

Teste de Software

Prof. Raul Sidnei Wazlawick



Erro humano

- Por melhores que sejam as técnicas de modelagem e especificação de software, por mais disciplinada e experiente que seja a equipe de desenvolvimento, sempre haverá um fator que faz com que o teste de software seja necessário: o *erro humano*.



Fundamentos

- Um **erro** (*error*) é uma diferença detectada entre o resultado de uma computação e o resultado correto ou esperado.
- Um **defeito** (*fault*) é uma linha de código, bloco ou conjunto de dados incorretos, que provocam um erro observado.
- Uma **falha** (*failure*) é um não funcionamento do software, possivelmente provocada por um defeito, mas também com outras causas possíveis.
- Um **engano** (*mistake*), ou *erro humano*, é a ação que produz ou produziu um defeito no software.



Verificação, Validação e Teste

- **Verificação** consiste em analisar o software para ver se ele está sendo construído de acordo com o que foi especificado.
- **Validação** consiste em analisar o software construído para ver se ele atende às verdadeiras necessidades dos interessados.
- **Teste** é uma atividade que permite realizar a verificação e a validação do software.



Depuração

- A *depuração* é a atividade que consiste em buscar a causa do erro, ou seja, o defeito oculto que a está causando.
- O fato de se saber que o software não funciona não significa que necessariamente se saiba qual ou quais são as linhas de código que provocam esse erro.




Teste de partes de software

- Frequentemente, partes do software precisam ser testadas isoladamente.
- Mas essas partes normalmente se comunicam com outras partes.

Stub



- Quando um componente *A* que vai ser testado chama operações de outro componente *B* que ainda não foi implementado, pode-se criar uma implementação simplificada de *B*, chamada *stub*, que será utilizada no lugar de *B*.



Exemplo: um stub para gerador de números primos

Função primo(n):integer

Caso n

1: retorna 2;

2: retorna 3;

3: retorna 5;

4: retorna 7;

5: retorna 11

Fim



Driver



- Por outro lado, muitas vezes é o módulo *B* que já está implementado, mas o módulo *A* que chama as funções de *B* ainda não foi implementado.
- Neste caso, deverá ser implementada uma simulação do módulo *A*, denominada *driver*.
- Essa simulação deverá chamar as funções do módulo *B* e, de preferência, executar todos os casos de teste necessários para testar *B*, de acordo com a técnica de teste adotada.




Níveis de Teste de Funcionalidade

- Unidade
- Integração
- Sistema
- Aceitação
- Ciclo de negócio
- Aceitação
- Operação



Teste de Unidade

- Os testes de unidade são os mais básicos e usualmente consistem em verificar se um componente individual do software (unidade) foi implementado corretamente.
- Esse componente pode ser um método ou procedimento, uma classe completa ou ainda um pacote de funções ou classes de tamanho pequeno a moderado.
- Usualmente, essa unidade ainda estará isolada do sistema do qual fará parte.



Exemplo (verificar se um método foi corretamente implementado em uma classe)

- Classe Livro
- Método `setIsbn(umIsbn:string)`.
- A especificação do método estabelece que ele deve trocar o valor do atributo ISBN do livro para o parâmetro passado, mas, antes disso, deve verificar se o valor do ISBN passado não corresponde a um ISBN já cadastrado (porque uma regra de negócio estabelece que dois livros não podem ter o mesmo ISBN).



Testes que devem ser feitos

- Inserir um ISBN que ainda não consta na base e verificar se o atributo do livro foi adequadamente modificado.
- Inserir um ISBN que já existe na base e verificar que a exceção foi sinalizada, como esperado.

Exemplo de driver para o teste

```
PROGRAMA DriverParaSetIsbn
(* 1 – entrada correta *)
REPITA
    isbn := geralIsbnAutomaticamente()
    ATÉ NÃO existeLivroNaBaseComIsbn(isbn)
    umLivro := obterUmLivroDaBase()
    umLivro.setIsbn(isbn)
    SE umLivro.getIsbn() = isbn ENTÃO
        Escreva('teste 1 correto')
    SENAÓ
        Escreva('falha encontrada no teste 1')
    FIM SE
(* 2 – entrada incorreta *)
umLivro := obterUmLivroDaBase()
REPITA
    outroLivro := obterUmLivroDaBase()
    ATÉ umLivro <> outroLivro
TENTE
    umLivro.setIsbn(outroLivro.getIsbn())
    Escreva('falha encontrada no teste 2 – não ocorreu exceção')
FIM TESTE
CAPTURE EXCEÇÃO
    SE EXCEÇÃO 'isbnInvalido' ENTÃO escreva ('teste 2 correto')
FIM CAPTURE
```

Ferramentas para teste

- *Junit*: um *framework* gratuitamente distribuído que permite inserir comandos específicos de verificação no programa que acusarão os erros caso venham a ser encontrados.
- *OCL Query-Based Debugger* (Hobatr & Malloy, 2001), que é uma ferramenta para depurar programas em C++ usando consultas formuladas em OCL (Object Constraint Language).
- Uma lista de *frameworks* de teste para mais de 70 linguagens de programação pode ser encontrada em:
 - en.wikipedia.org/wiki/List_of_unit_testing_frameworks



Teste de Integração

- Testes de integração são feitos quando unidades, como classes, por exemplo, estão prontas e testadas isoladamente e precisam ser integradas em um *build* para gerar uma nova versão de um sistema.



Técnicas de integração

- *big bang*
- *bottom up*
- *top-down*
- *sandwich*



Integração big bang

- Consiste em construir as diferentes classes ou componentes separadamente e depois integrar tudo junto no final.
- É uma técnica não incremental, utilizada no ciclo de vida Cascata com Sub-Projetos.
- Tem como vantagem o alto grau de paralelismo que se pode obter durante o desenvolvimento e o fato de não precisar de *drivers* e *stubs* durante a integração, mas como desvantagem tem o fato de não ser incremental (portanto, inadequada para o Processo Unificado e métodos ágeis).
- Além disso, a integração de muitos componentes ao mesmo tempo pode dificultar bastante a localização dos defeitos, pois estes poderão estar em qualquer um dos componentes.



Integração bottom up

- Consiste em integrar inicialmente os módulos de mais baixo nível, ou seja, aqueles que não dependem de nenhum outro, e depois ir integrando os módulos de nível imediatamente mais alto.
- Assim, um módulo só é integrado quando todos os módulos dos quais ele depende já foram integrados e testados.
- Dessa forma não é necessário escrever *stubs*, mas em compensação as funcionalidades de mais alto nível do sistema somente serão testadas tarde, quando os módulos de nível superior forem finalmente integrados.



Integração top-down

- Consiste em integrar inicialmente os módulos de nível mais alto, deixando os mais básicos para o fim.
- A vantagem está em verificar inicialmente os comportamentos mais importantes do sistema onde repousam as maiores decisões.
- Mas como desvantagem está o fato de que muitos *stubs* são necessários, e que o teste, para ser efetivo, precisa de bons *stubs*, caso contrário, ao se integrarem os módulos de nível mais baixo poderão ocorrer problemas inesperados.



Integração sandwich

- Consiste em integrar os módulos de nível mais alto da forma *top-down* e os de nível mais baixo da forma *bottom-up*.
- Esta técnica reduz um pouco os problemas das duas estratégias anteriores, mas seu planejamento é mais complexo.



Teste de Sistema

- Visa verificar se a versão corrente do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário que executa uma série de operações de sistema em uma interface (não necessariamente gráfica) e sendo capaz de obter os resultados esperados.

Exemplo (caso de uso a ser testado)

Caso de Uso: Comprar livros

1. [IN] O comprador informa sua identificação.
2. [OUT] O sistema informa os livros disponíveis para venda (título, capa e preço) e o conteúdo atual do carrinho de compras (título, capa, preço e quantidade).
3. [IN] O comprador seleciona os livros que deseja comprar.
4. O comprador decide se finaliza a compra ou se guarda o carrinho:
 - 4.1 Variante: Finalizar a compra.
 - 4.2 Variante: Guardar carrinho.

Variante 4.1: Finalizar a compra

- 4.1.1. [OUT] O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas.
- 4.1.2. [IN] O comprador seleciona um endereço para entrega.
- 4.1.3. [OUT] O sistema informa o valor do frete e total geral, bem como a lista de cartões de crédito já cadastrados para pagamento.
- 4.1.4. [IN] O comprador seleciona um cartão de crédito.
- 4.1.5. [OUT] O sistema envia os dados do cartão e valor da venda para a operadora.
- 4.1.6. [IN] A operadora informa o código de autorização.
- 4.1.7. [OUT] O sistema informa o prazo de entrega.

Variante 4.2: Guardar carrinho

- 4.2.1 [OUT] O sistema informa o prazo (dias) em que o carrinho será mantido.

Exceção 1a: Comprador não cadastrado

- 1a.1 [IN] O comprador informa seu CPF, nome, endereço e telefone.
- Retorna ao passo 1.

Exceção 4.1.2a: Endereço consta como inválido.

- 4.1.2a.1 [IN] O comprador atualiza o endereço.
- Avança para o passo 4.1.2.

Exceção 4.1.6a: A operadora não autoriza a venda.

- 4.1.6a.1 [OUT] O sistema apresenta outras opções de cartão ao comprador.
 - 4.1.6a.2 [IN] O comprador seleciona outro cartão.
- Retorna ao passo 4.1.5.

Objetivo	Caminho	Como testar	Resultados
Fluxo principal com variante 4.1	1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7	Um cliente cadastrado informa livros válidos, indica um endereço e cartão válidos e a operadora (possivelmente um <i>stub</i>) autoriza a compra.	Compra efetuada.
Fluxo principal com variante 4.2	1, 2, 3, 4, 4.2, 4.2.1	Um cliente cadastrado informa livros válidos, e guarda o carrinho.	Carrinho guardado.
Exceção 1a	1, 1a, 1a.1, 1, 2, 3, 4, 4.2, 4.2.1	Um cliente não cadastrado informa livros válidos e guarda o carrinho.	Cliente é cadastrado e o carrinho guardado.
Exceção 4.1.2a	1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.2a, 4.1.2a.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7	Um cliente cadastrado informa livros válidos, indica um endereço inválido, e depois um endereço válido; indica um cartão válido e a operadora (possivelmente um <i>stub</i>) autoriza a compra.	O endereço inválido é atualizado e a compra efetuada.
Exceção 4.1.6a	1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.6a, 4.1.6a.1, 4.1.6a.2, 4.1.5, 4.1.6, 4.1.7	Um cliente cadastrado informa livros válidos, indica um endereço e cartão válidos e a operadora (possivelmente um <i>stub</i>) não autoriza a compra. O cliente informa outro cartão válido e a operadora autoriza a compra.	Compra efetuada.

Figura 13-1: Exemplo de plano de teste de sistema para um caso de uso.



Teste de Aceitação

- O *teste de aceitação* é usualmente realizado pelo usuário ou cliente, usando a interface final do sistema.
- Ele pode ser planejado e executado exatamente como o teste de sistema, mas a diferença é que é realizado pelo usuário final ou cliente e não pela equipe de desenvolvimento.

Variações

- *Teste alfa:*
 - Efetuado pelo cliente ou seu representante de forma livre, sem o planejamento e formalidade do teste de sistema.
 - O usuário vai livremente utilizar o sistema e suas funções e, por isso, este teste também é chamado de teste de aceitação informal, em oposição ao *teste de aceitação formal* que deveria seguir o mesmo planejamento utilizado pelo teste de sistema.
- *Teste beta:*
 - Este teste é ainda menos controlado pela equipe de desenvolvimento.
 - No teste beta, versões operacionais do software são disponibilizadas para vários usuários que, sem acompanhamento direto nem controle por parte da empresa desenvolvedora, vão explorar o sistema e suas funcionalidades.
 - Normalmente versões beta de sistemas expiram após um período pré-determinado, quando então os usuários são convidados a fazer uma avaliação do sistema.



Teste de Ciclo de Negócio

- É uma abordagem possível tanto no teste de sistema quanto no teste de aceitação formal, e consiste em testar uma sequência casos de uso que corresponde a um possível ciclo de negócio da empresa.
- Assim, ao invés de testar os casos de uso isoladamente, o analista ou cliente vai testá-los no contexto de um ciclo de negócio.



Teste de regressão

- O teste de regressão é executado sempre que um sistema em operação sofre manutenção.
- O problema é que a correção de um defeito no software, ou modificação de alguma de suas funções pode ter gerado novos defeitos.
- Neste caso, devem ser executados novamente todos os testes de unidade das unidades alteradas, bem como os testes de integração e sistema sobre as partes afetadas.
- O teste de regressão tem esse nome porque se ao se aplicarem testes a uma nova versão na qual versões anteriores passaram, e essa nova versão não passar, então se considera que o sistema *regrediu*.



Testes Suplementares

- Interface com Usuário
- Performance (Carga, *Stress* e Resistência)
- Segurança
- Recuperação de Falha
- Instalação



Teste de interface com usuário

- Tem como objetivo verificar se a interface permite realizar as atividades previstas nos casos de uso de forma eficaz e eficiente.
- Mesmo que as funções estejam corretamente implementadas, o que já teria sido visto no teste de sistema, isso não quer dizer que a interface o esteja.
- Então, em geral, é necessário testar a interface de forma objetiva e específica.
- O teste de interface com usuário pode ainda verificar a conformidade das interfaces com normas ergonômicas que se apliquem.
 - Por exemplo, a NBR ISO 9241-11:2011



Teste de Performance

- Tenha o sistema requisitos de desempenho ou não, o teste de performance pode ser importante, especialmente nas operações que serão realizadas com muita frequência ou de forma iterativa.
- O teste consiste em executar a operação e mensurar seu tempo, avaliando se está dentro dos padrões definidos.



Tipos de teste de performance

- Carga
- Stress
- Resistência



Teste de carga

- É a forma mais simples de teste de performance.
- Normalmente o teste de carga é feito para uma determinada quantidade de dados ou transações, que se espera sejam típicos para um sistema, e avalia o comportamento do sistema em termos de tempo para estes dados ou transações.
- Desta forma, pode-se verificar se o sistema atende aos requisitos de performance estabelecidos e também pode-se verificar se existem gargalos de performance para serem tratados.



Teste de stress

- É um caso extremo de teste de carga.
- Procura-se levar o sistema ao seu limite máximo esperado de funcionamento para verificar como se comporta.
- Este tipo de teste é feito para verificar se o sistema é suficientemente robusto frente a situações anormais de carga de trabalho.
- O teste também ajuda a verificar quais seriam os problemas encontrados caso a carga do sistema ficasse acima do limite máximo estabelecido.



Teste de resistência

- É feito para verificar se o sistema consegue manter suas características de performance durante um longo período de tempo com uma carga nominal de trabalho.
- Os testes de resistência devem verificar basicamente o uso da memória ao longo do tempo para garantir que não existam perdas acumulativas de memória em função de lixo não recolhido, e também deverão verificar se não existe degradação de performance após um substancial período de tempo em que o sistema opera com carga nominal ou acima desta.

Testes de Segurança

- *Integridade:*
 - é uma forma de garantir ao receptor que a informação que ele recebeu é correta e completa.
- *Autenticação:*
 - é a garantia de que um usuário realmente é quem ele diz ser e que os documentos, programas e sites realmente sejam aqueles que se espera que sejam.
- *Autorização:*
 - é o processo de verificar se alguma pessoa ou sistema pode ou não acessar determinada informação ou sistema.
- *Confidencialidade:*
 - segurança de que quem não tem direito à informação não possa obtê-la.
- *Disponibilidade:*
 - é a segurança de quem tem direito à informação consiga obtê-la quando necessário.
- *Não repúdio:*
 - é uma forma de garantir que o emissor e receptor de uma mensagem não possam posteriormente alegar não ter enviado ou recebido a mensagem.



Teste de recuperação de falha

- Quando um sistema tem requisitos suplementares referentes a tolerância ou recuperação de falhas, estes devem ser testados separadamente.
- Basicamente, busca-se verificar se o sistema de fato atende aos requisitos especificados relacionados a esta questão.
- Normalmente trata-se de situações referentes a:
 - Queda de energia no cliente ou no servidor.
 - Discos corrompidos.
 - Problemas de comunicação.
 - Quaisquer outras condições que possam potencialmente provocar a terminação anormal do programa ou a interrupção temporária de seu funcionamento.



Teste de instalação

- Basicamente busca-se no teste de instalação verificar se o software não entra em conflito com outros sistemas eventualmente instalados em uma máquina, bem como se todas as informações e produtos para instalação estão disponíveis para os usuários instaladores.
- O teste de instalação também é associado com o teste de compatibilidade, onde se busca verificar se o sistema é compatível com diferentes sistemas operacionais, fabricantes de máquinas, *browsers* etc.



Técnicas de Teste

- *Testes **estruturais** ou caixa-branca:*
 - são testes que são executados com conhecimento do código implementado, ou seja, eles testam a estrutura do programa em si.
- *Testes **funcionais** ou caixa-preta:*
 - são testes executados sobre as entradas e saídas do programa sem que se tenha necessariamente conhecimento do seu código fonte.



Teste Estrutural

- Útil para teste de unidade
- Cada estrutura de controle deve ser testada para suas diferentes opções.



Complexidade Ciclomática

- Mede a complexidade de um programa em relação à quantidade de testes que devem ser feitos.
- Definição simplificada: é o número de comandos com ramificação de controle (branches) mais 1.

O que conta?

- IF-THEN: 1 ponto.
- IF-THEN-ELSE: 1 ponto.
- CASE: 1 ponto para cada opção, exceto OTHERWISE.
- FOR: 1 ponto.
- REPEAT: 1 ponto.
- OR ou AND na condição de qualquer das estruturas acima: acrescenta-se 1 ponto para cada OR ou AND (ou qualquer outro operador lógico **binário**, se a linguagem suportar, como XOR ou IMPLIES).
- NOT: não conta.
- Chamada de sub-rotina (inclusive recursiva): não conta.
- Estruturas de seleção e repetição em sub-rotinas ou programas chamados: não conta.

Testabilidade

- Complexidade ciclomática:
 - < 10 – fácil
 - 10 a 20 – médio risco
 - 20 a 50 – alto risco
 - > 50 – não testável

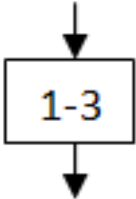
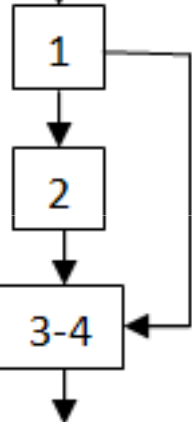
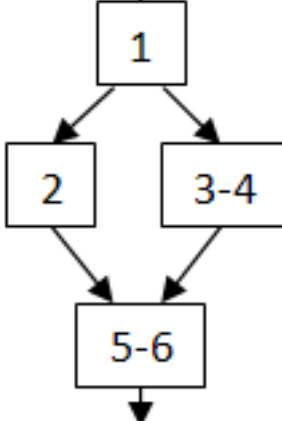
Exemplo: qual a complexidade ciclomática do programa abaixo?

```
01.  if num = 0 then
02.    fib := 0
03.  else
04.    if num = 1 then
05.      fib := 1
06.    else
07.      begin
08.        ultimoFib := 1;
09.        cont := 1;
10.        repeat
11.          penultimoFib := ultimoFib;
12.          ultimoFib := fib;
13.          fib := penultimoFib + ultimoFib;
14.          cont := cont + 1;
15.        until cont = num;
16.      end
17.    ;
18.  ;
```



Grafo de fluxo

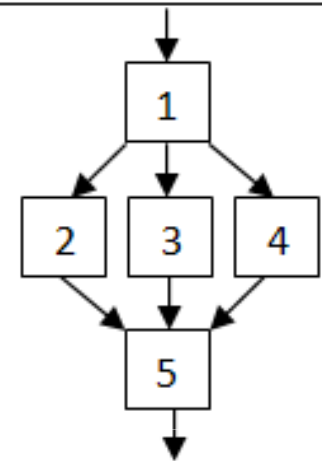
- O *grafo de fluxo* de um programa é obtido colocando-se todos os comandos em nós e os fluxos de controle em arestas.
- Comandos em sequência podem ser colocados em um único nó, e estruturas de seleção e repetição devem ser representadas através de nós distintos com arestas que indicam a decisão e a repetição, quando for o caso.

Regra	Código	Grafo
Comandos em sequência (dois ou mais)	01. <...>; 02. <...>; 03. <...>;	 <pre> graph TD Start(()) --> B1[1-3] B1 --> B2[2-3] B2 --> B3[3-4] B3 --> End(()) </pre>
If-Then	01. IF condição THEN 02. <...>; 03. ENDIF; 04. <...>;	 <pre> graph TD Start(()) --> B1[1] B1 --> B2[2] B1 --> B3[3-4] B2 --> B3 B3 --> End(()) </pre>
If-Then-Else	01. IF condição THEN 02. <...>; 03. ELSE 04. <...>; 05. ENDIF; 06. <...>;	 <pre> graph TD Start(()) --> B1[1] B1 --> B2[2] B1 --> B3[3-4] B2 --> B4[5-6] B3 --> B4 B4 --> End(()) </pre>

For	<pre> 01. FOR condição DO 02. <...>; 03. ENDFOR; 04. <...>; </pre>	<pre> graph TD Entry(()) --> 1[1] 1 --> 23[2-3] 23 --> 4[4] 4 --> Exit(()) 4 --> 1 </pre>
While	<pre> 01. WHILE condição DO 02. <...>; 03. ENDWHILE; 04. <...>; </pre>	<pre> graph TD Entry(()) --> 1[1] 1 --> 23[2-3] 23 --> 4[4] 4 --> Exit(()) 4 --> 1 </pre>
Repeat	<pre> 01. REPEAT 02. <...>; 03. UNTIL condição; 04. <...>; </pre>	<pre> graph TD Entry(()) --> 1[1] 1 --> 23[2-3] 23 --> 4[4] 4 --> Exit(()) 4 --> 1 </pre>

Case (para qualquer quantidade de casos)

```
01. CASE X OF  
02.   a: <...>;  
03.   b: <...>;  
04. OTHERWISE <...>;  
05. <...>;
```




```
01. IF cond1 OR cond2 THEN
02.   <...>;
03. ELSE
04.   <...>;
05. ENDIF;
06. <...>;
```

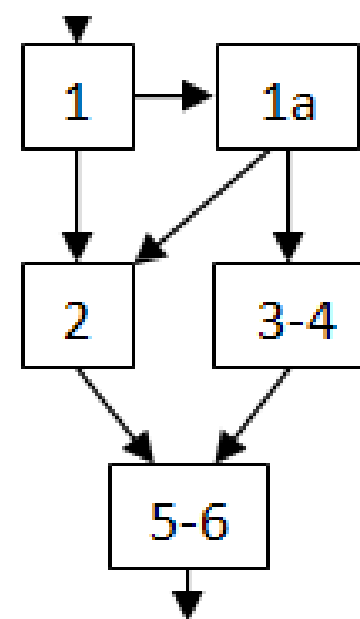


Figura 13-4: Regra para criação de grafo de fluxo para estruturas com condição OR.

```
01. IF cond1 AND cond2 THEN
02.   c1;
03. ELSE
04.   c2;
05. ENDIF;
06. c3;
```

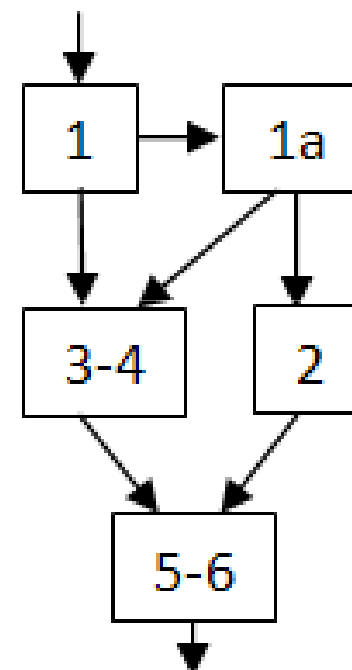


Figura 13-5: Regra para criação de grafo de fluxo para estruturas com condição AND.

Retornando ao exemplo

```
01.  if num = 0 then
02.    fib := 0
03.  else
04.    if num = 1 then
05.      fib := 1
06.    else
07.      begin
08.        ultimoFib := 1;
09.        cont := 1;
10.        repeat
11.          penultimoFib := ultimoFib;
12.          ultimoFib := fib;
13.          fib := penultimoFib + ultimoFib;
14.          cont := cont + 1;
15.        until cont = num;
16.      end
17.    ;
18.  ;
```

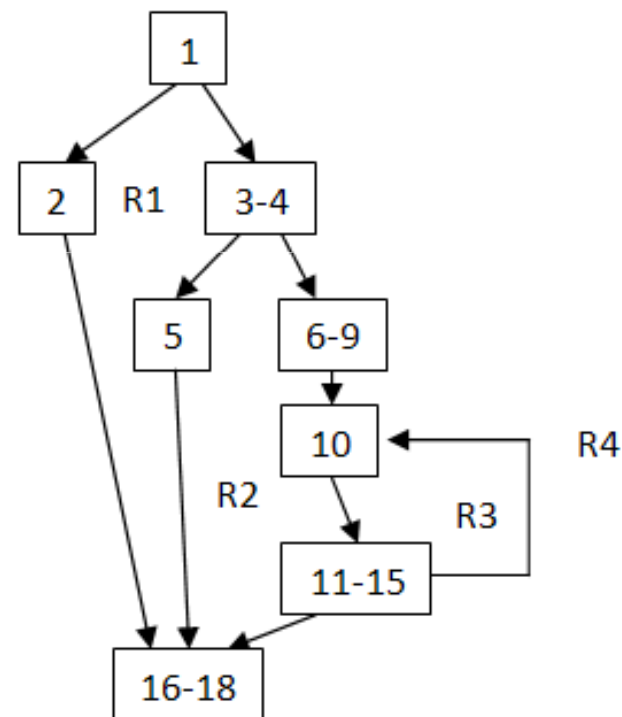



Figura 13-6: Grafo de fluxo do programa da Figura 13-2.



Caminhos Independentes

- O valor da complexidade ciclomática indica número *máximo* de execuções *necessárias* para exercitar todos os comandos do programa.
- Não apenas todos os comandos devem ser testados, mas todas as condições de controle.
- Assim, deve-se passar não só por todos os nodos do grafo de controle, mas por todas as arestas.
- Isso é feito pela determinação dos *caminhos independentes* do grafo, que são possíveis navegações do início ao fim do grafo.



Algoritmo para encontrar os caminhos independentes

- Inicialize o conjunto dos caminhos independentes com um caminho qualquer do início ao fim do grafo (usualmente pode ser o caminho mais curto).
- Enquanto for possível adicione ao conjunto dos caminhos independentes outros caminhos que passem por pelo menos uma aresta na qual nenhum dos caminhos anteriores ainda passou.

Exemplo

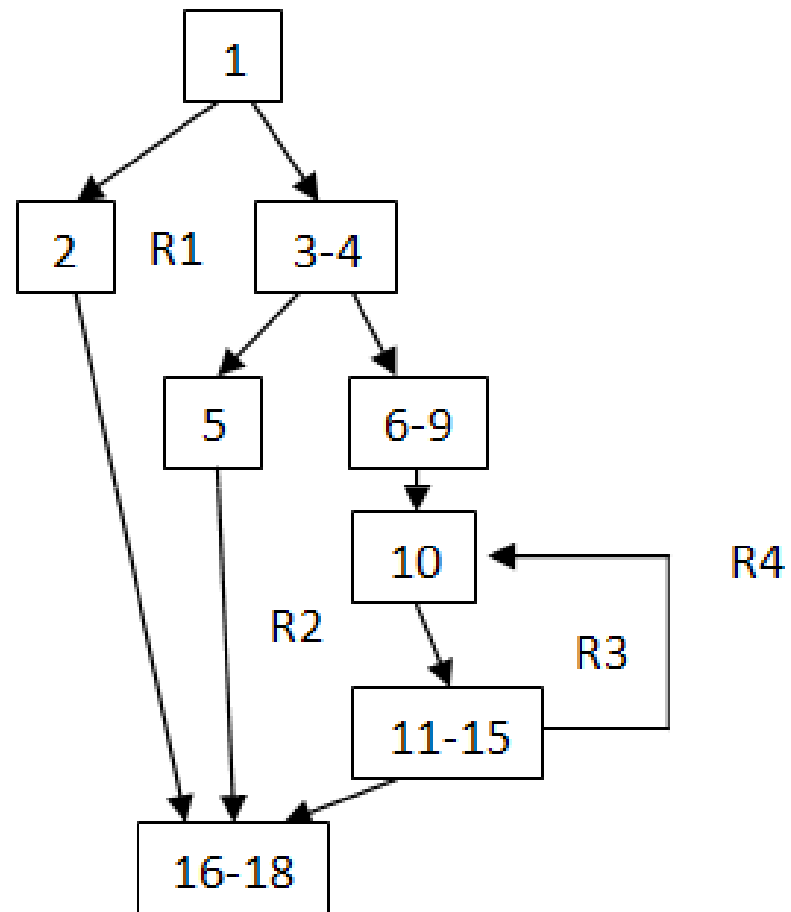


Figura 13-6: Grafo de fluxo do programa da Figura 13-2.

Exemplo

O caminho $c_1 = \langle 1, 2, 16-18 \rangle$.

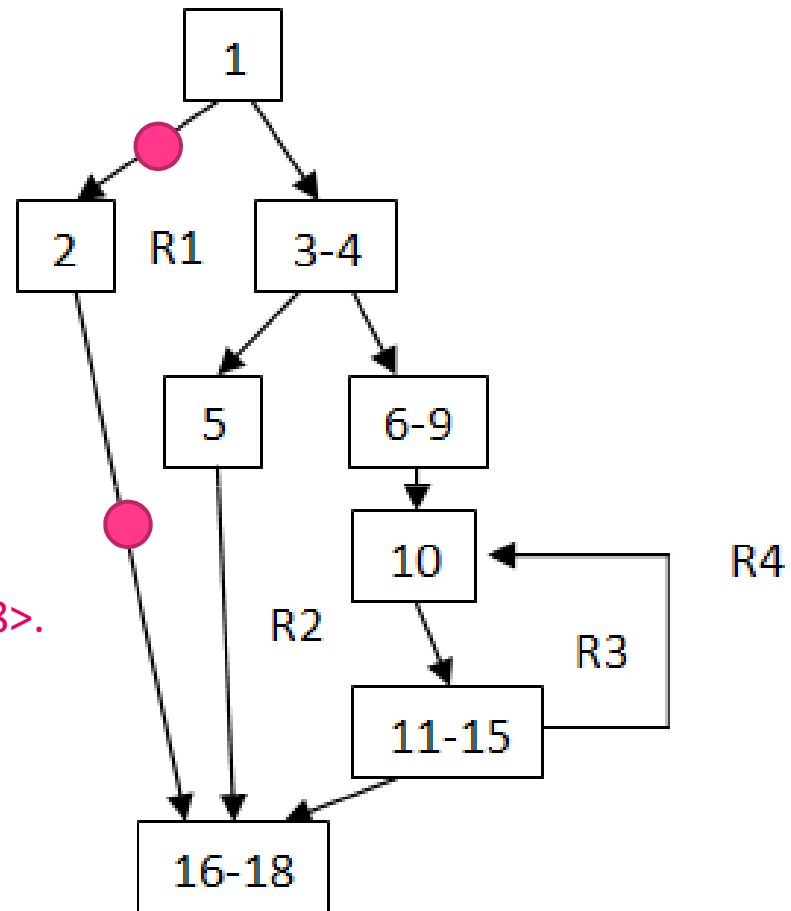


Figura 13-6: Grafo de fluxo do programa da Figura 13-2.

Exemplo

O caminho $c_1 = \langle 1, 2, 16-18 \rangle$.
O caminho $c_2 = \langle 1, 3-4, 5, 16-18 \rangle$.

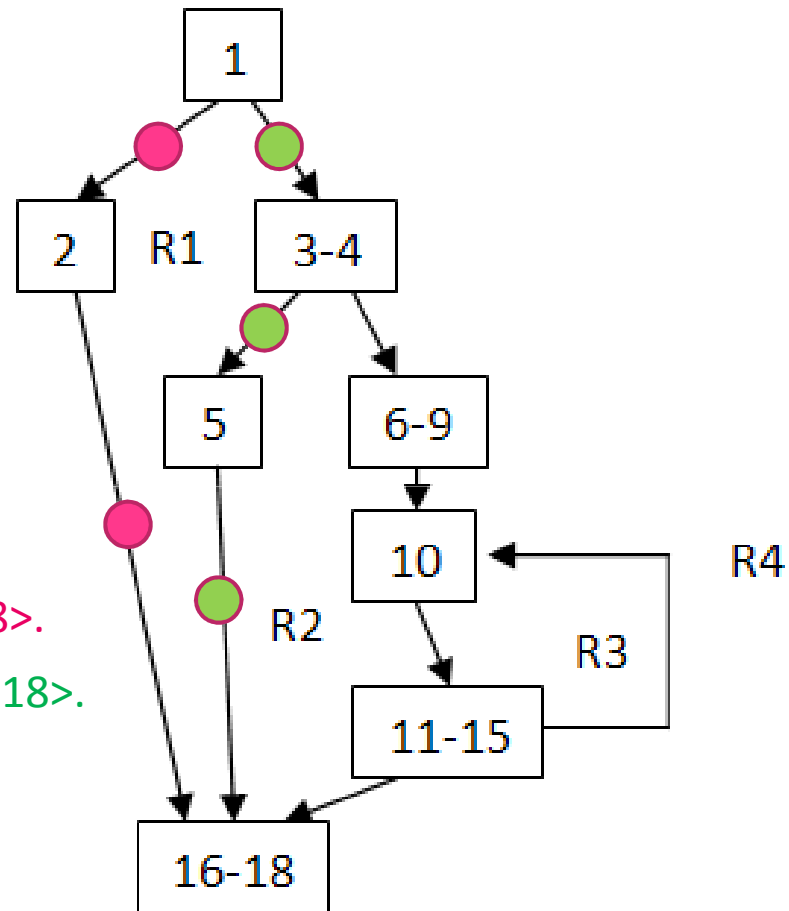


Figura 13-6: Grafo de fluxo do programa da Figura 13-2.

Exemplo

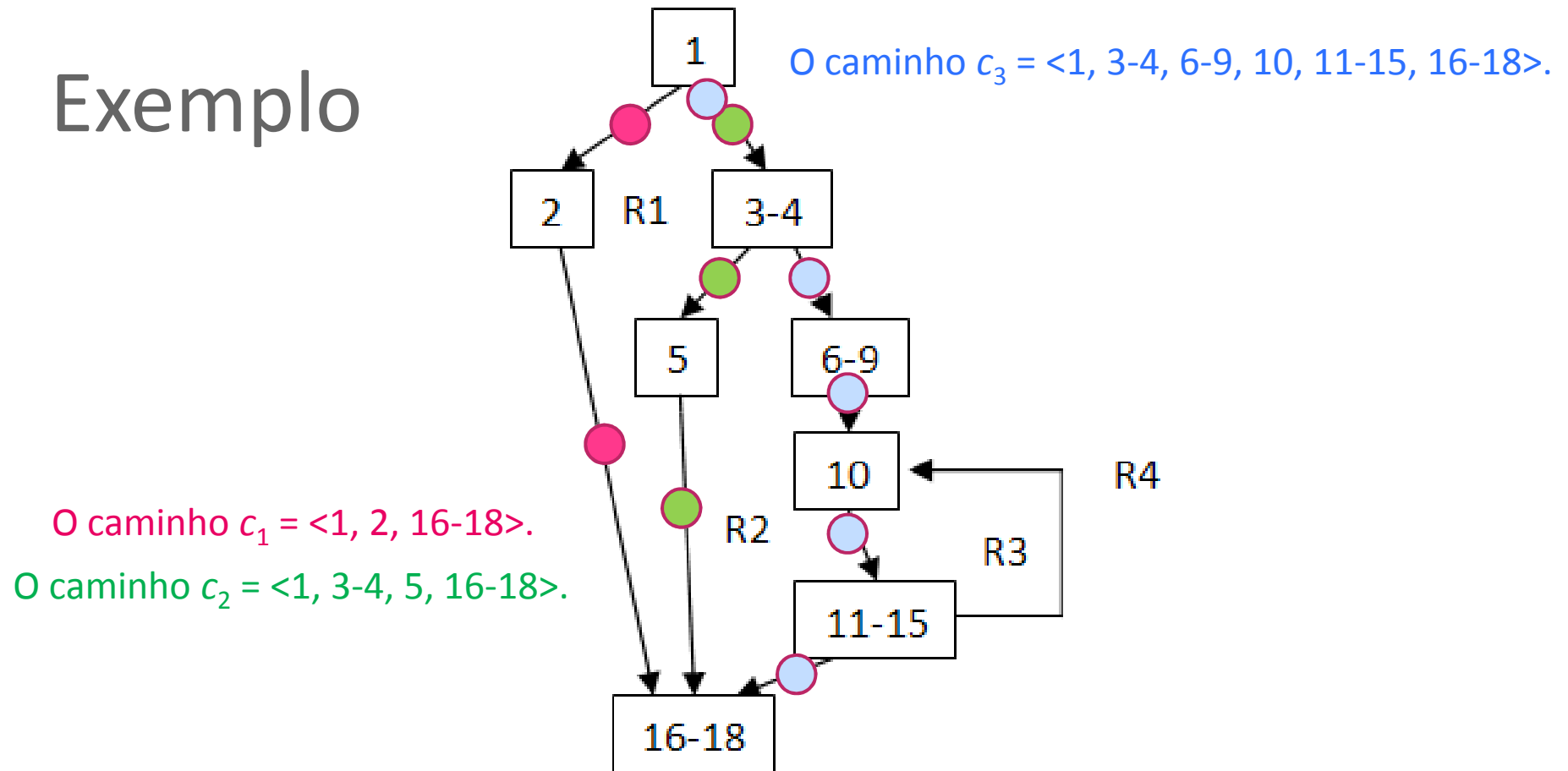


Figura 13-6: Grafo de fluxo do programa da Figura 13-2.

Exemplo

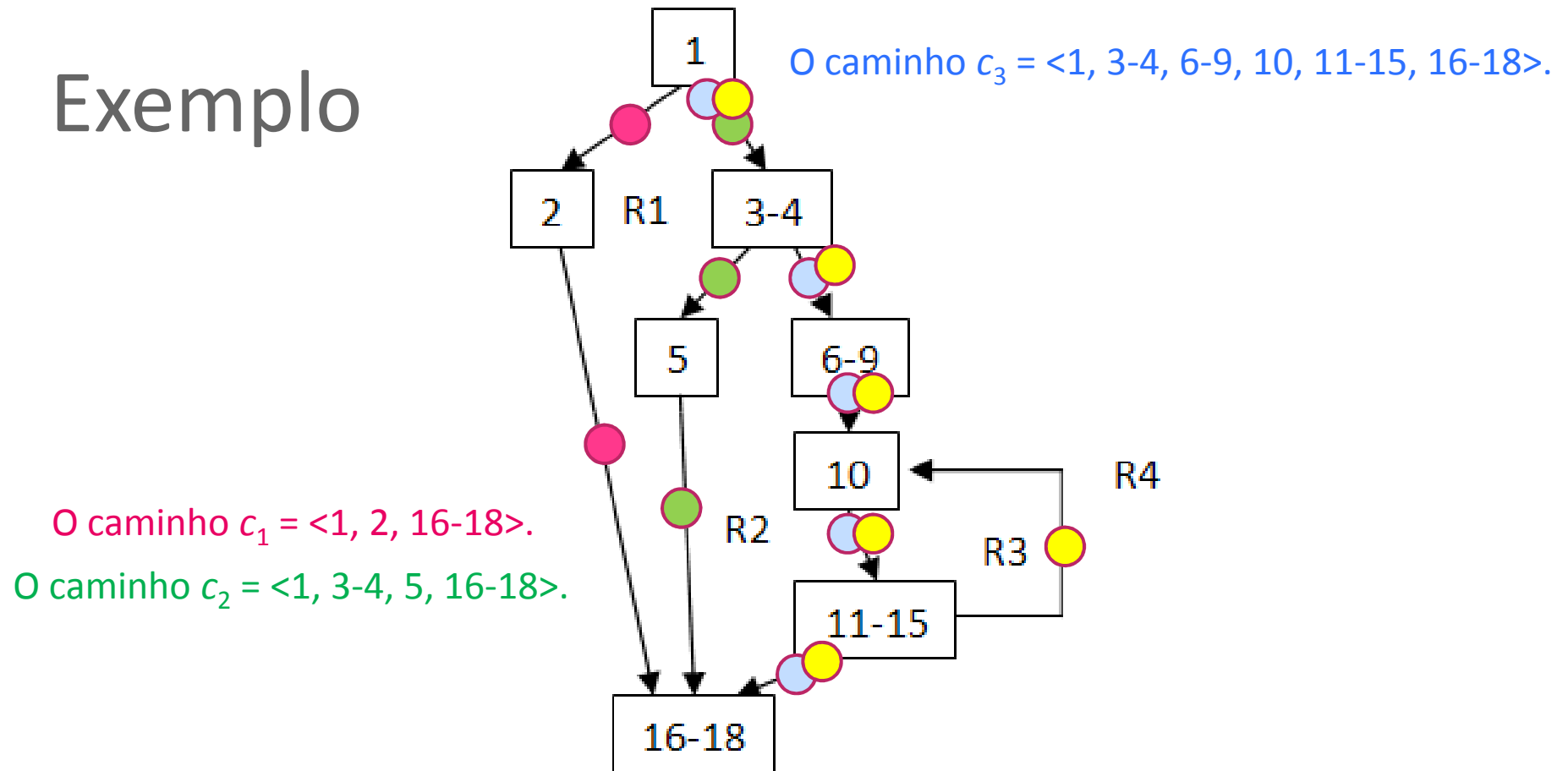


Figura 13-6: Grafo de fluxo do programa da Figura 13-2.

O caminho $c_4 = \langle 1, 3-4, 6-9, 10, 11-15, 10, 11-15, 16-18 \rangle$.



Casos de Teste

- Resta ainda definir os *casos de teste*, ou seja, quais dados de entrada levam o programa a executar cada um dos caminhos independentes e qual a saída esperada do programa para cada um destes casos.

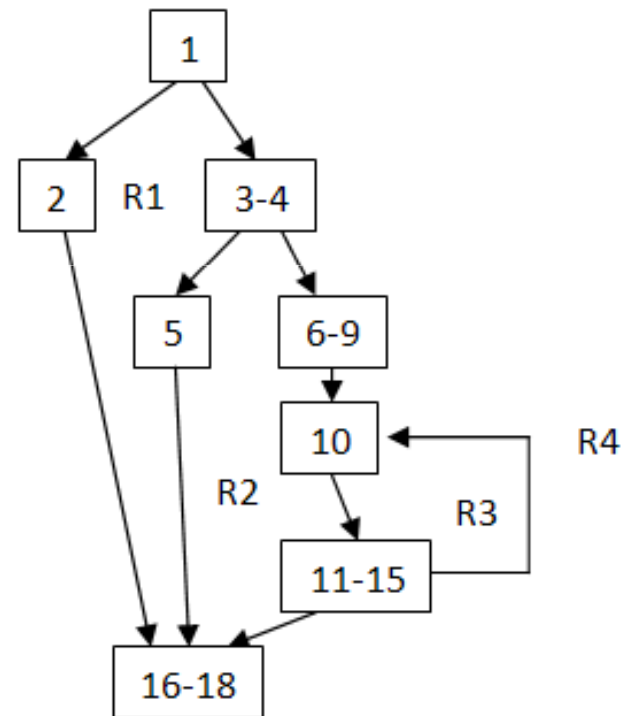
Exemplo

```

01.  if num = 0 then
02.    fib := 0
03.  else
04.    if num = 1 then
05.      fib := 1
06.    else
07.      begin
08.        ultimoFib := 1;
09.        cont := 1;
10.        repeat
11.          penultimoFib := ultimoFib;
12.          ultimoFib := fib;
13.          fib := penultimoFib + ultimoFib;
14.          cont := cont + 1;
15.        until cont = num;
16.      end
17.    ;
18.  ;

```

Caminho	Entrada	Saída esperada
c_1	num=0	fib=0
c_2	num=1	fib=1
c_3	num=2	fib=1
c_4	num=3	fib=2



$c_1 = \langle 1, 2, 16-18 \rangle$.

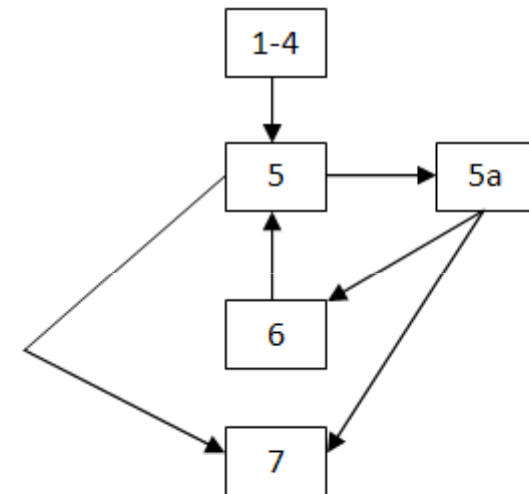
$c_2 = \langle 1, 3-4, 5, 16-18 \rangle$.

$c_3 = \langle 1, 3-4, 6-9, 10, 11-15, 16-18 \rangle$.

$c_4 = \langle 1, 3-4, 6-9, 10, 11-15, 10, 11-15, 16-18 \rangle$.

Exemplo com múltiplas condições

```
01. var anos : array [1..6] of integer;  
02.     coluna : integer;  
03. begin  
04.     coluna := 1;  
05.     while (anos[coluna] <> 1996) and (coluna < 6) do  
06.         coluna := coluna + 1;  
07. end
```



Caminho independente	Entrada	Saída esperada
c_1	anos=<1996, 0, 0, 0, 0, 0>	coluna=1
c_2	anos=<0, 0, 0, 0, 0, 0>	coluna=7
c_3	anos=<0, 1996, 0, 0, 0, 0>	coluna=2

c_1 : 1-4, 5, 7.

c_2 : 1-4, 5, 5a, 7.

c_3 : 1-4, 5, 5a, 6, 5, 7.

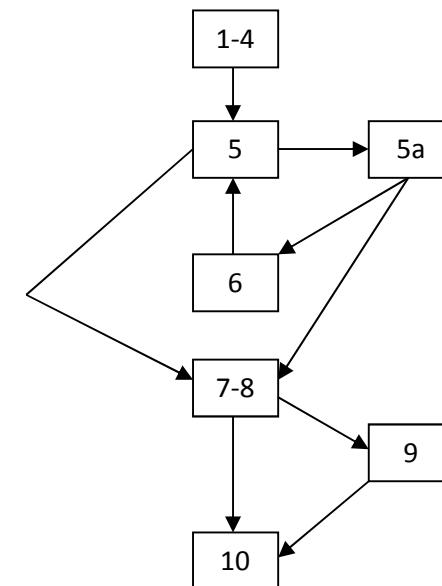


Caminhos impossíveis

- Algumas vezes, certos caminhos do grafo de fluxo simplesmente são impossíveis de testar porque a lógica do programa torna impossível passar por eles.

Exemplo de programa com caminhos impossíveis

```
01.  var anos : array [1..6] of integer;  
02.      coluna : integer;  
03.  begin  
04.      coluna := 1;  
05.      while (anos[coluna] <> 1996) and (coluna < 6) do  
06.          coluna := coluna + 1  
07.      ;  
08.      if (coluna = 6) and (anos[coluna] <> 1996) then  
09.          coluna := 7;  
10.  end
```





Limitações do Teste Estrutural

- Este tipo de teste não é capaz de identificar se o programa foi bem especificado.
 - Ou seja, o teste vai verificar se o programa se comporta de acordo com a especificação dada, mas não necessariamente se ele se comporta de acordo com a especificação *esperada*.
- Ele não necessariamente cobre potenciais problemas com estruturas de dados como *arrays* e listas.
- Ele não necessariamente trata situações típicas de programas orientados a objetos, especialmente quando se usa herança e polimorfismo.

Limitações do Teste Estrutural


- Funcionalidades ausentes não são testadas, pois a técnica preconiza apenas o teste daquilo que existe no programa.
 - Ou seja, se algum comando que *deveria* ter sido incluído no código, mas não o foi, o teste estrutural não será capaz de identificar isso.

```
01.  if num = 0 then
02.    fib := 0
03.  else
04.    if num = 1 then
05.      fib := 1
06.    else
07.      begin
08.        ultimoFib := 1;
09.        cont := 1;
10.        repeat
11.          penultimoFib := ultimoFib;
12.          ultimoFib := fib;
13.          fib := penultimoFib + ultimoFib;
14.          cont := cont + 1;
15.        until cont = num;
16.      end
17.    ;
18.  ;
```



Limitações do Teste Estrutural


- Algumas vezes o programa pode estar produzindo o resultado correto para uma entrada por mera coincidência, não significando que esteja correto.
- Só pode ser produzido *depois* que o código está escrito, o que não habilita seu uso com a técnica de **desenvolvimento dirigido pelo teste** que sugere que os casos de teste sejam definidos antes de se escrever o código.


- 
- Apesar dessas limitações a técnica é relevante e seu uso é importante para a detecção de vários tipos de defeitos no software, especialmente nas suas unidades mais básicas, pois ele é usado para garantir que todos os comandos e condições lógicas sejam executados pelo menos uma vez.




Teste Funcional

- Em várias situações a necessidade consiste em verificar a funcionalidade de um programa independentemente de sua estrutura interna.
- Um programa pode ter uma especificação, ou comportamento esperado, usualmente elaborado em um contrato de operação de sistema e o que se deseja saber é se ele efetivamente cumpre este contrato ou especificação.

- 
- Assim, no nível de integração de funções mais básicas do software será mais adequado realizar *testes funcionais*, que avaliam o comportamento de uma operação mais abrangente do que as operações elementares, porque essa técnica pode avaliar comportamentos que estão distribuídos em várias classes (coisa que a técnica estrutural não faz diretamente).

- 
- O padrão MVC (*Model View Control*), bastante empregado em sistemas de informação, sugere que um sistema deva ser dividido em pelo menos três camadas, sendo que a superior (*View*) corresponde à interface com usuário, e a intermediária (*Control*) é a que efetivamente se responsabiliza pela execução de processamento lógico sobre a informação.
 - Com o uso dessa técnica, todas as transformações da informação vão ocorrer encapsuladas pela controladora (uma classe com a função específica de encapsular estes comportamentos), e serão acessíveis pela interface através de chamadas a métodos dessa classe.
 - Quando os métodos fazem alteração de dados, são chamados de *operações de sistema* e quando fazem consulta a dados são chamados de *consultas de sistema*.

- 
- Essas operações e consultas devem ser especificadas por contratos bem definidos com parâmetros tipados, pré-condições, pós-condições e exceções.
 - O teste funcional então consistirá em verificar se em situação de normalidade (pré-condições atendidas) as pós-condições desejadas são realmente obtidas, e se em situações de anormalidade as exceções são efetivamente levantadas.



Particionamento de Equivalência

- Um dos princípios do teste funcional é a identificação de situações equivalentes.
 - Por exemplo, se um programa aceita um conjunto de dados (normalidade) e rejeita outro conjunto (exceção) então se pode dizer que existem duas *classes de equivalência* para os dados de entrada do programa, os dados aceitos e os dados rejeitados.
 - Pode ser impossível testar todos os elementos de cada conjunto, até porque esses conjuntos podem ser infinitos.
 - Então, o **particionamento de equivalência** vai determinar que pelo menos um elemento de cada conjunto seja testado.

Definição da Técnica 1/4

- Se as entradas válidas são especificadas como um ***intervalo de valores*** (por exemplo, de 10 a 20), então é definido:
 - um conjunto **válido** (10 a 20) e
 - dois **inválidos** (menor do que 10 e maior do que 20).

Definição da Técnica 2/4

- Se as entradas válidas são especificadas como uma ***quantidade de valores*** (por exemplo, uma lista com cinco elementos), então é definido:
 - um conjunto **válido** (lista com 5 elementos) e
 - dois **inválidos** (lista com menos de 5 elementos e lista com mais de 5 elementos).

Definição da Técnica 3/4

- Se as entradas válidas são especificadas como um *conjunto de valores aceitáveis* que podem ser tratados de forma diferente (por exemplo, os *strings* “masculino” e “feminino”), então é definido:
 - um conjunto **válido** para cada uma das formas de tratamento e
 - um conjunto **inválido** para outros valores quaisquer.

Definição da Técnica 4/4

- Se as entradas válidas são especificadas como uma condição do tipo “deve ser de tal forma” (por exemplo, uma restrição sobre os dados de entrada como “a data final deve ser posterior a data inicial”), então deve ser definido:
 - um conjunto **válido** (quando a condição é verdadeira) e
 - um **inválido** (quando a condição é falsa).

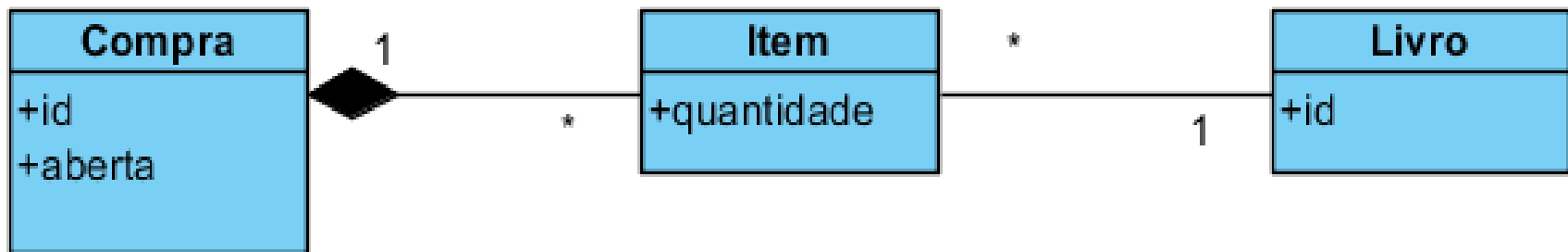


Sobre as saídas

- Os conjuntos de valores válidos devem ser definidos não só em termos de restrições sobre as entradas, mas também em função dos resultados a serem produzidos.
- Se a operação a ser testada puder ter dois comportamentos possíveis em função do valor de um dos parâmetros, então vão existir dois conjuntos distintos de valores válidos para aquele parâmetro, um para cada comportamento possível.

Exemplo

- adicionaLivro(idLivro, idCompra, quant)




O objetivo da operação consiste em adicionar na compra indicada um item que associe o livro indicado e uma quantidade.

Conjuntos válidos e inválidos

adicionaLivro(idLivro, idCompra, quant)

- Para idLivro:
 - **Válido:** se existe um livro com este id.
 - **Inválido:** se tal livro não existe.
- Para idCompra:
 - **Válido:** se existe uma compra com este id e ela está aberta
 - **Inválido:** caso a compra não exista.
 - **Inválido:** se a compra já está fechada.
- Para quantidade:
 - **Válido:** se for um valor maior ou igual a 1.
 - **Inválido:** para zero.

- 
- Em relação aos resultados da operação, pode-se ainda considerar que, se o livro já consta na compra, então ao invés de criar um novo item, deve-se somar a quantidade solicitada com a quantidade que já consta na compra.
 - Ou seja, existem dois comportamentos possíveis para um parâmetro de entrada, dependendo do valor deste e do estado interno dos objetos.
 - Neste caso, pode-se subdividir o conjunto de valores válidos relacionados com idLivro em dois conjuntos válidos:
 - um quando o idLivro passado não consta na compra e
 - outro quando o idLivro passado já consta na compra.



Então,

- Para idLivro:
 - *Válido: se existe um livro com este id e ele não consta na compra.*
 - *Válido: se existe um livro com este id e ele já consta na compra.*
 - *Inválido: se tal livro não existe.*

Plano de teste funcional para “adicionaLivro”

Tabela 13-3: Casos de teste funcional para uma operação de sistema.

Tipo	idLivro	idCompra	quant	Resultado esperado
Sucesso	um id já cadastrado que não consta na compra	um id já cadastrado de compra aberta	qualquer valor acima de 0	um item é criado e associado à compra e ao livro, com atributo quantidade = quant
Sucesso	um id já cadastrado que consta na compra	um id já cadastrado de compra aberta	qualquer valor acima de 0	o item do livro que já consta na compra tem seu valor incrementado com quant
Exceção	um id não cadastrado	um id já cadastrado de compra aberta	qualquer valor acima de 0	exceção: livro não cadastrado
Exceção	um id já cadastrado	um id não cadastrado	qualquer valor acima de 0	exceção: compra não cadastrada
Exceção	um id já cadastrado	um id já cadastrado de compra fechada	qualquer valor acima de 0	exceção: compra fechada
Exceção	um id já cadastrado	um id já cadastrado de compra aberta	0	exceção: quantidade não pode ser zero

Análise de Valor Limite



- Consiste em considerar não apenas um valor qualquer para teste dentro de uma classe de equivalência, mas um ou mais valores fronteiros com outras classes de equivalência quando isso puder ser determinado.

Valor limite

- Em domínios ordenados (números inteiros, por exemplo), esse critério pode ser aplicado.
- Por exemplo, se um programa exige uma entrada que para ser válida deve estar no intervalo $[n..m]$, então existem três classes de equivalência:
 - Inválida para qualquer $x < n$.
 - Válida para qualquer $x \geq n$ e $x \leq m$.
 - Inválida para qualquer $x > m$.

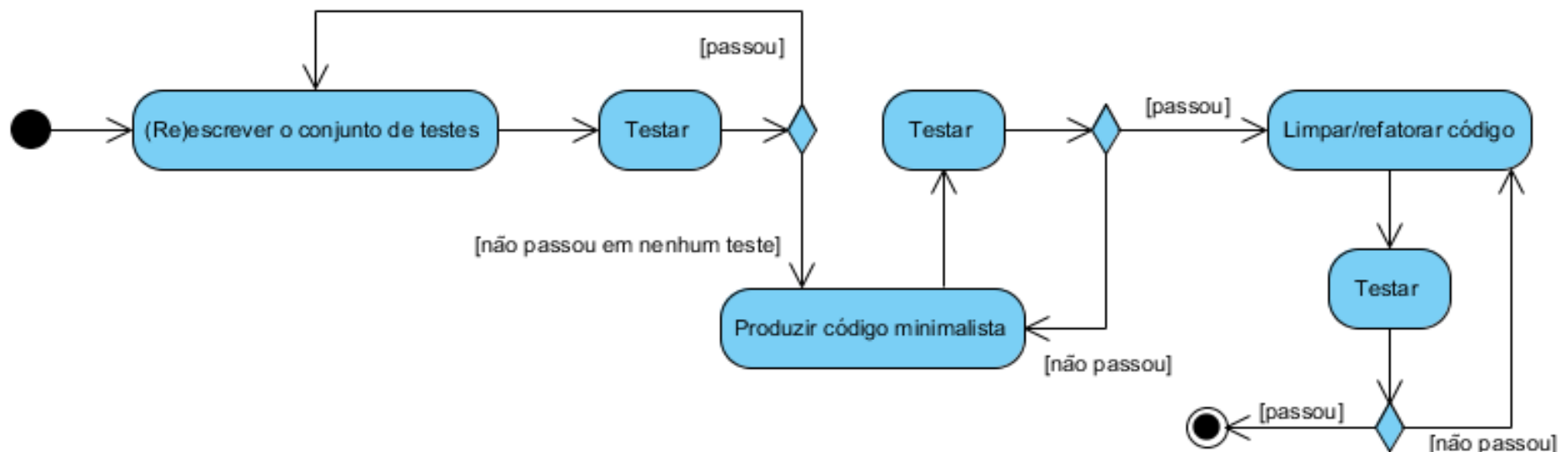


... $n-1$ [$n \dots m$] $m+1$...

- A análise de valor limite sugere que possíveis erros de lógica do programa não vão ocorrer em pontos arbitrários dentro desses intervalos, mas nos pontos onde um intervalo se encontra com outro.
- Então:
 - Para a primeira classe inválida, deve-se testar para o valor $n-1$.
 - Para a classe válida, deve-se testar os valores n e m .
 - Para a segunda classe inválida, deve-se testar para o valor $m+1$.

TDD – Desenvolvimento Orientado a Testes


- É uma técnica ou filosofia de programação que incorpora o teste no processo de produção de código da seguinte forma:





KISS e YAGNI

- Uma vez que os testes foram estabelecidos, o programador não deve implementar nenhuma funcionalidade além daquelas para as quais o teste foi criado.
- Isso evita que os programadores percam tempo criando estruturas que efetivamente não eram necessárias desde o início.
- Isso satisfaz os princípios:
 - *KISS (Keep it Simple, Stupid!)*
 - *YAGNI (You Ain't Gonna Need It)*

- 
- A técnica é bastante exigente em relação ao processo a ser seguido.
 - Um programador, por exemplo, que descubra a necessidade de adicionar um ELSE a um IF em um código já aprovado nos testes, deverá, antes de escrever código, escrever um teste para este ELSE, certificar-se que o teste falha para o programa atual, e somente depois adicionar o ELSE no código.
 - Pular etapas não é encorajado pela técnica




Medição em Teste

- *Métricas do processo de teste.*
 - As medidas relacionadas reportam a quantidade de testes requeridos, planejados, executados, etc., mas não o estado do produto em si.
- *Métricas de teste do produto.*
 - As medidas relacionadas reportam o estado do produto em relação à atividade de teste, como por exemplo, quantos defeitos foram encontrados, quantos estão em revisão, quantos foram resolvidos, etc.



Exemplos de métricas de processo de teste

- Número de testes previstos (sua necessidade já foi identificada).
- Número de testes planejados (casos de teste definidos).
- Número de testes executados, incluindo (a) testes que falharam e (b) testes que foram aprovados.



Exemplos de métricas relativas ao produto em teste

- Número de defeitos descobertos.
- Número de defeitos corrigidos.
- Distribuição dos defeitos por grau de severidade.
- Distribuição dos defeitos por módulo.


Exemplos de métricas compostas

- *Custo para encontrar um defeito* =
 - $\text{esforço total com atividades de teste} / \text{número de defeitos encontrados.}$
- *Adequação dos casos de teste* =
 - $\text{número de casos de teste real} / \text{número de casos de teste estimado.}$
- *Efetividade dos casos de teste* =
 - $\text{número de defeitos encontrados através de testes} / \text{número total de defeitos encontrados.}$



Depuração

- Depuração é o processo de remover defeitos de um programa.
- Ela normalmente inicia com atividades de teste, que identificam o problema, ou a partir de inspeções de código, ou ainda, a partir de relatos de usuários.
- No último caso, a equipe deve, no início do processo de depuração, tentar reproduzir o defeito relatado pelo usuário, o que nem sempre é possível ou fácil.

- 
- A depuração é uma atividade ainda artesanal, onde a quantidade e variedade de casos dificulta a elaboração de um processo padrão.
 - Porém, a adoção de boas técnicas de engenharia de software, como, integrações frequentes, controle de versões, desenvolvimento orientado a testes, etc., podem facilitar bastante o processo de descoberta e correção de defeitos.

quantidade

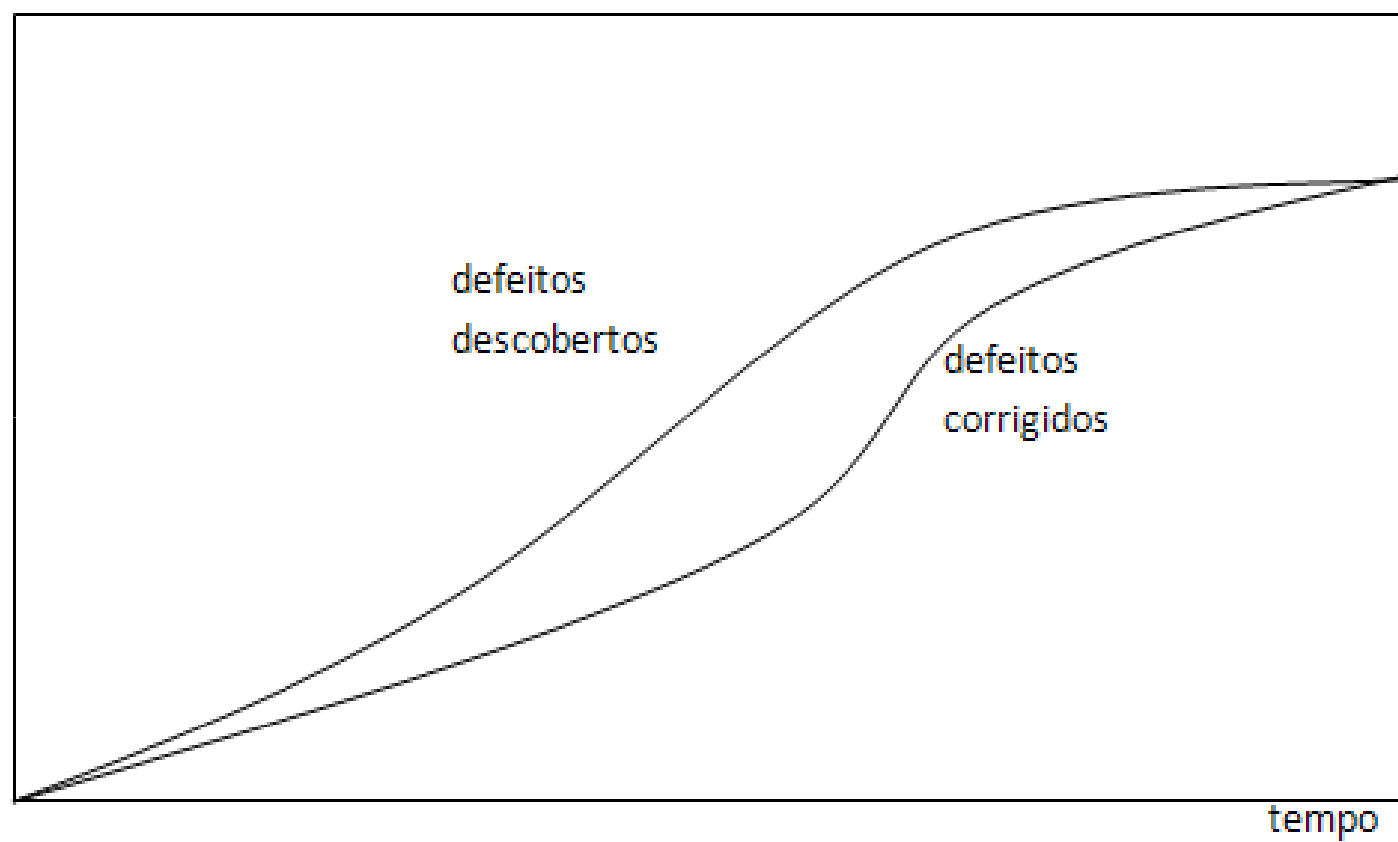


Figura 13-13: Evolução esperada das medidas de defeitos de produto ao longo de um projeto.