

Engenharia de Software

Prof. Vinicius Cardoso Garcia

vcg@cin.ufpe.br :: [@vinicius3w](https://twitter.com/vinicius3w) :: viniciusgarcia.me

[IF977] Engenharia de Software

<http://bit.ly/vcg-es>

Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-
Compartilhual 3.0 Não Adaptada



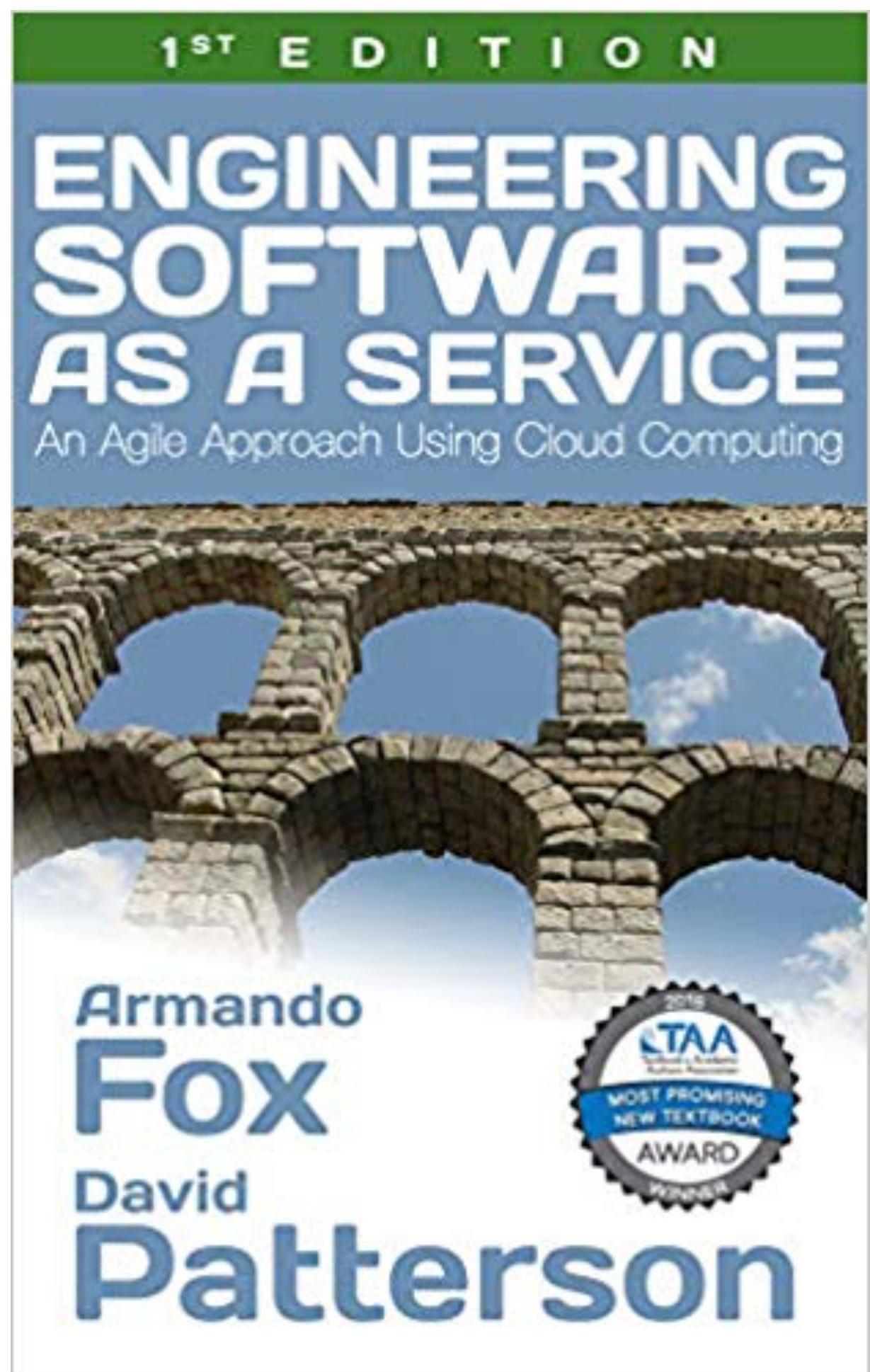
Mais informações visite

[http://creativecommons.org/licenses/by-nc-sa/
3.0/deed.pt](http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt)



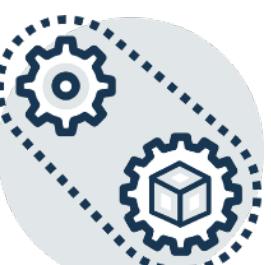
Referências

- A biblioteca do Desenvolvedor de Software dos dias de hoje
 - <http://bit.ly/31WYK5f>
- SWEBOK: Guide to the Software Engineering Body of Knowledge (SWEBOK)
 - <http://www.computer.org/web/swebok>
- Engineering Software as a Service: An Agile Approach Using Cloud Computing
 - <http://www.saasbook.info/>
- Marco Tulio Valente. Engenharia de Software Moderna
 - <https://engsoftmoderna.info/>



```
1 import { Selector, ClientFunction } from 'testcafe';
2
3 fixture `TDD Day Homepage`
4   .page('https://tddday.com');
5
6 test('Page should load and display the correct title', async t => {
7   const actual = Selector('h1').innerText;
8   const expected = 'TDD DAY 2019';
9   await t.expect(actual).eql(expected);
10});
```

Qual a diferença entre Teste Unitário, TDD e BDD?



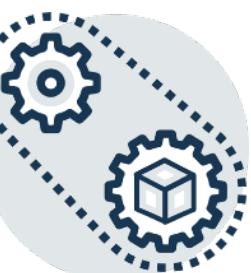
Teste Unitário

- Concentra-se em uma única **unidade de código**
 - geralmente uma função em um objeto ou módulo
- Ao tornar o teste **específico** para uma **única função**, o teste deve ser **simples, rápido de escrever** e **rápido de executar**
- Isso significa que você pode ter **muitos testes de unidade**
 - mais testes de unidade significam **mais erros detectados**
- Eles são especialmente úteis se você precisar **alterar** seu código
 - você pode alterar o código com **segurança** e confiar que outras partes do seu programa **não serão quebradas**



Isolamento

- Um teste de unidade deve ser **isolado** das dependências
 - i.e. nenhum acesso à rede e nenhum acesso ao banco de dados
- Existem ferramentas que podem substituir essas dependências por **falsificações** que você pode controlar
- Isso torna trivial testar todos os tipos de cenários que, de outra forma, exigiriam muita configuração
 - imagine ter que configurar um banco de dados inteiro apenas para executar um teste!?



Um exemplo

- Há um equívoco de que os teste unitários exigem uma sintaxe específica para serem escritos
- Essa chamada sintaxe “**estilo xUnit**” é comum em muitas ferramentas de teste um pouco mais antigas
- Abaixo temos um exemplo do "estilo xUnit", usando o **Mocha**:

```
suite('My test suite name', function() {
  setup(function() {
    //do setup before tests
  });

  teardown(function() {
    //clean up after tests
  });

  test('x should do y', function() {
    //test something
  });
});
```



Mas posso usar Javascript também

```
//suite: User

//test: Name should start empty
var user = new User();
if(user.getName() !== '') {
  throw new Error('User name should start as empty');
}

//test: Password should be hashed
var user = new user();
user.setPassword('hello');
if(user.getPassword() != bcrypt('hello')) {
  throw new Error('User password should be hashed with bcrypt');
}
```



O Básico

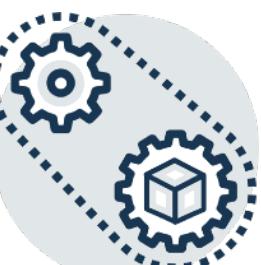
- Testes individuais, que testam uma coisa e são isolados um do outro
- Você pode usar scripts para criar testes unitários rudimentares para o seu código
- Porém, o uso de uma ferramenta de teste unitário real, como o Mocha ou o Jasmine, facilita a gravação de testes
 - e eles têm outros recursos úteis, como relatórios para quando os testes falham (o que facilita descobrir o que deu errado)



Quando escrever testes?

- Existem duas respostas principais para essa pergunta
 1. **Logo após implementar uma pequena funcionalidade.** Por exemplo, escreva alguns métodos e, em seguida, seus testes, que devem passar; isto é, programe um pouco, teste; programe mais um pouco, teste, etc
 2. Ou talvez, **escreva os testes primeiro**, antes de escrever algum método; esses testes não vão passar (no início); só depois que você prover uma implementação mais "real" para o método em questão
- Isto é, teste um código que apenas compila, falhe; implemente o código, teste, sucesso etc. Esse estilo de desenvolvimento chama-se **Test-Driven Development**

Fonte: <https://engsoftmoderna.info>



Quando escrever testes?

- Mas também existem outras respostas
- 3. Quando um usuário reportar um bug, **escreva um teste que reproduza o bug e que, portanto, vai falhar**. Em seguida, corrija o bug; quando isso acontecer, o teste vai passar (e você ganhou mais um teste para sua suíte de testes)
- 4. Quando estiver depurando o código; por exemplo, evite escrever um `System.out.println` para testar o resultado de um método; em vez disso, **escreva um método de teste** (pois a solução baseada em `println` assemelha-se a um teste manual; além disso, depois que remover o `println` o teste é "esquecido")

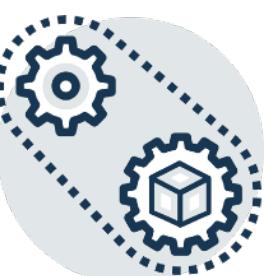
Fonte: <https://engsoftmoderna.info>



Princípios FIRST (para escrita de bons testes)

- **Fast**: testes devem executar rápido; para que possa executá-los sempre.
- **Independent**: não devem existir dependências entre testes; deve ser possível executar os testes em qualquer ordem; deve ser possível executar apenas alguns dos testes disponíveis.
- **Repeatable**: deve ser possível executar um teste em qualquer ambiente ou máquina; e o resultado deve ser o mesmo, independente deste ambiente
- **Self-checking**: ou os testes passam ou falham; simples assim e sem maiores interpretações; a própria IDE já avisa, via barra verde ou vermelha
- **Timely**: você deve escrever os testes o quanto antes; algumas vezes, mesmo antes de todo o código estiver pronto.

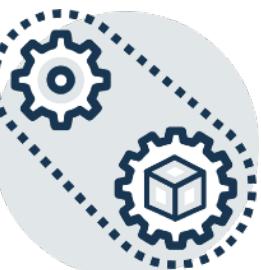
Fonte: Clean Code. Robert C. Martin



Testes Flaky

- Testes com resultados **não-determinísticos**: isto é, em uma execução podem passar; em outra execução, podem falhar
- Flaky = unreliable
- Isto é, não respeitam o R (Repeatable) dos princípios FIRST
- **Atrasam o desenvolvimento**: desenvolvedor perde tempo analisando um teste que falhou, pensando que a causa é a mudança que ele acabou de introduzir no código; no entanto, a falha deve-se a um teste flaky
- Problema mais comum do que se pode imaginar; por exemplo, chega a alcançar **16% dos testes realizados pelo Google em seus sistemas** (veja <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>)

Fonte: <https://engsoftmoderna.info>



Por que alguns testes tem comportamento flaky?

- Concorrência (65%): teste depende do resultado de uma outra thread, cuja execução ocorre de forma concorrente e, portanto, fora do controle do teste

```
1 @Test
2 public void testRsReportsWrongServerName() throws Exception {
3     MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
4     MiniHBaseClusterRegionServer firstServer =
5         (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
6     HServerInfo hsi = firstServer.getServerInfo();
7     firstServer.setHServerInfo(...);
8
9     // Sleep while the region server pings back
10    Thread.sleep(2000);
11    assertTrue(firstServer.isOnline());
12    assertEquals(2,cluster.getLiveRegionServerThreads().size());
13    ... // similarly for secondServer
14 }
```

E se "region server" demorar mais de 2s para dar o "ping"?

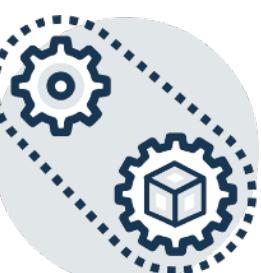
Fonte: An Empirical Analysis of Flaky Tests, FSE 2014.

Fonte: <https://engsoftmoderna.info>

Outros motivos para comportamento flaky? (2)

- **Cache**: em uma execução, o dado pode estar no cache; em outra, não
- **Falhas de rede**: teste de uma falha de comunicação, que nem sempre ocorre
- **Dependências** entre testes (isto é, não respeitam o I dos princípios FIRST).
 - Exemplo: teste X depende do valor de uma variável global ou do estado de um arquivo, que podem ser alterados por outro teste Y; assim, resultado de X depende se Y roda antes ou depois

Fonte: <https://engsoftmoderna.info>



Testing @ Google

- Unit testing is strongly encouraged and widely practiced at Google.
- All code used in production is expected to have unit tests, and the code review tool will highlight if source files are added without corresponding tests.
- Code reviewers usually require that any change which adds new functionality should also add new tests to cover the new functionality.
- Mocking frameworks (which allow construction of lightweight unit tests even for code with dependencies on heavyweight libraries) are quite popular

Fonte: <https://engsoftmoderna.info>

(apud Source: Software Engineering at Google. Fergus Henderson, <https://arxiv.org/abs/1702.01715>)



Testes automatizados

- **Testes unitários:** um único trecho de código (geralmente um objeto ou uma função) é testado, isolado de outras partes
- **Testes de integração:** vários trechos são testados juntos, por exemplo, testando o código de acesso ao banco de dados em um banco de dados de teste
- **Testes de aceitação:** teste automático de todo o aplicativo, por exemplo, usando uma ferramenta como o Selenium para executar automaticamente um navegador



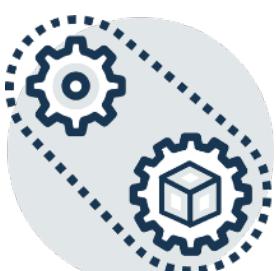
Test-Driven Development

- Um das mais práticas defendidas por métodos ágeis
- Kent Beck. Test-Driven Development: by Example, 2002
- TDD é um processo para quando você escreve seus testes antes do código, assim é possível ter uma cobertura de teste muito alta
- A cobertura do teste refere-se à porcentagem do seu código que é testada automaticamente

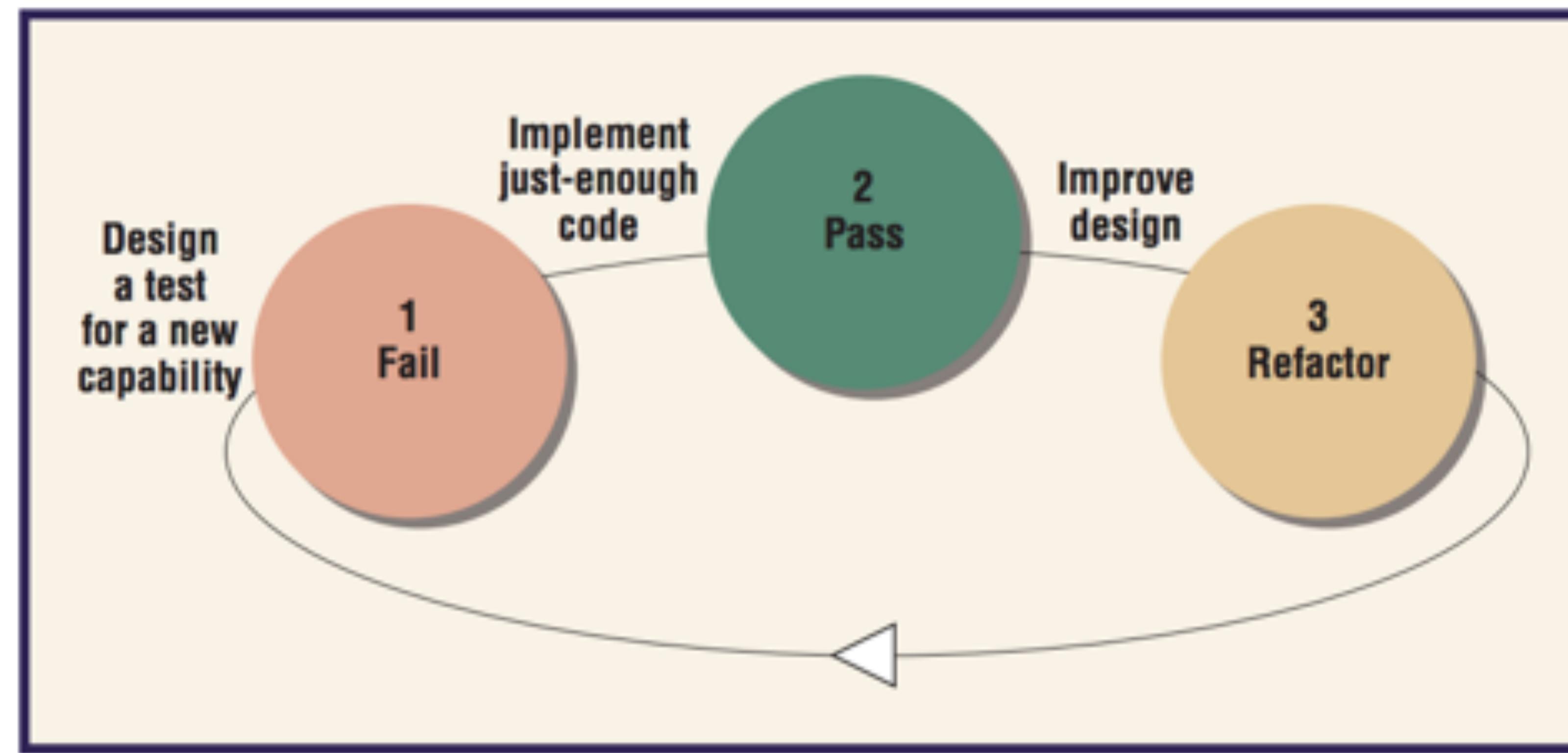


Passos do TDD

1. Comece escrevendo um teste
2. Execute o teste e quaisquer outros testes. Neste ponto, seu teste recém-adicionado deve falhar. Se não falhar aqui, pode não estar testando a coisa certa e, portanto, possui um bug
3. Escreva a quantidade mínima de código necessária para fazer o teste passar
4. Execute os testes para verificar as novas passagens de teste
5. Refatorar opcionalmente seu código
6. Repita a partir de 1



Ciclo TDD = Vermelho, Verde, Refactor

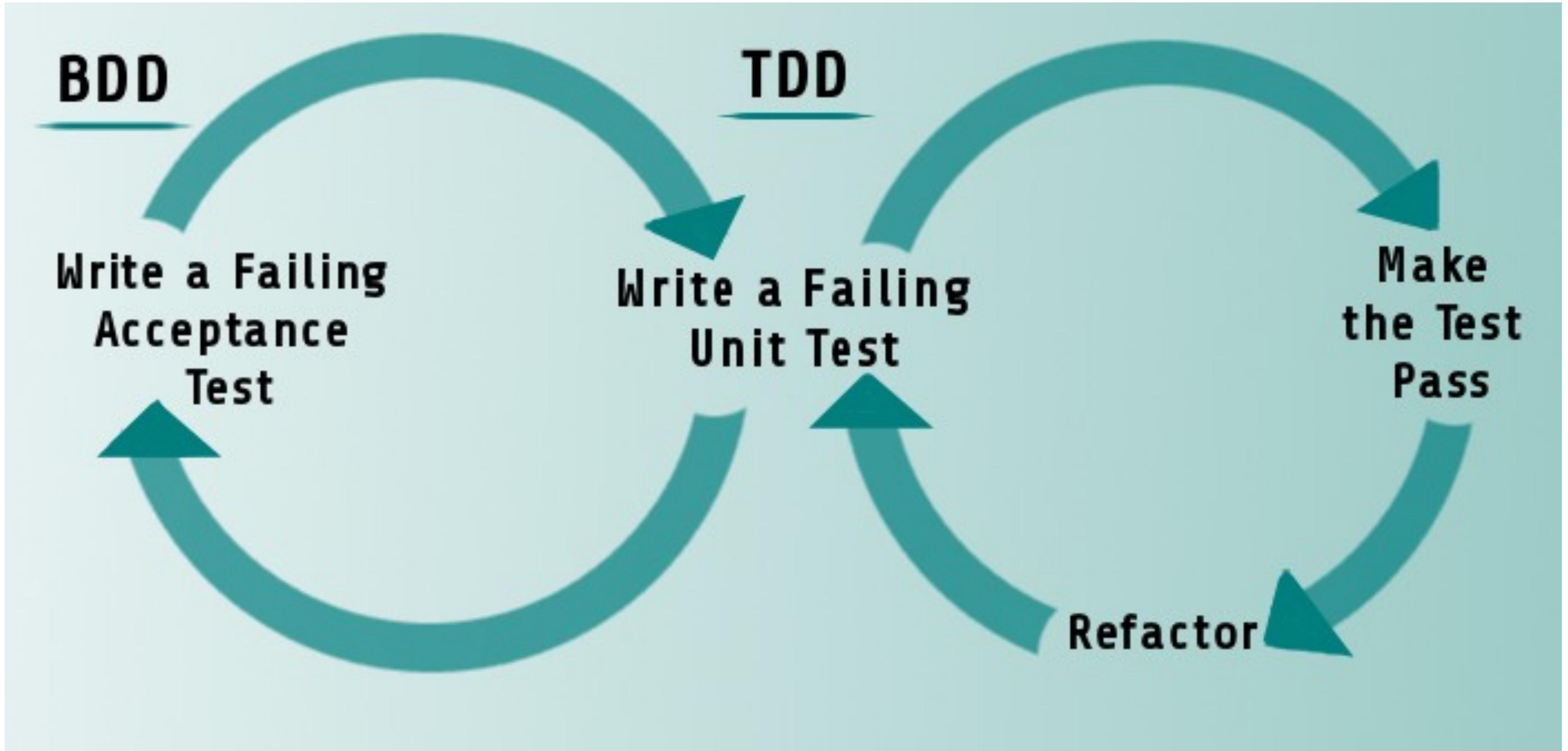


Source: TDD --The Art of Fearless Programming, IEEE Software, R. Jeffries, G. Melnik, 2007

Cobertura e Qualidade

- Os projetos TDD geralmente obtêm uma cobertura de código de 90 a 100%, o que significa que é fácil manter o código e adicionar novos recursos
- Isso ocorre porque você escreve um grande conjunto de testes, para que possa confiar no seu código e as alterações funcionem e também não quebrem nenhum outro código
- A coisa mais difícil do TDD para muitos desenvolvedores é o fato de você precisar escrever seus testes antes de escrever o código





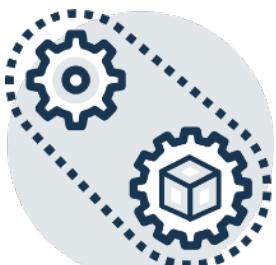
Behaviour-Driven Design

Ferramentas, Padrões e Arquiteturas



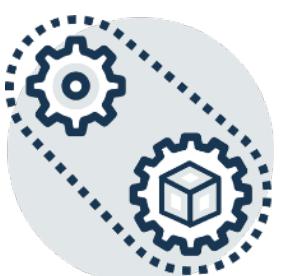
Behavior-Driven Design

- BDD é talvez a maior fonte de confusão
- Quando aplicado a testes automatizados, o BDD é um conjunto de práticas recomendadas para escrever ótimos testes
- BDD pode e deve ser usado junto com os métodos de teste unitário e TDD



Detalhes de implementação

- Uma das principais questões abordadas pelo BDD é o detalhe da implementação em testes unitários
- Um problema comum com testes unitários ruins é que eles dependem muito de como a função testada é implementada
- Isso significa que, se você atualizar a função, mesmo sem alterar as entradas e saídas, [eventualmente] também deverá atualizar o teste



Foco é no comportamento

- O BDD visa soluciona esse problema, mostrando como testar
- Você não deve testar a implementação, mas o comportamento



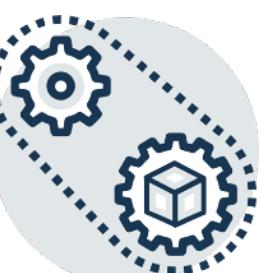
25



Exemplo

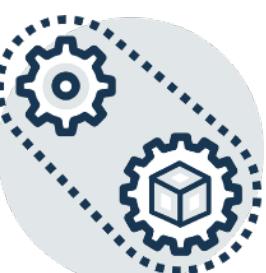
- Este é um teste de unidade de um contador de objetos imaginários
- Testamos que, depois de chamar `tick`, o valor deve ser `1`, o que parece fazer sentido
- Mas há um problema no teste, que depende completamente do fato de o contador iniciar em `0`
- Portanto, o teste depende de duas coisas.
 - O contador começa em `0`
 - Marcar incrementos de `1`
- O fato de o contador iniciar em `0` é um detalhe de implementação que é irrelevante para o comportamento da função `tick()`
- Portanto, ele não deve ter nenhuma influência no teste.
- A única razão pela qual escrevemos o teste dessa maneira é porque estávamos pensando na **implementação**, não no **comportamento**

```
suite('Counter', function() {  
  test('tick increases count to 1', function() {  
    var counter = new Counter();  
  
    counter.tick();  
  
    assert.equal(counter.count, 1);  
  });  
});
```



Qual é o cenário?

- O BDD sugere testar **comportamentos**; portanto, em vez de pensar em **como o código é implementado**, passamos um momento pensando em **qual é o cenário**
- Normalmente, você formula testes de BDD na forma de "isso deve fazer alguma coisa"
- Portanto, ao marcar um contador, ele deve aumentar a contagem em um
- A parte importante aqui é pensar no cenário, e não na implementação, pode levar você a projetar um teste melhor

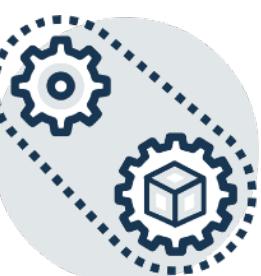


```
describe('Counter', function() {
  it('should increase count by 1 after calling tick', function() {
    var counter = new Counter();
    var expectedCount = counter.count + 1;

    counter.tick();

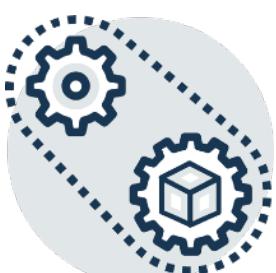
    assert.equal(counter.count, expectedCount);
  });
});
```

- Nesta versão do teste, que usa as funções de estilo BDD do Mocha, removemos os detalhes da implementação
- Em vez de confiar no contador começando em 0, estamos comparando com `counter.count + 1`, **o que faz muito mais sentido em termos de teste contra comportamento**
- Às vezes, seus requisitos mudam
- Vamos imaginar que, por algum motivo, o contador tenha que começar com algum outro valor
- Antes, teríamos que mudar o teste para acomodar isso, mas com a variante BDD não há necessidade de fazê-lo



Sumarizando

- Teste unitário fornece o **o quê**
- TDD fornece a você o **quando**
- BDD fornece a você o **como**
- Embora você possa usar cada um individualmente, combine-os para obter melhores resultados, pois eles se complementam muito bem

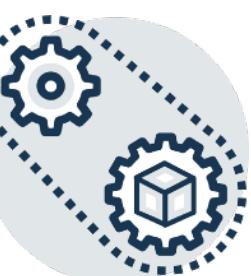


Testes de Integração (vs Testes de Unidade)

- Motivação: no exemplo abaixo, temos dois testes de unidade (que passam), mas o teste de integração falha



Fonte: <https://engsoftmoderna.info>

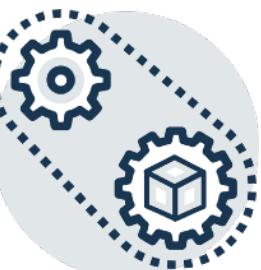


Mais um exemplo....

- 2 unit tests,
- 0 integration tests

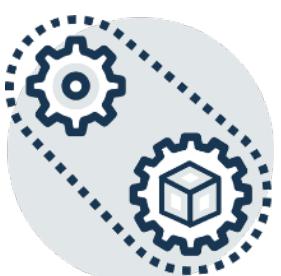


Fonte: <https://engsoftmoderna.info>



Testes de Integração (ou de Serviços)

- Teste que envolve **mais de uma classe ou componente/serviço**
- Pode-se usar um framework como o Mocha (+Chai) também para testes de integração
- Mas são mais difíceis de escrever e demoram mais tempo para executar:
 - Envolvem cenários e dependências mais complexos
 - Por isso, não faz mais sentido usar mocks
 - Usam bancos de dados reais; normalmente, com dados de teste
- Exemplo: simular uma compra em uma loja virtual; incluindo criar carrinho, adicionar produtos; associar carrinho a cliente; definir método de pagamento e entrega; fechar e gravar compra no BD; debitar cartão de crédito (usando um serviço de teste da própria operadora) etc





End to End
Test:

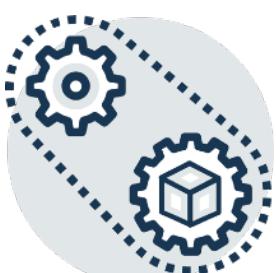
can a user
accomplish an
action?

✓ User sammy123 exists
in the database
(Action: account creation)

Teste fim a fim

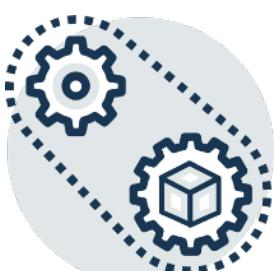
Teste fim a fim

- Hoje, o software está se tornando mais **complexo**
- As aplicações são criadas em **camadas e redes inteiras de subsistemas**, incluindo camadas de **UI e API, bancos de dados externos, redes** e até **integrações de terceiros**
- Quando um falha, o produto inteiro também falha, tornando a estabilidade de cada componente vital para o sucesso de um aplicativo
- Isso significa que há uma clara necessidade de testar o aplicativo inteiro, **do início ao fim** - nas camadas API e UI



O que é teste fim a fim?

- O teste fim a fim é uma **metodologia** usada no ciclo de vida de desenvolvimento de software para testar a **funcionalidade** e o **desempenho** de uma aplicação em circunstâncias e dados **semelhantes ao cenário de produção**
- O **objetivo** é simular a aparência de um cenário real do usuário do **início ao fim**
- A conclusão deste teste não é apenas para **validar o sistema em teste**, mas também para **garantir** que seus **subsistemas funcionem** e se **comportem** conforme o **esperado**.



Benefícios do teste fim a fim

- A realização de testes de ponta a ponta ajudará a garantir que o software esteja pronto para produção e evitar riscos após a liberação
 - **Integridade:** Os testes fim a fim validarão se o software é funcional em todos os níveis - do front ao back-end e em vários sistemas -, além de fornecer uma perspectiva do desempenho em diferentes ambientes
 - **Expande a cobertura de teste:** ao incorporar os diversos subsistemas diferentes em seu processo de teste, você expandirá efetivamente sua cobertura de teste e criará casos de teste adicionais que podem não ter sido considerados anteriormente
 - **Detecta bugs e aumenta a produtividade:** nos testes fim a fim, o software geralmente é testado após cada iteração, o que significa que você poderá encontrar e corrigir quaisquer problemas mais rapidamente. Isso reduzirá as chances de erros, tornando-o mais aprofundado no processo de teste (e também na produção), garantindo assim que os fluxos de trabalho da aplicação operem sem problemas
 - **Reduz os esforços e os custos dos testes:** com menos bugs, falhas e com testes abrangentes a cada passo, os testes fim a fim também diminuirão sua necessidade de repetir testes e, finalmente, os custos e o tempo associados a isso



Diferença entre teste fim a fim e de sistema

Teste fim a fim

Valida o sistema de software e os subsistemas interconectados

A ênfase principal está na verificação do fluxo completo do processo de ponta a ponta.

Durante a execução do teste, todas as interfaces, incluindo os processos de back-end do sistema de software, são levadas em consideração

É executado assim que o teste do sistema é concluído

O teste de ponta a ponta envolve a verificação de interfaces externas que podem ser complexas para automatizar. Portanto, o teste manual é preferido.

Teste de Sistema

Valida apenas o sistema de software de acordo com as especificações de requisitos

Ele verifica as funcionalidades e recursos do sistema.

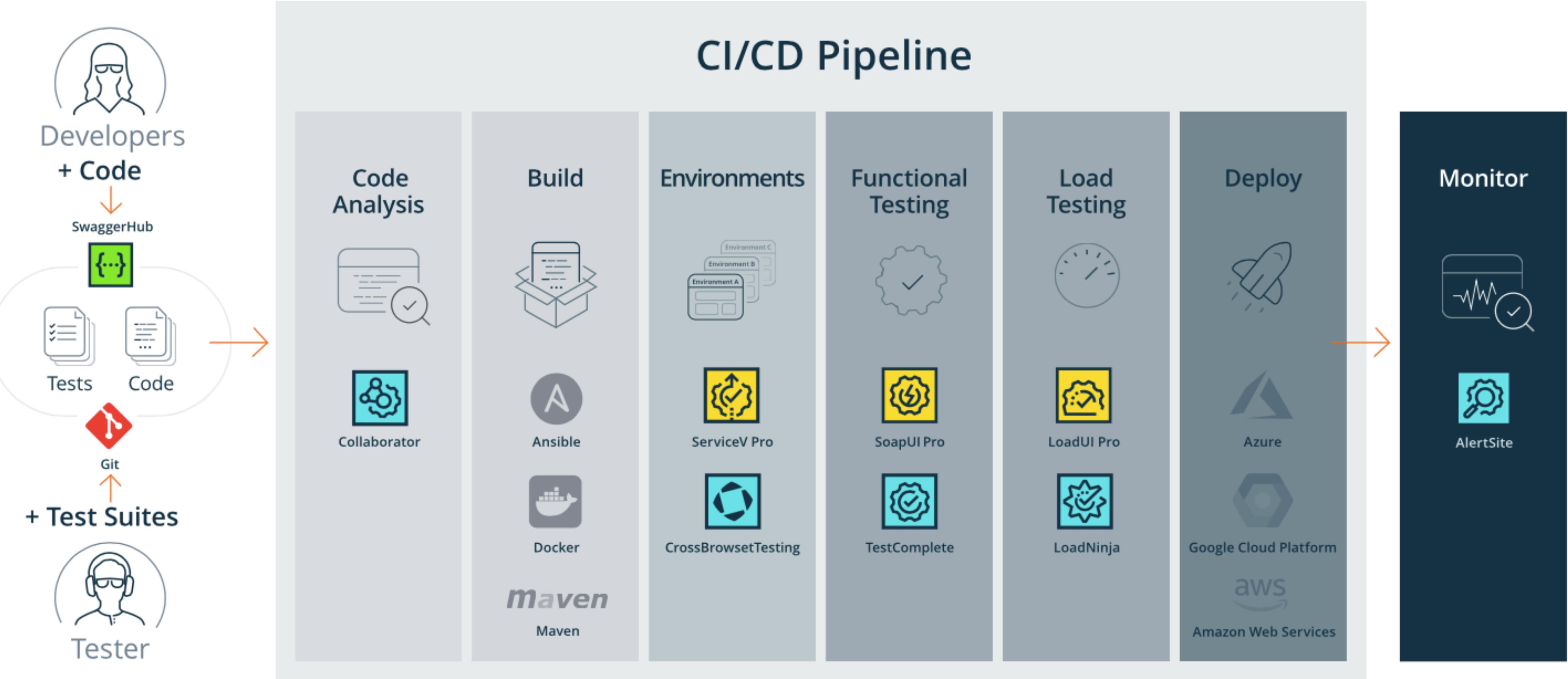
Somente testes funcionais e não funcionais serão considerados para teste.

É executado após o teste de integração

Manual e automação podem ser executados para teste do sistema.



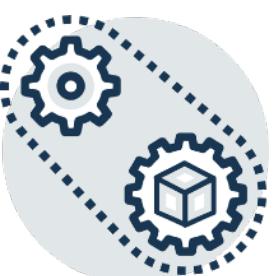
Impulsionone a qualidade em todo o pipeline



Por que testar fim a fim?

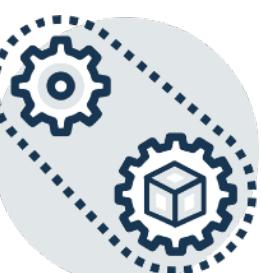
- Existem prós e contras em todos os métodos de teste
- O teste fim a fim é o mais próximo do teste real do usuário, uma das principais vantagens
- Quanto mais próximo o teste estiver de imitar o usuário, maior a probabilidade de ele encontrar problemas que o usuário possa enfrentar
- Se você quiser que um usuário teste os tweets no Twitter, pode dizer algo como:

Vá ao <https://twitter.com> e faça login. Clique na caixa de texto com o espaço reservado "O que está acontecendo?" E digite "Este é um tweet de teste". Clique no botão com o texto "Tweet". Agora, vá para a sua página de perfil e veja o primeiro tweet. O texto deve ser igual a "Este é um tweet de teste".



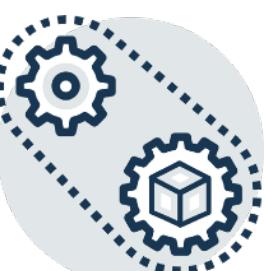
Algumas desvantagens do teste fim a fim são

- Eles são "caros", ou seja, levam muito tempo para serem executados
 - Todo teste requer que um navegador completo seja instanciado com eventos reais do navegador, o que leva mais tempo que os testes de unidade ou integração
- Ele é bom em encontrar problemas, mas não ajuda a resolver esses problemas
 - Seu teste fim a fim pode achar que o sistema de pagamento está com problemas, mas não informa qual dos 10 microsserviços causou o problema



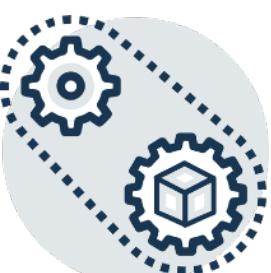
Qual framework de teste fim a fim escolher?

- **Test Cafe** - Esta é o mais recente framework de testes fim a fim e parece ser muito bom. Ele se integra à pilha do navegador, possui bom suporte ao navegador, suporta todas as estruturas de front-end, suporta a sintaxe do ES2015+ e também o typescript. Parece que precisa ter a versão paga para obter testes gravados
- **Puppeteer** - Esta é a solução de código aberto do Google. Parece leve e fácil de usar. É executado no Chromium. O Puppeteer é apresentado como um framework de testes que possui uma funcionalidade rica, melhor do que não ter testes fim a fim, mas não é uma solução completa



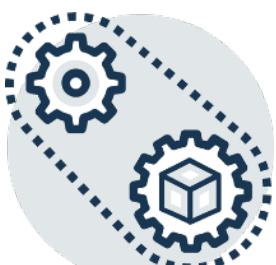
Cypress

- É um framework de teste de código aberto amigável para desenvolvedores
- O Cypress grava snapshots e vídeos de seus testes, possui um console de teste e é gratuito
- É fácil de começar a usar para desenvolvedores e engenheiros de controle de qualidade
- Atualmente, ele dá suporte a apenas variantes do Chrome, mas possui suporte para vários navegadores no roadmap
- Não possui suporte nativo a iframe, embora haja soluções alternativas
- O Cypress tem seu próprio sistema baseado em *promise* que você precisa usar (não pode usar as *promises* do ES6)



Leitura

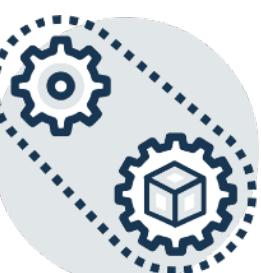
- Aqui está uma boa fonte para uma comparação detalhada do Cypress e Test Cafe
- <http://bit.ly/2WmtACH>, by Tamlyn Rhodes – Mar 29, 2018



Testes de Aceitação

- Teste **manual** realizado pelo cliente, com dados do cliente
 - Por exemplo, em métodos ágeis, uma HU somente é considerada **completa** após passar por **testes de aceitação**, realizados pelos usuários
- Assim, não é mais uma tarefa de **verificação**, mas de **validação**
 - **Verificação**: estamos fazendo o sistema corretamente? (de acordo com a especificação). Todos os testes anteriores eram tarefas de verificação
 - **Validação**: estamos fazendo o sistema correto (aquele que o cliente pediu)
- **Testes alfa**: possível primeira fase de um teste de aceitação, realizado com alguns usuários, em ambiente controlado (ex: máquinas desenvolvedores)
- **Testes beta**: teste de aceitação realizado com um grupo maior de usuários

Fonte: <https://engsoftmoderna.info>



Testes de SLA

- Conhecidos por Não-Funcionais
- Testes anteriores (com exceção dos testes de aceitação) verificam requisitos funcionais; logo, são testes funcionais
- Objetivo de testes funcionais: **encontrar bugs**
- **Teste de SLA:** verifica (ou valida) requisitos não-funcionais para atendimento ao nível de qualidade de provimento de serviço aos clientes
- **Teste de Performance:** simular o uso do sistema com alguma carga
- **Teste de Usabilidade:** simular o uso do sistema com alguns usuários
- **Teste de Falhas:** simular a queda de um data-center ou de alguns microservices

Fonte: <https://engsoftmoderna.info>



Verificação vs Validação

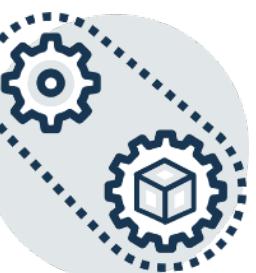


Fonte: <https://engsoftmoderna.info>

Explicando melhor o diagrama do slide anterior

- Assim como os testes do quadro da esquerda, **revisão de código** também é uma técnica de **verificação**
 - Porém, ao contrário dos testes listados, revisão de código é **manual**
- Assim como os testes do quadro da esquerda, **linters** são ferramentas que podem ser usados para **verificação de sistemas**
 - Porém, linters são **baseados em técnicas de análise estática** (eles apenas "leem" o código do sistema, sem executá-lo); já testes são técnicas que requerem **análise dinâmica** (isto é, eles executam o código)
 - Exemplos de linters: lint (C), FindBugs (Java), JSHint (JavaScript), etc
 - Linters também checam questões de estilo, bad smells etc.

Fonte: <https://engsoftmoderna.info>



Leitura

- Aula Refactorings do Prof. Marco Túlio
- Aula Prática (Java) do Prof. Marco Túlio



48

