Introdução a Teste de Software

Os slides a seguir usam o conteúdo do livro **Engenharia de Software Moderna**, de Marco Tulio Valente (Universidade Federal de Minas Gerais), disponível online no endereço https://engsoftmoderna.info/.

Teste de Software

Uma das práticas de programação mais valorizadas hoje em dia

Uma das práticas que sofreram mais transformações nos anos recentes.

No Desenvolvimento em Cascata

Antigamente – Desenvolvimento em Cascata

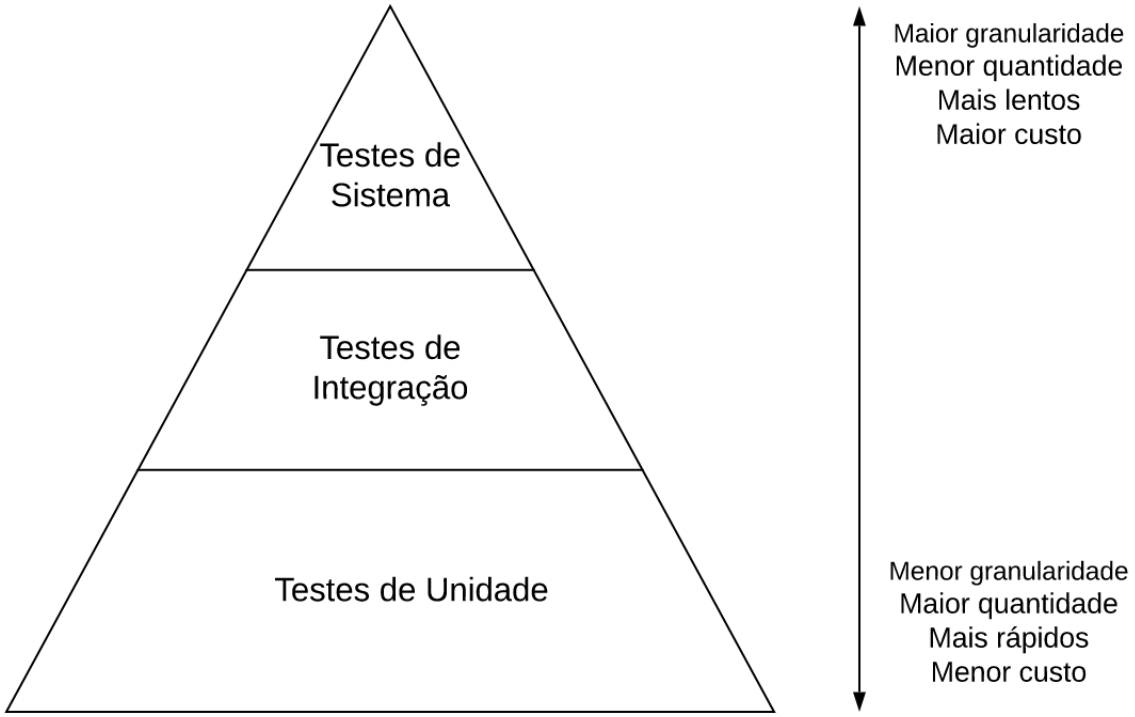
- Testes em uma fase separada, depois da análise de requisitos, projeto e codificação
- Manual
- Equipe separada (+ ou -)
- Apenas detectar bugs

Com Métodos Ágeis

Hoje em dia – Métodos Ágeis

- Muitos testes automatizados
- Construídos não mais depois da codificação
- Não mais existem grandes equipes de teste
  - Ou responsáveis por testes específicos
- Não só para detectar bugs, mas também para
  - Facilitar a manutenção
  - Verificar desempenho
  - Documentação

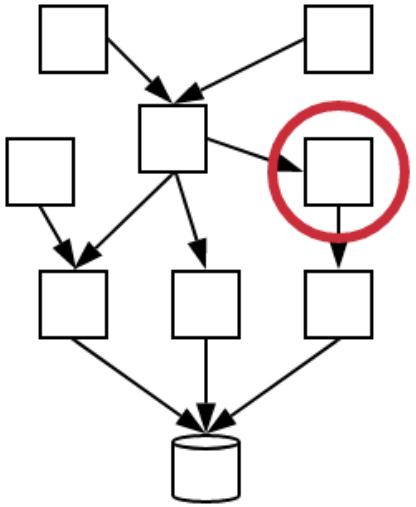
Pirâmide de Testes



Pirâmide de Testes

Testes de Unidade

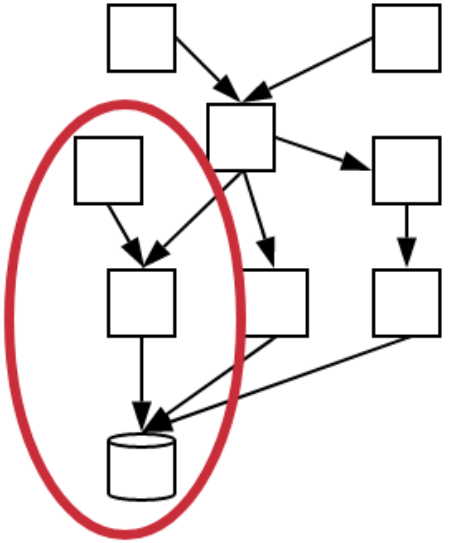
- Testam automaticamente pequenas partes do código
- Simples
- Fáceis de implementar
- Executam rapidamente
- Base da pirâmide



Pirâmide de Testes

Testes de Integração

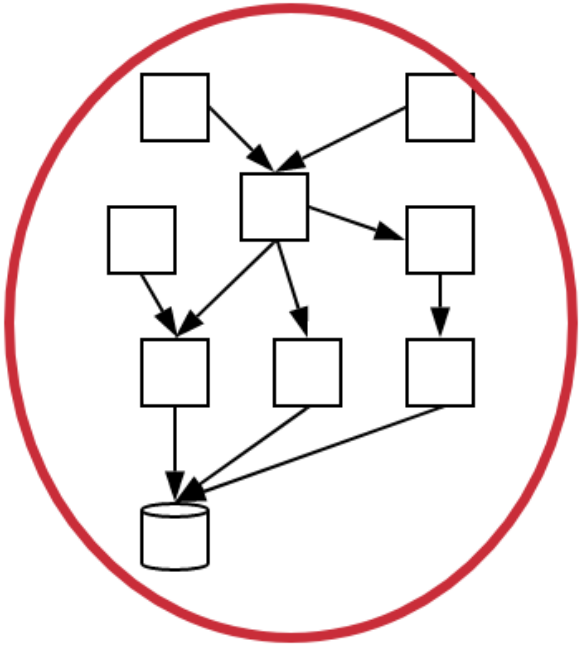
- Testes de Serviços
- Testam uma funcionalidade ou transação completa
- Usam várias classes, de pacotes distintos
- Podem usar componentes externos
- Mais esforço para serem implementados
- Executam de forma mais lenta



Pirâmide de Testes

Testes de Sistemas

- Testes de Interface com o Usuário
- Testes de ponta-a-ponta (end-to-end)
- Simulam uma sessão de uso do sistema por um usuário real
- Mais caros
- Mais lentos
- Menos numerosos



Testes de Unidade

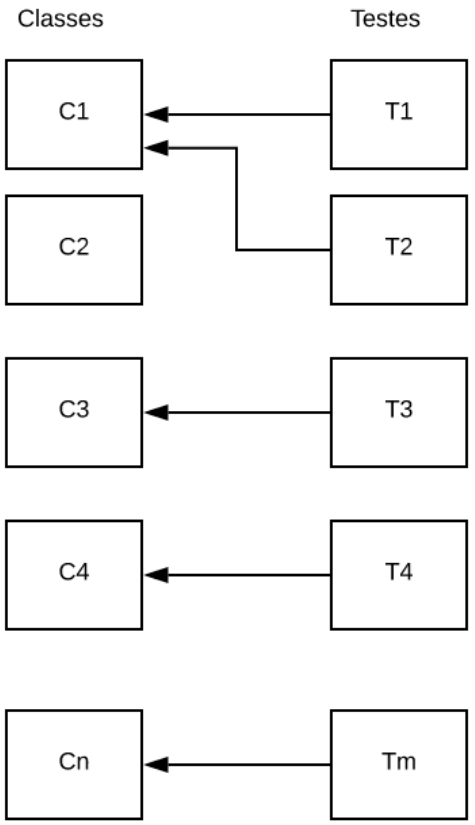
Testes automatizados de pequenas unidades de código, normalmente classes, as quais são testadas de forma isolada do restante do sistema.

Um teste de unidade é um programa que chama métodos de uma classe e verifica se eles retornam os resultados esperados.

Código do sistema dividido em dois grupos

- Classes que implementam os requisitos
- Classes de teste

Testes de Unidade



Testes de Unidade

Normalmente, implementados usando frameworks xUnit, onde x identifica a linguagem de programação

- Ex.: JUnit

```
public class Stack<T> {

  private ArrayList<T> elements = new ArrayList<T>();
  private int size = 0;

  public int size() {
    return size;
  }

  public boolean isEmpty() {
    return (size == 0);
  }

  public void push(T elem) {
    elements.add(elem);
    size++;
  }

  public T pop() throws EmptyStackException {
    if (isEmpty())
      throw new EmptyStackException();
    T elem = elements.remove(size-1);
    size--;
    return elem;
  }
}
```

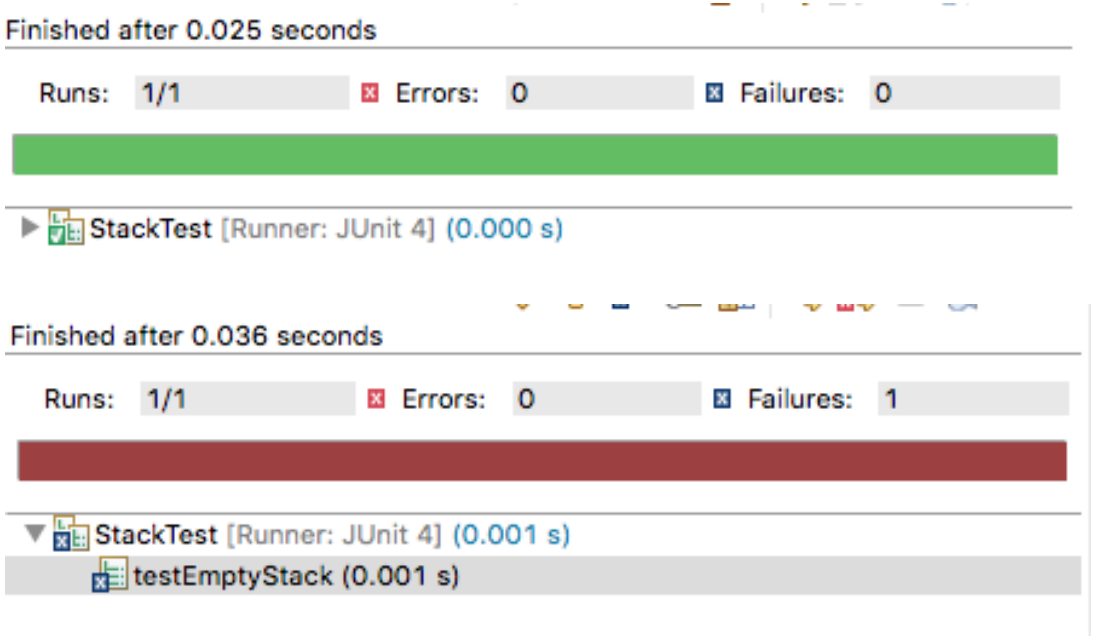
```
public class StackTest {

   @Test
   public void testEmptyStack() {
      Stack<Integer> stack = new
Stack<Integer>();
      boolean empty =
stack.isEmpty();
      assertTrue(empty);
   }

}
```

Testes de Unidade

IDEs oferecem opções para rodar apenas os testes (Run as Test)



```
public class StackTest {

   @Test
   public void testEmptyStack() {
     Stack<Integer> stack = new
Stack<Integer>();
     boolean empty = stack.isEmpty();
     assertTrue(empty);
   }

      ...
      ...
      ...
}
```

Ver código completo no Eclipse

Testes de Unidades - Definições

Teste: método que implementa um teste.

Fixture: estado do sistema que será testado por um ou mais métodos de teste, incluindo dados, objetos, etc.

Suíte de Testes (Test Suite): conjunto de todos os teste.

Sistema sob Teste (System Under Test, SUT): sistema que está sendo testado.

Quando Escrever Testes de Unidade

Após implementar uma pequena funcionalidade

Antes de implementar código de produção (TDD)

Quando o usuário reporta um bug, pode-se escrever um teste que reproduz o bug e depois corrigir o bug.

- Ganhamos um teste

Quando se estiver depurando um trecho de código

Não é recomendável deixar para escrever todos os testes depois que o sistema estiver pronto

O mesmo desenvolvedor de uma classe deve escrever os testes para ela

Os slides a seguir usam o conteúdo do livro **Software Testing: From Theory to Practice**, de Maurício Aniche (Universidade Técnica de Delft), disponível online no endereço sttp.site. Seguindo a orientação do livro, esses slides seguem a mesma licença do livro: CC BY-NC-SA 4.0 International.

Por que testar software?

Porque bugs estão em todos os lugares!

Por que testar software?

Considere o seguinte requisito:

Implemente um programa que, dada uma lista de números inteiros, retorna o menor e o maior número da lista.

Por que testar software?

Uma primeira implementação pode ser:

```
public class NumFinder {
  private int smallest = Integer.MAX_VALUE;
  private int largest = Integer.MIN_VALUE;

  public void find(int[] nums) {
    for(int n : nums) {
      if(n < smallest) smallest = n;
      else if(n > largest) largest = n;
    }
  }
```

Por que testar software?

Técnica comumente usada por desenvolvedores: executar o programa fazendo “pequenas checagens” para ver se ele funciona como esperado.

```
public class NumFinderMain {

  public static void main (String[] args) {
    NumFinder nf = new NumFinder();
    nf.find(new int[] {4, 25, 7, 9});

    System.out.println(nf.getLargest());
    System.out.println(nf.getSmallest());
  }
}
```

A saída desse programa é: 25 e 4
Então, podemos colocar o programa para o usuário usar.
Será?

Por que testar software?

Uma primeira implementação pode ser:

```
public class NumFinder {
  private int smallest = Integer.MAX_VALUE;
  private int largest = Integer.MIN_VALUE;

  public void find(int[] nums) {
    for(int n : nums) {
      if(n < smallest) smallest = n;
      else if(n > largest) largest = n;
    }
  }

  // getters for smallest and largest
}
```

E se tivermos os seguintes números como entrada: 4, 3, 2, 1?

Por que testar software?

Técnica comumente usada por desenvolvedores: executar o programa fazendo “pequenas checagens” para ver se ele funciona como esperado.

```
public class NumFinderMain {

  public static void main (String[] args) {
    NumFinder nf = new NumFinder();
    nf.find(new int[] {4, 3, 2, 1});

    System.out.println(nf.getLargest());
    System.out.println(nf.getSmallest());
  }
}
```

A saída desse programa é: -2147483648 e 1

Por que testar software?

Um primeira implementação pode ser:

```
public class NumFinder {
  private int smallest = Integer.MAX_VALUE;
  private int largest = Integer.MIN_VALUE;

  public void find(int[] nums) {
    for(int n : nums) {
      if(n < smallest) smallest = n;

      // O bug estava aqui
      if(n > largest) largest = n;
    }
  }

  // getters for smallest and largest
}
```

É um bug simples em um programa simples.
Mas há programas muito mais complexos que esse.

Terminologia

- Falha
- Falta
- Erro

Terminologia

- Falha
  - Ocorre quando o software não funciona como esperado. Geralmente são visíveis ao usuário.
- Falta (ou bug ou defeito)
  - Um problema, geralmente no código fonte, que faz com que o software se comporte incorretamente. Falhas são geralmente causadas por faltas.
- Erro
  - A ação humana que fez com que o sistema não funcionasse como esperado.

Terminologia

- Ou seja, um erro de uma desenvolvedor pode produzir uma falta na código fonte que pode resultar numa falha do sistema.

- Obs.: a existência de uma falta no código fonte não necessariamente leva a uma falha. Se o trecho de código que contém a falta nunca for executado, a falha nunca ocorrerá.

- Falha, falta e erro do exemplo NumFinder?

Terminologia

- Validação e Verificação

Terminologia

- Validação
  - Estamos construindo o software correto?
  - Tem a ver com as funcionalidades que o sistema oferece e o cliente para quem o sistema foi feito.
  - O sistema é realmente o que o cliente quer?
  - O sistema é realmente útil para o cliente?

Terminologia

- Verificação
  - Estamos construindo o sistema corretamente?
  - Tem a ver com o sistema fazer o que se espera dele, de acordo com as especificações.
  - A grosso modo, significa o sistema funcionar sem apresentar falhas.

Princípios de Testes de Software

- Ou porque testar software é tão difícil

Princípios de Testes de Software

- Visão simplista
  - Devemos incluir casos de testes até ser o bastante.
  - Não é tão simples assim.
- Quando parar de construir testes?
  - Objetivo: maximizar o número de defeitos encontrados, e minimizar os recursos usados para testar

Teste exaustivo é impossível

Princípios de Testes de Software

- Teste exaustivo é impossível
  - Imagine um sistema com 300 *flags* (ou elementos de configuração)
    - 2^{300} combinações que teriam que ser testadas

- Testes devem, então, ser priorizado

Bugs não são uniformemente distribuídos

Princípios de Testes de Software

Teste de software é capaz de mostrar a presença de bugs, mas não é capaz de mostrar a ausência de bugs.

Dijkstra

Princípios de Testes de Software

- Paradoxo do Pesticida
  - Se apenas uma única técnica ou estratégia de teste for sempre usada, ela perderá sua eficácia em algum ponto.

Princípios de Testes de Software

- Teste é dependente de contexto
  - Testar aplicação Web tem diferenças em relação a testar aplicação móvel, por exemplo.

Automação de Testes

Introdução a automação de testes de unidade com Junit.

Automação de Testes

Considerem os seguintes requisitos:

Implementem um programa que recebe como parâmetro uma cadeia de caracteres com um número romano e converte ele para um inteiro em arábico.

Em números romanos, as letras representam os seguintes valores:

I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000

As letras podem ser combinadas para formar números com a soma dos seus valores. Nesse caso, elas são ordenadas em ordem decrescente, por exemplo: VI = 5 + 1 = 6, VII é 7, CI é 101.

Há também o uso de subtração, por exemplo, 40 não é XXXX, e sim XL, nesse caso 50(L) – 10(X).

Automação de Testes

Vamos ver a implementação desse programa em Java e construir os casos de teste em JUnit no Eclipse.

Possíveis casos de teste seriam:

T1) Apenas uma letra: Ex. C deve ser igual a 100.

T2) Diferentes letras combinadas: Ex.: CLV deve ser 155.

T3) Notação com subtração: Ex.: CM deve ser 900.

Automação de Testes

- Passos para criar testes em JUnit
  - Criar um classe Java para implementar os testes
    - Convenciona-se que essa classe tenha o nome da similar ao da classe a ser testada, mais a palavra Test no final
  - Para cada caso de teste, escreve-se um método dessa classe, com tipo de retorno void e anotado com @Test
    - É importante que o nome de cada método de teste represente bem o caso de teste
  - O método de teste instancia a classe a ser testada e chama o método a ser testado.
  - O método de teste verifica por meio de assertivas se a resultado real coincide com o esperado.
    - Exemplo de assertiva: *Assertion.assertEquals(esperado, real)*

Automação de Testes

- A qualidade do código de teste também importa
  - Por exemplo, é importante evitar duplicação de código
  - Há literatura relacionada a qualidade de código de testes, por exemplo, artigos sobre test smells.

Automação de Testes

- Teste automatizado facilita a realização de refatorações
  - A cada refatoração todos os testes podem ser rapidamente executados novamente.

Automação de Testes

- A estrutura de casos de teste é similar
  - Estrutura AAA (*Arrange, Act and Assert)*
    - *Arrange*: Define e prepara os dados de entrada
    - *Act*: executa o comportamento a ser testado
    - *Assert*: verifica se o sistema se comportou como esperado

```
@Test
void convertSingleDigit() {
  // Arrange: we define the input values
  String romanToBeConverted = "C";

  // Act: we invoke the method under test
  int result = roman.convert(romanToBeConverted);

  // Assert: we check whether the output matches the expected result
  assertEquals(100, result);
}
```

Automação de Testes

- Vantagens do teste automatizado
  - São menos propensos a erros
  - São mais rápidos de serem executados
  - Trazem mais confiança para a realização de refatorações