

# Cláusula de Guarda

Reduzindo a Complexidade com o  
Padrão de Implementação  
Cláusula de Guarda



Online, 10 de Junho de 2021

Douglas Siviotti

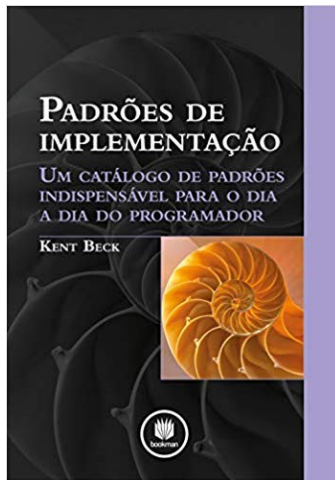


# Sobre a apresentação

Público Alvo: Desenvolvedores

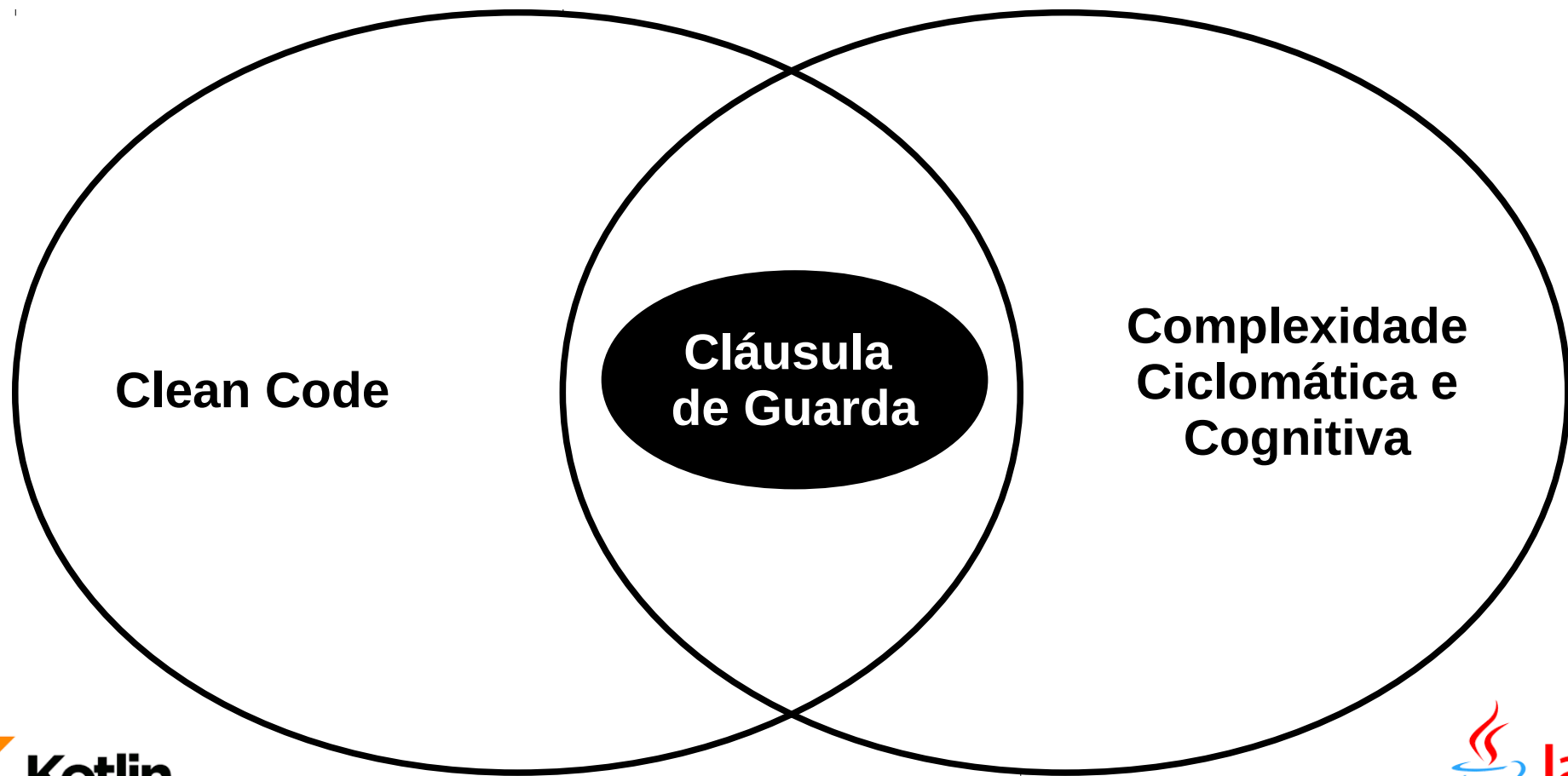
Organização: 37 Slides em  
4 partes (+- 30 minutos)

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 [artesoftware.com.br](http://artesoftware.com.br) e na plataforma Udemy.



## 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.**



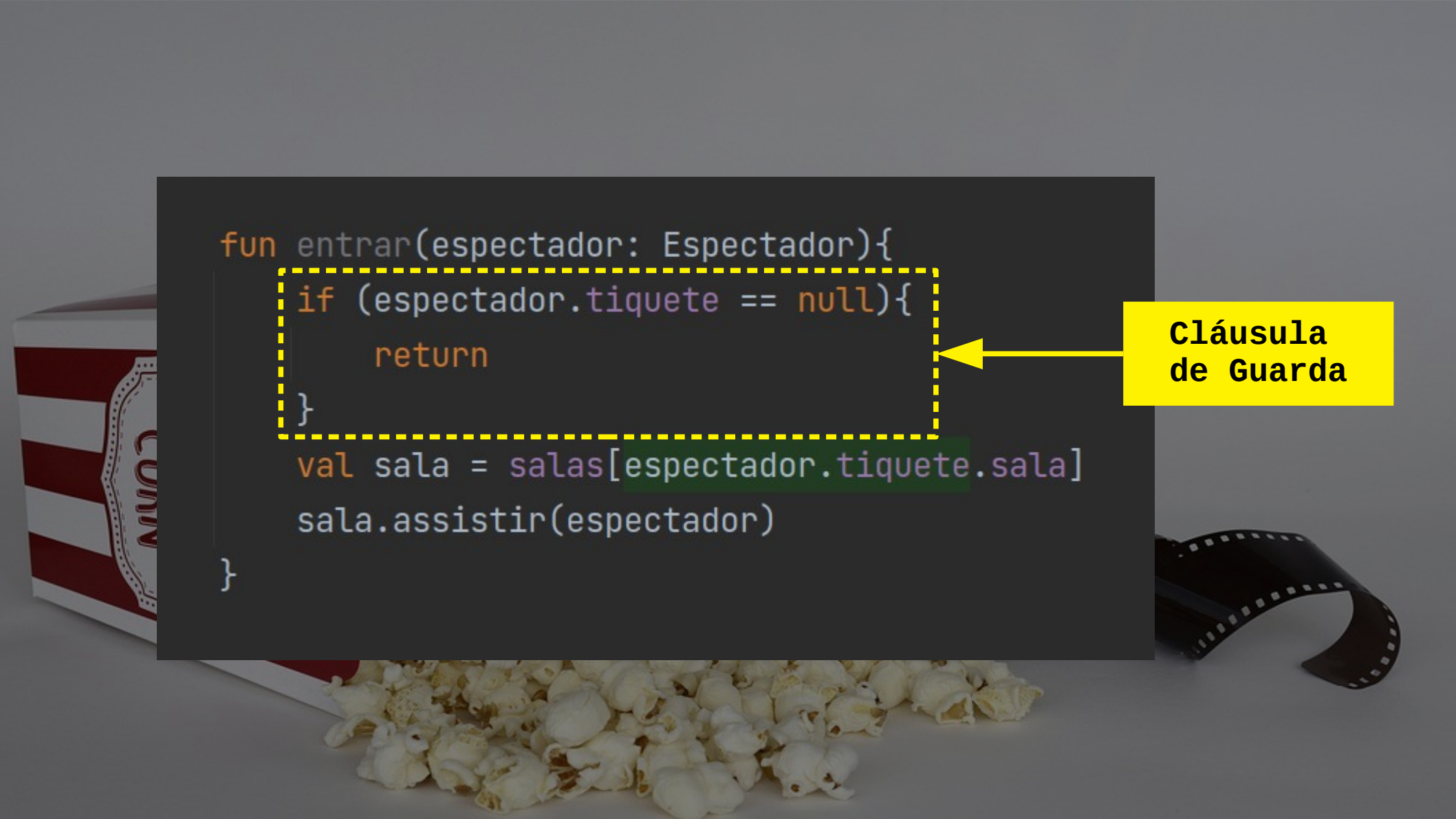


**objetivo:** Assistir o filme

**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**.

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**

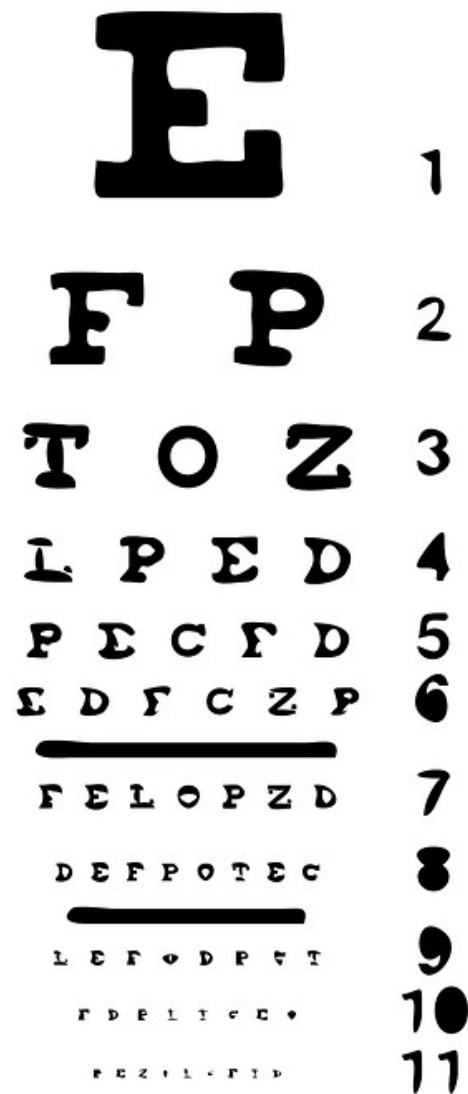
- Abertura de chaves
- Uso de espaços ou “tabs”
- “Code Style” da equipe
- Camel case, snake case etc
- Pulo de linha, espaços

```
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
- 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)

## 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

## 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
exception 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());  
    }  
}
```

h: 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).



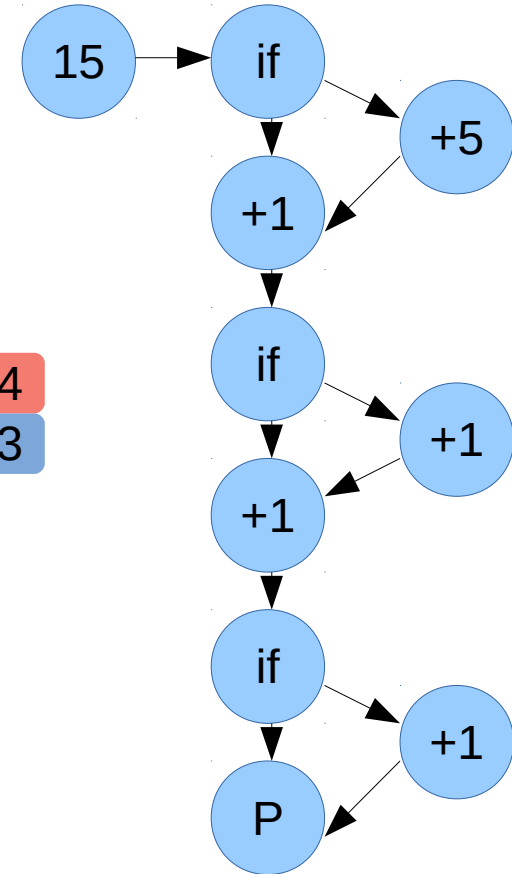
# Algoritmo Exemplo (Requisito 1)

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...



E o que a cláusula de guarda tem com isso?

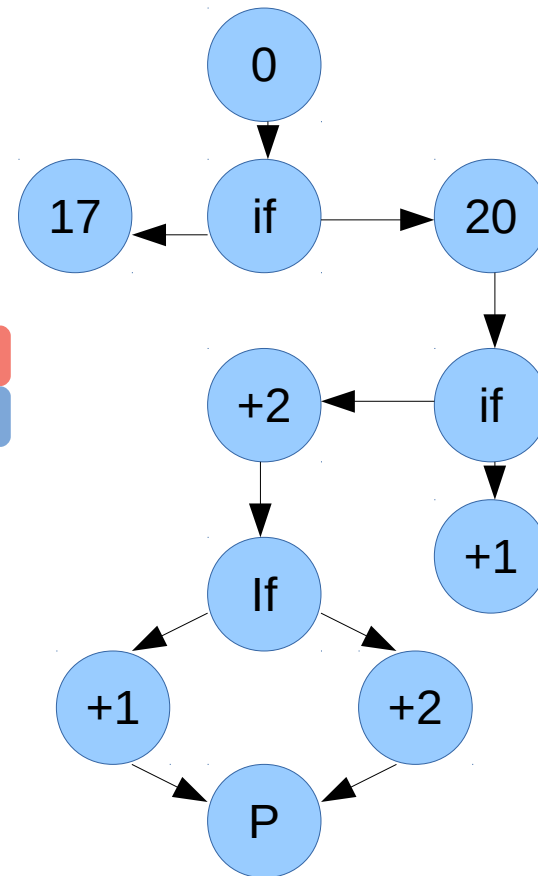


## Algoritmo Exemplo com Aninhamento (Requisito 2)

complexidade

```
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;  
    }  
    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

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

Teste do  
Requisito 1

```
    // 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. <b>Não é exatamente uma cláusula de  
* guarda</b> 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;  
    preco = 20 + 1 + 1;      // copo + 1 cob  
    if (!casquinha) return preco;  
    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));
```

```
}
```

Código 1	Ciclomática	4	Testabilidade (4) 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;
    
```

Ambos os algoritmos precisam de **4 cenários** de teste para cobrir 100% das linhas



```
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**

## A Cláusula de Guarda é

**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).

## A Cláusula de Guarda é

**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).

## A Cláusula de Guarda é

**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).



## conclusão

cláusula de guarda é um elemento de **design** de código e não uma questão de estilo, ela melhorara leitura, manutenção e testabilidade através da redução de **complexidade**

para encerrar...

# Obrigado!



Cláusula  
de Guarda

- Complexidade
- Clean Code