Aula 3

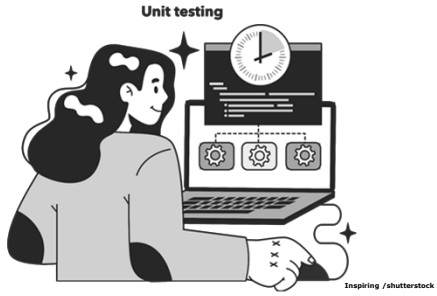
Testes de Software

Profª Maristela Weinfurter

1

Conversa Inicial

2



Inspiring /shutterstock

3

Semanche /shutterstock

4

Testes de componente

5

Testes de componente

- **O teste de componentes envolve a verificação sistemática dos componentes de nível mais baixo na arquitetura de um sistema**

AnnaKoles /shutterstock

6

Testes de componente

- **Dependendo da linguagem de programação usada para criá-los, esses componentes têm vários nomes, como *unidades*, *módulos* ou, no caso de programação orientada a objetos, *classes***

AnnaKoles /shutterstock

7

Testes de componente

- **Os testes correspondentes são, portanto, chamados de *testes de módulo*, *testes de unidade* ou *testes de classe***

AnnaKoles /shutterstock

8

Testes de componente

- **Um teste de componente normalmente testa apenas um único componente isoladamente do resto do sistema**

AnnaKoles /shutterstock

9

Testes de componente

- **Esse isolamento serve para excluir influências externas durante o teste: se um teste revelar uma falha, é obviamente atribuível ao componente que estamos testando**

AnnaKoles /shutterstock

10

Testes de componente

- **Também simplifica o design e a automação dos casos de teste, devido ao seu escopo restrito**

AnnaKoles /shutterstock

11

Testes de componente

- ***Test-first* é a abordagem de última geração para teste de componentes (e, cada vez mais, em níveis de teste mais altos também)**

AnnaKoles /shutterstock

12

Testes de componente

- A ideia é primeiro projetar e automatizar seus casos de teste e programar o código que implementa o componente como uma segunda etapa

AnnaKoles /shutterstock

13

Testes de componente

- Essa abordagem é fortemente iterativa: testamos o código com os casos de teste que já projetamos e, em seguida, estendemos e melhoramos o código do produto em pequenas etapas, repetindo até que o código cumpra seus testes

AnnaKoles /shutterstock

14

- Ficar sem o teste de componentes economiza esforço apenas ao preço de baixas taxas de descoberta de falhas e maiores esforços de diagnóstico

Semanche /shutterstock

15

TDD – Test Drive Development

16

TDD – Test Drive Development

LanKS /shutterstock

17

TDD – Test Drive Development

- Com TDD, conseguimos:
  - Código limpo (sem código desnecessário e/ou duplicado)
  - Código fonte dos testes como documentação dos casos de testes
  - Código confiável, logo, com mais qualidade

(...)

18

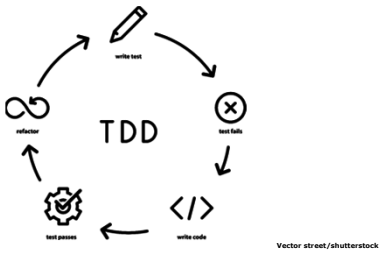
TDD – Test Drive Development

(...)

- Suporte para teste de regressão
- Ganho de tempo na depuração e correção de erros
- Desenvolvimento refatorado constantemente
- Baixo acoplamento do código

19

TDD – Test Drive Development

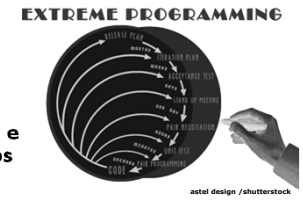


Vector street/shutterstock

20

TDD – Test Drive Development

- XP é um estilo de programação que foca na excelência do software por meio de técnicas de programação aliadas a uma comunicação clara e sem ruídos entre os desenvolvedores



astel design /shutterstock

21

TDD – Test Drive Development

- Com a aplicação do XP em nosso dia a dia, podemos observar que:
  - Conquistamos uma filosofia de desenvolvimento de software baseada nos valores de comunicação, feedback, simplicidade, coragem e respeito

22

TDD – Test Drive Development

- Utilizamos um conjunto de práticas comprovadamente úteis para melhorar o desenvolvimento de software. As práticas se complementam, amplificando seus efeitos. Eles são escolhidos como expressões dos valores

23

TDD – Test Drive Development

- Criamos um conjunto de princípios complementares, técnicas intelectuais para traduzir os valores em prática, o que é proveitoso quando não há uma prática útil para o seu problema específico

24

- **E, finalmente, desenvolvemos uma comunidade que compartilha esses valores e muitas das mesmas práticas**

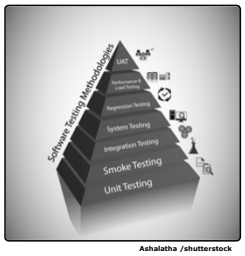
Semanche /shutterstock

25

JUnit e demais ferramentas

26

JUnit e demais ferramentas



Ashalatha /shutterstock

27

JUnit e demais ferramentas

- **Objetivos dos testes de unidade:**
  - **Permitir maior cobertura de teste do que testes funcionais**
  - **Aumentar a produtividade da equipe**
  - **Detectar regressões e limite à necessidade de depuração**

(...)

28

JUnit e demais ferramentas

(...)

- **Dar a confiança para refatorar e, em geral, fazer alterações**
- **Melhorar a implementação**
- **Documentar o comportamento esperado**
- **Ativar cobertura de código e outras métricas**

29

JUnit e demais ferramentas

- **Ferramentas:**
  - **Unit Testing Framework, Pytest e Locust para linguagem Python**
  - **XCTest para Swift**
  - **Test::Unit, RSpec e Minitest para Ruby**

(...)

30

JUnit e demais ferramentas

(...)

- Mocha, Jasmine, Jest, Protractor e Qunit para JavaScript
- PHPUnit para PHP
- NUnit para C#
- JUnit para Java

31

JUnit e demais ferramentas

- Dentre as vantagens do JUnit, encontram-se:
  - Criação rápida de testes com aumento de qualidade do código em desenvolvimento
  - Criação de uma hierarquia dos testes
  - Software livre e código aberto

(...)

32

JUnit e demais ferramentas

(...)

- Ganho de tempo no desenvolvimento ao diminuir o tempo de depuração
- Ser conhecido por um grande número de desenvolvedores
- Validação dos resultados dos testes com resposta imediata

33

JUnit e demais ferramentas

- Método de comparação – assertEquals()
  - O método assertEquals pode ser implementado de várias formas diferentes. Ele recebe como parâmetro o resultado do método que está sendo testado e o resultado esperado pelo desenvolvedor caso o método testado esteja correto. Os tipos desses valores passados como parâmetro podem ser vários (int, double, String, etc.)

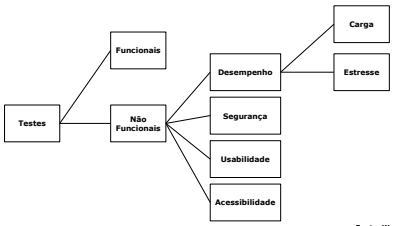
34

JUnit e demais ferramentas

- Métodos setUp() e tearDown()
  - O método setUp() é utilizado para sinalizar o início do processo de teste. Vem antes do método de teste
  - O método tearDown() sinaliza o final do processo, desfazendo o que o setUp() fez. Vem depois do método de teste

35

Taxonomia para testes funcionais e não funcionais



Fonte: Weinfurter, 2023.

36

JUnit e demais ferramentas

- Com base na proposta de esquema de taxonomia, vamos criar nossas meta-anotações personalizadas para folhas dessa estrutura de árvore:
  - @Functional
  - @Security
  - @Usability
  - @Accessiblity
  - @Load
  - @Stress

37

JUnit e demais ferramentas

- Observe que em cada anotação estamos utilizando uma ou mais anotações @Tag, dependendo da estrutura previamente definida

38

JUnit e demais ferramentas

```
package io.github.bonigarcia;

    import java.lang.annotation.ElementType;
    import java.lang.annotation.Retention;
    import java.lang.annotation.RetentionPolicy;
    import java.lang.annotation.Target;
    import org.junit.jupiter.api.Tag;

    @Target({ ElementType.TYPE, ElementType.METHOD })
    @Retention(RetentionPolicy.RUNTIME)
    @Tag("functional")
    public @interface Functional {
    }
```

39

- Todos exemplos utilizados aqui podem ser encontrados no repositório GitHub
  - Disponível em: <https://github.com/bonigarcia/mastering-junit5>

Semanche /shutterstock

40

Boas práticas em testes

41

Boas práticas em testes

- Garantir processos amigáveis às pessoas cria pessoas amigáveis aos processos

Net Vector/shutterstock

42

Boas práticas em testes

- Boas práticas relacionando testes ao processo
  - Uma forte infraestrutura de processo e uma cultura de processo são necessárias para alcançar melhor previsibilidade e consistência. Modelos de processo como o CMMI podem fornecer uma estrutura para construir essa infraestrutura

graphego/shutterstock

43

Boas práticas em testes

- Boas práticas relacionando testes ao processo
  - Integrar processos com tecnologia de forma inteligente é a chave para o sucesso de uma organização
  - Um banco de dados de processos, uma federação de informações sobre a definição e execução de vários processos, pode ser uma adição valiosa ao repertório de ferramentas de uma organização

44

Boas práticas em testes

- Boas práticas relacionando testes a pessoas
  - A chave para o gerenciamento bem-sucedido é garantir que as equipes de teste e desenvolvimento funcionem bem
  - Esse relacionamento pode ser aprimorado pela criação de um senso de propriedade nas metas abrangentes do produto

graphego/shutterstock

45

Boas práticas em testes

- Boas práticas relacionando testes a pessoas
  - Embora metas individuais sejam necessárias para as equipes de desenvolvimento e teste, é muito importante chegar a um entendimento comum das metas gerais que definem o sucesso do produto como um todo

46

Boas práticas em testes

- Boas práticas relacionando testes à tecnologia
  - Um repositório de defeitos pode ajudar na melhor automação das atividades de teste, auxiliando inclusive na escolha dos testes que provavelmente descobrirão defeitos

graphego/shutterstock

47

Boas práticas em testes

- Boas práticas relacionando testes à tecnologia
  - Métricas e medições, densidade de defeitos, taxa de remoção de defeitos e o cálculo dessas métricas são simplificados diante da integração entre essas ferramentas

48

Boas práticas em testes

- Boas práticas relacionando testes à tecnologia
  - A automação de testes e as ferramentas de automação eliminam o tédio e a rotina das funções de teste, além de aumentarem o desafio e o interesse nessas funções

49

- As três dimensões das melhores práticas não podem ser realizadas isoladamente
- Uma boa infraestrutura de tecnologia deve ser adequadamente apoiada por uma infraestrutura de processo eficaz e ser executada por pessoas competentes

Semanche /shutterstock

50

- Essas práticas recomendadas são interdependentes, autossuficientes e se aprimoram mutuamente
- Assim, a organização precisa ter uma visão holística dessas práticas e manter um bom equilíbrio entre as três dimensões

Semanche /shutterstock

51

Over the Shoulder, teste colaborativo

52

Over the Shoulder, teste colaborativo

- Quando um desenvolvedor termina a escrita de uma funcionalidade, ela deve ser mesclada com o projeto principal, por meio de uma ferramenta de versionamento e colaboração de desenvolvimento

53

Over the Shoulder, teste colaborativo

- Durante a "verificação de ombro", em uma tradução livre para de *over the shoulder*, o desenvolvedor mostrará cada critério de aceitação, o QA (Analista de Qualidade) pode pedir ao desenvolvedor para verificar outros casos que conhece e o designer pode validar se os projetos correspondem às especificações

54

Over the Shoulder, teste colaborativo

- **Esse tipo de verificação gera conhecimento entre os times e sobre o funcionamento do produto. A falha de comunicação pode gerar desacordos entre designers e desenvolvedores, mas, ao utilizarmos verificações em equipe para averiguação de como o produto deve ser e funcionar, evitamos a criação de alguns chamados por conta de um bug**

55

- **O processo de testes deve ser implementado para a busca constante de melhores resultados e os testes colaborativos/verificações de ombro são um importante caminho**
- **Reuniões curtas (5 a 10 minutos) entre desenvolvedores, analistas de negócios, designers e analistas de qualidade podem fortalecer o processo de testes colaborativos e o projeto de forma geral**

Semanche /shutterstock

56