Teste de Software

Prof. Eiji Adachi

Objetivos

- Testes manuais e Testes automatizados
- Automatização de testes com JUnit

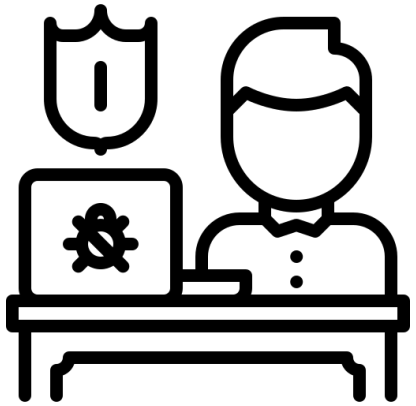
Testes Manuais

Testes Manuais

- Testes em que a execução dos casos de teste é feita manualmente pelo testador, sem o uso de ferramentas de apoio a automatização

Execução Manual

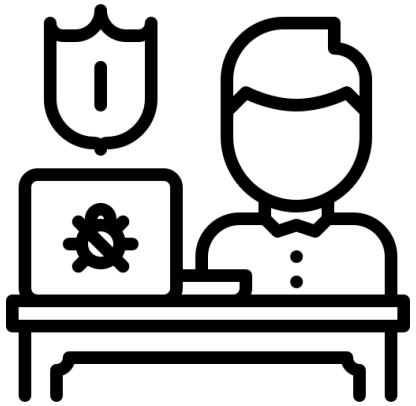
SOFTWARE



Execução Manual

Estímulo

SOFTWARE

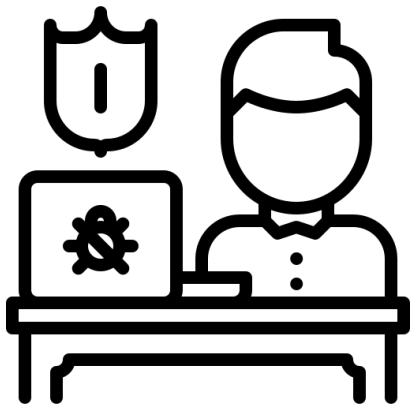


*Testador provê **estímulos** para exercitar um determinado comportamento do objeto de teste*

Execução Manual

Estímulo → SOFTWARE → Resultado Produzido

*Testador **observa** os resultados produzidos pelo objeto de teste*

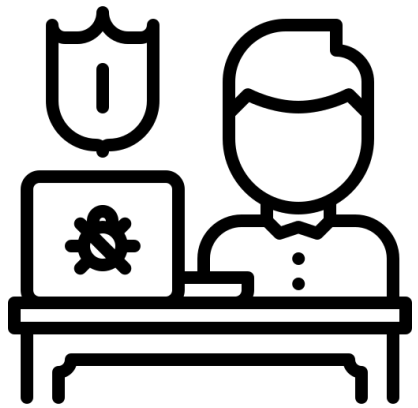


Execução Manual

Estímulo

SOFTWARE

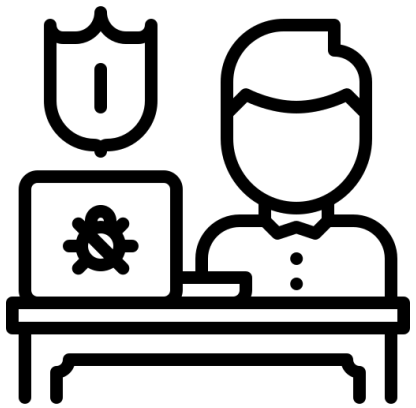
Resultado Produzido



*Testador **compara** os resultados produzidos pelo objeto de teste com os resultados esperados definidos por uma especificação*

Resultado Esperado

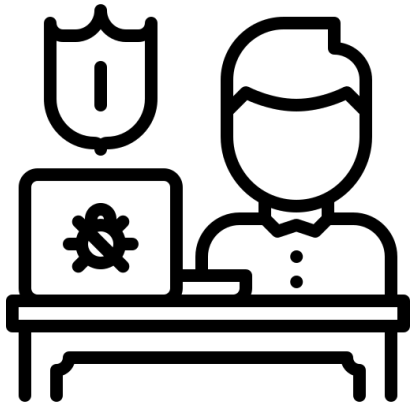
Execução Manual



Se resultado produzido é condizente com resultado esperado, o teste obteve sucesso

Resultado
Esperado

Execução Manual



Se resultado produzido é diferente do resultado esperado, o teste realçou uma falha no software sob teste

≠

Resultado
Esperado

Ex. de Teste com Execução Manual

- **Descrição:** Realização de um empréstimo de livros bem sucedido, seguindo os passos definidos do fluxo principal do caso de uso UC-0017
- **Tipo de Teste:** Funcionalidade
- **Nível de Teste:** Sistema
- **Pré-condição:**
  - Usuário já está logado no sistema
  - Usuário não está em período de suspensão
  - Lista de livros já está selecionada
- **Pós-condição:**
  - Empréstimo é registrado na base
  - Livros estão marcados como emprestados na base
  - Usuário recebe notificação por e-mail, incluindo lista de livros e data de devolução
- **Critério de aceitação:**
  - Todas as pós-condições são atendidas

11

Execução Manual

- Prós
  - Não requer conhecimento em ferramentas de automatização
  - Pode contar com experiência, criatividade e improviso do testador
- Contra
  - Alto custo para reprodução
  - Processo é cansativo e, por isso, mais suscetível a erros humanos

Automatização de Testes

Automatização de Testes de Software

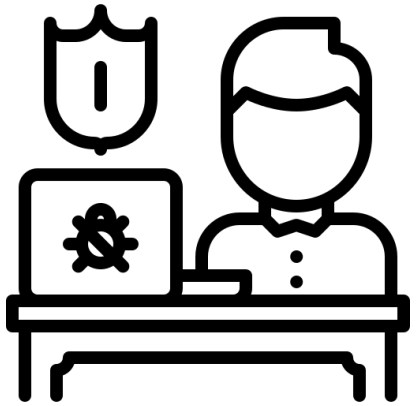
- Definição:
  - É o emprego de ferramentas de apoio para automatizar a execução do objeto de teste, a comparação entre resultados obtidos e esperados e a geração de relatórios detalhados

Automatização de Testes de Software

- Objetivos:
  - Reduzir custos e problemas associados à intervenção humana dos testes manuais
  - Permitir a execução dos testes com maior frequência durante o ciclo de desenvolvimento do software

Execução Automatizada

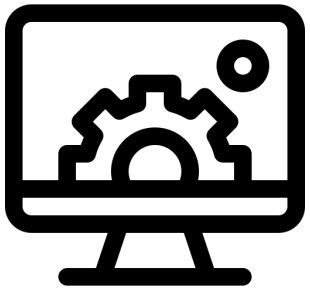
Script



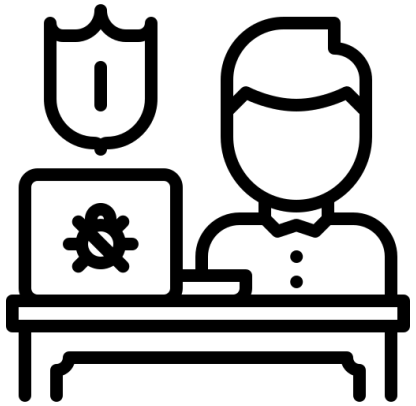
Testador cria um script descrevendo a configuração do objeto de teste, dos estímulos e dos resultados esperados

Execução Automatizada

Script



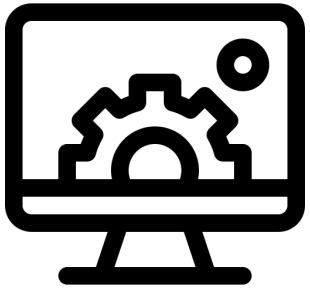
Ferramenta



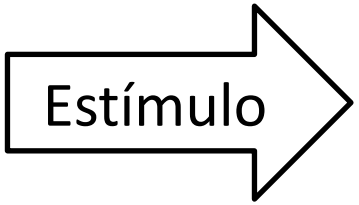
Ferramenta de automatização executa o script, estimulando o objeto de teste a produzir resultados

Execução Automatizada

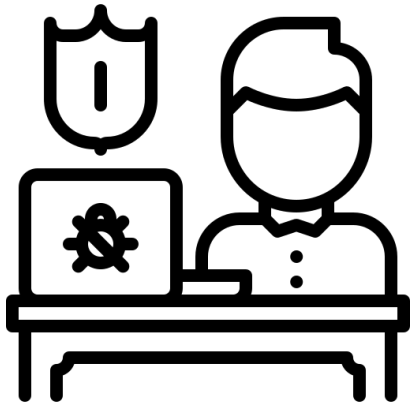
Script



Ferramenta

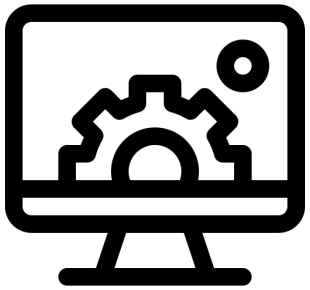


Ferramenta de automatização compara os resultados obtidos com os esperados

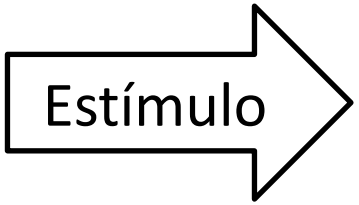


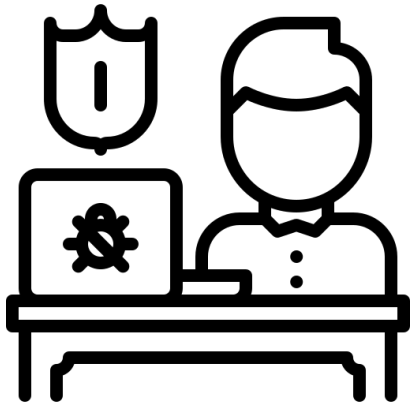
Execução Automatizada

Script



Ferramenta





Ferramenta de automatização produz relatório com dados sobre a execução dos testes

Relatório

Automatizado versus Manual

- Preparação:
  - Custo do automatizado é maior, pois requer instalação e configuração das ferramentas de automatização e, principalmente, criação dos scripts de automatização dos casos de teste

- Retorno:
  - Retorno da automatização é maior no médio-longo prazo, pois o custo inicial é pago com a possibilidade de executar os testes frequentemente

Automatizado versus Manual

- Confiabilidade:
  - Teste automatizado é mais confiável, no sentido de (supostamente) produzir os mesmos resultados sempre que executado

- Flexibilidade:
  - Teste manual é mais flexível, no sentido de que o testador pode descobrir e explorar durante a execução dos testes novas condições que não haviam sido previstas

Aplicação “de brinquedo”

Tipos de Triângulo

- Programa simples:
  - Entrada – três inteiros, cada um indicando o comprimento de um lado de um triângulo
  - Saída – Qual tipo de triângulo:
    - Equilátero
    - Isóceles
    - Escaleno
    - Não é triângulo

```
public String defineType(int l1, int l2, int l3)  {
        if ((l1 > (l2 + l3)) ||
              (l2 > (l1 + l3)) ||
              (l3 > (l1 + l2)))
        {
            return NOT_TRIANGLE;
        }
        else if ((l1 == l2) && (l2 == l3)){
            return EQUILATERAL;
        }
        else if ((l1 == l2) || (l1 == l3) || (l3 == l2))  {
            return ISOCELES;
        }
        else    {
            return SCALENE;
        }
    }
```

Cenário comum

- Programador escreve código e testa manualmente seu programa

```
Enter 3 sides: 10 10 10
Equilateral
Enter 3 sides: 10 15 10
Isoceles
Enter 3 sides: 10 15 8
Scalene
Enter 3 sides: 10 15 1
It is not a triangle
```

"Passou nos meus testes, então está bom!"
– Programador confiante

Cenário um pouco melhor

- Programador escreve um programa para automatizar seus testes

```
public static void main(String[] args)     {
        TriangleType triangleType = new TriangleType();
        int s1 = 10, s2 = 10, s3 = 10;

        String type = triangleType.defineType(s1, s2, s3);

        boolean success = true;

        if (!type.equals(TriangleType.EQUILATERAL))         {
              System.err.println(ERROR_MSG_FORMAT);
              success = false;
        }
        ...
        if (success)         {
              System.out.println(SUCCESS_MSG);
        }
}
```

***A**rrange*

***A**ct*

***A**ssert*

Consideração

- Programa que automatiza os testes é relativamente simples, mas:
  - Programador perde muito tempo escrevendo quantidade razoável de código “auxiliar”
    - Especialmente a parte “verificação” do código
  - Programador perde o foco do que realmente importa, que é escrever o script de teste
    - Importa = Partes “configuração” e “ação” do código

Arquitetura xUnit e JUnit

xUnit

- Kent Beck foi pioneiro na implementação de frameworks de apoio a execução automática de casos de teste
  - Implementação do SUnit (1998), para SmallTalk
  - Implementação do JUnit (2002), para Java
    - Escrito por Kent Beck (XP) e Erich Gamma (GoF Design Patterns)

- A arquitetura de software definida pelo SUnit e JUnit foi seguida por diversos outros frameworks de testes e é chamada de arquitetura xUnit

- Integração com as principais IDEs e ferramentas de build

O JUnit nasceu em um voo de Zurique para a OOPSLA de 1997 em Atlanta. Kent Beck estava voando com Erich Gamma, e o que mais dois nerds poderiam fazer em um voo longo senão programar? A primeira versão do JUnit foi construída lá, programada em pares e feito usando "teste-primeiro".

Martin Fowler

xUnit – Implementações

- Actionscript (FlexUnit)
- Ada (AUnit)
- C (CUnit)
- C# (NUnit)
- C++ (CPPUnit, CxxTest)
- Coldfusion (MXUnit)
- Delphi (DUnit)
- Erlang (EUnit)
- Eiffel (Auto-Test)
- Fortran (fUnit, pFUnit)
- Free Pascal (FPCUnit)
- Golang (Go JUnit report)
- Haskell (HUnit)
- JavaScript (JSUnit)
- Microsoft .NET (NUnit)
- Objective-C (OCUnit)
- OCaml (OUnit)
- Perl (Test::Class and Test::Unit)
- PHP (PHPUnit)
- Python (PyUnit and junit-xml)
- Qt (QTestLib)
- R (RUnit)
- Ruby (JUnit for Rspec)

JUnit

- Implementação Java da arquitetura xUnit
- Framework para apoiar a automatização de testes unitários
  - Disponibilizado como um arquivo .jar
    - Uma “pesquisa” mostra que o JUnit é a biblioteca Java mais comumente baixada do Maven Central Repo [1]
  - A partir do JUnit 4.x (2006), implementação do framework é baseada em annotations
  - Atualmente, está na versão 5.x

[1] https://www.diffblue.com/blog/java/most-popular-libraries-used-by-java-developers-2019/

```
@Test
public void testEquilateralTriangle( )   {
    TriangleType triangleType = new TriangleType();
    int s1 = 10, s2 = 10, s3 = 10;

    String type = triangleType.defineType(s1, s2, s3);

    assertEquals(TriangleType.EQUILATERAL, type );
}
```

***A**rrange*

***A**ct*

***A**ssert*

```
@Test
public void testEquilateralTriangle( )  {
    TriangleType triangleType = new TriangleType();
    int s1 = 10, s2 = 10, s3 = 10;

    String type = triangleType.defineType(s1, s2, s3);

    assertEquals(TriangleType.EQUILATERAL, type );
}
```

***A**rrange*

***A**ct*

***A**ssert*

Método assertEquals compara valor esperado com valor obtido e notifica o resultado da comparação ao framework, i.e., se o teste teve sucesso ou falhou.

JUnit

- @Test
  - Métodos com esta anotação são interpretados pelo framework como casos de testes automatizados
    - Devem ser “public void” e não-estáticos
  - Dentro destes métodos:
    - Configura-se o objeto de teste e o seu “contexto”
    - Definem-se as entradas e resultados esperados
    - Comparam-se os resultados obtidos com os resultados esperados
      - Comparação é feita com base nos métodos asserts, para que os resultados das comparações sejam corretamente notificadas ao framework

Classe Assertions[1]

- Classe que provê um conjunto de métodos para realizar assertivas em casos de testes JUnit
  - Métodos assert* são métodos que comparam dois objetos usando seus métodos equals e, caso sejam diferentes, notificam o JUnit esta diferença observada
    - Notificação é feita lançando um *Error*

[1]https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

assertEquals

- Existe um método assertEquals para todos os tipos primitivos, mas não é possível comparar tipos primitivos com seus respectivos objetos-wrappers:

```
Integer i1 = 1;
int i2 = 1;
assertEquals(i1.intValue(), i2);
assertEquals(i1, Integer.valueOf(i2));
```

assertEquals

- Para tipos Float e Double, é necessário passar uma margem de erro dentro da qual os dois valores ainda são considerados iguais

```
assertEquals(Math.PI, 3.14, 0.001);
```

assertTrue e assertFalse

- Para o tipo boolean, existem os métodos assertTrue e assertFalse

```
assertTrue(validation);
assertFalse(validation);
```

assertEquals

- Para o tipo String, como a comparação dos métodos assert é feita usando o método equals, a comparação é case sensitive

```
assertEquals("String", "string");
```

assertEquals

- Se necessário, devem ser usados os métodos de comparação da própria classe String

```
assertTrue("String".equalsIgnoreCase("string"));
```

assertEquals vs. assertSame

- Para objetos definidos por usuários, case se queira verificar que dois objetos possuem o mesmo estado, devemos implementar o método equals
  - Dica: você pode usar o gerador de código do Eclipse para criar o método equals dos seus objetos

- Mas caso se queira verificar que dois objetos são referências da mesma instância, devemos usar o método assertSame

assertNull

- Também existe um método para verificar se um determinado objeto é nulo:

```
assertNull(user.getName());
```

Versões em negação

- Usados para melhorar legibilidade do código dos casos de teste:
  - assertNotEquals
  - assertNotNull

Test Fixtures

- Um Test Fixture - em pt-br um “dispositivo de teste” - é um dispositivo que garante um ambiente controlado para a realização consistente de testes

Test Fixtures

- Em software, Test Fixture refere-se a uma pré-condição que deve ser garantida para que um teste seja executado corretamente
  - Ex.: O banco de dados precisa estar populado previamente com dados da aplicação para que um teste possa ser executado
  - Ex.: O usuário deve estar logado no sistema antes de se testar uma determinada funcionalidade
  - Ex.: O objeto deve estar num estado específico antes de se invocar um de seus métodos para testá-lo

Test Fixtures

- O JUnit fornece funcionalidades para realizar ações antes da execução dos testes
  - @BeforeEach
    - Métodos “public void” com esta anotação são executados uma vez antes de cada método anotado com @Test
  - @BeforeAll
    - Métodos “public static void” com esta anotação são executados apenas uma vez antes de todos os métodos anotados com @BeforeEach e @Test

Test Fixtures

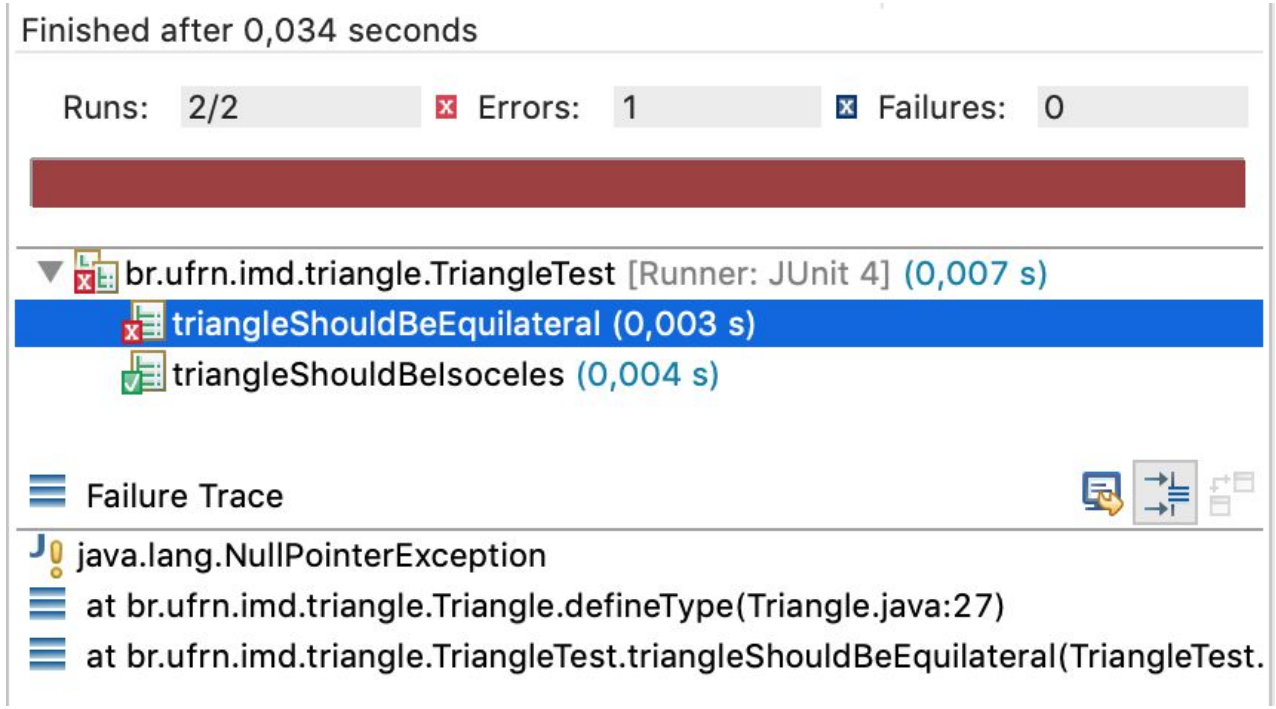
- O JUnit fornece funcionalidades para realizar ações após a execução dos testes
  - @AfterEach
    - Métodos “public void” com esta anotação são executados após cada método anotado com @Test
  - @AfterAll
    - Métodos “public static void” com esta anotação são executados apenas uma vez após todos os métodos anotados com @Test e @AfterEach

Test Fixtures

- A ordem da execução será:
  - @BeforeAll
  - @BeforeEach
  - @Test
  - @AfterEach
  - @AfterAll

Teste de Exceções

- Se um método @Test lançar uma exceção, o JUnit irá interpretar isto como um erro



Teste de Exceções

- Mas muitas vezes queremos de fato exercitar no objeto de teste o comportamento que exceções

```
public TriangleType defineType(Integer s1, Integer s2, Integer s3) {
    if (anyNull(s1, s2, s3)) {
        throw new NullPointerException();
    }

    if (anyNegative(s1, s2, s3)) {
        throw new IllegalArgumentException();
    }

    if (oneSideLargerThanSumOfOthers(s1, s2, s3)) {
        return TriangleType.NOT_TRIANGLE;
    }
    else if (allEqualSides(s1, s2, s3)) {
        return TriangleType.EQUILATERAL;
    }
    else if (twoEqualSides(s1, s2, s3)) {
```

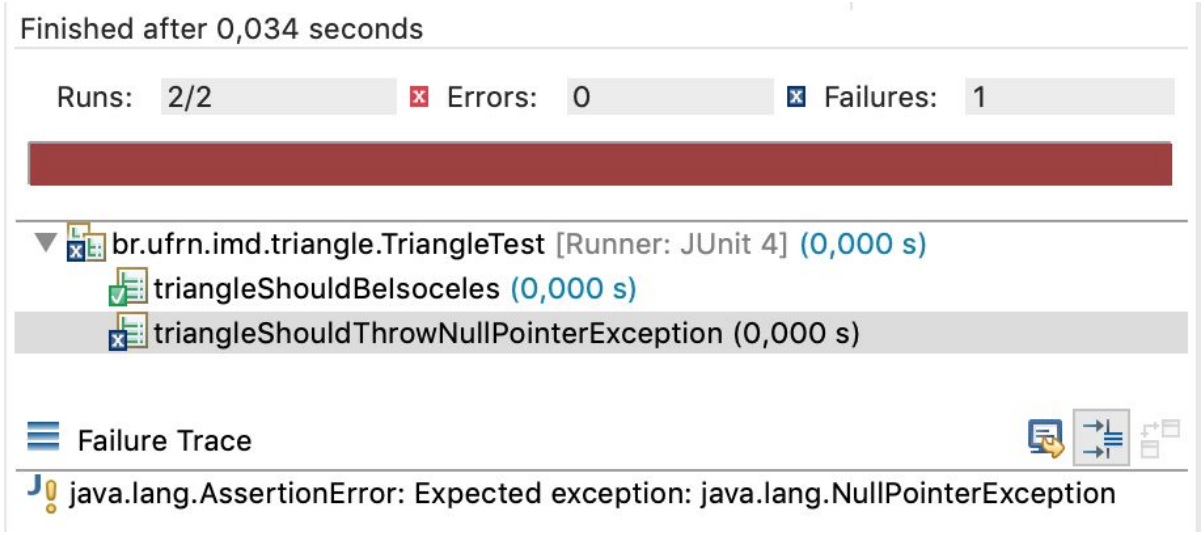
Teste de Exceções

- O JUnit provê o método assertThrows para especificar que o comportamento esperado para o método sob teste é o lançamento de uma exceção de um tipo específico

```
assertThrows(NullPointerException.class, () -> defineType(s1, s2, s3));
```

Teste de Exceções

- Se é especificada uma expectativa de exceção e o objeto de teste não lança uma durante sua execução, isto é considerado um teste que falha
  - Uma pós-condição não foi atendida



Testes Parametrizados

- É comum ocorrer muita repetição de código na construção dos testes JUnit

```java
@Test
public void triangleShouldBeEquilateral() {
    // Configuração
    final Integer s1 = 10;
    final Integer s2 = 10;
    final Integer s3 = 10;

    // Ação
    final TriangleType actual = t.defineType(s1, s2, s3);

    // Verificação
    Assert.assertEquals(TriangleType.EQUILATERAL, actual);
}
```

```java
@Test
public void triangleShouldBeIsoceles() {
    // Configuração
    final Integer s1 = 10;
    final Integer s2 = 8;
    final Integer s3 = 10;

    // Ação
    final TriangleType actual = t.defineType(s1, s2, s3);

    // Verificação
    Assert.assertEquals(TriangleType.ISOCELES, actual);
}
```

Só mudam os valores da entrada e do resultado esperado

Testes Parametrizados

- JUnit provê funcionalidade para definir ‘templates’ de testes, desacoplando o código que define a estrutura do teste da definição dos parâmetros de teste, i.e., dos valores de entrada e do resultado esperado

- Diferentes anotações permitem diferentes formas de se definir os parâmetros do caso de teste

Testes Parametrizados

- @ValueSource – permite definir um array de parâmetros de teste
  - Tipos permitidos são: String, int, long, double
  - Ex.: @ValueSource(ints = { 1, 2, 3 })

- @EnumSource – permite definir valores enumerados como parâmetros de teste
  - Ex.: @EnumSource(value = Months.class, names = {"JANUARY", "FEBRUARY"})

Testes Parametrizados

- @MethodSource – o valor retornado por um método é usado como os parâmetros de teste
  - Ex.: @MethodSource(names = ”geradorDeDados")

- @CsvSource – permite definir strings em formato CSV como parâmetros de teste
  - Ex.: @CsvSource({ ”1+2+3, 6", ”1+3+5, 9" })

Boas Práticas

- Convenções de nomenclatura:
  - Nome da classe de teste é definido como <nome_classe> + Test
  - Nome do método de teste deve explicar o que o teste faz:
    - Usar “should” (ou “deve”, caso use PT-BR) no nome do método:
      - itemsShouldBeCreated,
      - triangleShouldBeIsoceles, sidesShouldNotFormValidTriangle
      - expressaoInvalidaDeveLancarExcecao

Boas Práticas

- Cada método anotado com @Test deve encapsular um só caso de teste
  - Deve encapsular um só conjunto de entradas, saídas esperadas, pré e pós-condições
  - Facilita o diagnóstico das falhas observadas

Boas Práticas

- Durante o desenvolvimento:
  - Quando adicionar uma nova funcionalidade, escreva o teste primeiro
    - Você saberá que está pronto quando o teste rodar

- Durante a manutenção corretiva:
  - Quando encontrar uma falha, escreva um teste que demonstre a existência da falha
    - Você saberá que corrigiu o defeito quando o teste passar

Tutoriais recomendados

- https://www.vogella.com/tutorials/JUnit/article.html
- https://www.baeldung.com/parameterized-tests-junit-5
- https://howtodoinjava.com/junit5/junit5-test-suites-examples/
- https://blogs.oracle.com/javamagazine/post/beyond-the-simple-an-in-depth-look-at-junit-5s-nested-tests-dynamic-tests-parameterized-tests-and-extensions

Teste de Software

Prof. Eiji Adachi