INSTITUTO FEDERAL
Santa Catarina
Câmpus Gaspar

Teste de Software

Prof. Marcos Rodrigo momo, M.Sc.
marcos.momo@ifsc.edu.br

Gaspar, maio 2021.

Roteiro

- **Teste funcional**
- **Técnica estático**
- **Teste estrutural**

Roteiro

- Introdução ao teste estrutural (caixa branca)
- Técnicas de teste estrutural
  - Grafo de fluxo de controle
  - Teste e cobertura de declaração
- Critérios mais populares para o teste estrutural
  - Todos os nós
  - Todos os arcos
- Aplicabilidade e limitações
- Conclusões

Teste Estrutural

- Conforme discutido anteriormente, **técnicas e critérios** de teste fornecem ao projetista de *software* uma **abordagem sistemática** e **teoricamente fundamentada** para a condução da atividade de teste.
- Além disso, constituem um mecanismo que pode **auxiliar na garantia da qualidade dos testes** e na maior probabilidade em revelar defeitos no *software*.
- Várias técnicas podem ser adotadas para se conduzir e avaliar a qualidade da atividade de teste, sendo diferenciadas de acordo com a **origem das informações** utilizadas para estabelecer os requisitos de teste.

Imaginemos o seguinte cenário de teste
Teste funcional

- Programa recebe uma idade, entre 1 e 100 e deve responder se indivíduo é “maior” ou “menor de idade”.

- Baseado nessa especificação apenas, podemos elaborar alguns casos de teste:
- (1, menor), (17, menor) (18, maior), (19, maior), (100, maior)

Temos aqui um conjunto de limite de testes

Imaginemos o seguinte cenário de teste
Teste funcional

```
void maioridade(int idade) {
    if (idade == 1) printf("Menor");
    if (idade ==2) printf("Menor");
        .
        .
        .
    if (idade == 99) printf("Maior");
    if (idade == 100) printf("Maior");
}
```

O que acontece com o conjunto de testes?

- (1, menor), (17, menor) (18, maior), (19, maior), (100, maior)

Imaginemos o seguinte cenário de teste Teste funcional

O que acontece com o conjunto de testes?

- (1, menor), (17, menor) (18, maior), (19, maior), (100, maior)

- Ele irá executar uma parte do programa, vai deixar uma boa parte de comandos sem executar
- Isso não é nada bom, pois justamente na parte não testada pode haver erros

- Obviamente, esse é um cenário é bem irreal, mas serve para ilustrar qual é o propósitos de um teste estrutural
- Ou seja, teste estrutural, ao contrário do teste funcional, tem como objetivo exercitar os comandos do nosso programa.

Teste Estrutural – caixa branca

- O teste estrutural usa o código do programa para definir os requisitos de teste e deles derivar os casos de teste. O objetivo é observar as "estruturas" que compõem o programa e garantir que todas elas tenham sido executadas durante o teste.

Fonte: Dalamaro, 2020.

Teste Estrutural – caixa branca

- As técnicas de teste baseadas na estrutura, são utilizadas para explorar estruturas ou componentes do sistema nos mais diversos níveis.
- Estrutura de um sistema = tudo aquilo utilizado para a construção de um determinado *software* como, por exemplo:
  - estrutura do código
  - aspectos arquiteturais
  - estrutura de um menu
  - webpage.

Teste Estrutural

- Ao invés de verificar se um componente ou sistema funciona corretamente, conforme especificado, as técnicas de caixa branca irão focar na **garantia de que um determinado elemento dessas estruturas está executando conforme projetado**.
- Existem diversas técnicas para realização do teste caixa branca. Dentre elas:
  - Grafo de Fluxo de Controle;
  - Teste e Cobertura de Declaração.

Grafo de Fluxo de Controle

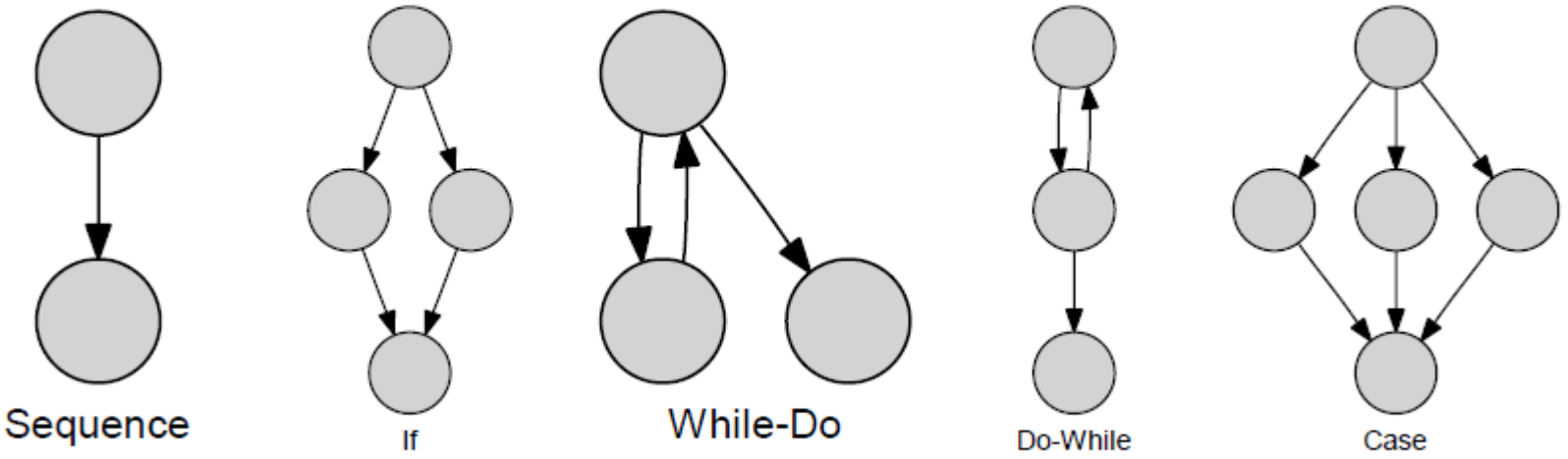
- O **Grafo de Fluxo de Controle** (GFC) ou **Grafo de Programa** fornece um mecanismo para representar os pontos de decisão e o fluxo de controle dentro de um código.
- É uma forma de abstrair as estruturas de um programa.
- São similares a um fluxograma, exceto pelo fato de que eles apresentam apenas as decisões.
- A elaboração do grafo se dá através da análise **exclusivamente** das declarações que afetam o fluxo de controle (if..then..else, for..do, do..while, repeat..until, switch case...).
- O grafo é elaborado utilizando-se dois símbolos:
  - **Nó ou vértice:** uma ou mais instruções que são executadas sempre em sequência. Uma vez executada a primeira instrução do nó, todas as outras serão executadas;
  - **Arco (também chamado de ramo ou de aresta):** são as linhas direcionadas que se conectam a quaisquer 2 nós e representam o fluxo de controle.

Grafo de Fluxo de Controle

- **Um programa pode ser decomposto em um conjunto de blocos disjuntos de comandos.**
- **Cada nó** representa um **bloco indivisível de comandos**.
- A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada.
- Os arcos indicam os possíveis fluxos de controle entre os blocos
- **Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando.**
- Então: um GFC é um grafo direcionado **(dígrafo)**, com um único **nó de entrada** e um **único nó de saída**, no qual cada vértice representa um bloco indivisível de comandos e cada arco representa um possível desvio de um bloco para outro.

Grafo de Fluxo de Controle

Para representar uma função ou um método como um GFC, as seguintes construções podem ser utililizadas:



Uso de GFC em testes

- Para usar grafos para projetar casos de teste deve-se proceder de acordo com os seguintes passos:

1. Defina o grafo
2. Percorra o grafo para atender a critérios selecionados (caminho de teste):

- caminho que começa em um nó inicial e termina em um nó final representam a execução de um caso de teste:

  - Alguns caminhos de teste podem ser executados por mais de um caso de teste;
  - Alguns caminhos de teste não podem ser executados por nenhum caso de teste.

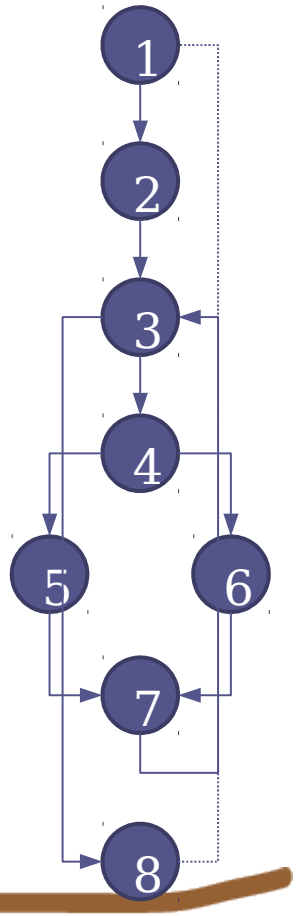
ou seja, ao ver um grafo, o testador deve ...

cobri-lo

Grafo de Fluxo de Controle

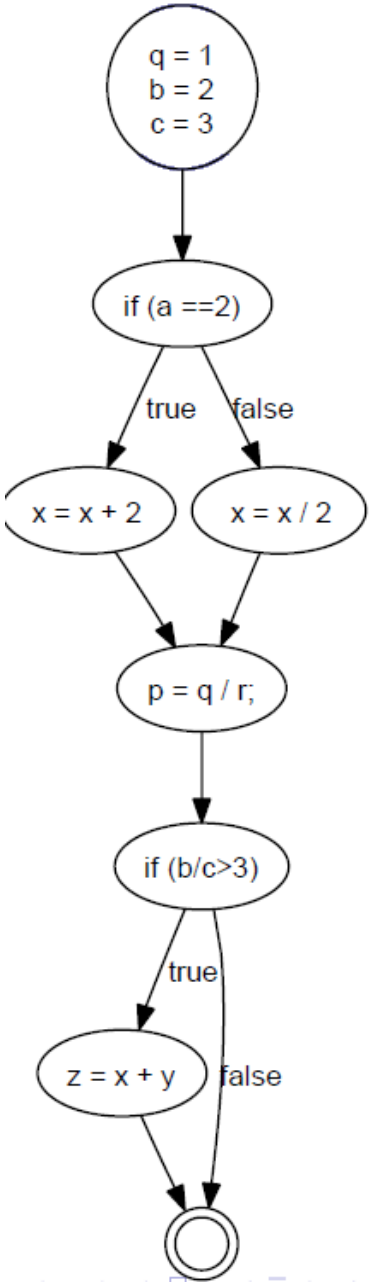
- Para desenhar um GFC, os passos a seguir devem ser executados:
  1. Fazer a análise do componente para identificar todas as estruturas de controle
  2. Adicionar um nó para cada declaração de decisão e expandir os nós representando a estrutura em um determinado ponto de decisão.

```
1.    a = Integer.parseInt(args[0]);
2.    b = Integer.parseInt(args[1]);
3.    while f1(a){
4.        if (f2(b)){
5.            a = b + 1;
6.        }else b = a + 1;
7.    }
8.    c = a + b;
```

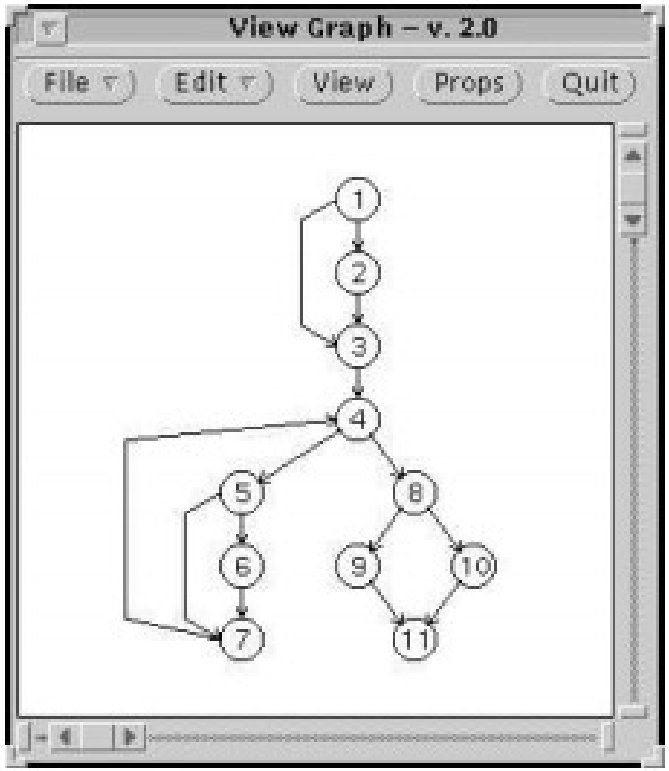


Exemplo

```
1   q = 1;
2   b = 2;
3   c = 3;
4   if (a ==2) {
5       x = x + 2;
6   } else {
7       x = x / 2;
8   }
9   p = q / r;
10  if (b/c>3) {
11      z = x + y;
12  }
```



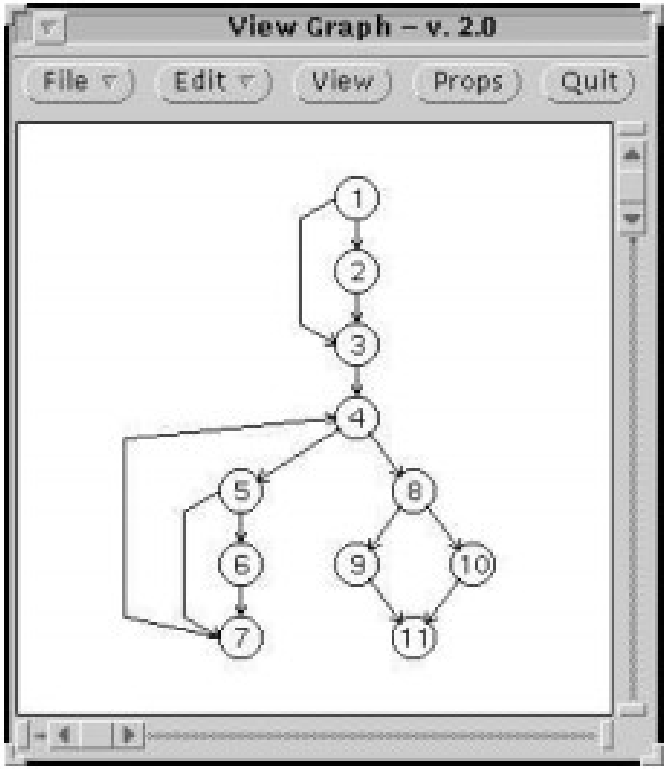
```
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em 'Silly
Pascal' (uma estranha variante do Pascal).  Um identificador valido deve comecar com uma
letra e conter apenas letras ou digitos.  Alem disso, deve ter no minimo 1 caractere e no
maximo 6 caracteres de comprimento
*****************************************************************************************/

            #include <stdio.h>
            main ()
/*  1 */    {
/*  1 */        char  achar;
/*  1 */        int length, valid_id;
/*  1 */        length = 0;
/*  1 */        valid_id = 1;
/*  1 */        printf ("Identificador: ");
/*  1 */        achar = fgetc (stdin);
/*  1 */        valid_id = valid_s(achar);
/*  1 */        if(valid_id)
/*  2 */        {
/*  2 */            length = 1;
/*  2 */        }
/*  3 */        achar = fgetc (stdin);
/*  4 */        while(achar != '\n')
/*  5 */        {
/*  5 */            if(!(valid_f(achar)))
/*  6 */            {
/*  6 */                valid_id = 0;
/*  6 */            }
/*  7 */            length++;
/*  7 */            achar = fgetc (stdin);
/*  7 */        }
/*  8 */        if(valid_id &&
                  (length >= 1) && (length <= 6))
/*  9 */        {
/*  9 */            printf ("Valido\n");
/*  9 */        }
/* 10 */        else
/* 10 */        {
/* 10 */            printf ("Invalid\n");
/* 10 */        }
/* 11 */    }
```




```
/* 2 */             return (1);
/* 2 */         }
/* 3 */         else
/* 3 */         {
/* 3 */             return (0);
/* 3 */         }
/* 4 */     }
```

O programa *Identifier.c*

- O primeiro bloco de comandos (nó) é caracterizado da linha 2 à linha 10. O segundo bloco é formado pelas linhas 11 a 13. O terceiro bloco refere-se à linha 14, e assim por diante.
- Ao todo, 11 nós constituem o GFC referente à função *main* do programa *identifier*, 4 nós constituem o grafo referente à função *valid_s* e 4 nós constituem o grafo da função *valid_f*.
- Na figura ao lado, é ilustrado o grafo obtido referente à função *main*, gerado pela ferramenta *ViewGraph* (ferramenta para visualização de grafos de fluxo de controle e informações de teste).



Definições úteis para os testes

- **Caminho**: sequência de arestas partindo de um nó origem a um nó destino
- **Caminho completo**: caminho de um nó de entrada a um nó de saída
- **Comprimento de um caminho**: número de nós ou número de arestas de um caminho
- **Ciclo (ou laço)**: um caminho em que um nó (ou aresta) é visitado mais de uma vez
- **Caminho livre de ciclos**: um caminho em que nenhum nó (ou aresta) é visitado mais de uma vez
- **Subcaminho**: subsequência de nós em um caminho

```
/*****************************************************************************
Identifier.c
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em 'Silly
Pascal' (uma estranha variante do Pascal).  Um identificador valido deve comecar com uma
letra e conter apenas letras ou digitos.  Alem disso, deve ter no minimo 1 caractere e no
maximo 6 caracteres de comprimento
*****************************************************************************/

                #include <stdio.h>
1               main ()
2  /*  1 */     {
3  /*  1 */         char  achar;
4  /*  1 */         int length, valid_id;
5  /*  1 */         length = 0;
6  /*  1 */         valid_id = 1;
7  /*  1 */         printf ("Identificador: ");
8  /*  1 */         achar = fgetc (stdin);
9  /*  1 */         valid_id = valid_s(achar);
10 /*  1 */         if(valid_id)
11 /*  2 */         {
12 /*  2 */             length = 1;
13 /*  2 */         }
14 /*  3 */         achar = fgetc (stdin);
15 /*  4 */         while(achar != '\n')
16 /*  5 */         {
17 /*  5 */             if(!(valid_f(achar)))
18 /*  6 */             {
19 /*  6 */                 valid_id = 0;
20 /*  6 */             }
21 /*  7 */             length++;
22 /*  7 */             achar = fgetc (stdin);
23 /*  7 */         }
24 /*  8 */         if(valid_id &&
                      (length >= 1) && (length < 6))
25 /*  9 */         {
26 /*  9 */             printf ("Valido\n");
27 /*  9 */         }
28 /* 10 */         else
29 /* 10 */         {
30 /* 10 */             printf ("Invalid\n");
31 /* 10 */         }
32 /* 11 */     }

                int valid_s(char ch)
/* 2 */         }
/* 3 */         else
/* 3 */         {
/* 3 */             return (0);
/* 3 */         }
/* 4 */     }

                int valid_f(char ch)
/* 1 */     {
/* 1 */         if(((ch >= 'A') &&
                    (ch <= 'Z')) ||
                   ((ch >= 'a') &&
                    (ch <= 'z')) ||
                   ((ch >= '0') &&
                    (ch <= '9')))
/* 2 */         {
/* 2 */             return (1);
/* 2 */         }
/* 3 */         else
/* 3 */         {
/* 3 */             return (0);
/* 3 */         }
/* 4 */     }
```

O caminho 2,3,4,5,6,7 é um caminho simples e livre de laços

```
/*****************************************************************************
Identifier.c
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em 'Silly
Pascal' (uma estranha variante do Pascal).  Um identificador valido deve comecar com uma
letra e conter apenas letras ou digitos.  Alem disso, deve ter no minimo 1 caractere e no
maximo 6 caracteres de comprimento
*****************************************************************************/
```

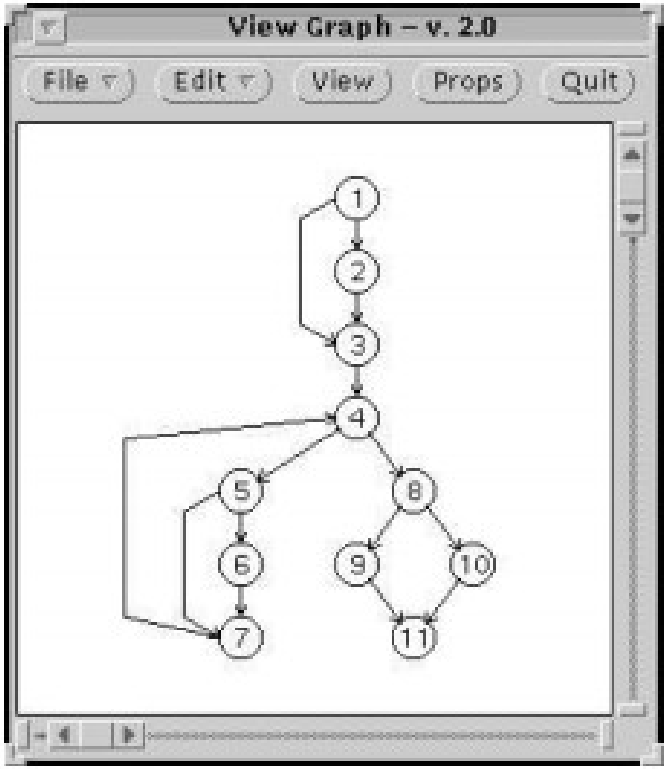
```
                #include <stdio.h>
1               main ()
2  /*  1 */     {
3  /*  1 */         char  achar;
4  /*  1 */         int length, valid_id;
5  /*  1 */         length = 0;
6  /*  1 */         valid_id = 1;
7  /*  1 */         printf ("Identificador: ");
8  /*  1 */         achar = fgetc (stdin);
9  /*  1 */         valid_id = valid_s(achar);
10 /*  1 */         if(valid_id)
11 /*  2 */         {
12 /*  2 */             length = 1;
13 /*  2 */         }
14 /*  3 */         achar = fgetc (stdin);
15 /*  4 */         while(achar != '\n')
16 /*  5 */         {
17 /*  5 */             if(!(valid_f(achar)))
18 /*  6 */             {
19 /*  6 */                 valid_id = 0;
20 /*  6 */             }
21 /*  7 */             length++;
22 /*  7 */             achar = fgetc (stdin);
23 /*  7 */         }
24 /*  8 */         if(valid_id &&
                      (length >= 1) && (length < 6))
25 /*  9 */         {
26 /*  9 */             printf ("Valido\n");
27 /*  9 */         }
28 /* 10 */         else
29 /* 10 */         {
30 /* 10 */             printf ("Invalid\n");
31 /* 10 */         }
32 /* 11 */     }
```

```
                int valid_s(char ch)
/* 2 */         }
/* 3 */         else
/* 3 */         {
/* 3 */             return (0);
/* 3 */         }
/* 4 */     }

                int valid_f(char ch)
/* 1 */     {
/* 1 */         if(((ch >= 'A') &&
                    (ch <= 'Z')) ||
                   ((ch >= 'a') &&
                    (ch <= 'z')) ||
                   ((ch >= '0') &&
                    (ch <= '9')))
/* 2 */         {
/* 2 */             return (1);
/* 2 */         }
/* 3 */         else
/* 3 */         {
/* 3 */             return (0);
/* 3 */         }
/* 4 */     }
```

O caminho 1,2,3,4,5,7,4,8,9,11 é um caminho completo

O programa *Identifier.c*

- O comando if (valid_id) (linha 10) ilustra um desvio de execução entre os nós do programa.
- Caso sejam exercitados os comandos internos ao if, tem-se um desvio de execução do nó 1 para o nó 2, representado no grafo pelo arco (1,2).
- Do contrário, se os comandos internos ao if não forem executados, tem-se um desvio do nó 1 para o nó 3, representado pelo arco (1,3).
- De maneira similar, obtêm-se os arcos (2,3), (3,4), (4,5), e assim por diante.



Exercícios – Desenhe o GFC (Grafos de Fluxo de Controle) para os seguintes códigos.

Exercício 1:

```
1.  public Triangulo(String args[ ]){
2.  int a,b,c;
3.  String resp = null;
4.  a = Integer.parseInt(args[0]);
5.  b = integer.parseInt(args[1]);
6.  c = integer.parseInt(args[2]);
7.  if (a==b)&&(b==c)
8.  resp = “equilatero”;
9.  if (((a==b)&&(b!=c))||((b==c)&&(a!=b))||
    ((a==c)&&(c!=b))
10.  resp = “isoceles”;
11.  if ((a!=b)&&(b!=c))
12.  resp = “escaleno”;
13.  system.out.println(“Tipo de triangulo:”+resp);
14.  }
```

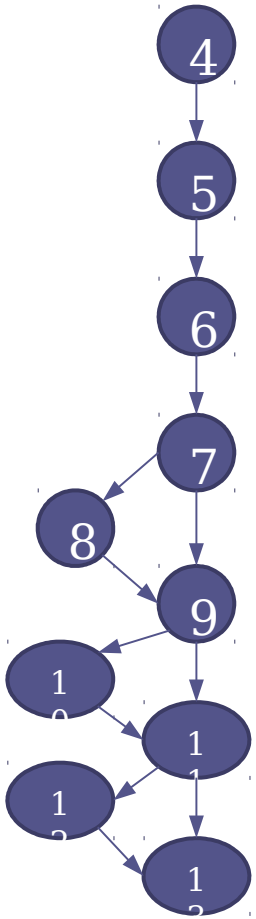
Exercício 2:

Cálculo do mdc (a, b) baseado no algoritmo de Euclides

```
1.      function mdc (int a, int b) {
2.          int temp, value;
3.          a := abs(a);
4.          b := abs(b);
5.          if (a = 0) then
6.                value := b;    // b é o MDC
7.          else if (b = 0) then
8.               exceção;
9.          else
10.                repeat
11.                     temp := b;
12.                     b := a mod b;
13.                     a := temp;
14.                until (b  =  0)
15.                value := a;
16.           end if;
17.           return value;
18.      end mdc
```

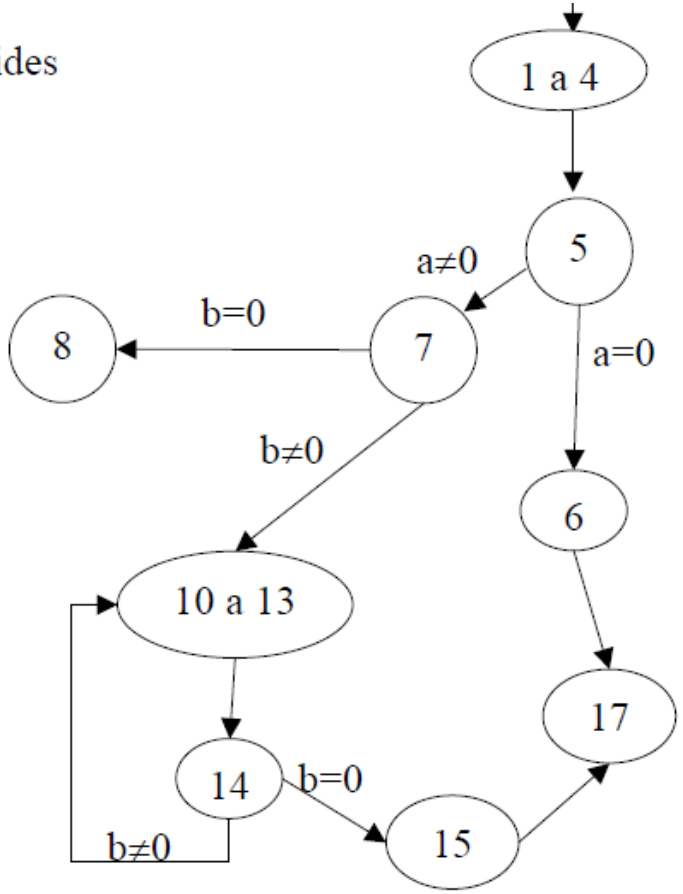
Solução - Exercício1 – Desenhe o grafo de programa correspondente ao código a seguir

```
1.  public Triangulo(String args[ ]){
2.  int a,b,c;
3.  String resp = null;
4.  a = Integer.parseInt(args[0]);
5.  b = integer.parseInt(args[1]);
6.  c = integer.parseInt(args[2]);
7.  if (a==b)&&(b==c)
8.  resp = “equilatero”;
9.  if (((a==b)&&(b!=c))||((b==c)&&(a!=b))||
      ((a==c)&&(c!=b))
10.  resp = “isoceles”;
11.  if ((a!=b)&&(b!=c))
12.  resp = “escaleno”;
13.  system.out.println(“Tipo de triangulo:”+resp);
14.  }
```



Solução – Exercício 2 – Desenhe o grafo de programa correspondente ao código a seguir:

```
Cálculo do mdc (a, b) baseado no algoritmo de Euclides
1.      function mdc (int a, int b) {
2.          int temp, value;
3.          a := abs(a);
4.          b := abs(b);
5.          if (a = 0) then
6.                value := b;    // b é o MDC
7.          else if (b = 0) then
8.               exceção;
9.          else
10.                repeat
11.                     temp := b;
12.                     b := a mod b;
13.                     a := temp;
14.                until (b  =  0)
15.               value := a;
16.          end if;
17.          return value;
18.     end mdc
```



Modelo de base → Modelo de teste

Como aplicar GFC para derivar os casos de teste????

Critérios para Teste Estrutural

1. **Critérios Baseados em Fluxo de Controle:** **c**onsidera somente nós e arestas
   - **Todos-Nós** - exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo de fluxo, ou seja, que cada comando do programa seja executado pelo menos uma vez;
   - **Todos-Arcos/Ramos** - requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez;
   - **Todos-Caminhos** - requer que todos os caminhos possíveis do programa sejam executados.

Critérios para Teste Estrutural

2. **Critérios Baseados em Fluxo de Dados:**
   - Requer que o grafo seja anotado com referências a variáveis exploram as interações que envolvem definições de variáveis e referências a tais definições para estabelecerem os requisitos de teste.
3. **Critérios Baseados na Complexidade:**
   - Utilizam informações sobre a complexidade do programa para derivar os requisitos de teste.
   - Um critério bastante conhecido dessa classe é o **Critério de McCabe,** que utiliza a **complexidade ciclomática** do GFC para derivar os requisitos de teste.
   - **Complexidade ciclomática** é uma métrica de software usada para indicar a complexidade de um programa de computador para medir a quantidade de caminhos de execução independentes de um código fonte.

Critérios para Teste Estrutural

- Os passos básicos para se aplicar um critério de teste caixa branca são os seguintes:
  1. A implementação do produto em teste é analisada.
  2. Caminhos através da implementação são escolhidos.
  3. Valores de entradas são selecionados de modo que os caminhos selecionados sejam executados.
  4. As saídas esperadas para as entradas escolhidas são determinadas.
  5. Os casos de testes são construídos.
  6. As saídas obtidas são comparadas com as saídas esperadas.
  7. Um relatório é gerado para avaliar o resultado dos testes.

Critérios para Teste Estrutural

- Grafo de Fluxo de Controle:
  - Define uma relação entre o caso de teste e a parte do programa exercitada por ele.
- Um caso de teste:
  - Corresponde a um caminho no grafo.
  - Corresponde a uma execução completa: do nó inicial até o final.
- Cada caso de teste corresponde a um caminho no grafo.
  - Etapas:
    - Construir o grafo de fluxo de programa.
    - Determinar os caminhos factíveis.
    - Selecionar um conjunto de caminhos factíveis para teste.
    - Gerar as entradas e os resultados esperados para os casos de teste correspondentes.

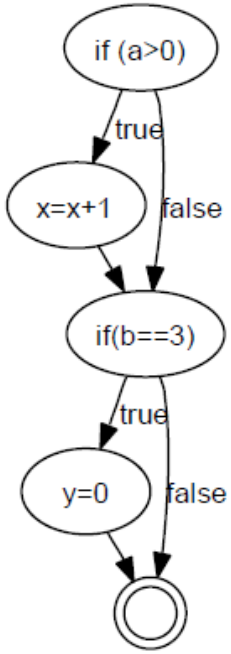
Níveis de Cobertura

- Diferentes níveis de cobertura podem ser definidos em função dos elementos do GFC.
- Cobertura: porcentagem dos requisitos que foram testados *versus* o total de requisitos gerados.
- Oito diferentes níveis de cobertura são definidos por Copeland (2004).
- Quanto maior o nível, maior o rigor do critério de teste, ou seja, mais caso de teste ele exige para ser satisfeito.
  - Nível 0 ← Nível 1 ← Nível 2 ← Nível 3 ← Nível 4 ← Nível 5 ← Nível 6 ← Nível 7

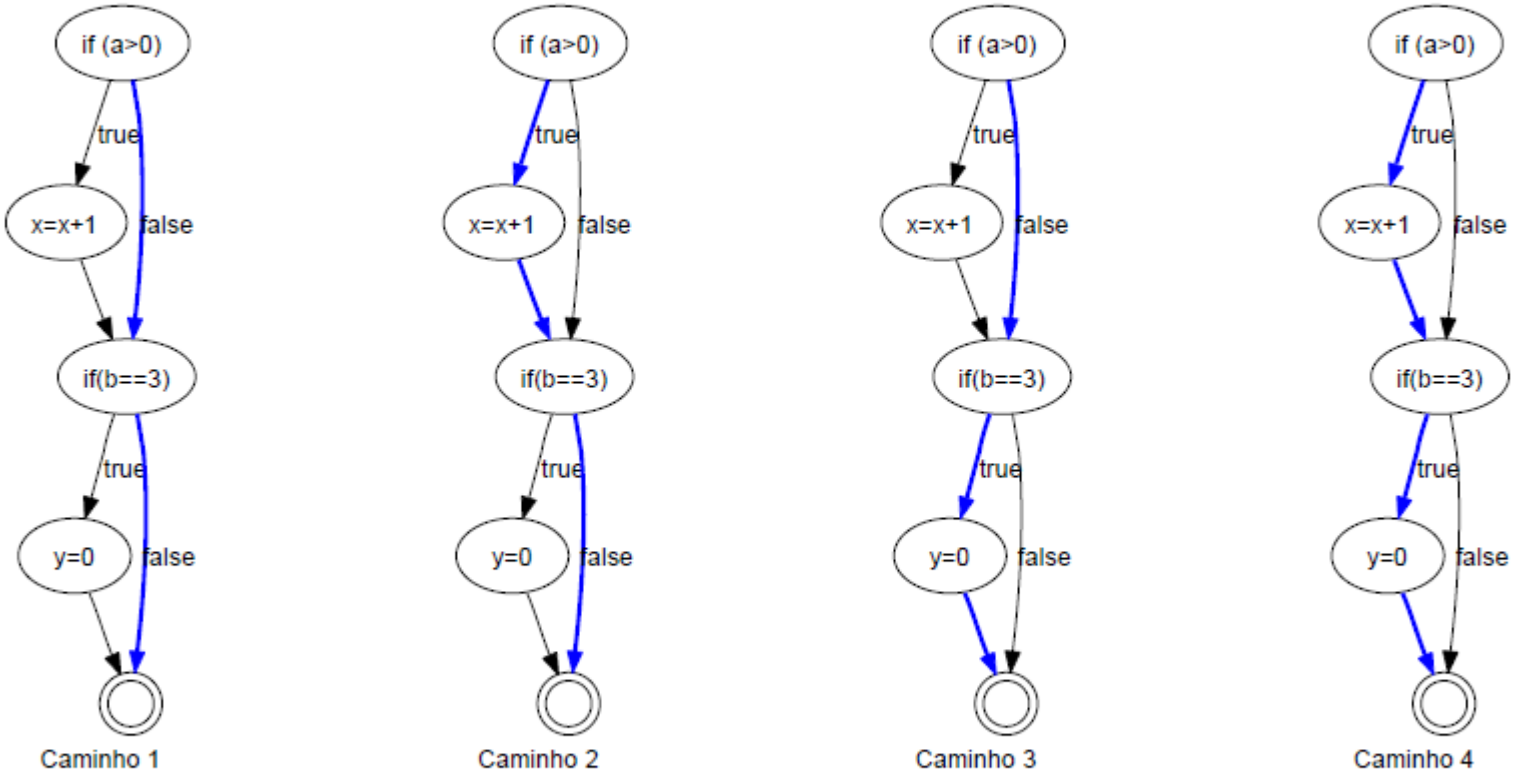
Níveis de Cobertura

- Nível 0: qualquer valor de cobertura inferior a 100% da cobertura de todos os comandos.
- Nível 1: 100% de cobertura de comandos.
  - Também chamado de cobertura de nós (critério **todos-nós**)

```
1  if (a>0){
2      x=x+1;
3  }
4  if (b==3){
5      y=0;
6  }
```



Nível 1: 100% de cobertura de comandos (requisito mínimo de teste)



Um caso de teste é suficiente para cobrir todos os comandos mas não todos os caminhos. Por exemplo, use a=6 e b=3 para cobrir o "Caminho 4".

Nível 1: 100% de cobertura de comandos (requisito mínimo de teste)

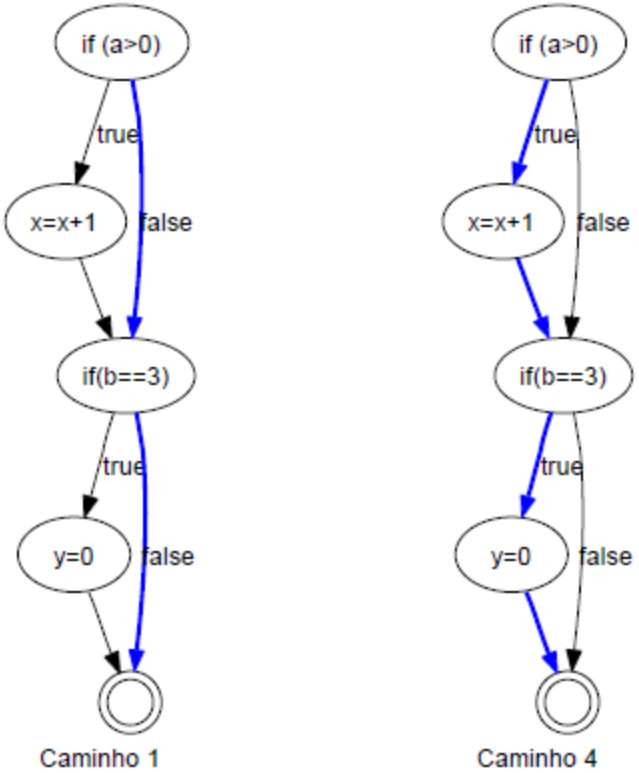
- Embora seja o nível mais baixo de cobertura pode ser difícil de ser atingida em alguns casos.
  - Código para situações excepcionais: falta de memória, disco cheio, arquivos ilegíveis, perda de conexão, dentre outras.
  - Pode ser difícil ou impossível simular tais situações excepcionais.
  - Nessas situações, o código correspondente permanece não testado.

Requisito não executável, ou seja, quando não existe um dado de teste no qual ele pode ser executado

Níveis de Cobertura (2)

- Nível 2: 100% de cobertura de decisões.
  - Também chamado de cobertura de arcos/arestas (critério **todos-arcos**).
  - Objetivo fazer cada comando de decisão assumir os valores `TRUE` e `FALSE`.

Nível 2: 100% de cobertura de decisões.



Dois casos de testes são suficientes para cobrir todos os arcos do GFC. Por exemplo, a=2, b=2 e a=4, b=3 satisfazem o critério **todos-arcos**.

- Nível 3: 100% de cobertura de condições.
  - Nem todos comandos de decisão são simples como o anterior.
  - Considere o exemplo abaixo:

```
1  if (a>0 && c==1){
2      x=x+1;
3  }
4  if (b==3 || d<0){
5      y=0;
6  }
```

  - Comando linha 2: requer que a>0 e c==1 sejam ambos TRUE.
    - Se a for 0, o comando c==1 pode nunca ser executado (linguagem de programação – curto-circuito).
  - Comando linha 5: requer que b==3 ou d<0 seja TRUE.

- Nível 3: 100% de cobertura de condições.
  - Para o exemplo em questão:

```
1  if (a>0 && c==1){
2      x=x+1;
3  }
4  if (b==3 || d<0){
5      y=0;
6  }
```

    - Dois casos de teste necessários para cobrir todas as condições:

      $\{a > 0, c = 1, b = 3, d < 0\}$ e $\{a \leq 0, c \neq 1, b \neq 3, d \geq 0\}$

  - Cobertura de condição é, em geral, melhor que cobertura de decisão: cada condição individual assume os valores TRUE e FALSE.

- Nível 4: 100% de cobertura decisões/condições.
  - Considere o exemplo abaixo:

```
1  if (x && y) {
2      conditionedStatement;
3  }
```

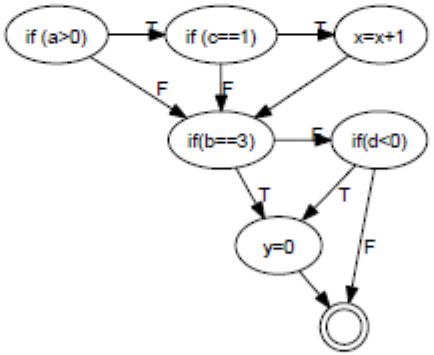
  - Cobertura de condições pode ser obtida com dois casos de testes:

    {x=TRUE, y=FALSE } e {x=FALSE, y=TRUE }

    mas o comando `conditionedStatement` não será executado.
  - O critério 100% de cobertura de decisões/condições requer que todas as combinações sejam testadas.

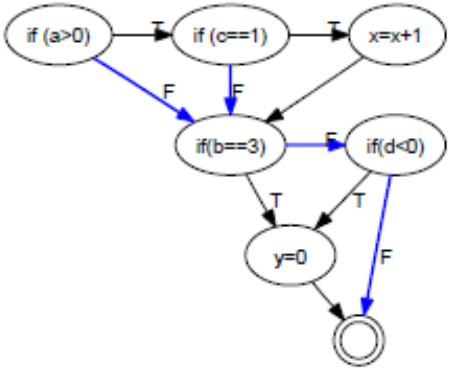
- Nível 5: 100% de cobertura de condições múltiplas.
  - Consiste em utilizar o conhecimento de como o compilador avalia condições múltiplas de determinado comando de decisão e utilizar essa informação na geração de casos de testes.
  - Considere um compilador que avalia condições múltiplas em uma decisão conforme ilustrado abaixo:

```
1  if (a>0 && c==1){
2      x=x+1;
3  }
4  if (b==3 || d<0){
5      y=0;
6  }
```



Obter 100% cobertura de condições múltiplas implica cobrir 100% dos critérios anteriores, mas não garante cobertura de todos-caminhos.

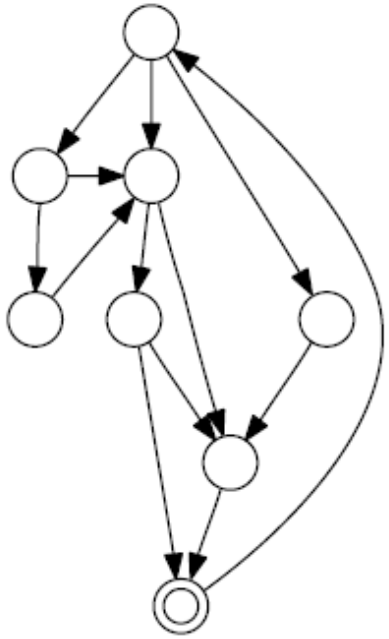
- Nível 5: 100% de cobertura de condições múltiplas.
  - Cobrir os arcos do grafo abaixo requer os seguintes casos de testes:



$a>0$	$c=1$	$b=3$	$d<0$
$a \leq 0$	$c=1$	$b=3$	$d \geq 0$
$a>0$	$c \neq 1$	$b \neq 3$	$d<0$
$a \leq 0$	$c \neq 1$	$b \neq 3$	$d \geq 0$ ←

- Nível 6: cobertura de loop.
  - Quando programas possuem loop, o número de caminhos possíveis pode ser infinito.
  - O número de caminhos pode ser reduzido limitando a execução do loop a:
    - 0 vezes.
    - 1 vez.
    - 2 vezes.
    - n vezes (sendo n um número de vezes padrão que o loop é executado).
    - m vezes (sendo m o número máximo de vezes que o loop pode ser executado).
    - m - 1 vezes.
    - m + 1 vezes.

- Nível 7: 100% de cobertura de caminhos.
  - Também conhecido como critério **todos-caminhos**.
  - Para programas sem loop o número de caminhos pode ser pequeno o suficiente e casos de testes podem ser construídos para cobri-los.
  - Para programas com loop o número de caminhos pode ser muito grande ou infinito, tornando-se impossível cobrir todos os caminhos de execução.



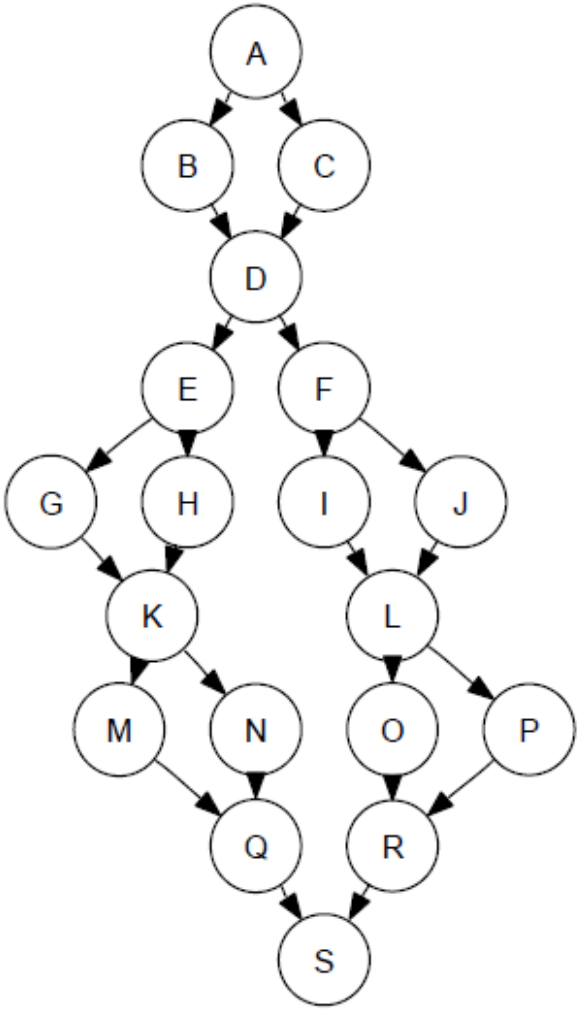
Teste do Caminho Básico

- Critério de teste estrutural baseado no fluxo de controle.
- Trabalho pioneiro desenvolvido por McCabe (1976).
- Baseado no conceito de Complexidade Ciclomática (calculada a partir do GFC).

Passos para aplicação

- Construir o GFC para o módulo do produto em teste.
- Calcular a Complexidade Ciclomática ($\mathcal{C}$).
- Selecionar um conjunto de $\mathcal{C}$ caminhos básicos.
- Criar um caso de teste para cada caminho básico.
- Executar os casos de testes.

Cálculo da complexidade ciclomática



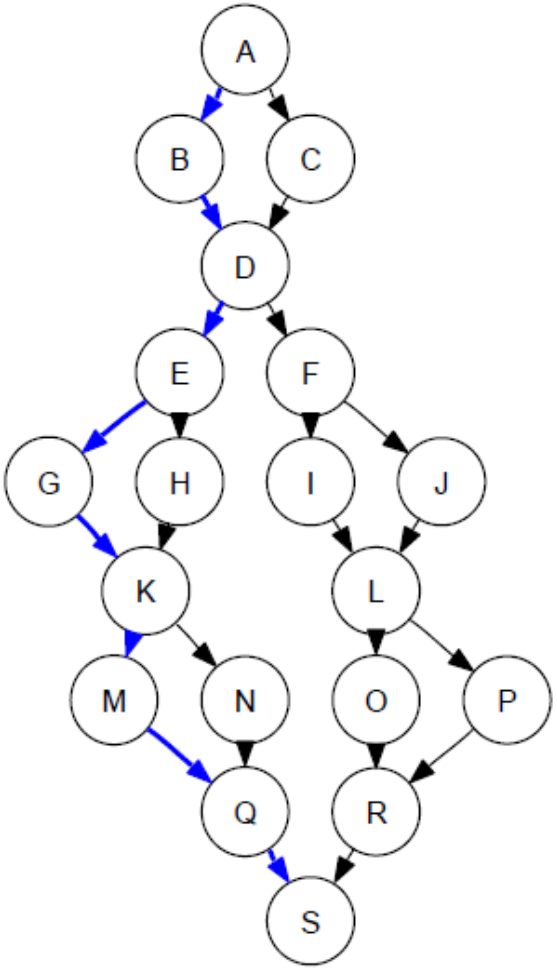
- Considere o GFC ao lado.
- McCabe (1976) define a Complexidade Ciclomática ($\mathcal{C}$) como:

  $\mathcal{C}$ = arcos - nós +2
  $\mathcal{C}$ = 24 - 19 +2
  $\mathcal{C} = 7$

  Para um GFC com p decisões binárias (dois arcos saindo):

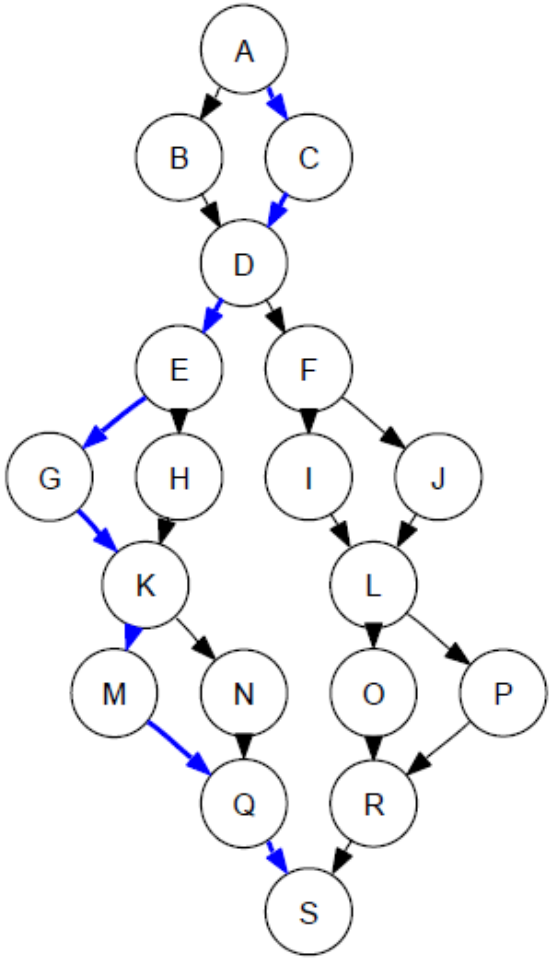
  $\mathcal{C} = p + 1$
- A complexidade ciclomática representa o número mínimo de caminhos independentes, sem loop, que gera todos os possíveis caminhos de um módulo.
- Em termos do GFC, cada caminho básico inclui pelo menos um arco que ainda não foi selecionado.
- Criar e executar $\mathcal{C}$ casos de testes (um para cada caminho básico) garante cobertura dos critérios **todos-nós** e **todos-arcos**.

Criação do conjunto de Caminhos Básicos – Passo 1



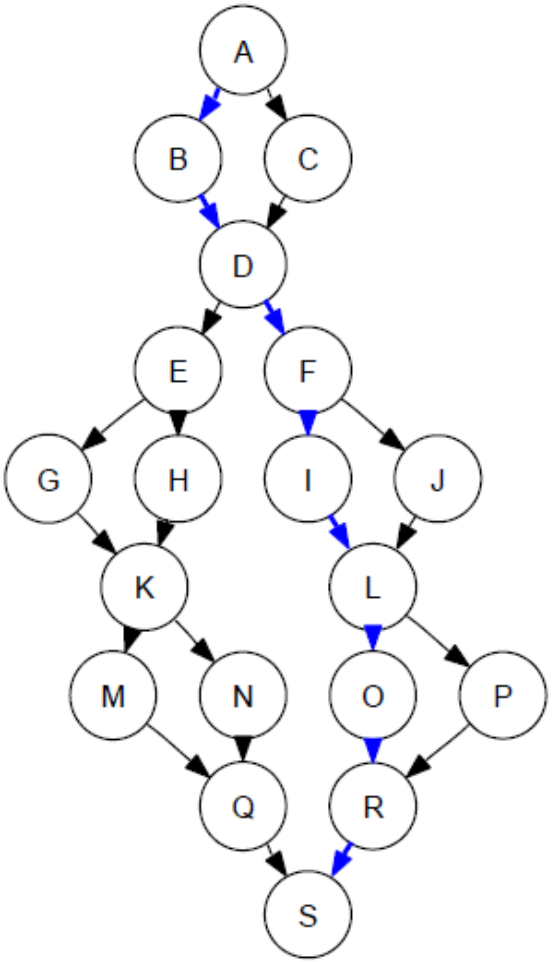
- Escolha um caminho básico. Esse caminho pode ser:
  - Caminho mais comum.
  - Caminho mais crítico.
  - Caminho mais importante do ponto de vista de teste.
- Caminho 1: ABDEGKMQS

Criação do conjunto de Caminhos Básicos – Passo 2



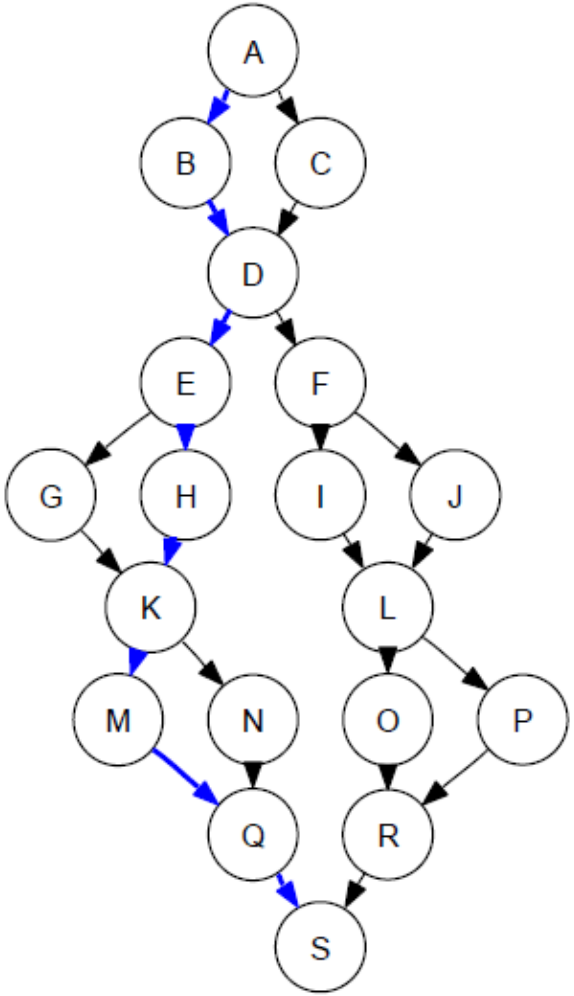
- Altere a saída do primeiro comando de decisão e mantenha o máximo possível do caminho inalterado.
- Caminho 2: ACDEGKMQS

Criação do conjunto de Caminhos Básicos – Passo 3



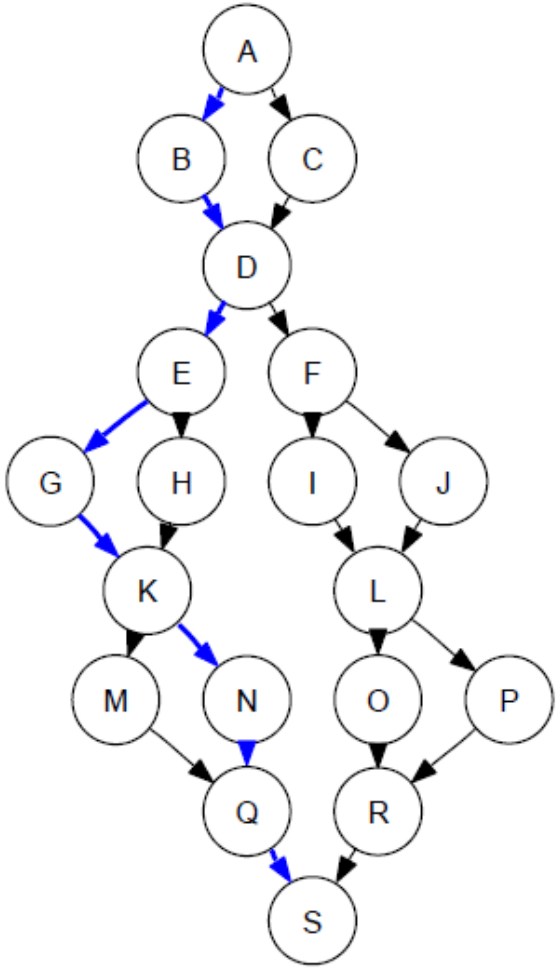
- A partir do caminho base alterar a saída do segundo comando de decisão.
- Caminho 3: ABDFILORS

Criação do conjunto de Caminhos Básicos – Passo 4



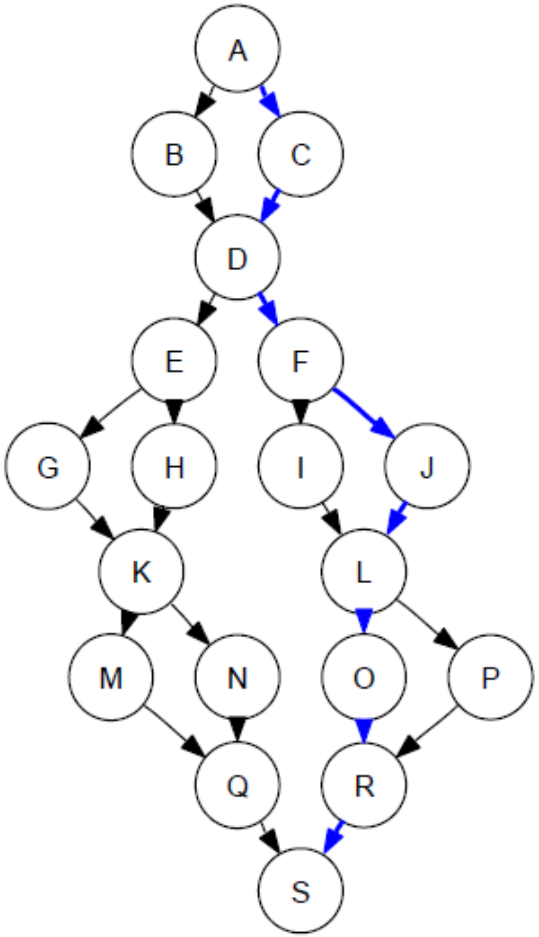
- A partir do caminho base alterar a saída do terceiro comando de decisão. Repetir esse processo até atingir o final do GFC.
- Caminho 4: ABDEHKMQS

Criação do conjunto de Caminhos Básicos – Passo 4



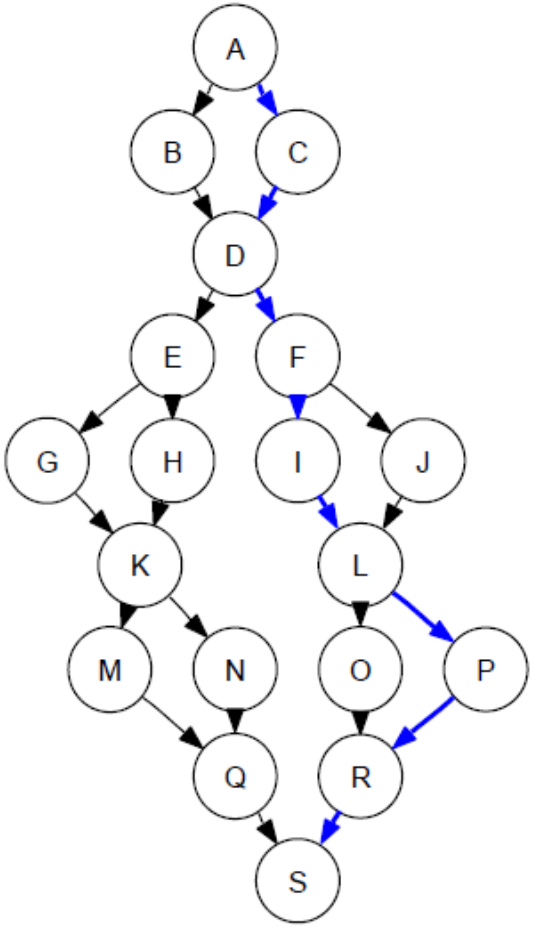
- Continuação do passo anterior.
- Caminho 5: ABDEGKNQS

Criação do conjunto de Caminhos Básicos – Passo 5



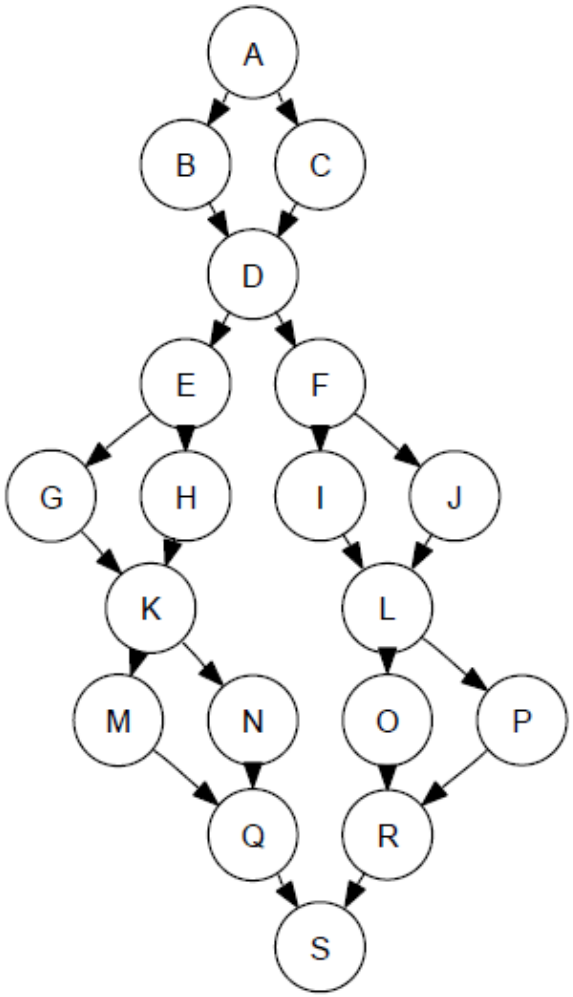
- Todas as decisões do caminho básico foram contempladas.
- A partir do segundo caminho, fazer as inversões dos comandos de decisão até o final do GFC.
- Esse padrão é seguido até que o conjunto completo de caminhos seja atingido.
- Caminho 6: ACDFJLORS

Criação do conjunto de Caminhos Básicos – Passo 5



- Continuação do passo anterior.
- Caminho 7: ACDFILPRS

Conjunto completo de Caminhos Básicos



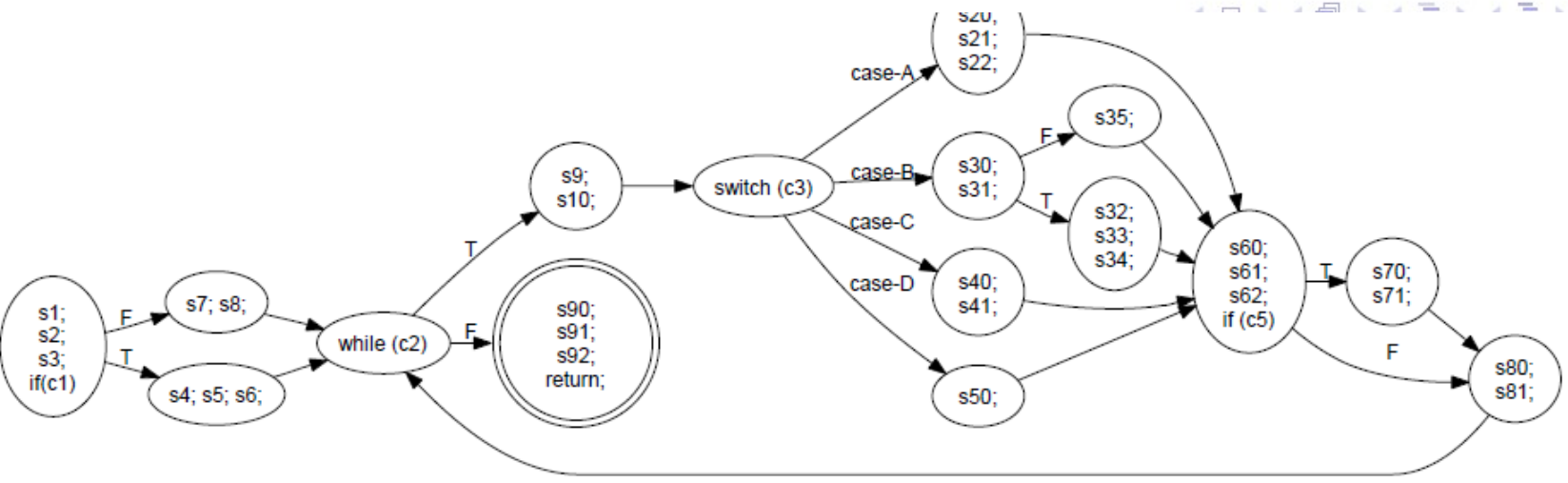
- Requisitos de testes derivado pelo critério.
  - ABDEGKMQS
  - ACDEGKMQS
  - ABDFILORS
  - ABDEHKMQS
  - ABDEGKNQS
  - ACDFJLORS
  - ACDFILPRS
- Conjunto criado não é único.
- Propriedade: o conjunto de teste que exercita os caminhos básicos também exercita todos-nós e todos-arcos do programa.

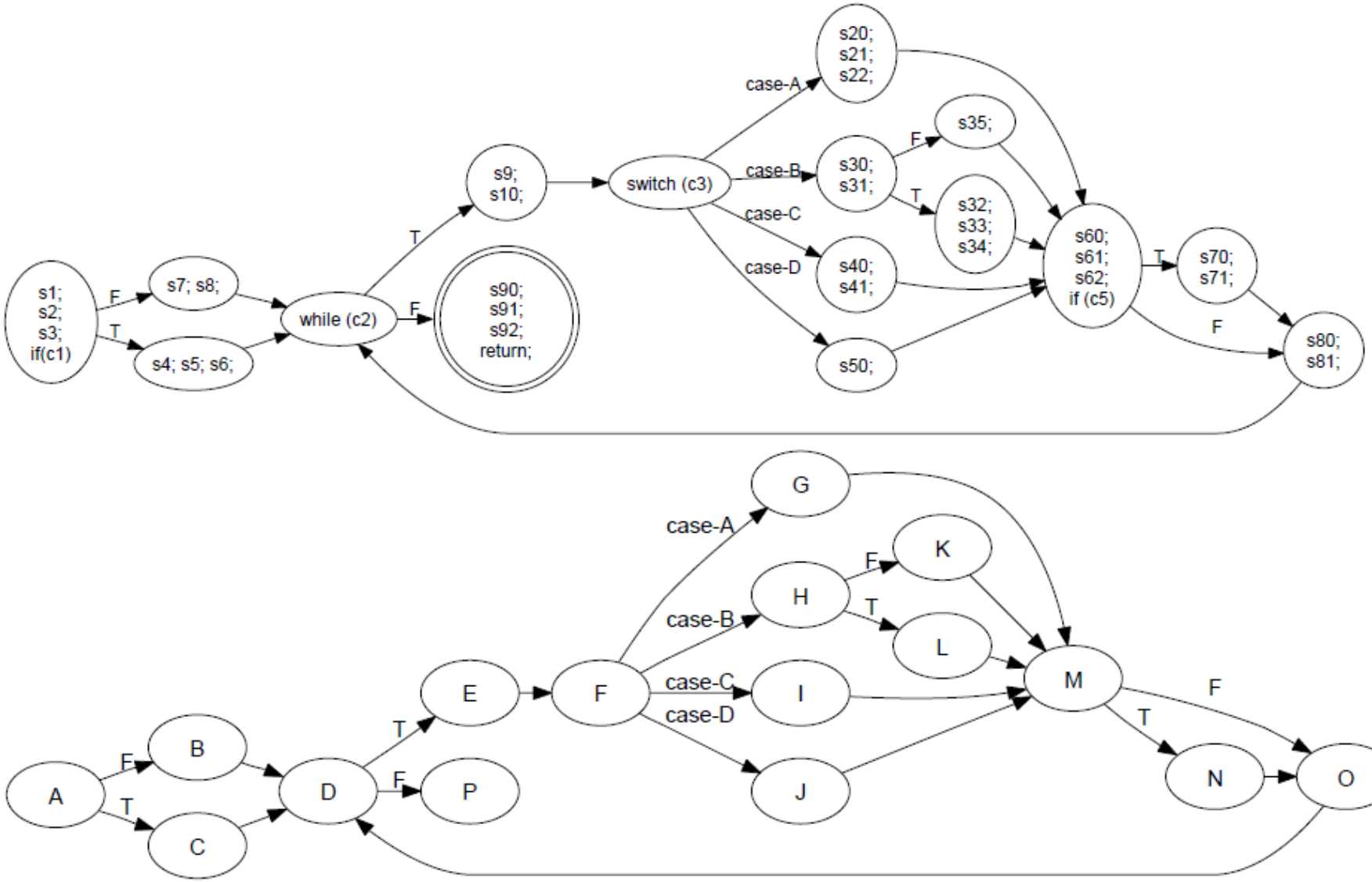
Exemplo

Considere o código Java abaixo:

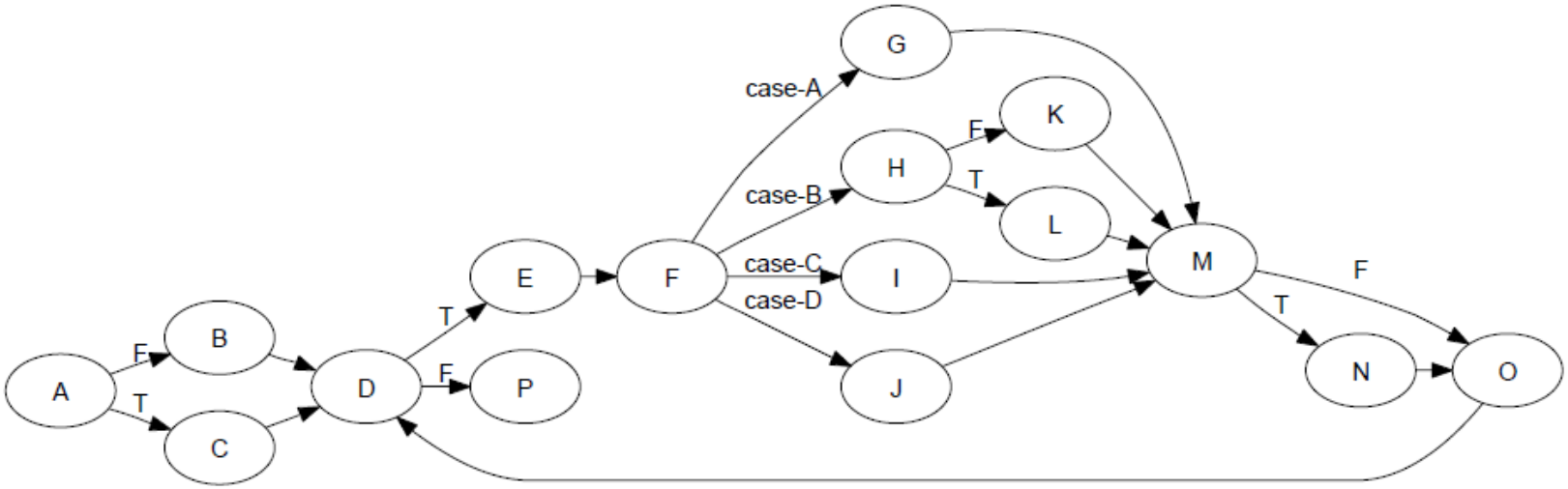
```
1   boolean evaluateBuySell (TickerSymbol ts) {
2   s1;
3   s2;
4   s3;
5   if (c1) {s4; s5; s6;}
6   else {s7; s8;}
7   while (c2) {
8      s9;
9      s10;
10     switch (c3) {
11        case-A:
12           s20;
13           s21;
14           s22;
15           break; // End of Case-A
16        case-B:
17           s30;
18           s31;
19           if (c4) {
20              s32;
21              s33;
22              s34;
23           }
24           else {
25              s35;
26           }
27           break; // End of Case-B
28        case-C:
29           s40;
30           s41;
31           break; // End of Case-C
32        case-D:
33           s50;
34           break; // End of Case-D
35        } // End Switch
36        s60;
37        s61;
38        s62;
39        if (c5) {s70; s71; }
40        s80;
41        s81;
42  } // End While
43  s90;
44  s91;
45  s92;
46  return result;
```

```
1   boolean evaluateBuySell (TickerSymbol ts) {
2   s1;
3   s2;
4   s3;
5   if (c1) {s4; s5; s6;}
6   else {s7; s8;}
7   while (c2) {
8      s9;
9      s10;
10     switch (c3) {
11        case–A:
12           s20;
13           s21;
14           s22;
15           break; // End of Case–A
16        case–B:
17           s30;
18           s31;
19           if (c4) {
20              s32;
21              s33;
22              s34;
23           }
24           else {
25              s35;
26           }
27           break; // End of Case–B
28        case–C:
29           s40;
30           s41;
31           break; // End of Case–C
32        case–D:
33           s50;
34           break; // End of Case–D
35        } // End Switch
36        s60;
37        s61;
38        s62;
39        if (c5) {s70; s71; }
40        s80;
41        s81;
42  } // End While
43  s90;
44  s91;
45  s92;
46  return result;
```




s1;
s2;
s3;
if(c1)
F
T
s7; s8;
s4; s5; s6;
while (c2)
T
F
s9;
s10;
s90;
s91;
s92;
return;
switch (c3)
case-A
case-B
case-C
case-D
s20;
s21;
s22;
s30;
s31;
F
T
s35;
s32;
s33;
s34;
s40;
s41;
s50;
s60;
s61;
s62;
if (c5)
T
F
s70;
s71;
s80;
s81;
A
F
T
B
C
D
T
F
E
P
F
case-A
case-B
case-C
case-D
G
H
I
J
K
L
M
F
T
N
O

Complexidade ciclomática



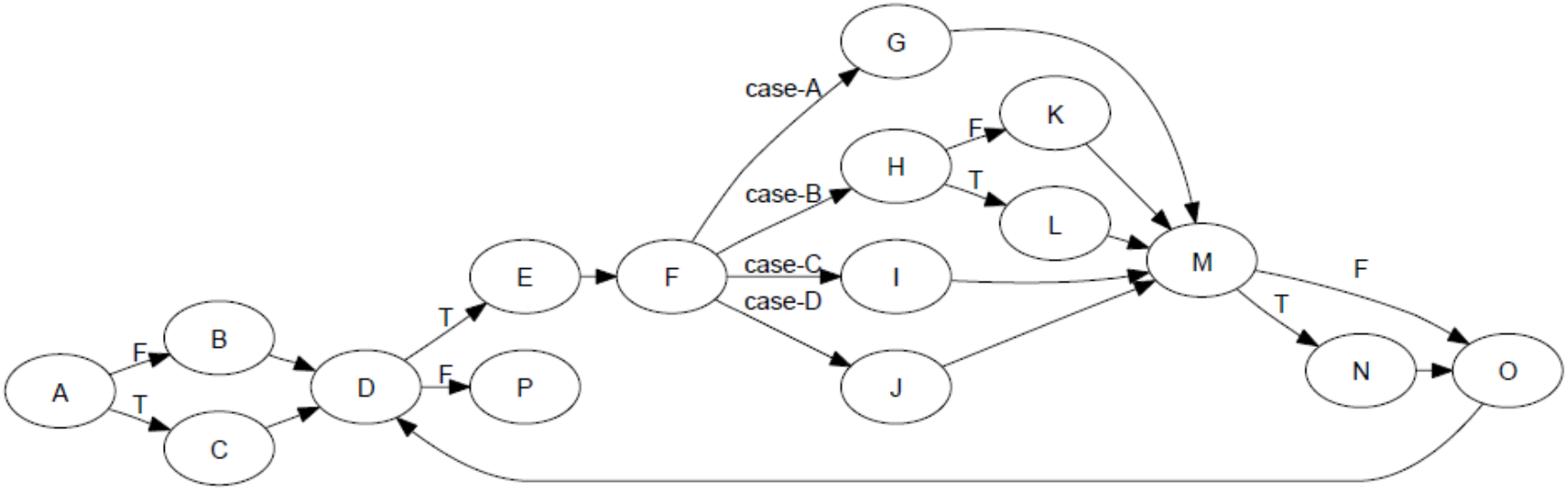
- Cálculo da complexidade ciclomática para o GFC acima:

$\mathcal{C}$ = arcos - nós + 2

$\mathcal{C}$ = 22-16+2

$\mathcal{C}$ = 8

Caminhos básicos



1. ABDP
2. ACDP
3. ABDEFGMODP
4. ABDEFHKMODP
5. ABDEFIMODP
6. ABDEFJMODP
7. ABDEFHLMODP
8. ABDEFIMNODP

Conjunto de Teste

1. ABDP
2. ACDP
3. ABDEFGMODP
4. ABDEFHKMODP
5. ABDEFIMODP
6. ABDEFJMODP
7. ABDEFHLMODP
8. ABDEFIMNODP

Caso Teste	C1	C2	C3	C4	C5
1	False	False	N/A	N/A	N/A
2	True	False	N/A	N/A	N/A
3	False	True	A	N/A	False
4	False	True	B	False	False
5	False	True	C	N/A	False
6	False	True	D	N/A	False
7	False	True	B	True	False
8	False	True	C	N/A	True

Aplicabilidade e Limitações

- Critérios de Fluxo de Controle são a "pedra fundamental" do teste de unidade.
- Devem ser aplicados a todos os módulos do software, em especial, nos mais críticos.
- Exigem habilidades de programação do testador para compreender o fluxo de controle do programa.
- Pode consumir tempo e recursos significativos para sua aplicação.

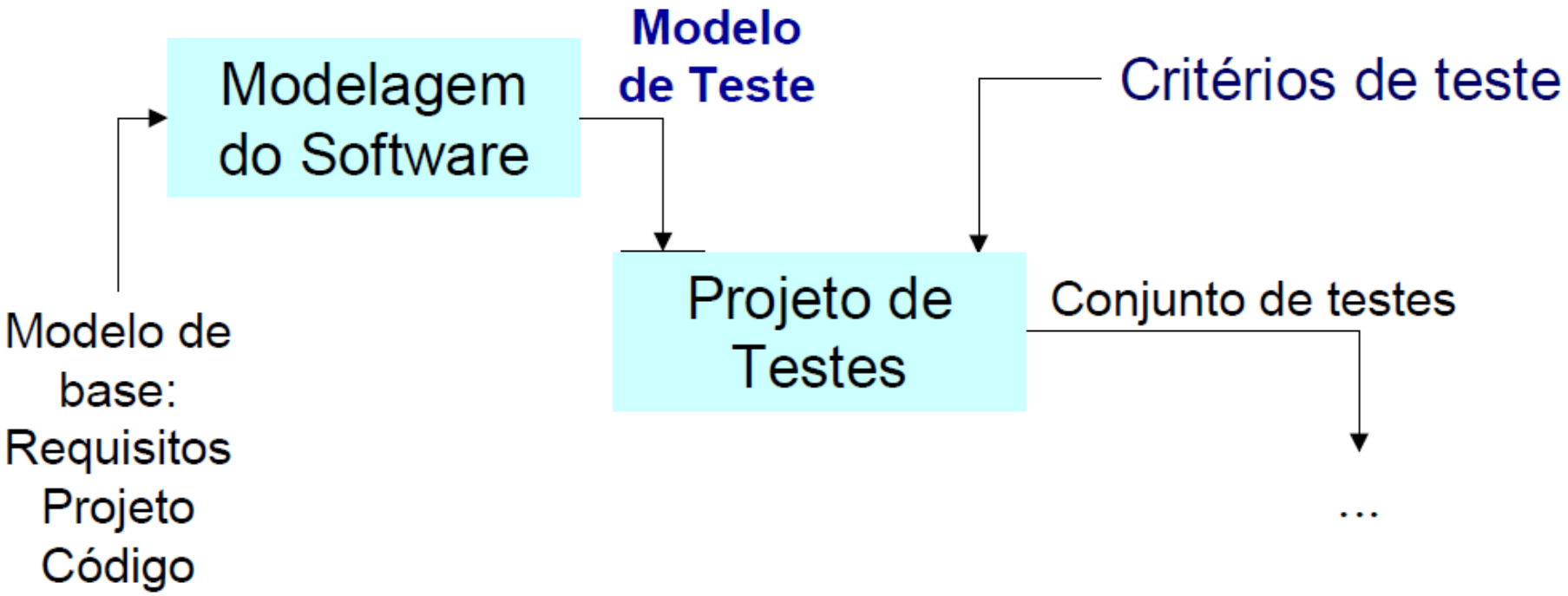
Teste funcional

Conclusão

Muitas vezes, tendemos a avaliar a qualidade do software simplesmente por aspectos comportamentais e negligenciamos as características computacionais que permanecem escondidas abaixo da interface do usuário.

Ao contrário da técnica de Teste Funcional, a técnica de Teste Estrutural é uma abordagem usada para projetar testes baseando-se na estrutura interna e na implementação de um programa.

Conclusão
Projeto de Teste



Referências

- Delamaro, Márcio Eduardo. Contexto geral – Teste estrutural. Disponível em: <https://www.coursera.org/learn/intro-teste-de-software/lecture/BPIkf/contexto-geral>. Ultimo acesso: março/ 2020.
- Justo, Daniela Sbizera Profª Dra. Introdução a teste de software. IFSC. Gaspar.