



Aula 7

Teste Estrutural (Caixa Branca)

Herysson R. Figueiredo
herysson.figueiredo@ufn.edu.br



Qualidade de Software

“Conformidade com requisitos funcionais e de desempenho, padrões de desenvolvimento documentados e características implícitas esperadas de todo software profissionalmente desenvolvido.”

- Corretude
- Confiabilidade
- Testabilidade



Garantia de Qualidade de Software

Conjunto de atividades técnicas aplicadas durante todo o processo de desenvolvimento

- Objetivo
 - Garantir que tanto o processo de desenvolvimento quanto o produto de software atinjam os níveis de qualidade especificados
 - **VV&T** – Verificação, Validação e Teste



Garantia de Qualidade de Software

- **Validação:** Assegurar que o produto final corresponda aos requisitos do usuário
 - Estamos construindo o produto certo?
- **Verificação:** Assegurar consistência, completitude e corretitude do produto em cada fase e entre fases consecutivas do ciclo de vida do software
 - Estamos construindo corretamente o produto?
- **Teste:** Examina o comportamento do produto por meio de sua execução



Terminologia

DEFEITO - ERRO - FALHA

- **Defeito** - deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha;
- **Erro** - item de informação ou estado de execução inconsistente
- **Falha** - evento notável em que o sistema viola suas especificações.



Defeitos no Processo de Desenvolvimento

- A maior parte é de origem humana
- São gerados na comunicação e na transformação de informações
- Continuam presentes nos diversos produtos de software produzidos e liberados (10 defeitos a cada 1000 linhas de código)
- A maioria encontra-se em partes do código raramente executadas



Defeitos no Processo de Desenvolvimento

- **Principal causa:** tradução incorreta de informações;
- Quanto antes a presença do defeito for revelada, menor o custo de correção do defeito e maior a probabilidade de corrigi-lo corretamente
- **Solução:** introduzir atividades de VV&T ao longo de todo o ciclo de desenvolvimento



Teste x Depuração

- **Teste**
 - Processo de execução de um programa como objetivo de revelar a presença de erros.
 - Contribuem para aumentar a confiança de que o software desempenha as funções especificadas.



Teste x Depuração

- **Depuração**
 - Consequência não previsível do teste. Após revelada a presença do erro, este deve ser encontrado e corrigido.



Teste de *software*

“Teste de software é o processo de executar programas com o objetivo de encontrar defeitos”

É uma atividade essencial para se garantir a qualidade do software

É uma das últimas atividades que fará a revisão do produto.

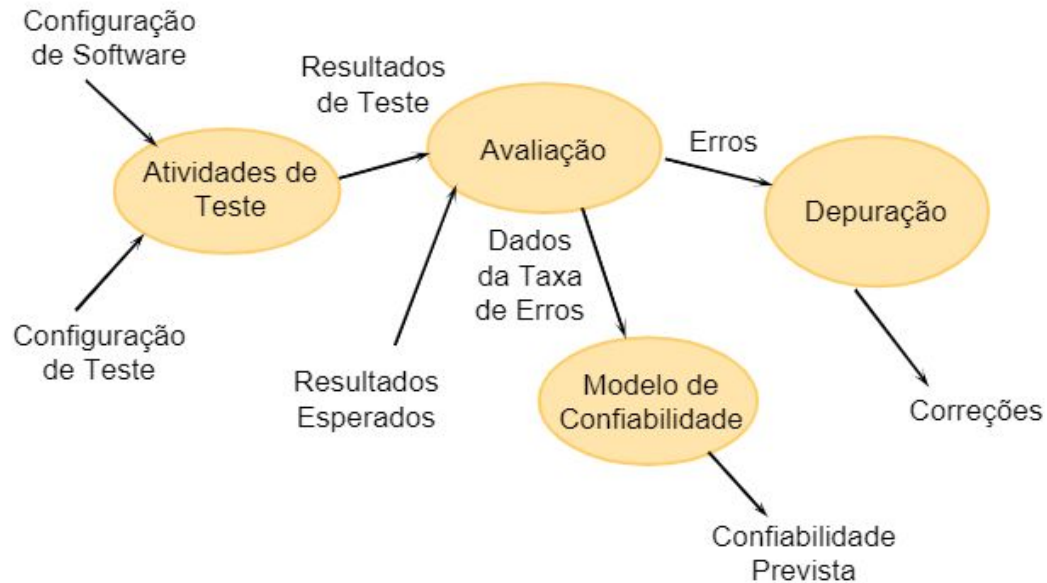


Teste de Software

Limitações

- Não existe um algoritmo de teste de propósito geral para provar a corretude de um programa;
- Em geral, é indecidível se dois caminhos de um mesmo programa ou de diferentes programas computam a mesma função;
- É indecidível se existe um dado de entrada que leve à execução de um dado caminho de um programa; isto é, é indecidível se um dado caminho é executável ou não

Teste de Software





Teste de Software

Fases de Teste

- Teste de Unidade
- Teste de Integração
- Teste de Sistema



Teste de Software

Fases de Teste

- **Teste de Unidade**
 - Identificar erros de lógica e de implementação em cada módulo do software, separadamente;
- **Teste de Integração**
- **Teste de Sistema**



Teste de Software

Fases de Teste

- Teste de Unidade
- Teste de Integração
 - Identificar erros associados às interfaces entre os módulos do software
- Teste de Sistema



Teste de Software

Fases de Teste

- Teste de Unidade
- Teste de Integração
- Teste de Sistema
 - Verificar se as funções estão de acordo com as especificações e se todos os elementos do sistema combinam-se adequadamente



Teste de Software

Etapas do Teste

- Planejamento
- Projeto de casos de teste
- Execução do programa com os casos de teste
- Análise de resultados



Caso de Teste

Especificação de uma entrada para o programa e a correspondente saída esperada

- Entrada: conjunto de dados necessários para uma execução do programa
- Saída esperada: resultado de uma execução do programa
- Oráculo

Um bom caso de teste tem alta probabilidade de revelar um erro ainda não descoberto



Projeto de caso de teste

- O projeto de casos de teste pode ser tão difícil quanto o projeto do próprio produto a ser testado
- Poucos programadores/analistas gostam de teste e, menos ainda, do projeto de casos de teste
- O projeto de casos de teste é um dos melhores mecanismos para a prevenção de defeitos
- O projeto de casos de teste é tão eficaz em identificar erros quanto a execução dos casos de teste projetados



Cenário de Testes

Cenários de testes ou suítes de teste, de acordo com a norma IEEE 829-2008 (IEEE 829-2008, 2008), cenário de teste é um conjunto de casos de teste relacionados, que comumente testam o mesmo componente ou a mesma funcionalidade do sistema.



Técnicas e Critérios de Teste

Técnica Funcional - Caixa-Preta

- Particionamento de equivalência
- Análise de valor limite
- Grafo Causa-Efeito
- Teste combinatorial
- Error Guessing

TÉCNICA DE TESTE CAIXA-BRANCA



Teste Estrutural - Caixa-Branca

- Baseado na estrutura do objeto de teste (classes, funções, módulos...)
- Podem ser usadas em todos os níveis de testes, mas as técnicas que estudaremos agora são mais comuns de serem usadas no teste de componentes ou integração entre componentes.
- Mediação via cobertura de teste
- Geralmente feito pelo Desenvolvedor



Casos de Teste

Os casos de teste no teste estrutural devem:

- Garantir que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez
- Exercitem todas as decisões lógicas em seus lados verdadeiro e falso
- Executem todos os ciclos nos seus limites e dentro de seus intervalos operacionais
- Exercitem as estruturas de dados internas

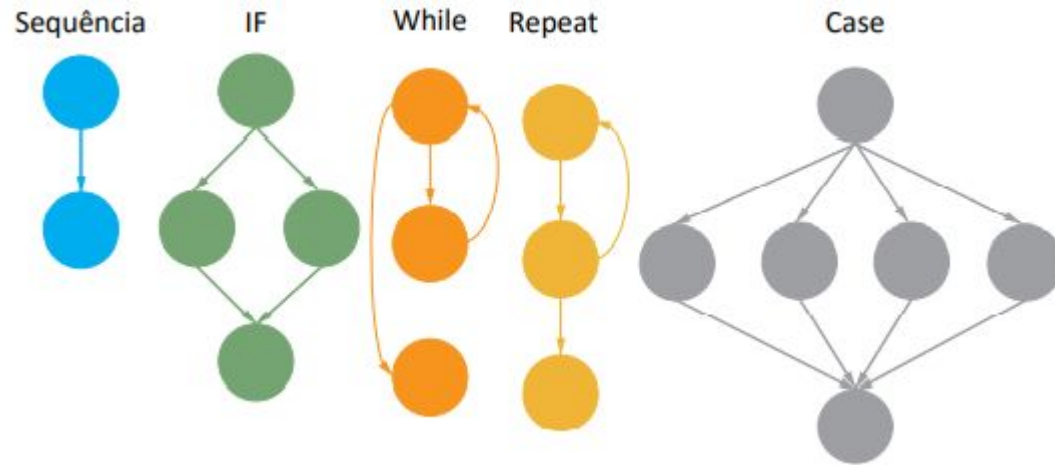


Notação Grafo de Fluxo

O grafo de fluxo mostra o fluxo de controle

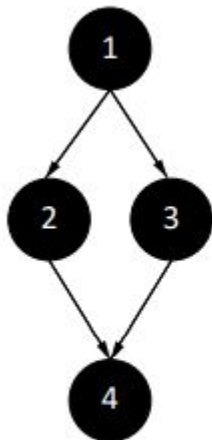
- Nós representam um ou mais processos
- Arestas representam o fluxo de controle
- Regiões do grafo são áreas limitadas pelas arestas e nós (incluindo a área fora do grafo)

Notação Grafo de Fluxo



Notação Grafo de Fluxo

Grafo de Fluxo de Controle



Detalhamento da representação de cada nó

1	1	inteiro calculo hextraPercentual(real horasTrabalhadas) {
	2	inteiro horasMes, horasExtras;
	3	horasMes = 40;
	4	SE (horasExtras > 40){

2	5	horasExtras = horasTrabalhadas - 40;
	6	return (horasExtras *100)/horasMes;

3	7	SENÃO
	8	return 0;

4	9	}
	10	}

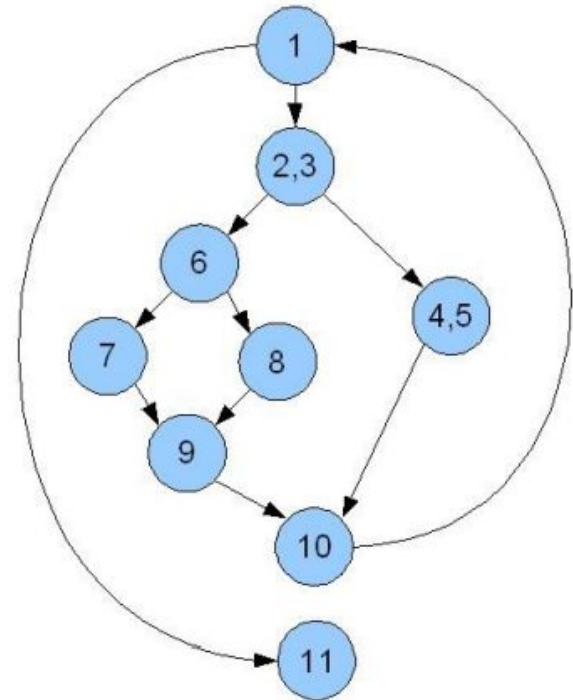


Derivando o grafo de fluxo a partir de PDL

```
1: enquanto existir registro faça  
2:   leia registro  
3:   se registro.campo1 = 0 então  
4:     processar registro e armazenar em buffer  
5:     incrementar contador  
6:   senão se registro.campo2 = 0 então  
7:     resetar contador  
8:   senão  
9:     processar registro e armazenar em arquivo  
10:  fimse  
11: fimenquanto
```

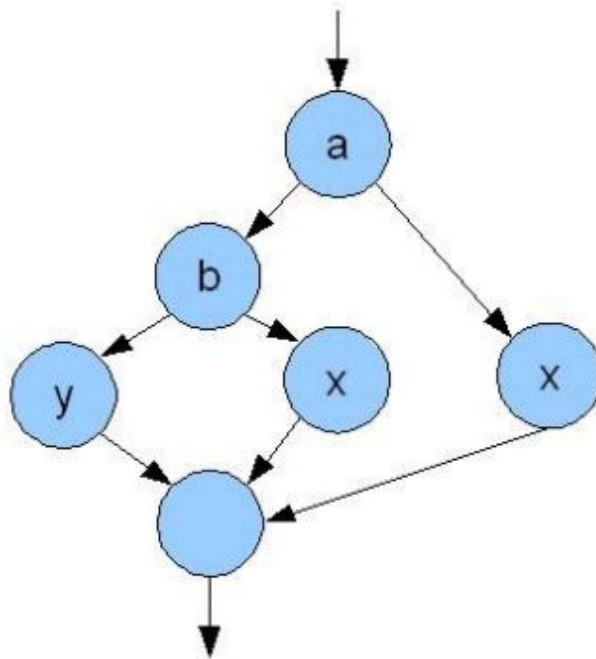
Derivando o grafo de fluxo a partir de PDL

```
1: enquanto existir registro faça  
2:   leia registro  
3:   se registro.campo1 = 0 então  
4:     processar registro e armazenar em buffer  
5:     incrementar contador  
6:   senão se registro.campo2 = 0 então  
7:     resetar contador  
8:   senão  
9:     processar registro e armazenar em arquivo  
10:  fimse  
11: fimenquanto
```



Notação Grafo de Fluxo

...
se a ou b então
 procedimento
x
senão
 procedimento
y
fimse
...



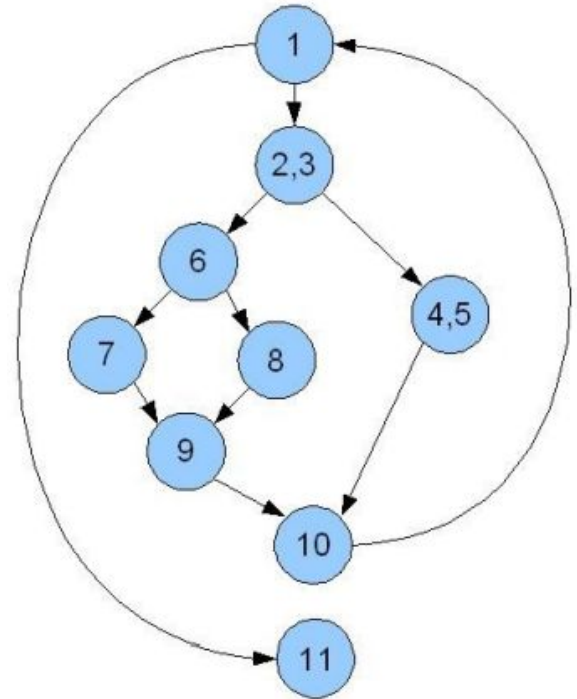
Caminhos Independentes

Caminhos independentes:

1. 1-11
2. 1-2-3-4-5-10-1-11
3. 1-2-3-6-8-9-10-1-11
4. 1-2-3-6-7-9-10-1-11

O caminho:

1-2-3-4-5-10-1-2-3-6-8-6-10-1-11 **NÃO** é independente





Prática

```
public static int buscaBinaria(int[] array, int valor) {  
    int inicio = 0;  
    int fim = array.length - 1;  
    int retorno = -1;  
    while (inicio <= fim) {  
        int meio = (inicio + fim) / 2;  
        if (array[meio] == valor) {  
            retorno = meio;  
            fim = -1;  
        } else if (valor > array[meio]) {  
            inicio = meio + 1;  
        } else {  
            fim = meio - 1;  
        }  
    }  
    return retorno;  
}
```




Técnicas estruturais

No contexto de teste de software, as **técnicas estruturais** são métodos de teste que se concentram na análise da estrutura interna do código-fonte do software.



Técnicas estruturais

as três técnicas estruturais principais são:

- Critério baseado na complexidade
- Critérios baseados em fluxo de controle
- Critérios baseados em fluxo de dados



Critério baseado na complexidade

Este critério refere-se a uma técnica de teste de software que se concentra na complexidade do código-fonte para determinar quais partes do programa precisam ser testadas de forma mais abrangente.



Critério baseado na complexidade

A ideia por trás desse critério é que partes do código mais complexas têm maior probabilidade de conter erros, e, portanto, devem ser submetidas a testes mais rigorosos.



Critério baseado na complexidade

Um dos métodos mais comuns para avaliar a complexidade do código-fonte é o uso de métricas de complexidade, como o índice de complexidade ciclomática (também conhecido como "*Complexity Cyclomatic Number*" ou "*Cyclomatic Complexity*").



Critério baseado na complexidade

Um dos métodos mais comuns para avaliar a complexidade do código-fonte é o uso de métricas de complexidade, como o **índice de complexidade ciclomática** (também conhecido como "*Complexity Cyclomatic Number*" ou "*Cyclomatic Complexity*").



Critério baseado na complexidade

A complexidade ciclomática é uma medida numérica que representa o **número de caminhos independentes** através de um programa. **Quanto maior** a complexidade ciclomática, **maior é a complexidade** do código.



Critério baseado na complexidade

A complexidade ciclomática é calculada com base em estruturas de controle no código, como loops, condicionais e chamadas de função. Ela ajuda a determinar a quantidade mínima de testes necessários para garantir a cobertura completa do código.



Critério baseado na complexidade

A **complexidade ciclomática** é calculada usando a fórmula de McCabe, que é baseada nas estruturas de controle presentes no código-fonte. A fórmula leva em consideração o número de nós de decisão e o número de arestas do grafo de controle do programa.



Complexidade Ciclomática

Formula

$V(G) = R$ - onde R é o número de regiões do grafo de fluxo.

$V(G) = E - N + 2$ - onde E é o número de arestas (setas) e N é o número de nós do grafo G.

$V(G) = P + 1$ - onde P é o número de nós-predicados contidos no grafo G (só funciona se os nós-predicado tiverem no máximo duas arestas saindo.)

Procedimento média(valor[])

i=1;

soma=0;

total.entrada=0;

total.válidas=0

Faca-Enquanto (valor[i]≠-999 **E** total.entrada<100)

 incremente total.entrada de 1;

Se (valor[i]≥min **E** valor[i]≤max)

Então

 incremente total.válidas de 1;
 soma = soma + valor[i]

Fim-Se

 incremente i de 1;

Fim-Enquanto

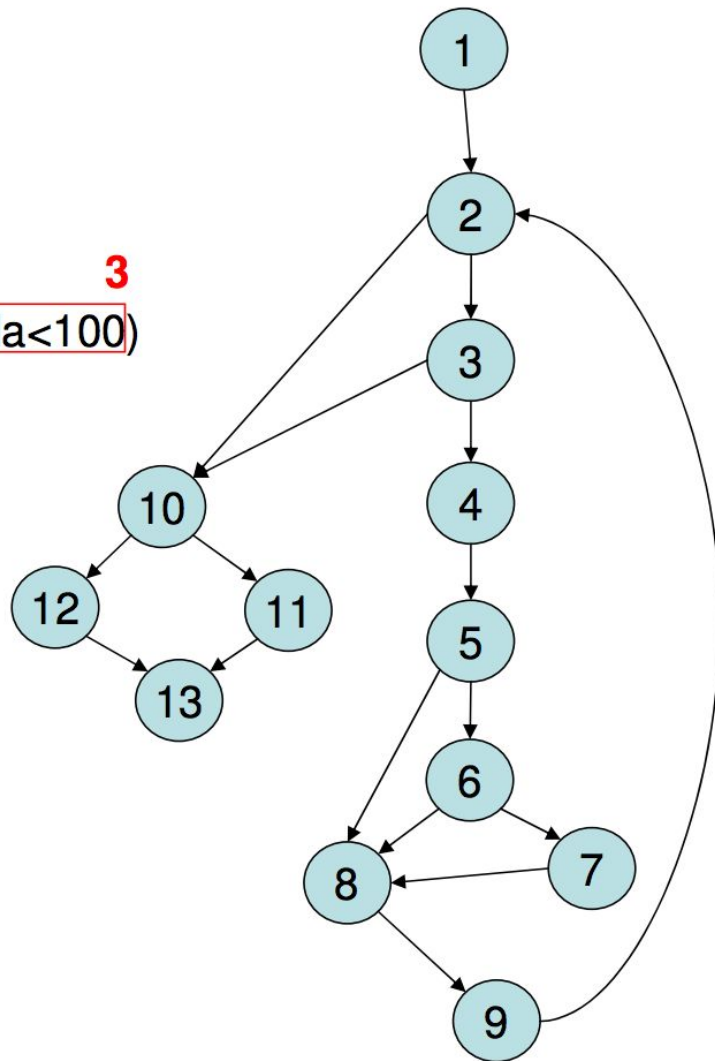
Se total.válidas>0

Então média = soma/total.válidas;

Senão média = -999;

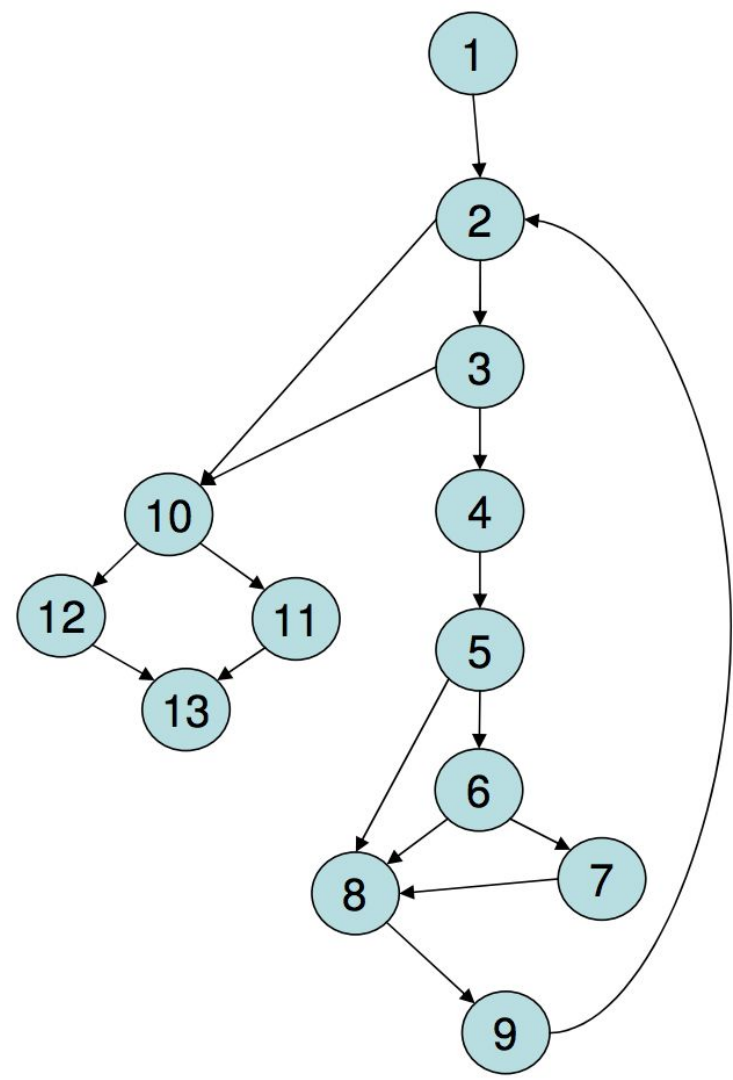
Fim-Se

Fim média



Complexidade Ciclomática

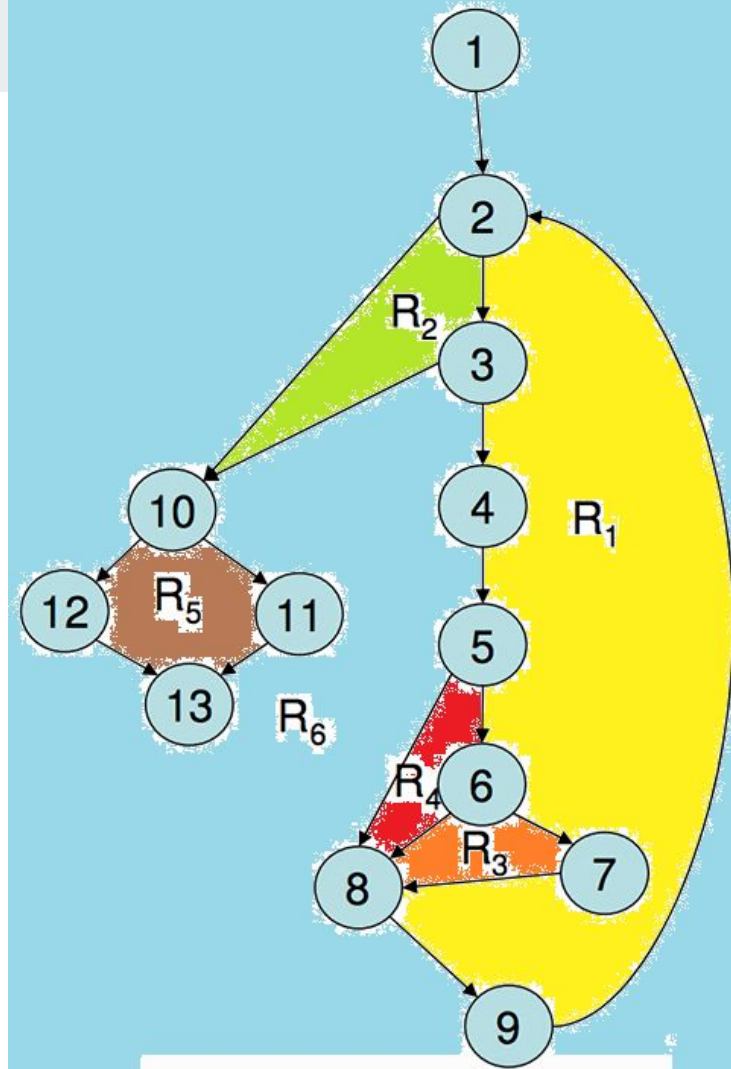
$$V(G) = R$$



Complexidade Ciclomática

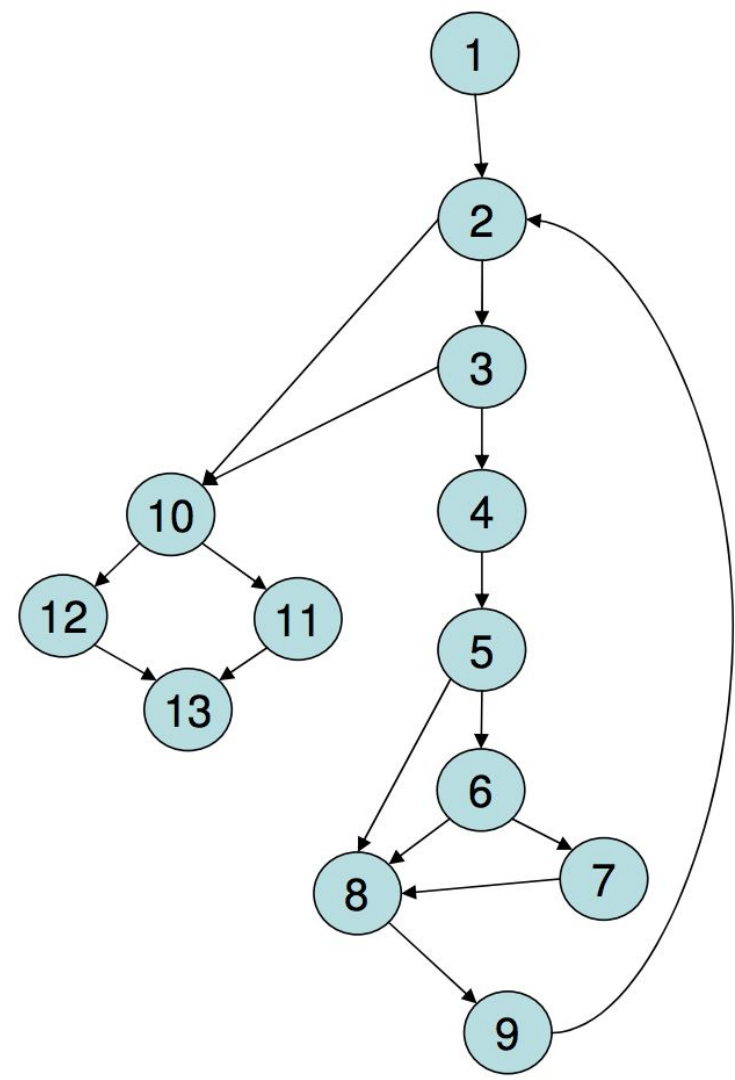
$$V(G) = R$$

Temos 6 regiões.



Complexidade Ciclomática

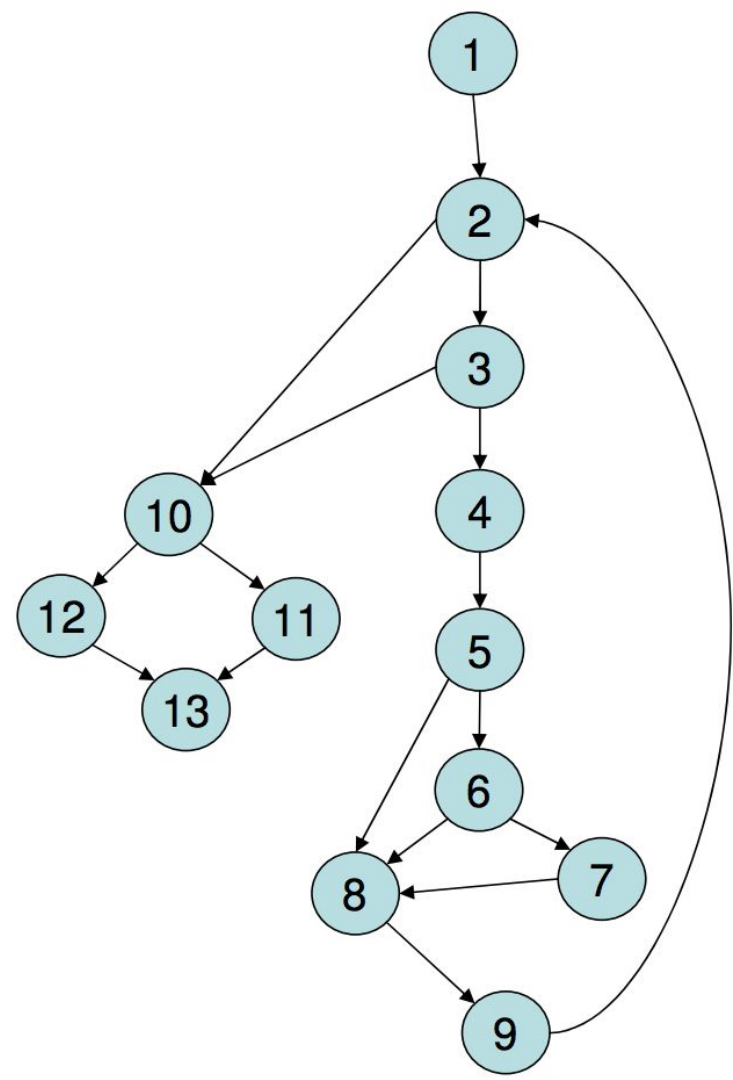
$V(G) = E - N + 2$ - onde E é o número de arestas (setas) e N é o número de nós do grafo G.



Complexidade Ciclomática

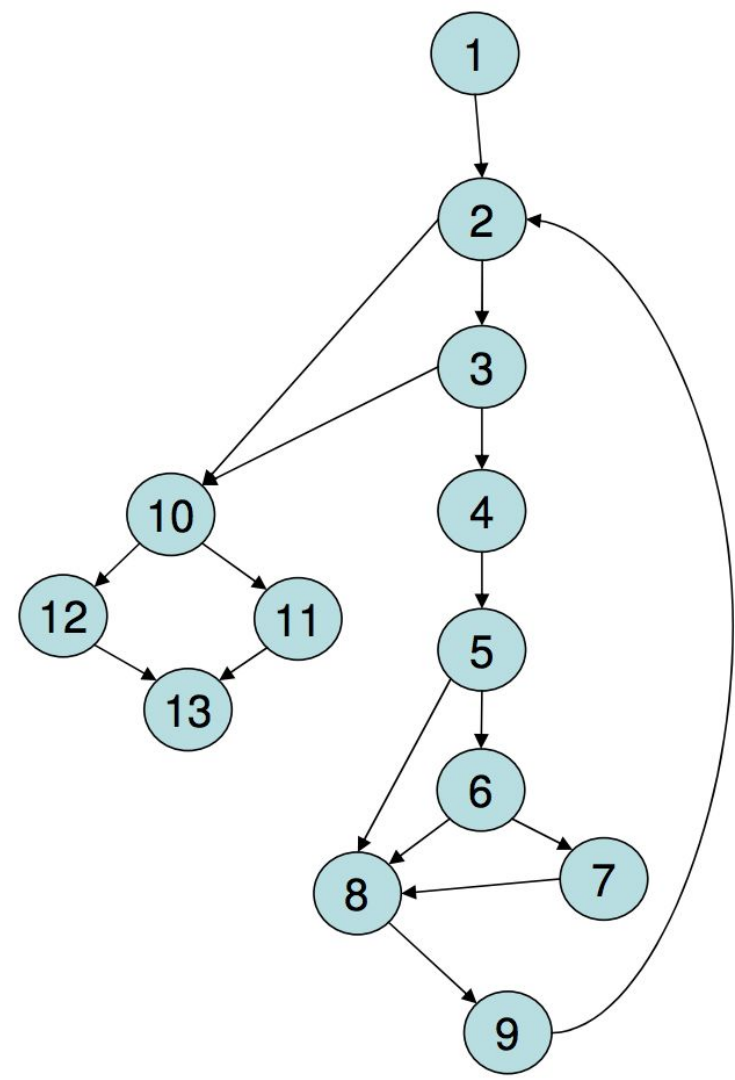
$V(G) = E - N + 2$ - onde E é o número de arestas (setas) e N é o número de nós do grafo G.

$$V(G) = 17 \text{ arestas/setas} - 13 \text{ nós} + 2 = 6$$



Complexidade Ciclomática

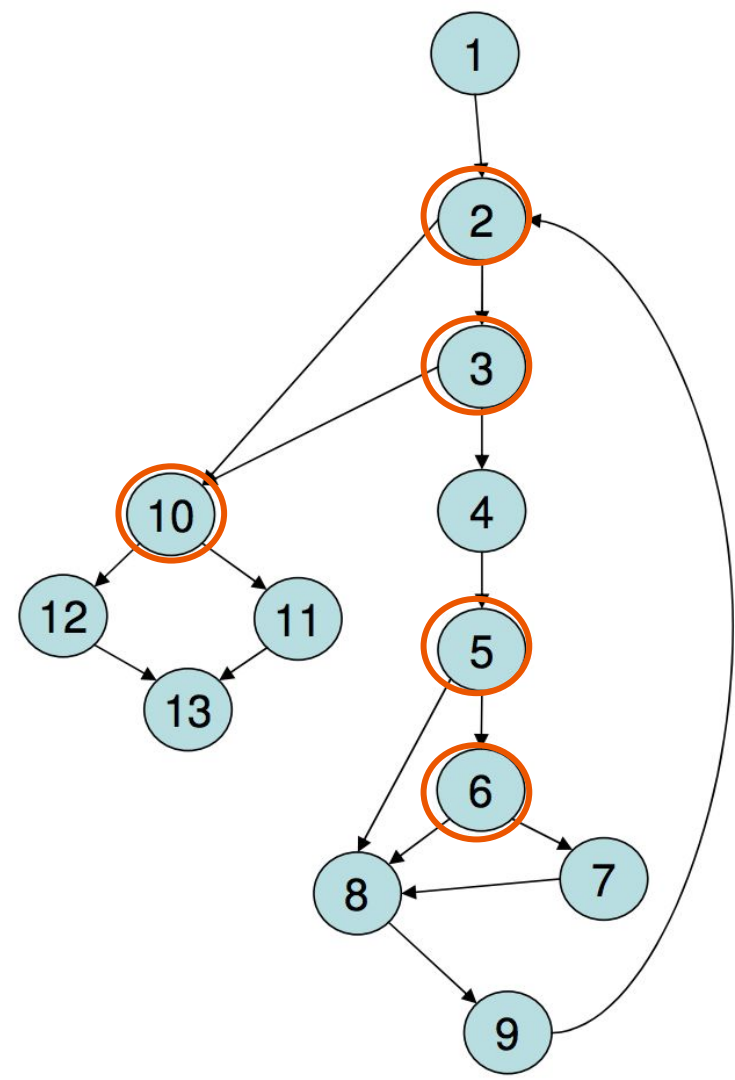
$V(G) = P + 1$ - onde P é o número de nós-predicados contidos no grafo G .



Complexidade Ciclomática

$V(G) = P + 1$ - onde P é o número de nós-predicados contidos no grafo G .

$V(G) = 5 \text{ nós-predicados} + 1 = 6$





Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle se concentram na lógica do programa, ou seja, **como o controle do programa é direcionado** através das instruções.



Critérios baseados em fluxo de controle

Fluxos de controle se referem à ordem em que as instruções de um programa de computador são executadas. Em outras palavras, fluxos de controle descrevem **como o programa "flui"** de uma parte para outra, decidindo quais **caminhos seguir** com base em condições e decisões.



Critérios baseados em fluxo de controle

Existem várias estruturas de controle que afetam o fluxo de controle em um programa, incluindo:

- `if`, `"else if"` e `"else"`
- `"for"`, `"while"` e `"do-while"`
- Chamadas de função
- Saltos e desvios



Cr terios baseados em fluxo de controle

Alguns cr terios baseados em fluxo de controle incluem:

- Cobertura de declara  o (Statement Coverage);
- Cobertura de decis  o (Decision Coverage);
- Cobertura de caminho (Path Coverage);
- Cobertura de fluxo de controle (Control Flow Coverage);



Cobertura de declaração (*Statement Coverage*)

Essa técnica envolve garantir que cada declaração no código-fonte seja executada pelo menos uma vez durante a execução dos testes. Isso ajuda a verificar a cobertura básica do código.



Cobertura de declaração (*Statement Coverage*)

```
void fun(int x) {  
    if (x > 0) {  
        cout << "Valor positivo" << endl;  
    } else {  
        cout << "Valor não-positivo" << endl;  
    }  
}
```

Para atingir 100% de cobertura de declaração, você deve garantir que ambas as declarações de saída ("cout") dentro de cada ramificação do condicional sejam executadas em seus testes.



Cobertura de decisão (*Decision Coverage*)

Nesse caso, o objetivo é garantir que todas as decisões (geralmente associadas a estruturas de controle como condicionais "if" e "switch") sejam testadas em todas as suas ramificações. Isso ajuda a identificar possíveis erros lógicos no código.



Cobertura de decisão (*Decision Coverage*)

```
void fun(int x) {  
    if (x > 0) {  
        cout << "Valor positivo" << endl;  
    } else {  
        cout << "Valor não-positivo" << endl;  
    }  
}
```

Exemplo: Continuando com o código anterior, para atingir 100% de cobertura de decisão, você deve garantir que todos os resultados possíveis da condição "x > 0" sejam testados. Isso significa que você precisa criar testes para ambas as ramificações, com um teste para "x" sendo maior que zero e outro teste para "x" não sendo maior que zero.



Cobertura de caminho (*Path Coverage*)

Essa técnica busca testar todos os caminhos possíveis através do código, considerando todas as combinações de decisões e loops. Isso é mais abrangente do que a cobertura de decisão, mas também mais trabalhoso de implementar.



Cobertura de decisão (*Decision Coverage*)

Para atingir 100% de cobertura de caminho, você precisaria criar testes para todas as combinações possíveis de "a", "b" e "c", cobrindo todos os caminhos possíveis, como $(a > 0)$, $(b > 0)$, $(c > 0)$, e assim por diante.

Quais são estas combinações?

```
int calculate(int a, int b, int c) {  
    int result = 0;  
    if (a > 0) {  
        result = b + c;  
    } else if (b > 0) {  
        result = a + c;  
    } else {  
        result = a + b;  
    }  
    return result;  
}
```



Cobertura de fluxo de controle (*Control Flow Coverage*)

Garante que diferentes fluxos de controle sejam testados, incluindo execuções alternativas e excepcionais. Isso é particularmente útil para identificar problemas relacionados a exceções e manipulação de erros.



Cobertura de decisão (*Decision Coverage*)

Para atingir 100% de cobertura de fluxo de controle, você precisaria criar testes que cubram todas as partes do código, incluindo os blocos try, except e else, a fim de garantir que todos os fluxos de controle sejam testados adequadamente. Isso incluiria testar uma divisão bem-sucedida, uma divisão por zero e uma exceção de valor inválido.

```
def divide(a, b):  
    try:  
        result = a / b  
    except ZeroDivisionError:  
        result = "Divisão por zero"  
    except ValueError:  
        result = "Valor inválido"  
    else:  
        result = "Divisão bem-sucedida"  
    return result
```



Cr terios baseados em fluxo de dados

Os cr terios baseados em fluxo de dados concentram-se na an lise de como os dados fluem atrav s do programa e como as vari veis s o usadas e modificadas.



Cobertura de uso de variável (*Variable Usage Coverage*)

Envolve a verificação de todas as variáveis para garantir que elas sejam inicializadas, usadas e modificadas corretamente ao longo do programa.



Cobertura de uso de variável (*Variable Usage Coverage*)

Para atingir 100% de cobertura de uso de variável, você deve criar testes que abranjam todas as operações que envolvem a variável "saldo", garantindo que ela seja usada corretamente em todas as partes do código.

```
def calcular_saldo(inicial, transacao):  
    saldo = inicial  
    saldo += transacao  
    return saldo
```




Cobertura de definição-uso (*Def-use Coverage*)

Nesse caso, o objetivo é garantir que cada definição (atribuição de valor a uma variável) seja seguida por um uso (leitura) da variável. Isso ajuda a identificar possíveis problemas de lógica de programação.



Cobertura de definição-uso (*Def-use Coverage*)

Para atingir 100% de cobertura de definição-uso, você deve criar testes que garantam que a variável "maximo" seja definida antes de ser usada em todas as partes do código, incluindo o laço "for".

```
int encontrar_maximo(int array[], int inicio, int fim) {  
    int maximo = array[inicio];  
    for (int i = inicio + 1; i < fim; i++) {  
        if (array[i] > maximo) {  
            maximo = array[i];  
        }  
    }  
    return maximo;  
}
```



Cobertura de uso-define (*Use-Def Coverage*)

Ao contrário da técnica anterior, aqui o foco é garantir que cada uso de variável seja precedido por uma definição em algum lugar do programa. Isso ajuda a evitar o uso de variáveis não inicializadas.



Cobertura de uso-define (*Use-Def Coverage*)

Suponha que você tenha uma função em Java que lê um arquivo e calcula a soma dos números dentro dele.

```
int calcular_soma_arquivo(String nomeArquivo) {  
    int soma = 0;  
    try {  
        BufferedReader br = new BufferedReader(new FileReader(nomeArquivo))  
        String linha;  
        while ((linha = br.readLine()) != null) {  
            int numero = Integer.parseInt(linha);  
            soma += numero;  
        }  
        br.close();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return soma;  
}
```



Cobertura de fluxo de dados (*Data Flow Coverage*):

Essa técnica analisa como os dados fluem através das estruturas do programa, identificando possíveis problemas de dependências de dados e conflitos.



Cobertura de definição-uso (*Def-use Coverage*)

Para atingir 100% de cobertura de fluxo de dados, você deve criar testes que cubram todos os cenários, incluindo a entrada de números pares e ímpares, garantindo que todos os possíveis fluxos de dados sejam testados.

```
int contar_pares(int lista[], int tamanho) {  
    int contador = 0;  
    for (int i = 0; i < tamanho; i++) {  
        if (lista[i] % 2 == 0) {  
            contador++;  
        }  
    }  
    return contador;  
}
```



Atividade

Pesquisar um dos critérios de teste estrutural abaixo:

1. Critério baseado na complexidade
2. Critério baseado em fluxo de controle
3. Critério baseado em fluxo de dados
 - a. Rapps e Weyuker
 - b. Potenciais-Usos

Atividade

```
1 #include <iostream>
2
3 void processNumber(int number) {
4     if (number % 2 == 0) {
5         std::cout << "The number is even." << std::endl;
6     } else {
7         std::cout << "The number is odd." << std::endl;
8     }
9
10    int remainder = number % 4;
11    switch (remainder) {
12        case 0:
13            std::cout << "The remainder is 0." << std::endl;
14            break;
15        case 1:
16            std::cout << "The remainder is 1." << std::endl;
17            break;
18        case 2:
19            std::cout << "The remainder is 2." << std::endl;
20            break;
21        default:
22            std::cout << "The remainder is not 0, 1, or 2." << std::endl;
23    }
24 }
25
26 int main() {
27     for (int i = 1; i <= 5; i++) {
28         std::cout << "Processing number " << i << ":" << std::endl;
29         processNumber(i);
30         std::cout << "-----" << std::endl;
31     }
32
33     return 0;
34 }
```


Atividade

```
1 double complexLogarithmicFunction(int n) {
2     double result = 0;
3     if (n <= 0) {
4         result = -1.0;
5     } else {
6         for (int i = 1; i <= n; i++) {
7             if (i % 2 == 0) {
8                 result -= std::log2(i) + std::sqrt(i);
9             } else {
10                 result += std::log2(i) - std::sqrt(i);
11             }
12
13             if (i > n / 2) {
14                 if (result > 0) {
15                     result *= 2.0;
16                 } else {
17                     result /= 2.0;
18                 }
19             }
20         }
21
22         if (result >= 0) {
23             for (int j = 1; j < n; j++) {
24                 result += std::pow(j, 3);
25             }
26         } else {
27             result = -result;
28         }
29     }
30     return result;
31 }
```



Referências

BRAGA, Pedro Henrique Cacique. Teste de Software. Pearson Education do Brasil. São Paulo. 2016. Disponível na Biblioteca Virtual. DELAMARO, Márcio;

MALDONADO, José Carlos, JINO, Mario. Introdução ao teste de software. Elsevier. Rio de Janeiro. 2007.

PRESSMAN, Roger S. Engenharia de software. 5. Ed. Rio de Janeiro: McGraw Hill, 2002.