TESTE E QUALIDADE DE SOFTWARE

João Choma Neto

joao.choma@gmail.com

Unicesumar – Maringá – 2023/2

ANTERIORMENTE

TESTE FUNCIONAL

VISÃO PARA TESTAR (1)

- Segundo Pressman (2011), qualquer produto de engenharia pode ser **testado** por uma de duas maneiras:
  - Conhecendo a **função** para o qual um produto foi projetado para realizar
  - Construir testes que demonstram que cada uma das funções é totalmente operacional

CAIXA PRETA

- A primeira abordagem de teste usa uma visão externa e é chamada de teste **caixa-preta**
- Faz referência a **testes** realizados na **interface** do software
- Examina alguns aspectos fundamentais de um sistema, com **pouca preocupação** em relação à **estrutura lógica** interna do software

FRAMEWORKS PARA TESTE FUNCIONAL

JUnit (Java): Ele suporta testes de unidade, testes de integração e testes funcionais.

TestNG (Java): TestNG é uma alternativa ao JUnit para testes de unidade e funcionais em Java.

pytest (Python): Ele é fácil de usar e oferece recursos avançados de descoberta automática de testes e geração de relatórios.

NUnit (C#): Ele oferece suporte a parametrização de testes e outras funcionalidades avançadas.

FRAMEWORKS PARA TESTE FUNCIONAL

Cucumber (Várias Linguagens): O Cucumber é uma ferramenta de teste de aceitação que utiliza a linguagem Gherkin para escrever cenários de teste em linguagem natural. Ele é frequentemente usado para testes funcionais.

Selenium (Web Applications): O Selenium é uma ferramenta popular para testar aplicativos da web. Ele permite a automação de testes de interface do usuário em navegadores.

FRAMEWORKS PARA TESTE FUNCIONAL

Robot Framework (Várias Linguagens): O Robot Framework é uma estrutura genérica de automação de teste que pode ser usada para testes funcionais e de aceitação, suportando várias linguagens de programação.

Jest (JavaScript/Node.js): O Jest é um framework de teste para JavaScript e Node.js. É especialmente útil para testar aplicativos React e possui recursos como "snapshot testing".

PHP Unit (PHP): O PHPUnit é um framework de teste para PHP, projetado para testes de unidade e funcionais. Ele segue uma abordagem semelhante ao JUnit.

CLASSES DE EQUIVALÊNCIA

- Classes de equivalência ajudam a organizar e simplificar a criação de casos de teste, permitindo que você escolha representantes de um grupo de dados que compartilham características semelhantes
- Isso é particularmente útil quando se trata de testar diferentes valores de entrada que devem ser tratados de maneira semelhante pelo sistema

CLASSES DE EQUIVALÊNCIA

- Classes de equivalência dividem o conjunto de dados de entrada em grupos ou classes que devem ser tratados da mesma maneira pelo software
- Ao testar um valor em uma classe, você pode fazer suposições sobre o comportamento do sistema em relação a outros valores na mesma classe.

CLASSES DE EQUIVALÊNCIA

- **Cenário: Valores Positivos e Negativos**
- Classes de Equivalência: Valores positivos, valores negativos e zero.

- Exemplo de Casos de Teste:
    - Teste com -10 (valor negativo)
    - Teste com 5 (valor positivo)
    - Teste com 0.

CLASSES DE EQUIVALÊNCIA

- **Cenário: Notas em uma Avaliação**
- Classes de Equivalência:
    - Notas abaixo da faixa válida (por exemplo, < 0)
    - Notas válidas (por exemplo, 0 a 10)
    - Notas acima da faixa válida (por exemplo, > 10)
- Exemplo de Casos de Teste:
    - Teste com -2 (nota abaixo da faixa válida)
    - Teste com 7.5 (nota válida)
    - Teste com 12 (nota acima da faixa válida)

TESTE ESTRUTURAL

VISÃO PARA TESTAR

- Segundo Pressman (2011), qualquer produto de engenharia pode ser **testado** por uma de duas maneiras:

VISÃO PARA TESTAR (1)

- Segundo Pressman (2011), qualquer produto de engenharia pode ser **testado** por uma de duas maneiras:
  - Conhecendo a **função** para o qual um produto foi projetado para realizar
  - Construir testes que demonstram que cada uma das funções é totalmente operacional

VISÃO PARA TESTAR (2)

- Segundo Pressman (2011), qualquer produto de engenharia pode ser **testado** por uma de duas maneiras:
  - Conhecendo o **funcionamento interno** de um produto, podem ser realizados testes para garantir que “tudo se encaixa”
  - Construir testes que demonstram que as operações internas foram realizadas de acordo com as especificações

VISÃO PARA TESTAR (2)

- Segundo Pressman (2011), qualquer produto de engenharia pode ser **testado** por uma de duas maneiras:
  - Conhecendo o **funcionamento interno** de um produto, podem ser realizados testes para garantir que "tudo se encaixa"
  - Construir testes que demonstram que as operações internas foram realizadas de acordo com as especificações

CAIXA BRANCA

- A segunda abordagem requer uma visão interna e é chamada de teste **caixa-branca**.
- Fundamenta-se em um exame rigoroso do detalhe procedimental.
- Os caminhos lógicos do software e as colaborações entre componentes são testados.

TESTE ESTRUTURAL

- Objetivo principal é garantir que o código-fonte seja testado de maneira abrangente, com ênfase na cobertura de todas as partes do código
- Testar:
  - Instruções
  - Caminhos de execução
  - Ramificações condicionais

TESTE ESTRUTURAL

- Objetivo principal é garantir que o código-fonte seja testado de maneira abrangente, com ênfase na cobertura de todas as partes do código
- Testar:
  - Teste de cobertura de código
  - Teste de caminho
  - Teste de ramificação
  - Teste de mutação

TESTE ESTRUTURAL

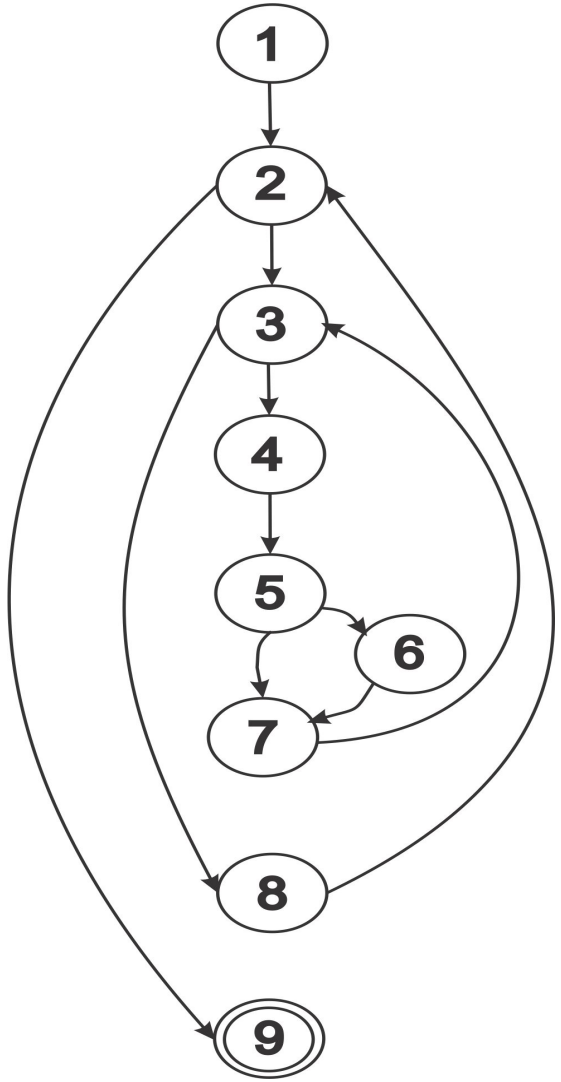
- Propriedades que devem ser avaliadas no teste

1. Baseados em **complexidade**
2. Baseados em **fluxo de controle**
3. Baseados em **fluxo de dados**

- Todo critério de teste é composto por um conjunto requisitos de teste
- Caminhos, laços de repetição, definição e uso de variáveis

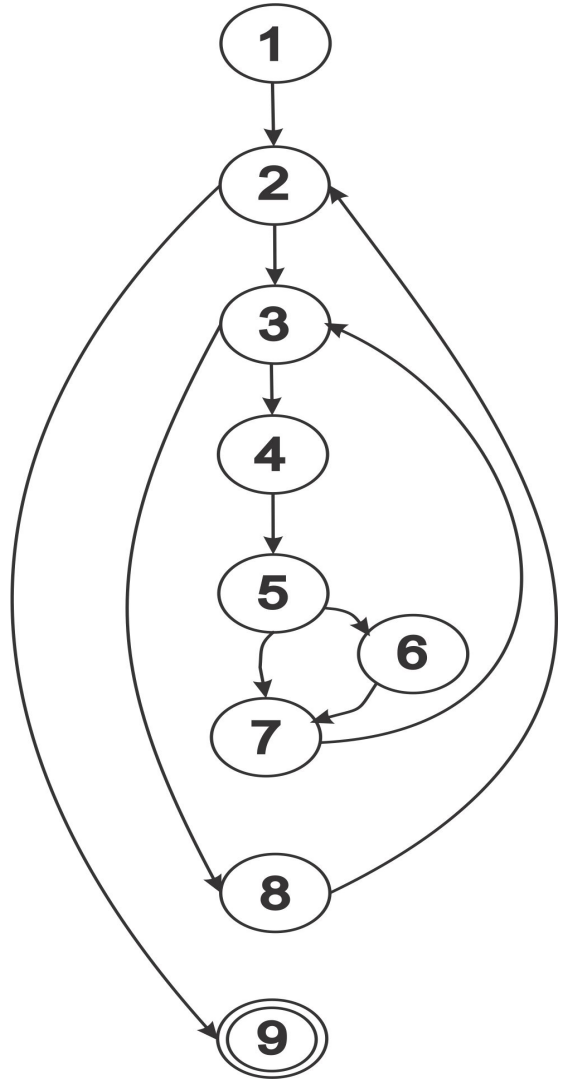
Abstração do código Grafo de Fluxo de Controle

- O comportamento do código fonte de um programa pode ser representado por Grafo de Fluxo de Controle
  - Um nó corresponde a uma instrução
  - As arestas denotam o potencial fluxo de controle entre as instruções



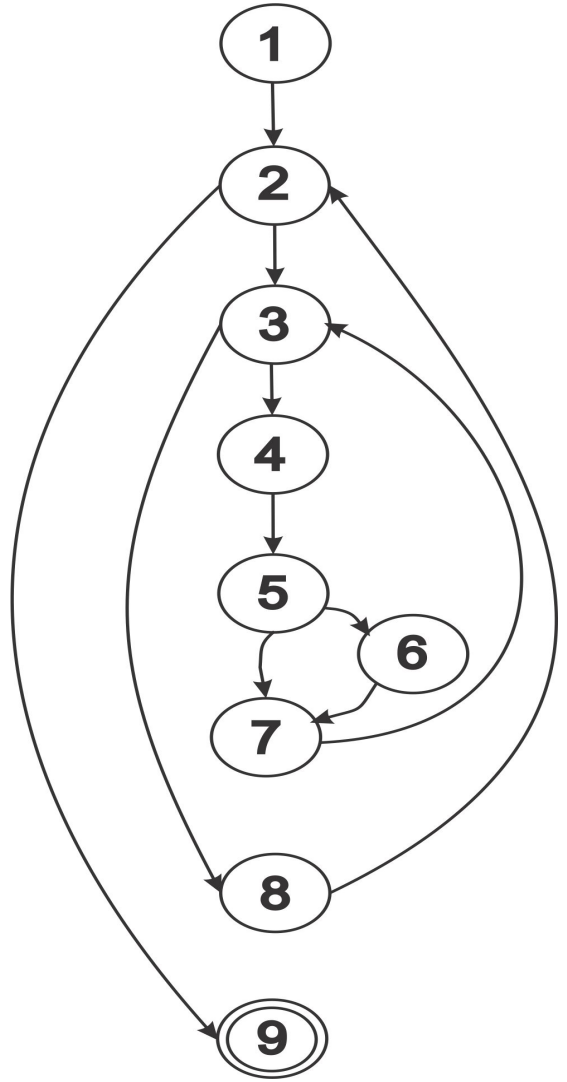
FLUXO DE CONTROLE

- É a sequência de passos que o computador segue para executar as operações do programa
  - Sequência
  - Condicionais
  - Estruturas de repetição



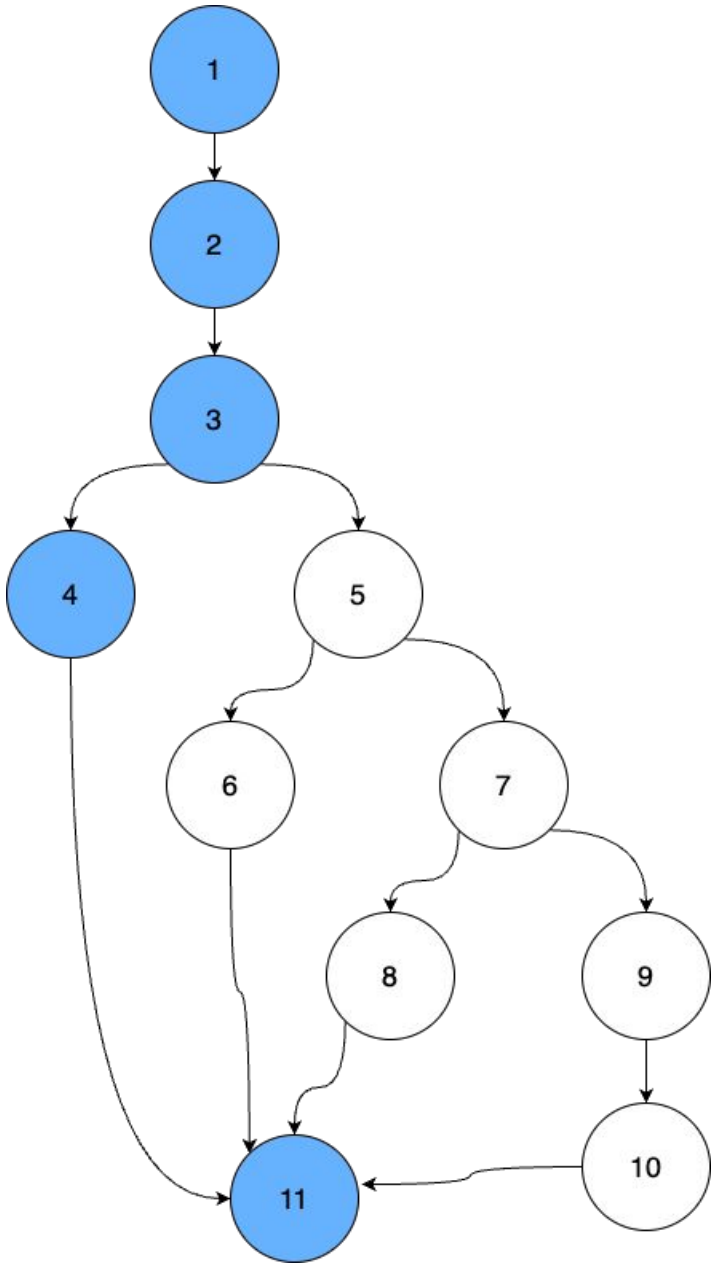
FLUXO DE DADOS

- O fluxo de dados descreve como os dados são lidos, processados e transmitidos



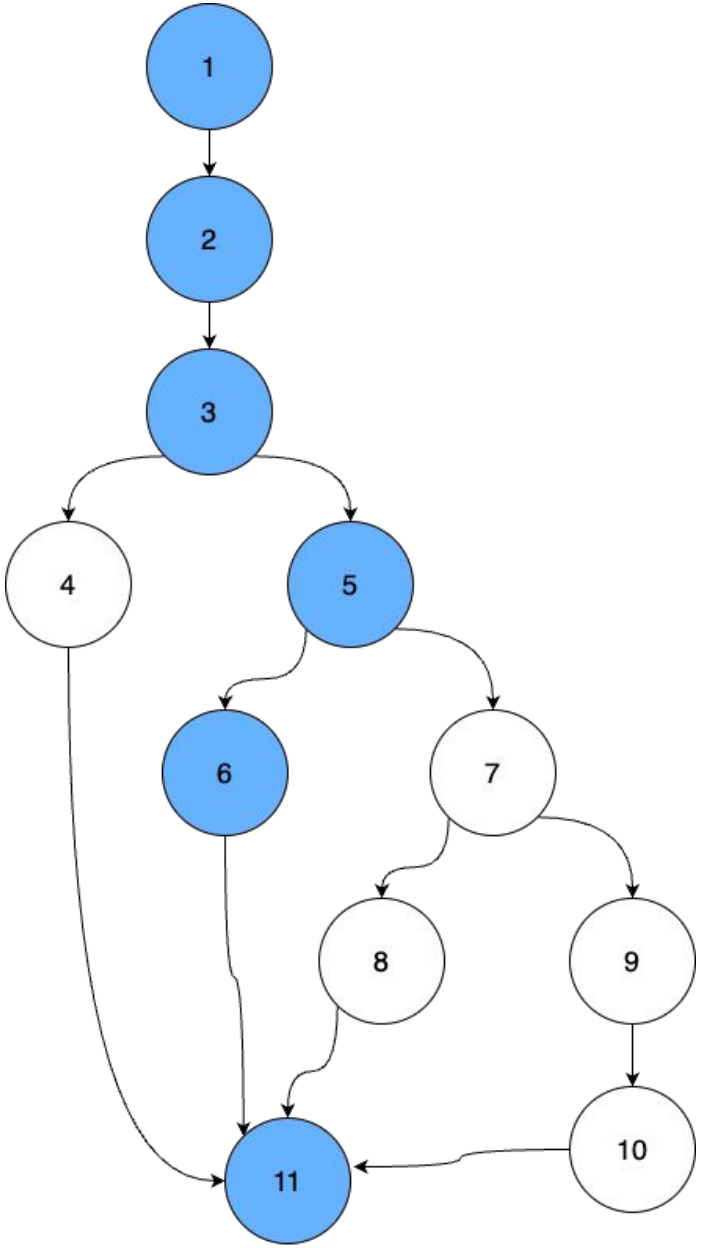
```
1. public class PnE {
2.   public static void calcularValor(int valor) {
3.       if (valor < 0) {
4.               System.out.println("Valor negativo");
5.       } else if (valor > 100) {
6.           System.out.println("Valor maior que 100");
7.       } else if (valor >= 0 && valor <= 100) {
8.           System.out.println("Valor entre 0 e 100");
9.       } else {
10.          System.out.println("Esta linha nunca será executada.");
11.      }
12.  }

13.      public static void main(String[] args) {
14.              calcularValor(-50);
15.      }
16. }
```



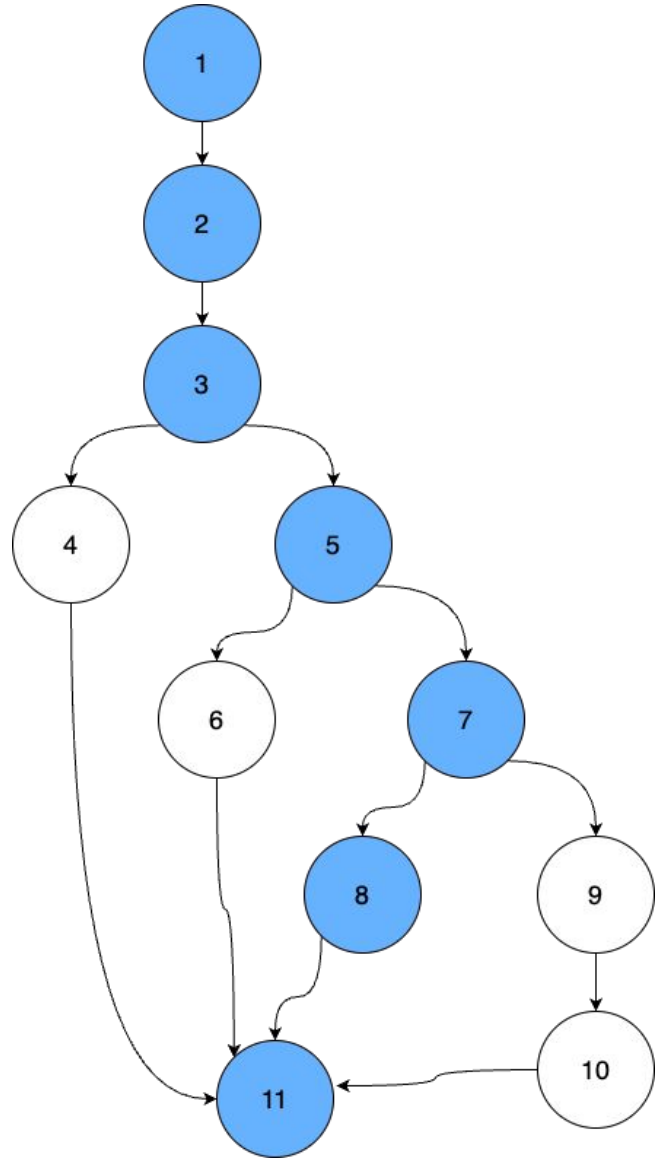
```
1. public class PnE {
2.   public static void calcularValor(int valor) {
3.       if (valor < 0) {
4.               System.out.println("Valor negativo");
5.       } else if (valor > 100) {
6.           System.out.println("Valor maior que 100");
7.       } else if (valor >= 0 && valor <= 100) {
8.           System.out.println("Valor entre 0 e 100");
9.       } else {
10.          System.out.println("Esta linha nunca será executada.");
11.      }
12.  }

13.      public static void main(String[] args) {
14.              calcularValor(101);
15.      }
16. }
```



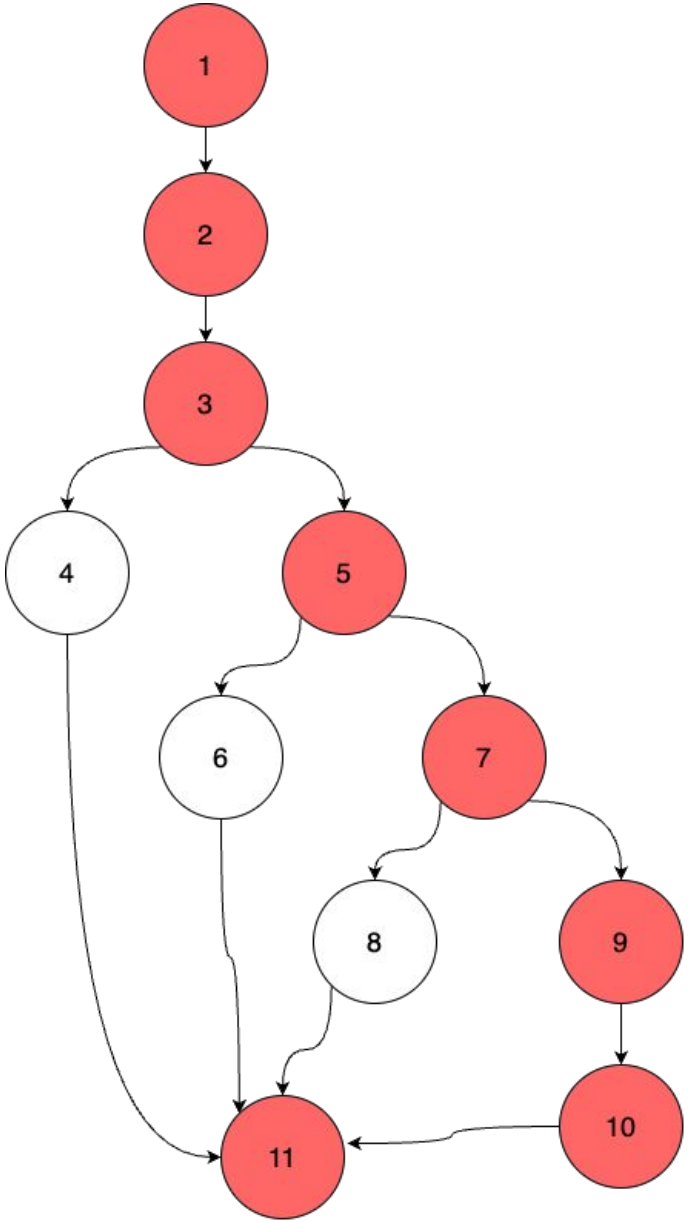
```
1. public class PnE {
2.   public static void calcularValor(int valor) {
3.       if (valor < 0) {
4.               System.out.println("Valor negativo");
5.       } else if (valor > 100) {
6.           System.out.println("Valor maior que 100");
7.       } else if (valor >= 0 && valor <= 100) {
8.           System.out.println("Valor entre 0 e 100");
9.       } else {
10.          System.out.println("Esta linha nunca será executada.");
11.      }
12.  }

13.      public static void main(String[] args) {
14.              calcularValor(75);
15.      }
16. }
```



```
1. public class PnE {
2.   public static void calcularValor(int valor) {
3.       if (valor < 0) {
4.               System.out.println("Valor negativo");
5.       } else if (valor > 100) {
6.           System.out.println("Valor maior que 100");
7.       } else if (valor >= 0 && valor <= 100) {
8.           System.out.println("Valor entre 0 e 100");
9.       } else {
10.          System.out.println("Esta linha nunca será executada.");
11.      }
12.   }

13.      public static void main(String[] args) {
14.              calcularValor(??);
15.      }
16. }
```

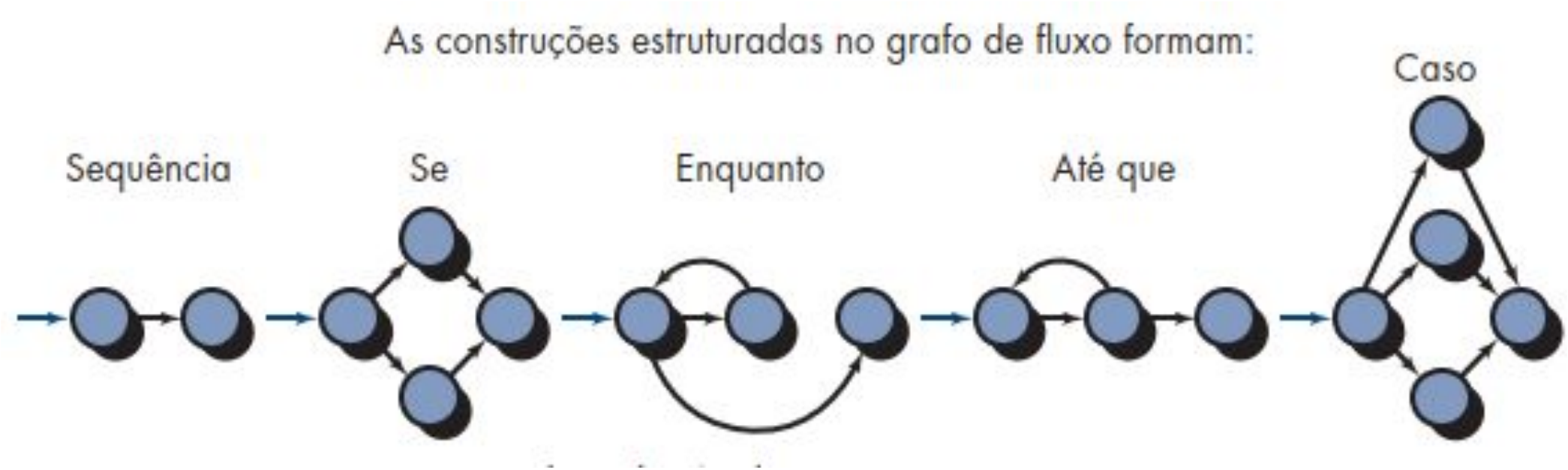


TESTE DE CAMINHO BÁSICO

- O teste de caminho básico é uma técnica de teste caixa-branca.
- Permite derivar uma medida da complexidade lógica de um projeto e usar essa medida como guia para definir um conjunto base de caminhos de execução.
- Casos de teste criados para exercitar o conjunto básico executam com certeza todas as instruções de um programa pelo menos uma vez durante o teste (Pressman, 2011).

CAMINHO BÁSICO

- A ideia por trás do caminho básico é identificar e testar caminhos que percorrem diferentes partes do código, como instruções, decisões condicionais e loops.
- Os caminhos básicos ajudam a garantir que todas as partes do código sejam testadas pelo menos uma vez.



TESTE DE CAMINHO BÁSICO

- O **grafo de fluxo** representa o fluxo de controle lógico.

Como encontrar caminhos básicos?

- Identificação de caminhos: identificar todos os caminhos possíveis de execução no código-fonte.
- Isso inclui caminhos que passam por instruções simples, estruturas de controle de fluxo, como condicionais (if/else) e loops (for/while), e qualquer outra estrutura de decisão.

Como encontrar caminhos básicos?

- Simplificação: eliminar caminhos redundantes ou irrelevantes.
- Desenvolvimento de casos de teste: Com os caminhos básicos identificados, são criados casos de teste que sigam esses caminhos. Cada caso de teste visa testar um caminho específico, fornecendo entradas e condições de teste apropriadas.
- Execução de testes: os casos de teste são executados no programa, e os resultados são avaliados

Quantos caminhos procurar?

- O cálculo da complexidade ciclomática fornece a resposta.

- Para calcular a complexidade ciclomática de McCabe, você pode usar a fórmula a seguir: **V(G) = E - N + 2**
- Onde:
  - V(G) é a complexidade ciclomática.
  - E é o número de arestas no grafo de fluxo de controle.
  - N é o número de nós no grafo de fluxo de controle.

ATIVIDADE

1. Interpretar o código e criar um grafo de fluxo de controle
2. Com base no grafo calcular a complexidade ciclomática
3. Com base no grafo definir quais são os caminhos possíveis

Entrega: Tirar foto e enviar no formulário de envio de atividades

```
public class ExemploCalculoComplexidade {

    public static void main(String[] args) {
        int a = 5;
        int b = 10;

        if (a > b) {
            System.out.println("a é maior que b");
        } else {
            System.out.println("a não é maior que b");
        }

        for (int i = 0; i < 3; i++) {
            System.out.println("Iteração " + (i + 1));
        }
    }
}
```

```
public class ExemploCalculoComplexidade2 {

    public static void main(String[] args) {
        int x = 5;
        int y = 10;
        int z = 0;

        if (x > y) {
            z = x + y;
        } else if (x < y) {
            for (int i = 0; i < 3; i++) {
                z += i;
            }
        } else {
            z = x * y;
        }

        System.out.println("O valor de z é: " + z);
    }
}
```

TESTE E QUALIDADE DE SOFTWARE

João Choma Neto

joao.choma@gmail.com

Unicesumar – Maringá – 2023/2