



# QUALIDADE E TESTE DE SOFTWARE

---

Daniel Callegari – Aula 03

# Professores

## **RICARDO BECK**

Professor Convidado

Possui um histórico profissional de mais de 20 anos de experiência em Qualidade de Software com certificação internacional em Teste de Software (CSTE), tendo trabalhado com um extenso portfólio de produtos multinacionais aplicado a estratégias e processos no estado da arte. Há 10 anos se dedica à gerência de projetos, atuando como Scrum Master/Product Owner e, mais recentemente, sendo gerente sênior de P&D na HP Brasil. Possui certificações em Gestão de Pessoas pela FGV-Rio, SCM, PO e Agile Coaching pela Scrum Alliance, além de Fotografia pela ESPM.

## **DANIEL CALLEGARI**

Professor PUCRS

Doutor em Ciência da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul. Possui Especialização em Gestão Empresarial (Sebrae / ANFE / Itália) e Certificações Microsoft e IBM. Associado da Sociedade Brasileira de Computação (SBC). Foi sócio-diretor de duas empresas de tecnologia e sempre equilibrou a atuação acadêmica com o mundo empresarial. Atualmente é professor Adjunto da PUCRS e Coordenador do Curso de Ciência de Dados e Inteligência Artificial. Tem ampla experiência na área da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: bancos de dados, engenharia de software, gerenciamento de projetos. Durante o doutorado, desenvolveu um algoritmo para reconfiguração dinâmica de projetos de software e realocação de recursos, que ganhou o prêmio de primeiro lugar em um congresso da área. Atualmente ministra disciplinas principalmente de engenharia de software, banco de dados e programação, além de coordenar equipes em cooperação com a Dell Computadores do Brasil.

# *Ementa da disciplina*


Introdução aos conceitos de teste unitário, teste de integração, teste de UI.  
Introdução aos conceitos de garantia de qualidade de software. Estudo de métricas voltados ao controle de qualidade no desenvolvimento de software.

# APRESENTAÇÃO

Daniel Antonio Callegari é Doutor em Ciência da Computação pela PUCRS.

Possui Especialização em Gestão Empresarial (Sebrae / ANFE / Itália) e Certificações Microsoft e IBM.

Associado da Sociedade Brasileira de Computação (SBC). Foi sócio-diretor de empresas de tecnologia e sempre equilibrou a atuação acadêmica com o mundo empresarial.

Tem mais de 20 anos de experiência na área da Computação. Atualmente é professor de disciplinas de engenharia de software, banco de dados e programação, além de atuar como coordenador de times de desenvolvimento de software em cooperação com a **PUCRS** online  **uol** edtech. Dell Computadores do Brasil.

# ORGANIZAÇÃO

## Primeira parte da disciplina

- Introdução aos principais conceitos sobre qualidade e teste de software com o professor convidado;
- Exemplos de aplicação.

## Segunda parte da disciplina

- Resgate dos principais conceitos da área;
- Aprofundamento técnico e teórico;
- Exemplos práticos de código e ferramentas.

# INTRODUÇÃO

## Parte 1

- Introdução
- Revisão de Conceitos
- Terminologia
- Técnicas
- Tipos de Teste

# INTRODUÇÃO E REVISÃO DE CONCEITOS

Qualidade e Teste de Software

Objetivo:

“Assegurar que o software cumpra com as suas especificações e atenda às necessidades dos clientes ”

...através de um processo que permeia todo o ciclo de vida do produto.

# INTRODUÇÃO E REVISÃO DE CONCEITOS

## Verificação e Validação de Software

### VERIFICAÇÃO da correção, frente aos requisitos

Feita pela equipe; verificando se atende especificações; à procura de DEFEITOS.

Critérios: {consistência, clareza, segurança, ...} dos requisitos.

→ “*O produto foi construído corretamente?*”

### VALIDAÇÃO com o cliente, no ambiente final

Junto com o cliente; atende ao que o cliente quer?; à procura de PROBLEMAS.

Critérios: {usabilidade, desempenho, portabilidade, ...} do produto.

→ “*O produto correto foi construído?*”



# INTRODUÇÃO E REVISÃO DE CONCEITOS

Dois tipos de técnicas:

## Estáticas

Não requerem que o sistema seja executado

→ Exemplo: **revisões**

## Dinâmicas

Requerem trabalhar com uma representação executável do sistema

→ Exemplo: **testes**

# INTRODUÇÃO E REVISÃO DE CONCEITOS

Dois tipos de técnicas:

## Estáticas

Não requerem que o sistema seja executado

→ Exemplo: **revisões**

- Identificam a correspondência entre um artefato (programa, projeto, diagrama) e sua especificação.
- Não demonstram se o artefato é operacionalmente útil ou se suas características não-funcionais atendem aos requisitos desejados.
- Exemplos: *revisões formais, peer review, inspeção, walkthrough* etc.

## Dinâmicas

Requerem trabalhar com uma representação executável do sistema

→ Exemplo: **testes**

# INTRODUÇÃO E REVISÃO DE CONCEITOS

Dois tipos de técnicas:

## Estáticas

Não requerem que o sistema seja executado

→ Exemplo: **revisões**

## Dinâmicas

Requerem trabalhar com uma representação executável do sistema

→ Exemplo: **testes**

- Envolvem executar uma implementação do software com dados de teste e examinar suas saídas e seu comportamento operacional.
- Objetivos: Encontrar **defeitos** e Avaliar **qualidade**

# NÍVEIS DE TESTE DE SOFTWARE

## Teste de Unidade

- Cada componente é testado separadamente.
- Definição de componente variável (classe, módulo, procedimento, ...)

## Teste de Integração

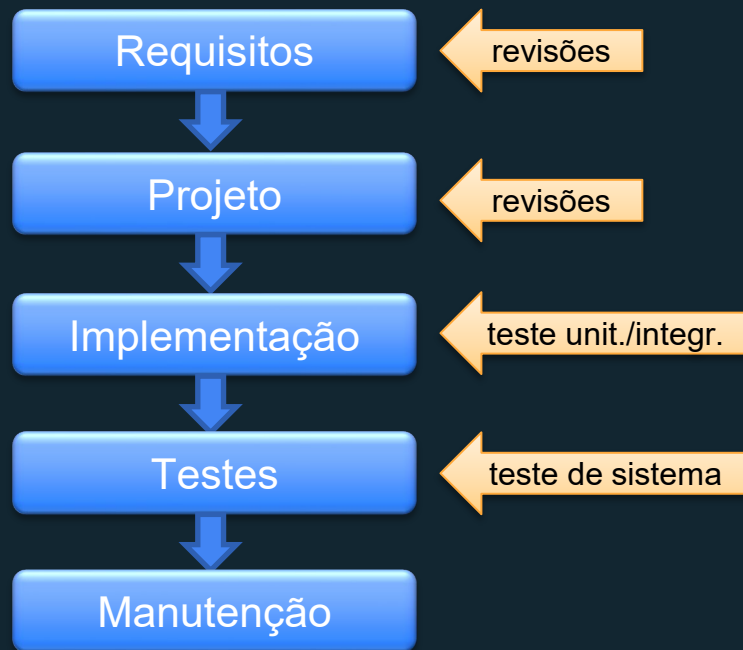
- Testa as interfaces entre os componentes.

## Teste de Sistema

- Extremo do teste de integração.
- Analisa as funcionalidades do sistema como um todo.

## Teste de regressão

- Re-teste após reparo de falhas ou inclusão



# TERMINOLOGIA

**CASO DE TESTE** é um subconjunto de entradas e saídas planejadas, para um ambiente controlado de execução.

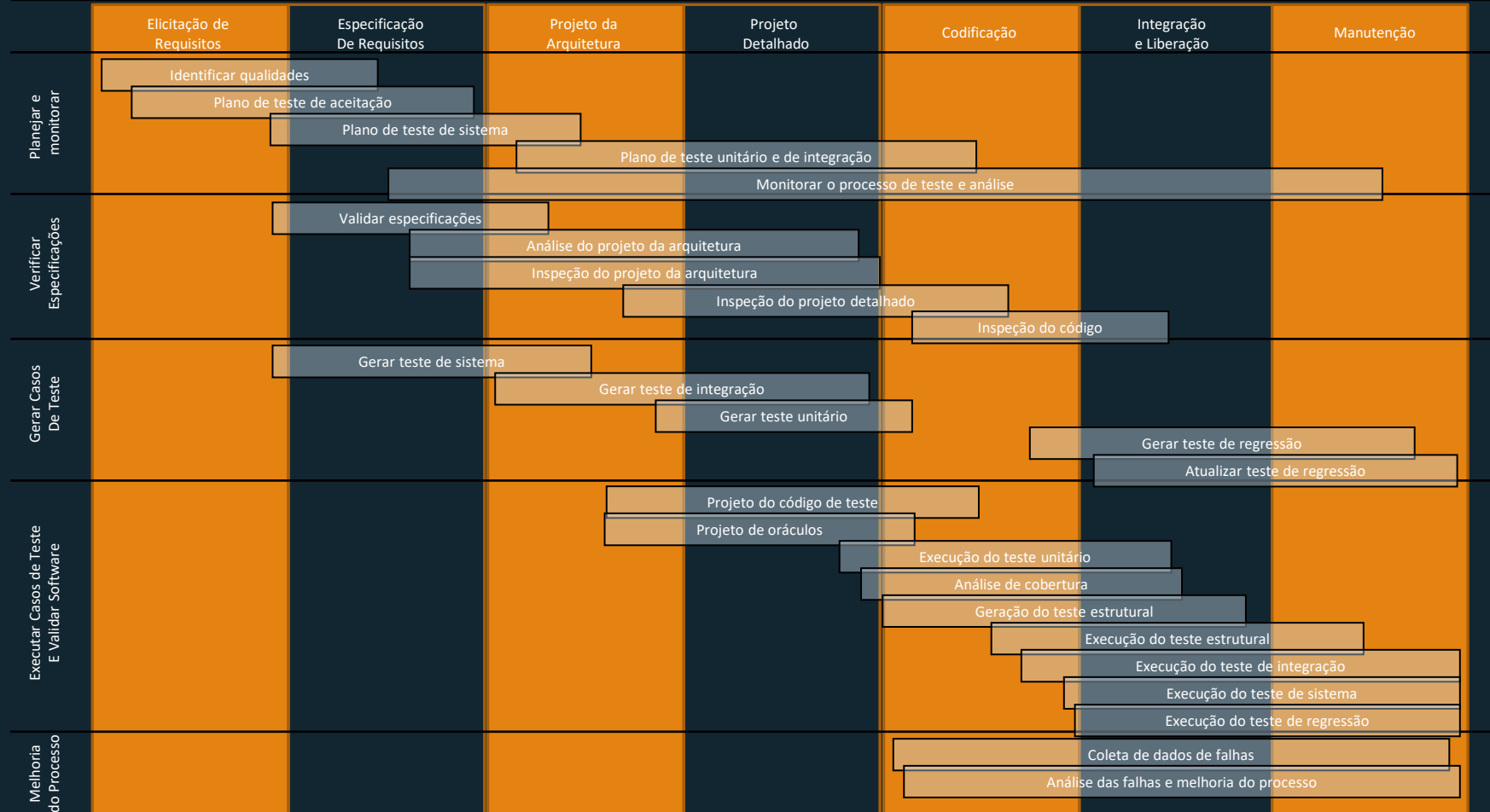
**ROTEIRO DE TESTE** é conjunto de casos de teste.

“O programador comete um **ENGANO**,  
introduzindo um **DEFEITO** (ex. comando incorreto),  
gerando um **ERRO** (estado incorreto),  
que manifesta-se como uma **FALHA** (saída para o usuário)”.

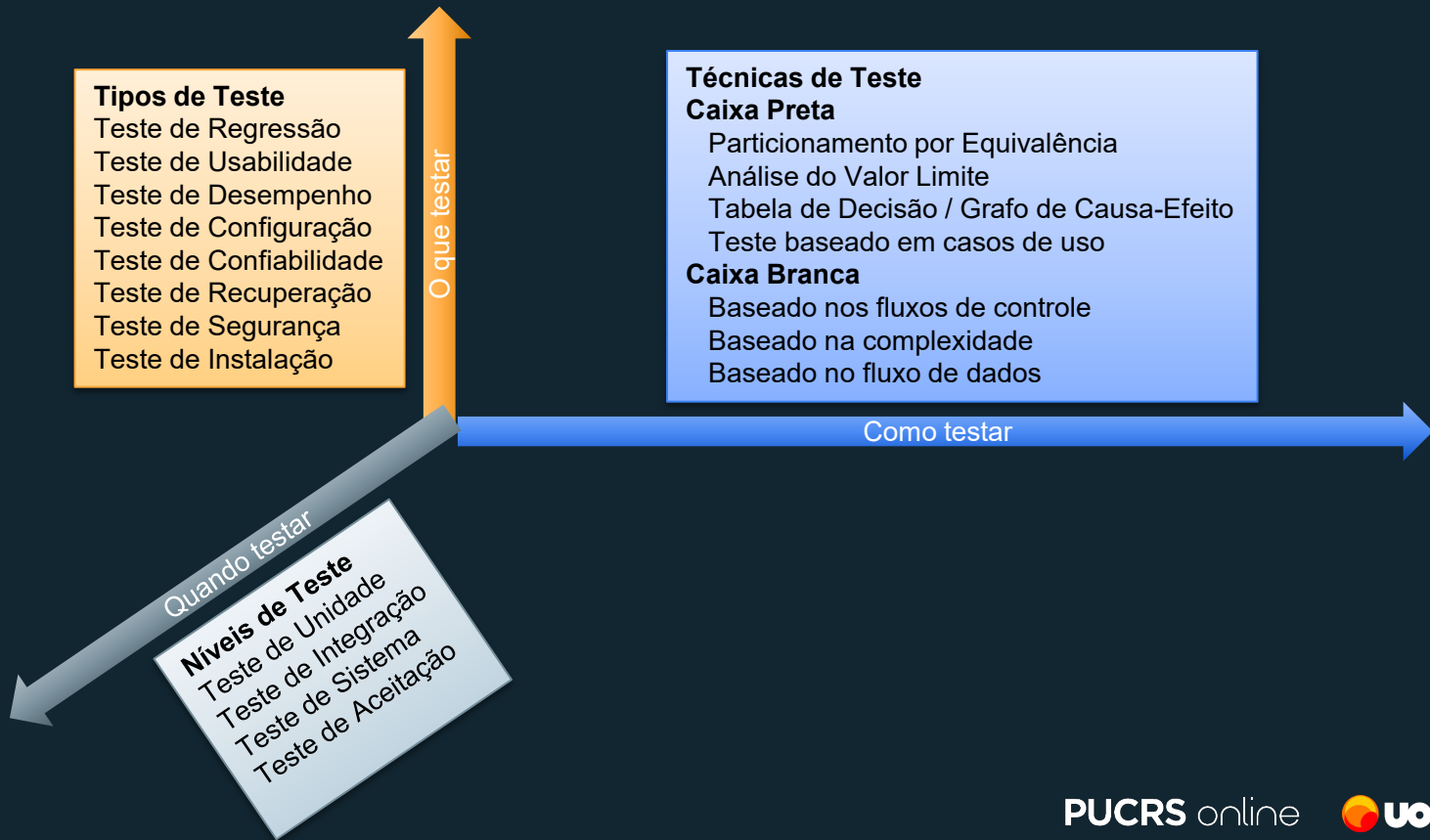
O desafio:

*Definir um subconjunto mínimo de casos de teste  
com a maior probabilidade de apontar erros.*

# Atividades de Teste e Análise ao Longo do Ciclo de Vida do Software



# DIMENSÕES DE TESTE



# TÉCNICAS FUNDAMENTAIS DE TESTE



## Caixa Preta

- ..ou.. **Teste funcional** [o que o software faz e não como ele faz]
- Baseado na especificação formal
- “Não conheço o código-fonte. Olho para os requisitos”
- Uso mais comum: no nível de SISTEMA



## Caixa Branca

- ..ou.. **Teste estrutural** [sobre o código-fonte]
- Baseado na estrutura do programa
- “Valido o comportamento interno da unidade”
- Uso mais comum: nos níveis de UNIDADE e INTEGRAÇÃO





# Técnica: Caixa Preta (Teste Funcional)

## Vantagens

- Independe da linguagem/paradigma
- Pode ser manual ou automatizado
- Pode ser usada em todos os níveis

## Limitações

- Depende de uma boa especificação de requisitos
- Depende da capacidade de interpretar os requisitos (pessoas)
- Pode não identificar funcionalidades ausentes
- Depende do gerente de projetos identificar quais as funcionalidades mais relevantes

Roteiro de Testes ( <i>template</i> )			
ID: xxx	DESCRIÇÃO: (objetivo, ref. reqs)		
	PRÉ-COND:		
ID do Req. Func.		ENTRADA	SAÍDA ESPERADA
	Caso de Teste 1		
	Caso de Teste 2		
	...		



# Técnica: Caixa Branca (Teste Estrutural)

Ocorre sobre o código-fonte

- Baseado nos fluxos de controle (GFC – Grafo dos fluxos de controle)
- Baseado na complexidade (McCabe)
- Baseado no fluxo de dados (Def-Uso)

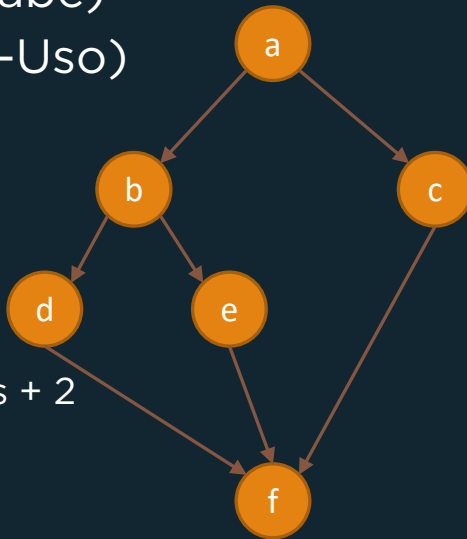
## Complexidade Ciclomática

(métrica do número de caminhos independentes)

$C$  = número de regiões

$C$  = número de arcos – número de nodos + 2

$C$  = número de nodos predicativos + 1



# CASOS REAIS

0800 Unitan started  
1000      stepped - action ✓  
1300 030 MP - MC 1.480000000  
030 PRO - 2 1.130400000  
      conv 2.130670000  
Relays 6w in 033 failed speed speed test  
      in return 1100 tank.  
1100 Started Cosine Tape (Sine chest)  
1525 Started Multiplier Adder Test.  
1545  
Relay #70 Panel F  
      (mouth) in relay.  
First actual case of bug being found.

Harvard, 1947 – Primeiro bug em computadores

You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

Musisz zrestartować swój komputer. Wciśnij i przytrzymaj przez kilka sekund przycisk Power lub wciśnij Restart.

## Empresa anuncia recolhimento de bombas de insulina vulneráveis a hackers

por  
**Victoria Song**

publicado em  
7 de outubro de 2021 @ 15:15



O fabricante de dispositivos médicos Medtronic expandiu seu recall (recolhimento de produto) de controles remotos para suas bombas de insulina MiniMed 508 e MiniMed Paradigm.

O motivo? Os dispositivos são um potencial risco de segurança cibernética. Segundo a Food and Drug Administration (FDA), dos EUA, pessoas não autorizadas podem acessar os dispositivos para alterar a quantidade de insulina fornecida a um paciente.

'Reinicie tudo a cada 149 horas' é uma solução para um bug em aviões da Airbus de mais de US\$ 300 milhões

por  
Andrew Liszewski

publicado em  
30 de julho de 2019 @ 19:05



Ter que reiniciar um notebook lento depois de alguns dias de uso é uma pequena inconveniência que precisamos enfrentar com um aparelho que custou alguns milhares de reais. E quando isso acontece com um avião comercial de mais de US\$ 300 milhões?

Companhias aéreas que não realizaram uma atualização de software em certos modelos do Airbus A350 estão sendo instruídas a desligar completamente a aeronave e ligá-la novamente a cada 149 horas para não correr o risco de "[...] perder parcialmente ou totalmente alguns dos sistemas ou funções do avião", de acordo com a EASA (Agência da União Europeia de Segurança de Aviação).

**PUCRS** online



# NÍVEIS DE TESTE

## Parte 2

- Teste no nível de unidades
- Teste no nível de integração
- Teste no nível de sistema
- Teste de aceitação

# Teste no nível de UNIDADES

## UNIT TESTING:

- Drivers e Stubs
- Ferramentas:
  - Junit (para Java)
  - Jest (para Javascript)
  - etc.

(?) O que é uma unidade mesmo?

“A menor unidade testável em um sistema”

- Diversas “granularidades”
  - uma função/procedimento/método
  - uma classe
  - uma página web
  - um script
  - um módulo
  - um componente
  - ...

# Teste no nível de UNIDADES

## Exemplo em Javascript com Jest

funcoes.js

```
const funcoes = {  
  ...  
  
  codigoValido: function (codigo) {  
    return (codigo >= 100 && codigo <= 999);  
  }  
  
  ...  
}
```

funcoes.test.js

```
...  
  
test('Codigo 100 deve ser permitido', () => {  
  expect( funcoes.codigoValido(100) )  
    .toBeTruthy();  
});  
  
...
```

# Teste no nível de UNIDADE

## Exemplo em Java com JUnit

Identificador.java

```
package com.profdcallegari.geraclassesjava;

public class Identificador {

    public static boolean isValidado (String id) {

        ...

    }

}
```

IdentificadorTest.java

```
package com.profdcallegari.geraclassesjava;

import org.junit.Test;
import static org.junit.Assert.*;

public class IdentificadorTest {

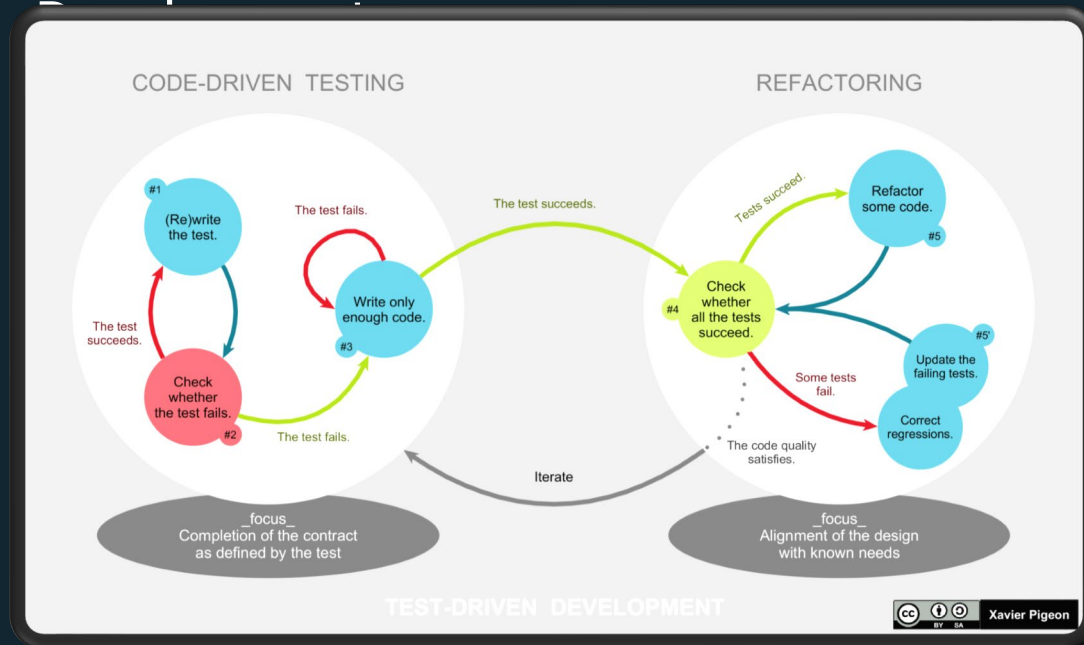
    @Test
    public void deveAceitarIds () {
        final String[] validos = {
            "id",
            "a1",
            "_var",
            "MAIUSC",
            "camelCase"
        };

        for (String id : validos) {
            assertTrue(Identificador.isValidado(id));
        }

        ...
    }
}
```

# Teste no nível de INTEGRAÇÃO

## TDD: Test-Driven





# Teste no nível de INTEGRAÇÃO

## Ciclo de desenvolvimento orientado a testes (TDD)

1. **Adicione** um teste.
2. Execute todos os testes. O novo teste **deve falhar**, e por motivos esperados.
3. Escreva o código **mais simples** que **passa** no novo teste.
4. Todos os testes agora devem **passar**.
5. **Refatore** conforme necessário, usando testes após cada refatoração para garantir que a funcionalidade seja preservada. Por exemplo:
  - mova o código para onde ele mais logicamente pertence
  - remova código duplicado
  - torne nomes auto documentados
  - divida métodos em partes menores
  - reorganize hierarquias de herança
6. **Repita**

# Teste no nível de INTEGRAÇÃO

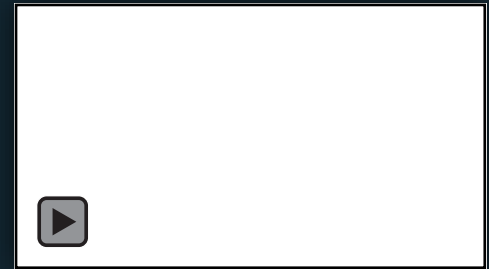
Seguem naturalmente a partir do teste de UNIDADES e também das inspeções.

Embora funcionem em separado, as unidades devem ser testadas em conjunto (suas interfaces):

- Efeitos colaterais dado o comportamento sistêmico

Pode-se reaproveitar Casos de Teste, Drivers de Teste

Obs.: unidades (componentes) podem ser: comerciais, reusáveis adaptados ou



# Teste no nível de INTEGRAÇÃO

## Dicas de estratégia:

Executar os testes de integração de forma incremental, para fornecer *feedback* rápido para o time.

Dependendo do acoplamento entre projeto e teste (montagem incremental dos módulos), utiliza-se código de apoio (*drivers*, *stubs* e *instrumentação*).

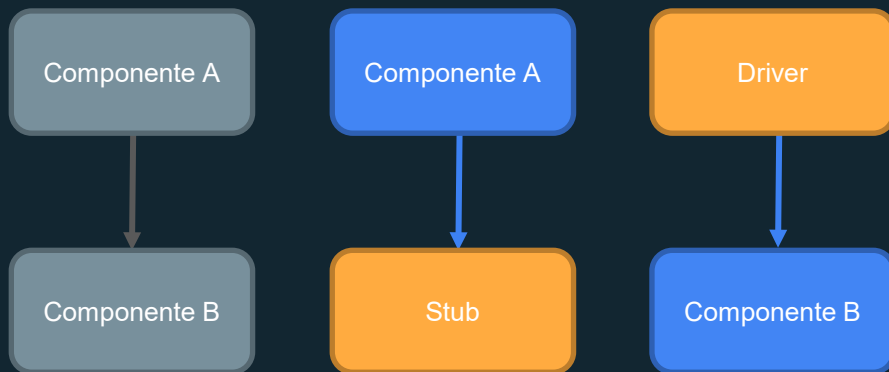
# Teste no nível de INTEGRAÇÃO

## DRIVERS

Responsáveis pelo controle dos testes  
Exercitam as UNIDADES sob teste

## STUBS

Simulam comportamento da UNIDADE  
Utilizando o mínimo de esforço



Atenção: Falhas detectadas durante o teste de integração podem ser um sinal de teste de UNIDADE insatisfatório.

Ex. em 2004, uma vulnerabilidade de *buffer overflow* numa biblioteca de PNGs criou problemas de segurança em browsers e clientes de email no Windows, no Linux e no MacOS.

# Teste no nível de INTEGRAÇÃO

## Caso Real

No foguete Ariane 4, o módulo de cálculo da inclinação horizontal foi testado e utilizado sem que se verificasse nenhum problema.

O cálculo da inclinação levava em conta a velocidade do foguete.

No foguete Ariane 5, o mesmo módulo – já testado, inclusive na prática – foi reutilizado.

## O que ocorreu

O foguete Ariane 5 saiu do controle e teve de ser destruído.

As investigações concluíram que:

- A velocidade do Ariane 5 era muito superior à do Ariane 4.
- Ao receber valores maiores, o módulo de cálculo da inclinação gerou um erro de *overflow* ...



[https://www.youtube.com/watch?v=PK\\_yguLapgA](https://www.youtube.com/watch?v=PK_yguLapgA)

# Teste no nível de INTEGRAÇÃO

## TOP-DOWN



Usamos DRIVERS e STUBS

Vamos substituindo os stubs sucessivamente (escrevendo o código dos stubs)

## BOTTOM-UP




Combinamos unidades (componentes)

Escrevemos drivers (@Test)

Chamamos os componentes


# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 

Este caso possui características específicas que precisam ser consideradas na integração:

- a) Comportamento dependente do estado
- b) Encapsulamento
- c) Herança
- d) Polimorfismo e ligação dinâmica
- e) Classes abstratas

# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 

## a) Comportamento dependente do estado


- os testes devem considerar o **estado** em que se encontra o objeto no momento que os métodos são ativados e as relações de **dependência** entre eles.

## b) Encapsulamento

- eventualmente os testes podem necessitar de acesso a **informação privada** de maneira a poder verificar se um método funciona corretamente ou não.
- muitas vezes os métodos **alteram o estado** interno do objeto sem uma resposta direta correspondente.



# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 


## c) Herança

- o projeto dos testes deve considerar o **efeito** dos métodos novos e dos métodos sobrescritos.
- casos de teste novos têm de ser escritos para estes;
- alguns métodos necessitam ser apenas re-testados com os testes já existentes e alguns não precisam nem ser re-testados.

## d) Polimorfismo e ligação dinâmica

- uma única chamada pode estar ligada a **diferentes métodos**, dependendo do estado da computação.

# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 

## e) Classes abstratas

- é necessário testá-las sem ter pleno conhecimento de como podem ser **instanciadas**.
- necessitam de alguma implementação de uma classe derivada.

# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP)



Para o teste intraclasses (teste das unidades)

1. Se a classe é abstrata, deriva-se um conjunto de instanciações para cobrir os casos significativos.
2. Projetam-se casos para cobrir os métodos herdados e sobrecarregados, incluindo construtores.
3. Projeta-se um conjunto de casos de teste baseados na máquina de estados que especifica o comportamento da classe.
4. Complementa-se o conjunto de testes a partir das relações estruturais derivadas do código fonte.
5. Projetam-se testes para verificar as exceções que podem ser lançadas

# Teste no nível de INTEGRAÇÃO

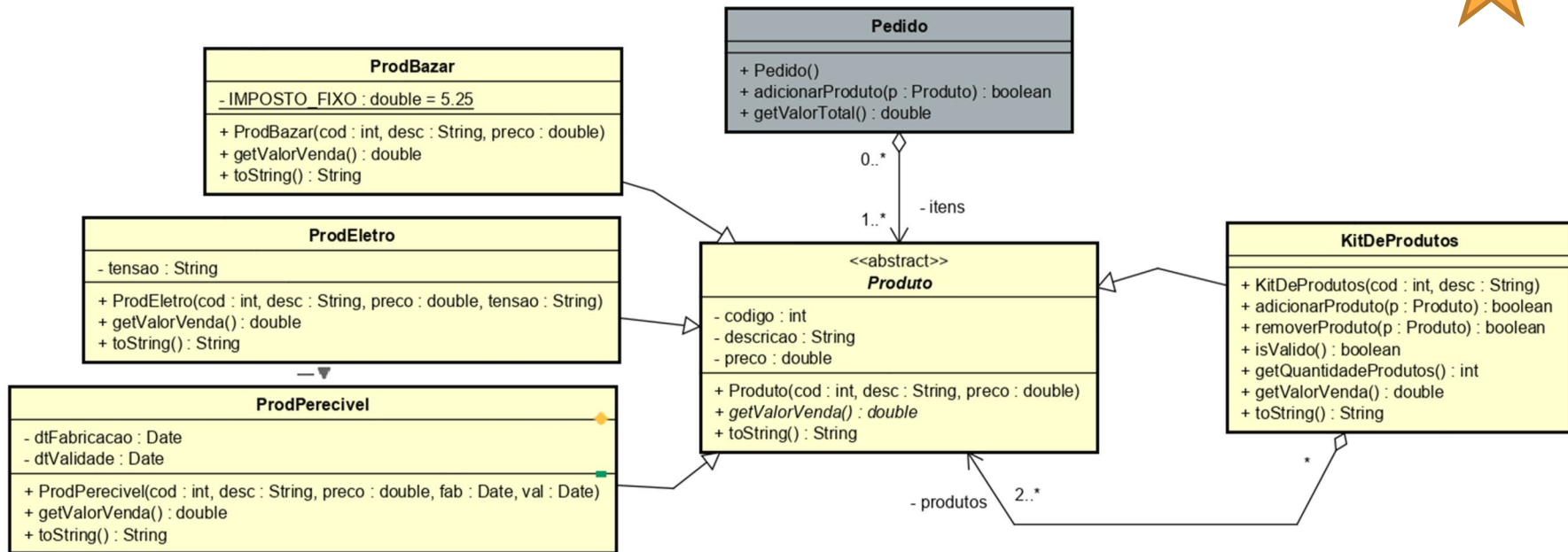
Recomendações especiais para o caso de software orientado a objetos (OOP)



Para o teste interclasse (teste de integração)

1. Identifica-se a hierarquia de unidades a serem testadas de forma incremental.
2. Projeta-se um conjunto de casos de teste funcionais para a classe alvo.
3. Acrescentam-se testes que cubram o fluxo de dados entre as chamadas.
4. Verificam-se as exceções propagadas através das classes.
5. Integram-se os testes polimórficos com testes que visam à interação entre chamadas polimórficas e ligações dinâmicas.

# Teste no nível de INTEGRAÇÃO



# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 

## Testando a classe abstrata *Produto* e suas derivadas

- A classe *Produto* é abstrata, portanto necessita-se de uma implementação.
  - No caso, optou-se por testar seus métodos a partir da classe *ProdBazar*.
- A classe *ProdBazar* não possui um ciclo de vida relevante que justifique sua modelagem por um diagrama de estados.
  - Especificam-se os casos de teste usando as técnicas tradicionais.
- Para os demais tipos de produto segue-se o mesmo raciocínio.

# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP)



## Classe ProdBazar

Métodos	Verificar
Construtor	Código: -1,0,1,500,9999,10000 Descrição: "", "Banana" Preço: -1.0,0.0,1.0,100.0
toString()	Formatação da string: "10, Banana, R\$ 5.90"
getValorVenda()	Se o preço de venda é 5,25% maior que o preço informado no construtor (R.N.)

## Classe ProdEletro \*

Método	Verificar
Construtor	Verificar apenas os valores para tensão: 110 ou 220 (p/os demais valores válidos)
getValorVenda()	Se o valor de venda é igual ao preço informado no construtor
toString()	Se inclui a tensão especificada

\* Observe que vários aspectos já foram testados na classe base.

## Classe ProdPerecivel

Método	Verificar
Construtor	dt/fab < dt/val (ok) dt/fab == dt/val (erro) dt/fab > dt/val (erro)
getValorVenda()	Se dt/fab > dt/val+10 dias Se dt/fab == dt/val+10 dias Se dt/fab < dt/val+10 dias
toString()	Se inclui a data de fabricação e a data final.

# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP)




## Testando a classe `KitDeProdutos`

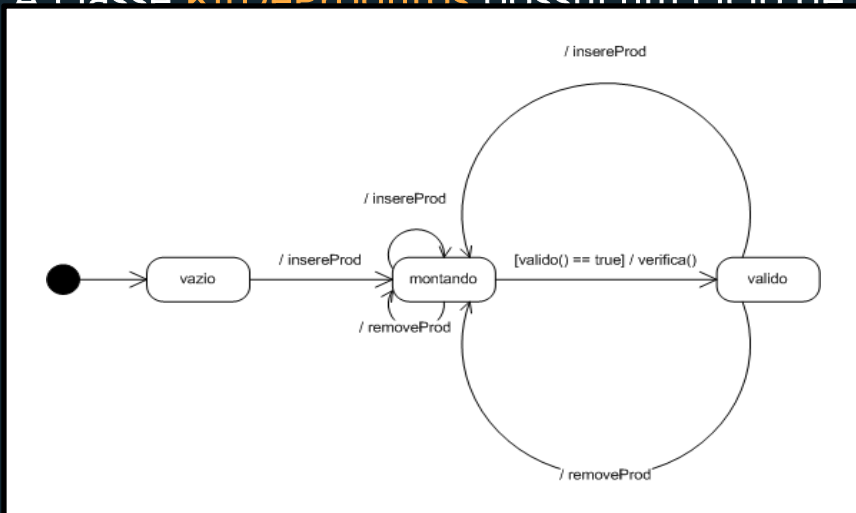
- Os atributos `codigo` e `descricao`, assim como o método `toString`, são herdados sem alterações de `Produto`. Logo, não precisam ser re-testados.
- O atributo `preco` sofre efeito dos métodos de `KitDeProdutos` (o preço varia com a quantidade de produtos no kit), logo precisa ser re-testado (o atributo `preco` sempre armazena o somatório dos preços dos produtos do kit). O método `getValorVenda` calcula os descontos do valor de venda.
- Os métodos `adicionarProd`, `removerProd`, `getQuantidadeProdutos`, `isValido` e `getValorVenda` possuem implementações novas e devem ser testados.
- Note que `isValido` e `getValorVenda` devem ser testados em teste interclasse.



# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 

A classe **KitDeProdutos** possui um ciclo de vida que justifica sua descrição por um



# Teste no nível de INTEGRAÇÃO

Recomendações especiais para o caso de software orientado a objetos (OOP) 

```
@Test
public void kitComDoisProdutos_quantidade2 () {
    try {
        ProdBazar pb1 = new ProdBazar(10, "Caneta mágica", 2.0);
        ProdBazar pb2 = new ProdBazar(20, "Lápis mágico", 3.5);

        KitDeProdutos kit = new KitDeProdutos(1, "Kit de Natal");
        kit.adicionarProduto(pb1);
        kit.adicionarProduto(pb2);

        assertEquals(2, kit.getQuantidade());
        assertEquals(true, kit.isValido());
    } catch (ProdutoInvalidoException e) {
        fail("kitComDoisProdutos_quantidade2");
    }
}
```

# Testes de Sistema, Aceitação e Regressão

Os testes de Sistema, Regressão e Aceitação lidam com o **comportamento** de um sistema como um todo, mas com diferentes propósitos.

O Teste de Sistema pode ser considerado um passo final no teste de integração.

É a etapa final do teste de sistema e sua especificação.

Teste de Sistema	Teste de Aceitação	Teste de Regressão
Verifica contra uma especificação de requisitos	Verifica a adequação às necessidades do usuário	Verifica novamente os casos de teste aprovados em versões anteriores do sistema.
Executado pelo time de testes (idealmente independente)	Executado pelo time de testes com o envolvimento do usuário	Executado pelo time de desenvolvimento de testes
Verifica a correção e completude do produto	Valida a utilidade e a satisfação com o produto	Protege contra alterações indesejadas

# Teste de Regressão

Os Testes de **Regressão** procuram verificar novamente os casos de teste aprovados em versões anteriores do sistema.

O objetivo é proteger contra alterações indesejadas quando modificamos ou acrescentamos funcionalidades.

Por isso, talvez um nome mais apropriado seja “**Teste de não-regressão**”.

As técnicas de teste de regressão incluem:

- Reteste completo
- Seleção de testes
- Priorização de casos de teste
- Híbrido

# Teste no nível de SISTEMA

O Teste de Sistema avalia:

- Comportamento funcional
- Requisitos de qualidade (p.ex. confiabilidade, usabilidade, desempenho e segurança)

É útil também para detectar defeitos externos da interface entre hardware e software (ex. *race-conditions*, *deadlocks*, interrupções, tratamento de exceções, uso ineficiente da memória).

Serve como um bom “cenário de ensaio” para o Teste de Aceitação.

# Teste no nível de SISTEMA

## Em suma:

Com o sistema já desenvolvido (com as unidades testadas e integradas), deve-se voltar aos requisitos e observar se o produto os atende de acordo com a especificação.

“Executar as funcionalidades do sistema  
sob o ponto de vista do usuário final,  
procurando por falhas  
em relação aos objetivos planejados”

Atenção: Isso não garante a satisfação do usuário (o foco é nas especificações!)

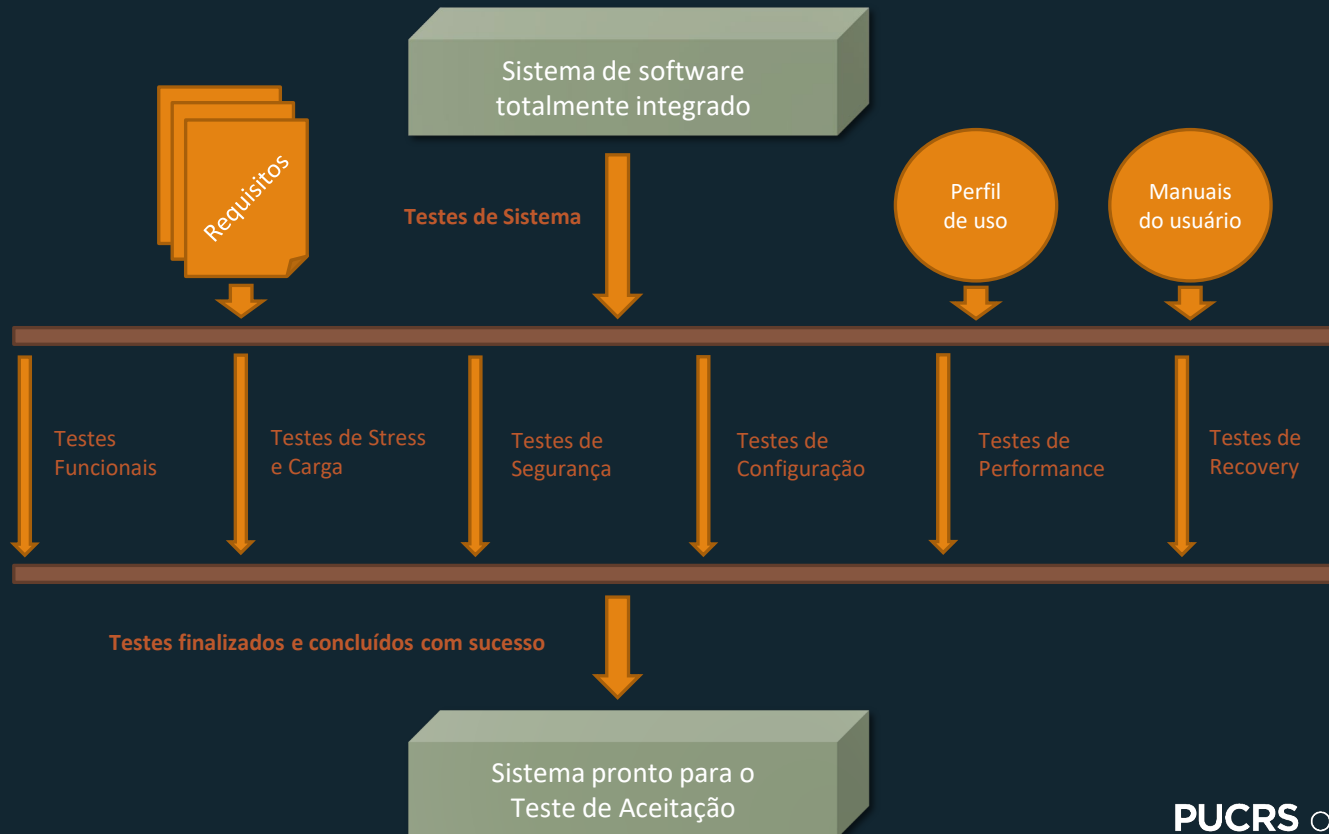
# Teste no nível de SISTEMA

“O sistema atende aos requisitos?”

- predomínio dos testes funcionais (Caixa Preta)
- executados por uma equipe distinta
- em um ambiente controlado, que simula o ambiente de produção.
- Obs.: em um processo de desenvolvimento iterativo, o teste ocorre sobre um incremento  
(chamando-se, portanto, “Teste de Release”).

O resultado de um teste de sistema é o **comportamento observado** deste sistema.

# Teste no nível de SISTEMA

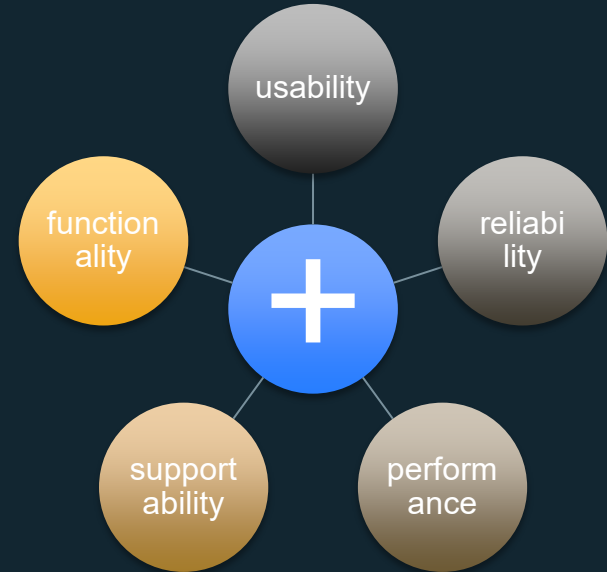




# Teste no nível de SISTEMA

Características de qualidade de software (FURPS+ Model)

- Funcionalidade
- Confiabilidade
- Portabilidade
- Usabilidade
- Eficiência
- Manutenibilidade



# Teste no nível de SISTEMA

## Funcionalidade

- Eficácia
- Produtividade
- Segurança
- Satisfação

## Confiabilidade

- Maturidade
- Tolerância a Falhas
- Recuperabilidade
- Conformidade

## Eficiência

- Relação ao Tempo
- Utilização de Recursos
- Conformidade

## Manutenibilidade

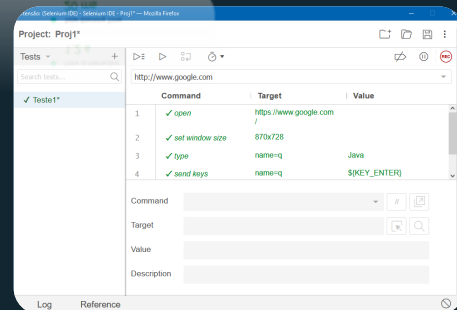
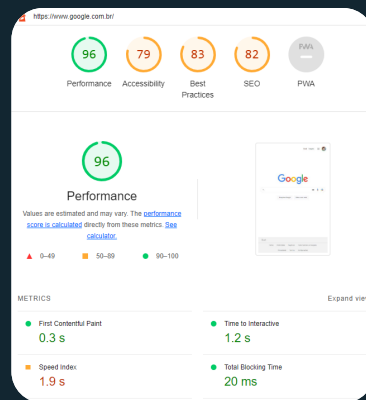
- Analisabilidade
- Modificabilidade
- Estabilidade
- Testabilidade
- Conformidade

## Portabilidade

- Adaptabilidade
- Capacidade para Instalação
- Coexistência
- Conformidade

## Usabilidade

- Inteligibilidade
- Apreensibilidade
- Operacionalidade
- Atratividade
- Conformidade



# PRÁTICA

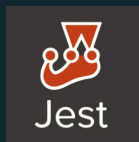
## Parte 3

- Uso de ferramentas de teste
- Teste de unidades
- Teste de integração
- Teste de sistema



# DEMOS

- JUnit para Java
- Jest para Javascript/Node
- Selenium para Teste de Sistema (Web)
- Lighthouse (Google)



# CONCLUSÃO

## Parte 4

- Sugestões e Dicas
- Conclusão

# SUGESTÃO

Montar e disponibilizar dashboards para o time de desenvolvimento e operações



# Sugestões para consulta adicional

Os 25 erros de software mais perigosos

<https://www.sans.org/top25-software-errors/>



- Ranking of each Top 25 entry,
- Links to the full CWE entry data,
- Data fields for weakness prevalence and consequences,
- Remediation cost,
- Ease of detection,
- Code examples,
- Protection Methods

OWASP

<https://owasp.org/>



**Who is the OWASP® Foundation?**

The Open Web Application Security Project® (OWASP) is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

## Sugestões para consulta adicional

O código-fonte do JUnit (e suas *issues*) estão disponíveis no GitHub

<https://github.com/junit-team/junit5>



no





**“Qualidade não é testável.  
Se ela não existir antes de  
você começar a testar, ela  
não existirá quando o  
teste  
estiver terminado.”**

Pressman

**PUCRS** online  **uol**edtech.