Fundamentos Engenharia de Software

UNIDADE 07

Testes estruturais

Esta Unidade apresenta a técnica de estrutural. O início se dá com uma introdução a respeito do objetivo principal da técnica e esclarecimento da diferença entre os testes funcionais e estruturais. Ao longo da Unidade, estão apresentados em detalhes os conceitos e exemplos de aplicação da técnica de teste estrutural caminho básico. Para finalizar, é apresentado o conceito de Test Driven Development (TDD) e as possibilidades de ferramentas para automação dos testes.

Introdução ao teste estrutural

Na Unidade anterior, aprendemos que os testes funcionais não dependem de código implementado, pois podem ser gerados a partir das especificações de *softwar*e. Já o teste estrutural, também conhecido como teste de caixa-branca, tem como foco o código implementado e procura garantir a cobertura do código.

O teste estrutural avalia o comportamento interno do componente de software e pode ajudar a responder à pergunta-chave:

> **Quais casos de testes adicionais são necessários para revelar falhas que não são aparentes usando somente testes de caixa-preta?**

As técnicas de teste estrutural, também chamadas de testes, se baseiam na estrutura de um componente ou sistema.

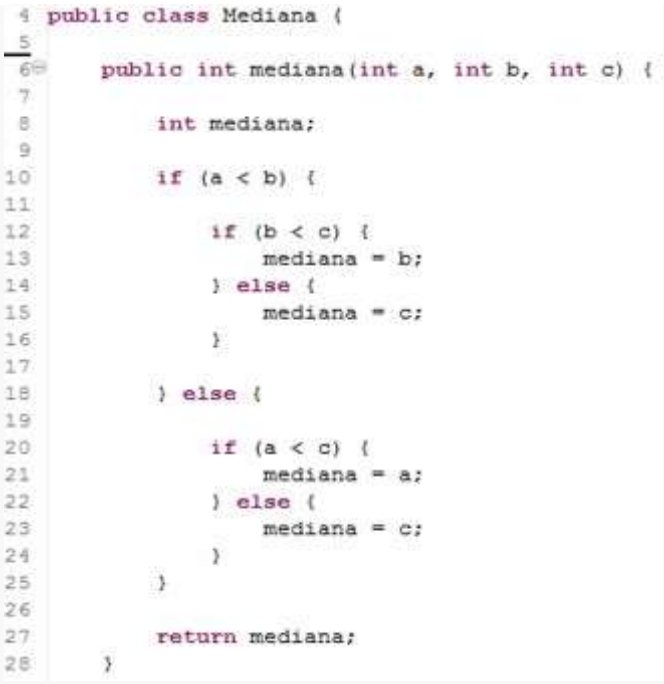
Um conceito muito importante na aplicação de técnica de testes estruturais é o de **cobertura**, uma medida dos elementos de um componente ou sistema que são cobertos pelos testes estruturais. O objetivo é que se tenha um conjunto de testes que percorra todos os elementos estruturais de um código a ser testado.

A seguir, confira a fórmula para calcular o percentual de cobertura de testes de um determinado conjunto de elementos estruturais (código-fonte):

> **Cobertura = qtd. elementos estruturais executados / qtd. elementos estruturais existentes * 100.**

Está apresentado na Figura 1 um exemplo de um código-fonte que calcula a mediana, usando três valores de entrada. Cada comando foi considerado um elemento estrutural e os valores 1, 2 e 3, assumidos para o teste.

Figura 1: Exemplo de um código-fonte



```
4  public class Mediana {
5
6      public int mediana(int a, int b, int c) {
7
8          int mediana;
9
10         if (a < b) {
11
12             if (b < c) {
13                 mediana = b;
14             } else {
15                 mediana = c;
16             }
17
18         } else {
19
20             if (a < c) {
21                 mediana = a;
22             } else {
23                 mediana = c;
24             }
25         }
26
27         return mediana;
28     }
```

A figura apresenta um exemplo de um código que calcula a mediana e as linhas do código que foram executadas, considerando que os valores de entrada para o teste foram: 1, 2 e 3. Fonte: O autor (2021).

O percentual de cobertura do código, considerando o teste realizado, seria igual a 50%, em que:

Cobertura = 4 / 8 * 100 = 50

Caminho básico: critério para elaboração de testes estruturais

Assim como para elaborar os testes funcionais é necessário utilizar critérios, nos testes estruturais ocorre o mesmo desafio. Nos estruturais é necessário estabelecer um conjunto de casos de testes que garanta que todas as instruções codificadas para um elemento de *software* sejam executadas pelo menos uma vez.

O uso de critérios é importante para evitar que parte do código seja testado repetidas vezes, enquanto parte do código não seja executada nenhuma vez.

Um dos critérios utilizados para estabelecer os casos de testes estruturais é o **caminho básico**.

O caminho básico é uma técnica de teste de caixa-branca, proposta por Tom McCabe, em 1976. Esta medida serve como um guia para definir um conjunto de caminhos de execução. A partir dos caminhos de execução é possível identificar quantos casos

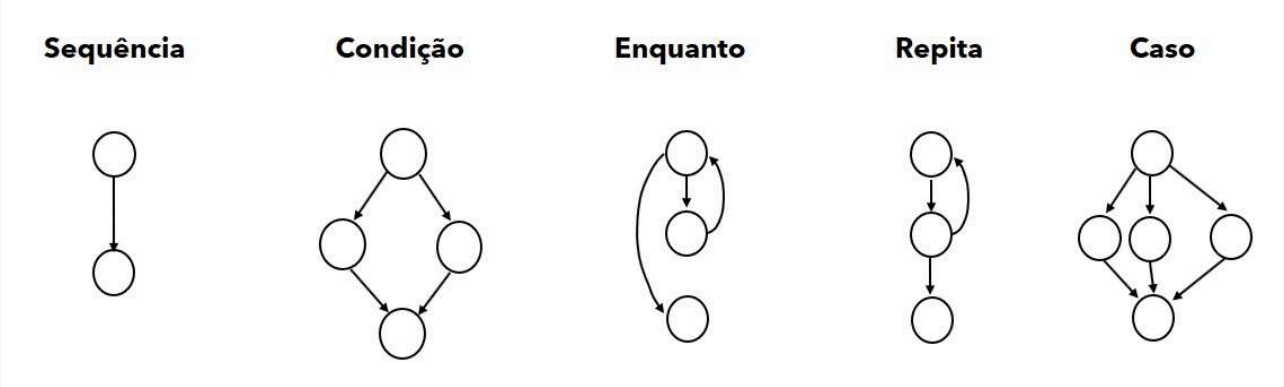
de testes são necessários para percorrer todos os fluxos do código-fonte.

Grafo de fluxo

A partir do código-fonte é gerado um grafo de fluxo, a partir do qual é possível identificar a complexidade lógica e, consequentemente, o conjunto de caminhos de execução.

A construção do grafo é feita de acordo com os padrões apresentados na Figura 2.

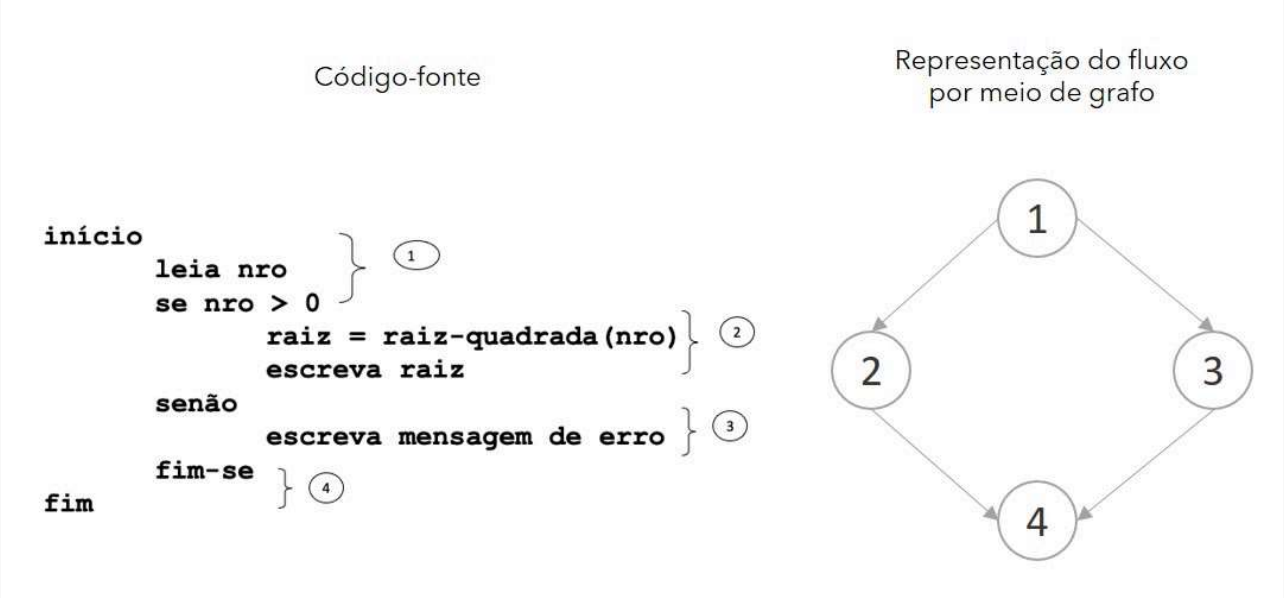
Figura 2: Formas de representação em um grafo das estruturas de um código-fonte



A figura apresenta as formas de representação em um grafo das estruturas básicas de um código-fonte. Fonte: O autor (2021).

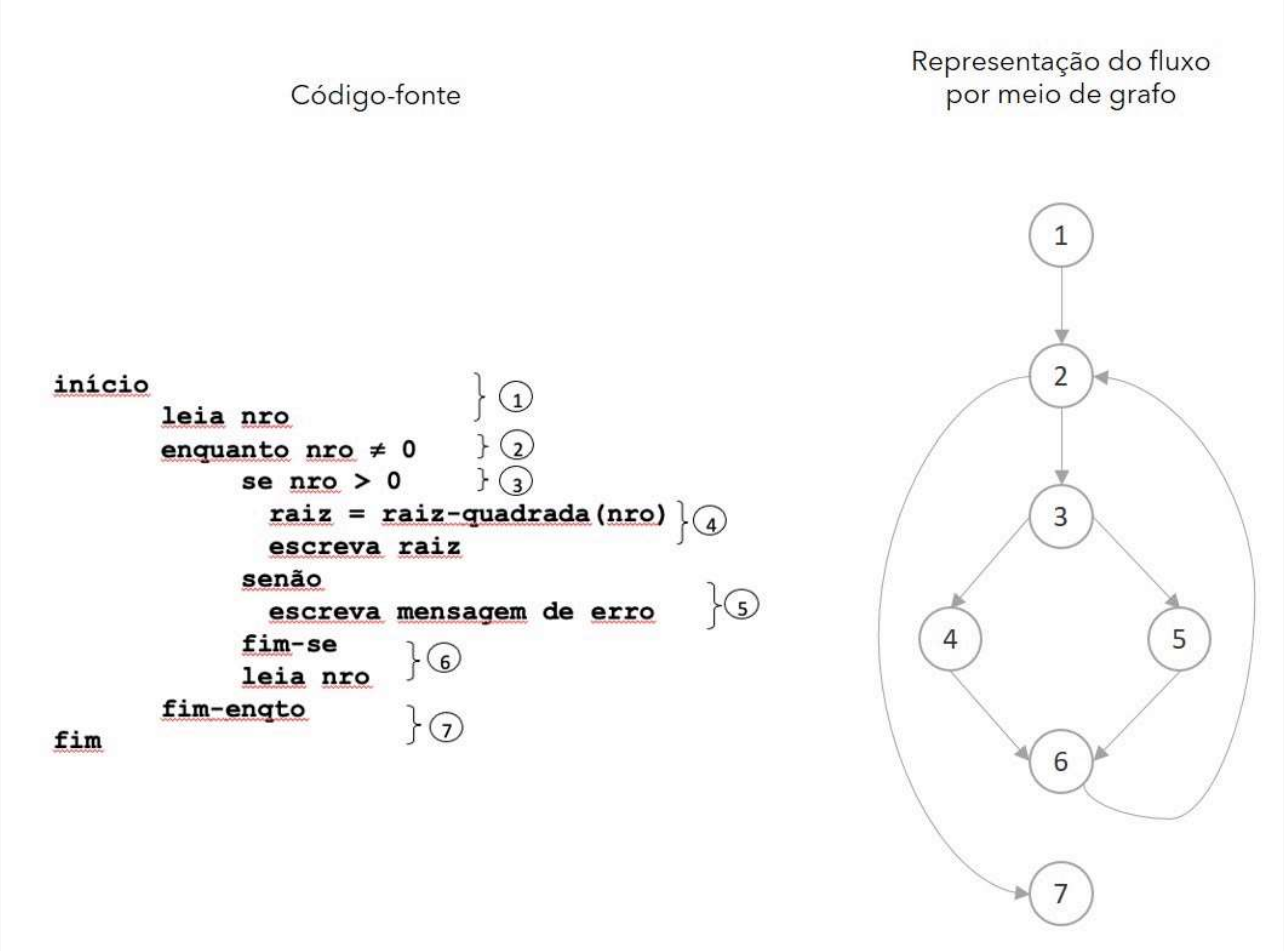
A Figura 3, apresentada a seguir, exemplifica como um fluxo de um código-fonte com estruturas sequenciais e de condição seria representado por meio de um grafo. Já a Figura 4 exemplifica como um fluxo de um código-fonte com estruturas sequenciais, de condição e de repetição seriam representadas por meio um grafo.

Figura 3: Exemplo de um código-fonte com estruturas sequenciais e de condição



A Figura exemplifica como um fluxo de um código-fonte com estruturas sequenciais e de condição seria representado por meio de um grafo. Fonte: O autor (2021).

Figura 4: Exemplo de um código-fonte com estruturas sequenciais, de condição e de repetição



A Figura exemplifica como um fluxo de um código-fonte com estruturas sequenciais, de condição e de repetição seria representado por meio de um grafo. Fonte: O autor (2021).

Com o auxílio do grafo apresentado, é possível identificar os caminhos independentes.

Caminhos independentes

Os caminhos independentes, por sua vez, são ciclos que não contêm outros ciclos embarcados. Deverá existir pelo menos um caso de teste para cada caminho independente existente. Desta forma, garante-se 100% de cobertura de execução do código.

Tomando como exemplo o grafo gerado na Figura 4, existem três caminhos independentes, listados a seguir:

- C1: 1,2,7.
- C2: 1,2,3,4,6,2,7.

- C3: 1,2,3,5,6,2,7.

Cada um dos caminhos inclui pelo menos um nó que ainda não foi executado pelos caminhos anteriores. No caso dos caminhos C2 e C3, o que os diferencia é a execução dos nós 4 e 5, respectivamente. Um exemplo de um caminho independente **inválido** está listado a seguir:

- C4: 1,2,3,4,6,2,3,5,6,2,7.

C4 não é um caminho independente, pois embora apresente uma sequência diferente, não inclui nenhum outro nó ainda não considerado em C1, C2 ou C3.

Ou seja, caminho independente é qualquer caminho que introduza pelo menos um novo conjunto de comandos.

Existe uma maneira para calcular os caminhos independentes, que é por meio da identificação da complexidade ciclomática do código analisado.

Complexidade ciclomática

Para confirmar de uma maneira mais segura quantos caminhos independentes existem em um código, basta calcular a complexidade ciclomática. A complexidade ciclomática é uma métrica de *software* que fornece a medida quantitativa da complexidade lógica de um programa.

O cálculo pode ser realizado por meio da fórmula apresentada a seguir:

- CC = A – N + 2, em que:
    - CC = Complexidade ciclomática.
    - A = número de arcos (arestas) do grafo.
    - N = número de nós do grafo.

Considerando o exemplo da Figura 3, teríamos:

- CC = 4 – 4 + 2.

CC = 2.

Ou seja, teríamos dois caminhos independentes:

C1 = 1, 2, 4.

C2 = 1, 3, 4.

Considerando o exemplo da Figura 4, teríamos:

- CC = 8 – 7 + 2.

CC = 3.

Ou seja, teríamos três caminhos independentes, confirmando a avaliação manual, exemplificada na seção **caminhos independentes.**

> **Lembre-se:** os caminhos independentes indicam quantos fluxos distintos podem ser percorridos em um código-fonte. O cálculo da complexidade ciclomática identifica quantos caminhos independentes existem em um código. O seu cálculo é extremamente útil quando o código é mais longo e complexo. Quanto mais longo e complexo o código, mais difícil se torna a identificação manual dos caminhos independentes. Portanto, o cálculo da complexidade ciclomática confirma quantos caminhos independentes existem no código analisado. Se houver um caso de teste para cada caminho independente, será garantida 100% de cobertura do código na execução desses testes.

Uma outra forma mais simples de calcular a complexidade ciclomática é: **contar a quantidade de estruturas de seleção e de repetição + 1.**

Casos de testes

O último passo do uso do critério do caminho básico é planejar casos de testes para cada um dos caminhos independentes identificados.

Considerando o exemplo da Figura 4, foram identificados três caminhos independentes:

- C1: 1,2,7.
- C2: 1,2,3,4,6,2,7.
- C3: 1,2,3,5,6,2,7.

Agora, é necessário planejar casos de testes que façam com que os caminhos independentes sejam percorridos pelo menos uma vez.

Confira alguns exemplos de parâmetros de entrada que fariam com que o caminho independente fosse executado:

- C1: valor de entrada = 0.
- C2: valor de entrada = 1.
- C3: valor de entrada = -1.

Analise o código-fonte e faça o teste de mesa para verificar como os fluxos de execução são diferentes de acordo com os três cenários de entrada planejados. Estes três casos de testes garantem que todas as instruções do código foram executadas, e, se houver alguma falha, o teste revelará.

EXERCÍCIO

De acordo com o código-fonte abaixo, realize as seguintes ações:

- Desenhar o grafo de fluxo.
- Calcular a complexidade ciclomática.
- Identificar os caminhos independentes.
- Definir um caso de teste para cada caminho independente.

```
public static int buscaBinaria( int[] array,
   int valor )

{       int esq = 0;

         int dir = array.length - 1;

         int valorMeio;

         while ( esq <= dir ) {

                  valorMeio = esq + ((dir -
   esq) / 2);

                  if ( array[valorMeio] < valor
   ) {
```

```
                esq = valorMeio + 1;

            } else if( array[valorMeio] >
    valor ) {

                dir = valorMeio - 1;

            } else {

                return valorMeio;

            }

        }

        return -1;

    }
```

Após realizar o exercício, assista à videoaula desta Unidade para acompanhar a resolução do exercício apresentado.

Caminho básico

Na aplicação da técnica estrutural, um conceito que tem ganhado atenção e adoção, principalmente em times ágeis, é o *Test Driven Development* (TDD). A seção a seguir descreve qual é a filosofia por trás deste conceito.

TDD

TDD é uma técnica ou filosofia que incorpora testes ao processo de produção de código. O desenvolvedor recebe a especificação da nova funcionalidade e programa um conjunto de testes automatizados para um código que não existe. Esse conjunto de testes deve ser executado e falhar. O código é desenvolvido apenas para passar nos testes; então, é refatorado para atender aos padrões de qualidade e testado novamente.

Depois que os testes foram estabelecidos, o programador não deve implementar nenhuma funcionalidade. O objetivo é evitar que os programadores percam tempo com estruturas desnecessárias. Isso atende ao princípio *Keep it Simple, Stupid* (KISS), que indica como tolice fazer algo a mais do que o necessário.

A aplicação de testes, de uma maneira geral, seja testes funcionais ou testes estruturais, tem demandado a disponibilidade de ferramentas que apoiem desde a gestão até a automatização deles.

As ferramentas são bastante variadas, voltadas para os diversos níveis de testes, tipos de requisitos, tipos de linguagens etc.

Confira a seguir algumas ferramentas que, atualmente, têm sido utilizadas com um certo destaque na indústria de desenvolvimento de *software*.

Automação e gestão dos testes

A seguir, estão relacionadas algumas ferramentas atualmente utilizadas para a automação e gestão dos testes.

- ComplexGraph: constrói o gráfico e calcula complexidade ciclomática, ou seja, no de testes necessário para 100% de cobertura.

https://code.google.com/archive/p/complexgraph/downloads

- Eclemma: análise de cobertura dos testes.

https://www.eclemma.org/

- JUnit: biblioteca para automatização dos testes unitários para Java.

https://www.eclipse.org/community/eclipse_newsletter/2017/october/article5.php

- PyUnit: biblioteca para automatização dos testes unitários para Python.

http://pyunit.sourceforge.net/

- Selenium: *framework* para testes em aplicativos *web*.

https://www.selenium.dev/

- Robotium: voltado para testes em aplicativos de Android.

https://www.methodsandtools.com/tools/robotium.php

- Watir: automação de testes da Web, em código aberto, baseados em bibliotecas de código Ruby.

http://watir.com/

- TestRails: gerenciamento de casos de testes.

https://www.gurock.com/testrail/

- TestLinks: gerenciamento de testes.

https://testlink.org/

- Jira: gerenciamento de testes.

https://support.atlassian.com/jira-software-server/

Existem muitas outras ferramentas disponíveis para a automação de testes. Cada uma delas pode ter uma ênfase diferente, como: a plataforma (*web, desktop ou mobile*), o tipo de linguagem (Java, Python, Ruby etc.), os tipos de requisitos (funcionais ou não funcionais) e a gestão dos testes (controle da qualidade dos testes, coleta de métricas de testes etc.).

A área de tecnologia evolui continuamente, assim como as ferramentas. No momento em que estiver estudando esta Unidade, pode ser que novas ferramentas tenham sido disponibilizadas pelo mercado. Portanto, sugerimos que faça uma pesquisa

exploratória para conhecer quais são as disponíveis atualmente para automação de testes e para quais tipos de testes são mais indicadas.

Foi apresentada nesta Unidade a técnica de testes estruturais. Foi detalhado o uso do critério do caminho básico, utilizado para a elaboração e a execução dos casos de testes estruturais. Para entender a aplicação desse critério, foram discutidos alguns conceitos, tais como completude, grafo de fluxo, caminhos independentes e complexidade ciclomática. O objetivo principal do teste estrutural é elaborar casos de testes que levem à execução de 100% do código avaliado. Desta forma, garantimos que, se há alguma falha, em alguma parte do código, este conjunto de testes irá revelá-lo. A adoção dos testes tem sido cada vez maior devido à complexidade de desenvolvimento e à necessidade de entrega de produtos de software com mais qualidade. Por conta disso, novos modelos, conceitos e ferramentas têm sido propostos. Ao final da Unidade, foi apresentado o conceito de TDD e alguns exemplos de ferramentas de automação de testes. O TDD tem como objetivo encorajar o programador a codificar os casos de testes antes mesmo de o código ter sido construído. Estes casos de testes facilitam posteriormente a automatização dos testes. Sabemos que a atividade de testes é importante e trabalhosa, portanto, diversas ferramentas têm sido propostas para apoiar nesta atividade. A Unidade foi concluída com uma lista resumida de ferramentas disponíveis com este objetivo, mas encorajamos que você faça a sua pesquisa para ampliar as possibilidades de aplicação.

Referências

BRAGA, P. H. C. **Testes de software**. São Paulo: Pearson Education do Brasil, 2016. Disponível em: https://plataforma.bvirtual.com.br/Leitor/Publicacao/150962/epub/0.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software: uma abordagem profissional.** 8. ed. Porto Alegre: AMGH, 2016. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788580555349/.

WAZLAWICK, R. S. **Engenharia de software: conceitos e práticas**. 2. ed. Rio de Janeiro: Elsevier, 2019.

DEVCAST. **E aí? Como você testa seus códigos? Devmedia, Tecnologias**, s.d. Disponível em: https://www.devmedia.com.br/e-ai-como-voce-testa-seus-codigos/39478. Acesso em: 19 maio 21.

DEVMEDIA. **Teste unitário com JUnit e ComplexGraph. Devmedia, Tecnologias**, 2014. Disponível em: https://www.devmedia.com.br/teste-unitario-com-junit-e-complexgraph/31382. Acesso em: 19 maio 21.