pode ser nulo!");  
  }  
+1  if (divisor.intValue() == 0) { // pré-condição 3: divisor não é zero  
    throw new ArithmeticException("0 divisor não pode ser zero!");  
  }  
  return dividendo.divide(divisor);  
}
```

Ciclomática

4

Cognitiva

3

**Código 2**

1. Se a complexidade ciclomática não se altera, será que o número de cenários necessários é o mesmo ao utilizar cláusulas de guarda?

Resposta: Depende do algoritmo, mas geralmente SIM

2. Como a cláusula de guarda impacta testes de unidade?

**a. Se é mais fácil entender fica mais fácil testar**

( se está difícil testar é porque está muito complexo)

**b. É possível separar o código de guarda do principal**

(ler e entender cada parte como um método separado)

```
106
107 /**
108  * Este versão utiliza algumas cláusulas de guarda e já dá pra perceber que uma
109  * parte do é checagem de pré-condições enquanto a segunda é o cálculo de fato.
110  */
111 public int precoSorvete3(boolean premium, boolean casquinha, int coberturas) {
112     if (! premium && casquinha)
113         throw new IllegalArgumentException("Somente premium tem casquinha");
114     if (! casquinha && coberturas > 1)
115         throw new IllegalArgumentException("Somente premium + casquinha pode ter mais de
116     if (coberturas > 3)
117         throw new IllegalArgumentException("O máximo de coberturas permitido é 3");
118     int preco = 15 + 1 + 1; // copo + 1 cob
119     if (!premium) return preco;
120     preco = 20 + 1 + 1; // copo + 1 cob
121     if (!casquinha) return preco;
122     return (coberturas > 1) ? preco + 2 : preco + 1;
123 }
124
```

1

2

Separação mental em duas partes:

1. Pré-condições
2. Código principal



```

61 @Test
62 public void testPrecoSorvete3() {
63     // Pré-condições
64     assertThrows(IllegalArgumentException.class, () -> // PC1: casquinha sem premium
65         { sorvete.precoSorvete3( premium: false, casquinha: true, coberturas: 1); });
66     assertThrows(IllegalArgumentException.class, () -> // PC2: cobertura sem casquinha
67         { sorvete.precoSorvete3( premium: true, casquinha: false, coberturas: 3); });
68     assertThrows(IllegalArgumentException.class, () -> // PC3: muitas coberturas
69         { sorvete.precoSorvete3( premium: true, casquinha: true, coberturas: 5); });
70     // Sorvete Comum
71     assertEquals(17, sorvete.precoSorvete3( premium: false, casquinha: false, coberturas: 1));
72     // Sorvete Premium
73     assertEquals(22, sorvete.precoSorvete3( premium: true, casquinha: false, coberturas: 1));
74     // Premium casquinha
75     assertEquals(23, sorvete.precoSorvete3( premium: true, casquinha: true, coberturas: 1));
76     // Premium Completo
77     assertEquals(24, sorvete.precoSorvete3( premium: true, casquinha: true, coberturas: 3));
78 }

```

Estes 7 cenários cobrem 100% das linhas

1

2

```
126 * Este versão utiliza desloca a checagem de pré-condições para outro mét. 1 4 26 ^ v
127 */
128 public int precoSorvete4(boolean premium, boolean casquinha, int coberturas) { 2
129 1 checkParametros(premium, casquinha, coberturas)
130     int preco = 15 + 1 + 1; // copo + 1 cob
131     if (!premium) return preco;
132 2 preco = 20 + 1 + 1; // copo + 1 cob
133     if (!casquinha) return preco;
134     return (coberturas > 1) ? preco + 2 : preco + 1;
135 }
136
137 void checkParametros(boolean premium, boolean casquinha, int coberturas) { 1
138     if (! premium && casquinha)
139         throw new IllegalArgumentException("Somente premium tem casquinha");
140     if (! casquinha && coberturas > 1)
141         throw new IllegalArgumentException("Somente premium + casquinha pode ter mais de
142     if (coberturas > 3)
143         throw new IllegalArgumentException("O máximo de coberturas permitido é 3");
144 }
```

```

81 ▶ @Test    public void testPrecoSorvete4Precondicoes() {
82         assertThrows(InvalidArgumentException.class, () -> // PC1: casquinha sem premium
83         { sorvete.precoSorvete3( premium: false, casquinha: true, coberturas: 1); });
84     1  assertThrows(InvalidArgumentException.class, () -> // PC2: cobertura sem casquinha
85         { sorvete.precoSorvete3( premium: true, casquinha: false, coberturas: 3); });
86         assertThrows(InvalidArgumentException.class, () -> // PC3: muitas coberturas
87         { sorvete.precoSorvete3( premium: true, casquinha: true, coberturas: 5); });
88     }
89

```

```

90 ▶ @Test    public void testPrecoSorvete4() {
91         // Sorvete Comum
92         assertEquals(17, sorvete.precoSorvete4( premium: false, casquinha: false, coberturas: 1));
93         // Sorvete Premium
94         assertEquals(22, sorvete.precoSorvete4( premium: true, casquinha: false, coberturas: 1));
95     2  // Premium casquinha
96         assertEquals(23, sorvete.precoSorvete4( premium: true, casquinha: true, coberturas: 1));
97         // Premium Completo
98         assertEquals(24, sorvete.precoSorvete4( premium: true, casquinha: true, coberturas: 3));
99     }

```



```
public void add(int index, E element) {  
    ① rangeCheckForAdd(index);  
    {  
        modCount++;  
        ② final int s;  
        Object[] elementData;  
        if ((s = size) == (elementData = this.elementData).length)
```

A version of rangeCheck used by add and addAll.

```
private void rangeCheckForAdd(int index) { ①  
    if (index > size || index < 0)  
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));  
}
```

```
constructor(text: String, val rowCount: Int = DEFAULT_ROW_COUNT)
```

1 ^ v

```
15     val size = ROW_SIZE * rowCount
```

```
16     check(value: text.length == size) { "Table must have size=$size and rows=$rowCount. Cont"
```

```
17     check(value: text.toSet().size == text.length) { "Duplicated elements: ${filterDuplicate"
```

```
18     checkValidChar(text)
```

```
19     var temp = mutableListOf<Char>()
```

```
20     val rowsTemp = mutableListOf<List<Char>>()
```

```
21     val mapTemp = mutableMapOf<Char, Int>()
```

```
22     text.forEachIndexed { index, char ->
```

```
23         temp.add(char)
```

```
24         mapTemp[char] = index % 10
```

```
25         if ((index + 1) % ROW_SIZE == 0) { //10, 20, 30, 40, 50, 60
```

```
26             rowsTemp.add(temp.toList())
```

```
27             temp = mutableListOf()
```

```
28         }
```

```
29     }
```

```
30     rows = rowsTemp.toList()
```

```
31     map = mapTemp.toMap()
```

```
32 }
```

Cláusulas de guarda “barulhentas” em construtores ajudam a criar objetos com estados válidos (IllegalStateException)



1. Cláusula de guarda não é questão de estilo, mas **design de código**
2. Melhora a leitura, entendimento e manutenção
  - Separa as **pré-condições** do código principal
  - Reduz a **complexidade cognitiva** (medida de qualidade Sonarqube)
3. Reduz ou evita impactos e efeitos colaterais em **código cliente**
4. Aumenta a testabilidade, sem reduzir o número de cenários
  - Código menos complexo é **mais fácil** de testar (conclusão 2)
  - Permite testes/cenários mais **precisos e especializados**

para encerrar...

# Obrigado!



Cláusula  
de Guarda

- Complexidade
- Clean Code