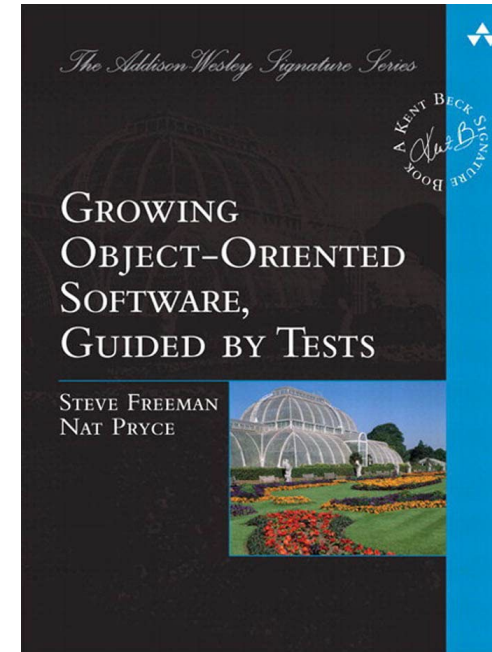
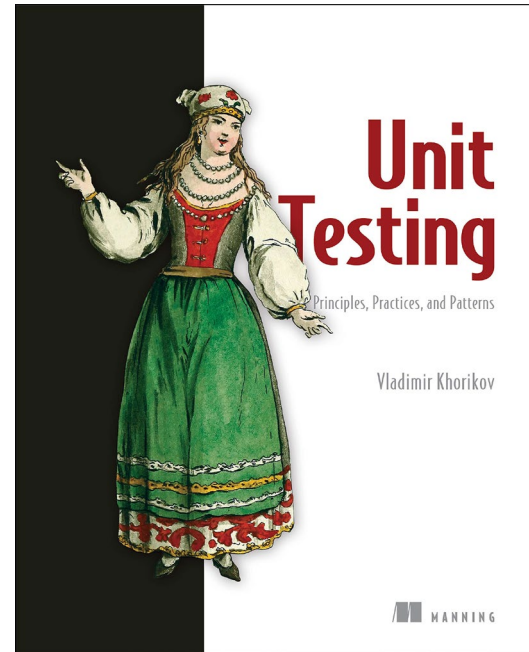
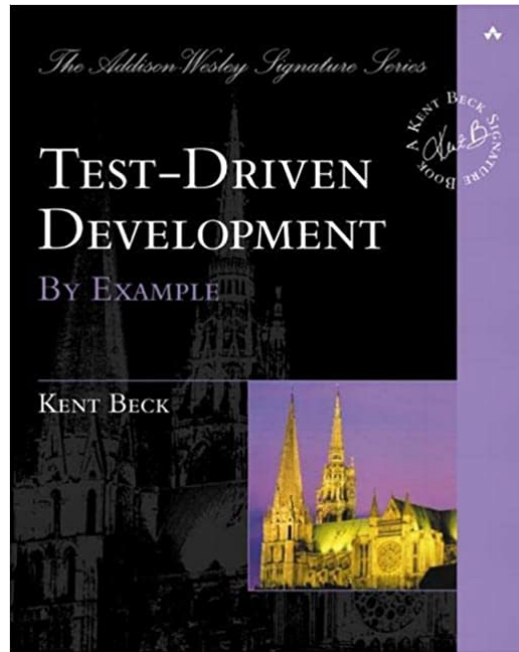


# Testes Unitários vs. Integrados e Aspectos de Qualidade

Autores: Bruno e Cristian

# Bibliografia



# Organização das Unidades

- Unidade I: trata das diferenças entre testes unitários e de integração, além dos conceitos de SUT e Collaborator bem como Dependências compartilhadas, privadas e externas ao processo.
- Unidade II: trata mais profundamente das características de um teste e implicações na qualidade de um software como produto.

# Unidade I: Testes Unitários vs. Integrados

# É um Teste Unitário ou Integrado?

C3: Sistema de folha de pagamento da Chrysler. Tinha sido feito em COBOL. O projeto depois foi migrado pra Smalltalk, voltou pra COBOL e depois foi trocado por um ERP. Fowler com isso reforça que Agile e XP não são garantias de sucesso por si só.

- Não existe resposta unânime
- Existem 2 visões que respondem a pergunta
  - Escola de Londres: “estilo” da comunidade de programadores surgida em Londres nos anos 2000. Conhecidos como “mockistas”.
  - Escola Clássica ou Detroit: “clássico” porque boa parte do mundo aprendeu com TDD nos livros de Kent Beck. Martin Fowler no artigo [Mocks Aren't Stubs](#) explica que o termo surgiu no projeto C3 da Chrysler em Chicago, berço do movimento XP iniciado pela liderança de Kent Beck em 1996.
- Existem muitas definições do que seja um teste unitário
- Khorikov define 3 propriedades essenciais de todo teste que podemos considerar “unitário”

# Atributos de um Teste Unitário

- Verifica uma pequena porção de código (chamamos de Unidade)
- Executa rapidamente
- Roda de forma isolada → aqui se origina divergências entre a escola clássica e de Londres
- Vamos analisar um cenário de negócio para poder compreender mais profundamente o que o autor quer mostrar

# Vamos montar uma narrativa de negócio com JUnit

Abordagem chamada de Comportamental (BDD-like)

Exemplo: Cliente faz compras  
numa Loja



Arrange, Act, Assert : mesmo que Dado, Quando e Então para a comunidade de BDD

```
public class ClienteFazComprasEstiloDetroitTests {  
    @Test  
    void compraBemSucedida_quandoEstoqueSuficiente() {  
        // Arrange  
        var loja = new Loja();  
        loja.adicionaProdutoAoEstoque(new Produto( nome: "Shampoo"), quantidade: 10);  
        var cliente = new Cliente();  
  
        // Act  
        var sucesso : boolean = cliente.compra(loja, new Produto( nome: "Shampoo"), quantidade: 5);  
  
        // Assert  
        assertTrue(sucesso);  
        assertEquals( expected: 5, loja.obtemTotalEstoque(new Produto( nome: "Shampoo")));  
    }  
}
```

```
public class Cliente {
```

```
    2 usages
```

```
    public boolean compra(Loja loja, Produto produto, int quantidade) {  
        return true;
```

```
    }
```

```
}
```

```

public class Loja {
    1 usage
    public void adicionaProdutoAoEstoque(Produto produto, int quantidade) {
        // TODO: este método é um stub (tipo de test double). Precisa ser removido em novas refatorações.
    }

    1 usage
    public int obterTotalEstoque(Produto shampoo) {
        return 5; // TODO: este método é um stub (tipo de test double). Precisa ser removido em novas refatorações.
    }
}

```

✓ Tests passed: 1 of 1 test – 30 ms

```

> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
> Task :jacocoTestReport
BUILD SUCCESSFUL in 1s
4 actionable tasks: 3 executed, 1 up-to-date
14:23:38: Execution finished ':test --tests "org.example.unittest.ClienteFazComprasEstiloDetroitTests"'.

```

```

@Test
void compraFalha_quandoEstoqueForInsuficiente() {
    // Arrange
    var loja = new Loja();
    loja.adicionaProdutoAoEstoque(new Produto( nome: "Shampoo"), quantidade: 10);
    var cliente = new Cliente();

    // Act
    var sucesso : boolean = cliente.compra(loja, new Produto( nome: "Shampoo"), quantidade: 15);

    // Assert
    assertFalse(sucesso);
    assertEquals( expected: 10, loja.obtemTotalEstoque(new Produto( nome: "Shampoo")));
}

```

✖ Tests failed: 1, passed: 1 of 2 tests – 34 ms

expected: <false> but was: <true>

Expected :false

Actual :true

[<Click to see difference>](#)

org.opentest4j.AssertionFailedError: expected: <false> but was: <true>

# O código não satisfaz novos cenários. É hora da implementação satisfatória.

Kent Beck afirma “Dirty hands first”. Não faça perfeição. Atenda os requisitos mas com bom senso. Depois de tudo “verde”, aí sim refatore seguindo boas práticas de codificação e padrões de projeto. Obs.: evitei refatorar muito para ser mais breve

@Test

```
void compraBemSucedida_quandoEstoqueSuficiente() {
```

```
    // Arrange
```

```
    var loja = new Loja();
```

```
    loja.adicionaProdutoAoEstoque(new Produto( nome: "Shampoo"), quantidade: 10);
```

```
    var cliente = new Cliente();
```

```
    // Act
```

```
    var sucesso : boolean = cliente.compra(loja, new Produto( nome: "Shampoo"), quantidade: 5);
```

```
    // Assert
```

```
    assertTrue(sucesso);
```

```
    assertEquals( expected: 5, loja.obtemTotalEstoque(new Produto( nome: "Shampoo")));
```

```
}
```

@Test

```
void compraFalha_quandoEstoqueForInsuficiente() {
```

```
    // Arrange
```

```
    var loja = new Loja();
```

```
    loja.adicionaProdutoAoEstoque(new Produto( nome: "Shampoo"), quantidade: 10);
```

```
    var cliente = new Cliente();
```

```
    // Act
```

```
    var sucesso : boolean = cliente.compra(loja, new Produto( nome: "Shampoo"), quantidade: 15);
```

```
    // Assert
```

```
    assertFalse(sucesso);
```

```
    assertEquals( expected: 10, loja.obtemTotalEstoque(new Produto( nome: "Shampoo")));
```

```
}
```

```
public class Cliente {
```

4 usages

```
| public boolean compra(Loja loja, Produto produto, int quantidade) {  
|     var estoqueInsuficiente : boolean = !loja.temEstoqueSuficiente(produto, quantidade);  
  
|     if (estoqueInsuficiente) return false;  
  
|     loja.removeProdutoEstoque(produto, quantidade);  
  
|     return true;  
| }  
}
```



```
public class Loja {
```

3 usages

```
public Map<Produto, Integer> estoque = new HashMap<>();
```

2 usages

```
public void adicionaProdutoAoEstoque(Produto produto, int quantidade) {  
    estoque.put(produto, quantidade);  
}
```

2 usages

```
public long obterTotalEstoque(Produto produtoAFiltrar) {  
    var registroEstoque : Optional<Entry<Produto, Integer>> = estoque  
        .entrySet() Set<Entry<Produto, Integer>>  
        .stream() Stream<Entry<Produto, Integer>>  
        .filter((registro) → registro.getKey().getNome().equals(produtoAFiltrar.getNome()))  
        .findFirst();  
  
    var quantidadeTotalPorProdutoFiltrado : Integer = registroEstoque.get().getValue();  
  
    return quantidadeTotalPorProdutoFiltrado;  
}
```

Map é uma sequencia entry1, entry2, ...

Onde entry é uma tupla (Key, Value)

(produto, quantidadeNoEstoque),  
(k2, v2),  
...

Produto : objeto de Produto  
quantidadeNoEstoque : wrapper de Integer

Atenção: Map não é coleção, mas pode  
assumir uma visão de coleção com uso de  
entrySet

```

public void removeProdutoEstoque(Produto produtoAFiltrar, int quantidadeARemover) {
    var registroEstoque : Optional<Entry<Produto, Integer>> = estoque
        .entrySet() Set<Entry<Produto, Integer>>
        .stream() Stream<Entry<Produto, Integer>>
        .filter((registro) → registro.getKey().getNome().equals(produtoAFiltrar.getNome()))
        .findFirst();

    var quantidadeNoEstoque : Integer = registroEstoque.get().getValue();

    if (quantidadeARemover > quantidadeNoEstoque) throw new RuntimeException("Estoque insuficiente!");

    registroEstoque.get().setValue(quantidadeNoEstoque - quantidadeARemover);
}

```

✓	✓ Test Results	36 ms
✓	✓ ClienteFazComprasEstiloDetroitTests	36 ms
✓	✓ compraBemSucedida_quandoEstoqueSuficiente()	34 ms
✓	✓ compraFalha_quandoEstoqueForInsuficiente()	2 ms

Nova Refatoração

```
public class Loja {
```

2 usages

```
    public Map<Produto, Integer> estoque = new HashMap<>();
```

2 usages

```
    public void adicionaProdutoAoEstoque(Produto produto, int quantidade) {  
        estoque.put(produto, quantidade);  
    }
```

2 usages

```
    public long obtemTotalEstoque(Produto produtoAFiltrar) {  
        var registroEstoque : Optional<Entry<Produto, Integer>> = busqueUmRegistroDeEstoque(produtoAFiltrar);  
  
        var quantidadeTotalPorProdutoFiltrado : Integer = registroEstoque.get().getValue();  
  
        return quantidadeTotalPorProdutoFiltrado;  
    }
```










3 usages

```
    public void removeProdutoEstoque(Produto produtoAFiltrar, int quantidadeARemover) {  
        var registroEstoque : Optional<Entry<Produto, Integer>> = busqueUmRegistroDeEstoque(produtoAFiltrar);  
        var quantidadeNoEstoque : Integer = registroEstoque.get().getValue();  
  
        registroEstoque.get().setValue(quantidadeNoEstoque - quantidadeARemover);  
    }
```

```
public boolean temEstoqueSuficiente(Produto produtoAVerificar, int quantidade) {  
    var registroDeEstoque : Optional<Entry<Produto, Integer>> = busqueUmRegistroDeEstoque(produtoAVerificar);  
  
    var quantidadeNoEstoque : Integer = registroDeEstoque.get().getValue();  
  
    return quantidade ≤ quantidadeNoEstoque;  
}
```

3 usages

```
public Optional<Map.Entry<Produto, Integer>> busqueUmRegistroDeEstoque(Produto produtoABuscar) {  
    return estoque  
        .entrySet() Set<Entry<Produto, Integer>>  
        .stream() Stream<Entry<Produto, Integer>>  
        .filter((registro) → registro.getKey().getNome().equals(produtoABuscar.getNome()))  
        .findFirst();  
}
```

Element	Missed Instructions	Cov.	Missed Branches	Cov.
 <a href="#">ProductionCode</a>		50%		n/a
 <a href="#">Loja</a>		100%		100%
 <a href="#">Cliente</a>		100%		n/a
 <a href="#">Produto</a>		100%		n/a
Total	3 of 114	97%	0 of 2	100%

Teste de comportamento (BDD-style) nos permite alcançar patamares maiores de cobertura com poucas linhas de código

Tem muito o que melhorar... O foco é na relação entre SUT e Collaborator. Aliás, o que são?

# Introdução ao SUT e Collaborator

- SUT = System Under Test
- Observando os testes, quem é o SUT?
  - SUT é quase sempre uma Classe específica. A Classe Cliente é o nosso SUT.
  - Por quê? Cliente faz uma compra. O autor da ação é Cliente e adotamos a perspectiva dele para executar a compra numa loja.
  - A Classe Cliente está mais acabada. As classes que provavelmente vão se modificar mais será Loja e Produto.



# Introdução ao SUT e Collaborator

- Collaborator = auxiliador para que SUT consiga ser implementado
- Um SUT pode ter vários Collaborators.
  - Loja é um Collaborator: suporta o método de compra e guarda o estoque
  - Produto: também é um Collaborator

Agora, tudo o que fizemos foi um  
Teste Unitário?

Não é tão simples dizer. As  
Escolas de Londres e Detroit  
divergem.

Em que ponto divergem? É o  
chamado “Problema do Isolamento”

# Problema do Isolamento: Escola de Londres

- Para a Escola de Londres, se sua classe tem uma dependência com outra classe ou várias, você precisa substituir todas as dependências por *test doubles* (será explicado)
- Podemos nos focar numa classe a ser testada exclusivamente separando seu comportamento de qualquer influência externa
- Principal benefício: se um teste falhar, você sabe exatamente onde falhou, sem ficar suspeitando de outras classes vizinhas.
  - Classes tem capacidade de inserir dependência circular, nos enganando no debugging, mandando para o início de tudo
- Seu projeto tem muitas classes interconectadas? Acha impossível criar testes unitários?
  - Test doubles pode colocar um fim nisso, substituindo dependências imediatamente de uma classe de forma recursiva

# Test double (Dublê de teste) e Mock

- O que é um test double?
  - Tem significado amplo, descrevendo todos os tipos de dependências fake e não preparadas para produção num teste
- O que é um Mock?
  - Mock é apenas um tipo de dublê de teste
  - O foco dele é nos ajudar a testar o fluxo de um objeto forjando estados internos de collaborators.
- Existem outros dublês como o dummy, stub, spy, in-memory database, etc

# Problema do Isolamento: Escola de Londres

- Tem uma Classe de Implementação (onde vai ficar o código para produção)? Crie uma Classe correspondente de teste
  - Exemplo: Cliente.java, Loja.java são a implementação e ClienteFazComprasTests.java é o teste
- A Loja eventualmente vai precisar se conectar com um Banco de Dados ou API que estão numa classe externa, certo? Aplique algum tipo de test double.

A Escola de Londres se  
“diferenciou” da abordagem  
antiga, dita “clássica” ou Detroit

Só tivemos conhecimento de que a abordagem anterior era de fato a clássica devido ao surgimento da nova visão trazida por Londres



# Problema do Isolamento: Escola Clássica

- O código que implementa “O Cliente compra numa Loja” feito, se observado pela Escola de Londres, é classificado como “clássico”
- A Escola Clássica enxerga o problema do isolamento da seguinte maneira:
  - A classe Loja é usada da maneira mais próxima do código em produção (production-ready)
  - Verificamos com isso que efetivamente testamos tanto Cliente quanto Loja e não apenas Cliente
  - Qualquer bug encontrado em Loja vai afetar Cliente mesmo que o código de Cliente esteja correto
  - Não temos isolamento de Cliente para Loja. Cliente depende de Loja.

Evoluindo para abordagem  
“mockista” ou de Londres

# Problema do Isolamento: Escola de Londres

```
public class ClienteFazComprasEstiloLondresTests {  
    @Test  
    void compraBemSucedida_quandoEstoqueSuficiente() {  
        // Arrange  
        var lojaMockada : Loja = mock(Loja.class);  
        var produtoAComprar : Produto = mock(Produto.class);  
  
        when(lojaMockada.temEstoqueSuficiente(produtoAComprar, quantidade: 5)).thenReturn(value: true);  
  
        var cliente = new Cliente();  
  
        // Act  
        var sucesso : boolean = cliente.compra(lojaMockada, produtoAComprar, quantidade: 5);  
  
        // Assert  
        assertTrue(sucesso);  
  
        verify(lojaMockada, atMostOnce()).removeProdutoEstoque(produtoAComprar, quantidadeARemover: 5);  
    }  
}
```

@Test

```
void compraFalha_quandoEstoqueForInsuficiente() {  
    // Arrange  
    var lojaMockada : Loja = mock(Loja.class);  
    var produtoAComprar : Produto = mock(Produto.class);  
  
    when(lojaMockada.temEstoqueSuficiente(produtoAComprar, quantidade: 5)).thenReturn(value: false);  
  
    var cliente = new Cliente();  
  
    // Act  
    var sucesso : boolean = cliente.compra(lojaMockada, produtoAComprar, quantidade: 5);  
  
    // Assert  
    assertFalse(sucesso);  
  
    verify(lojaMockada, never()).removeProdutoEstoque(produtoAComprar, quantidadeARemover: 5);  
}
```

# Problema do Isolamento: Tipos de Dependências

# Shared, Private e Out-of-Process

- Dependência Compartilhada (Shared)
  - Tem potencial de afetar o resultado de vários testes
  - Exemplos
    - Uso de campos estáticos mutáveis. Quando se altera o valor num determinado teste, caso outro venha fazer uso, pode gerar resultados diferentes num mesmo processo de execução dos testes
    - Bancos de Dados tradicionais costumam gerar o mesmo efeito num conjunto de testes.

# Shared, Private e Out-of-Process

- Dependência Privada (Private)
  - Logicamente, não é compartilhada
  - Isto é, caso um teste venha utilizá-la, não tem capacidade de afetar os outros testes
- **IMPORTANTE:** é a propriedade que define um teste unitário para Escola de Londres

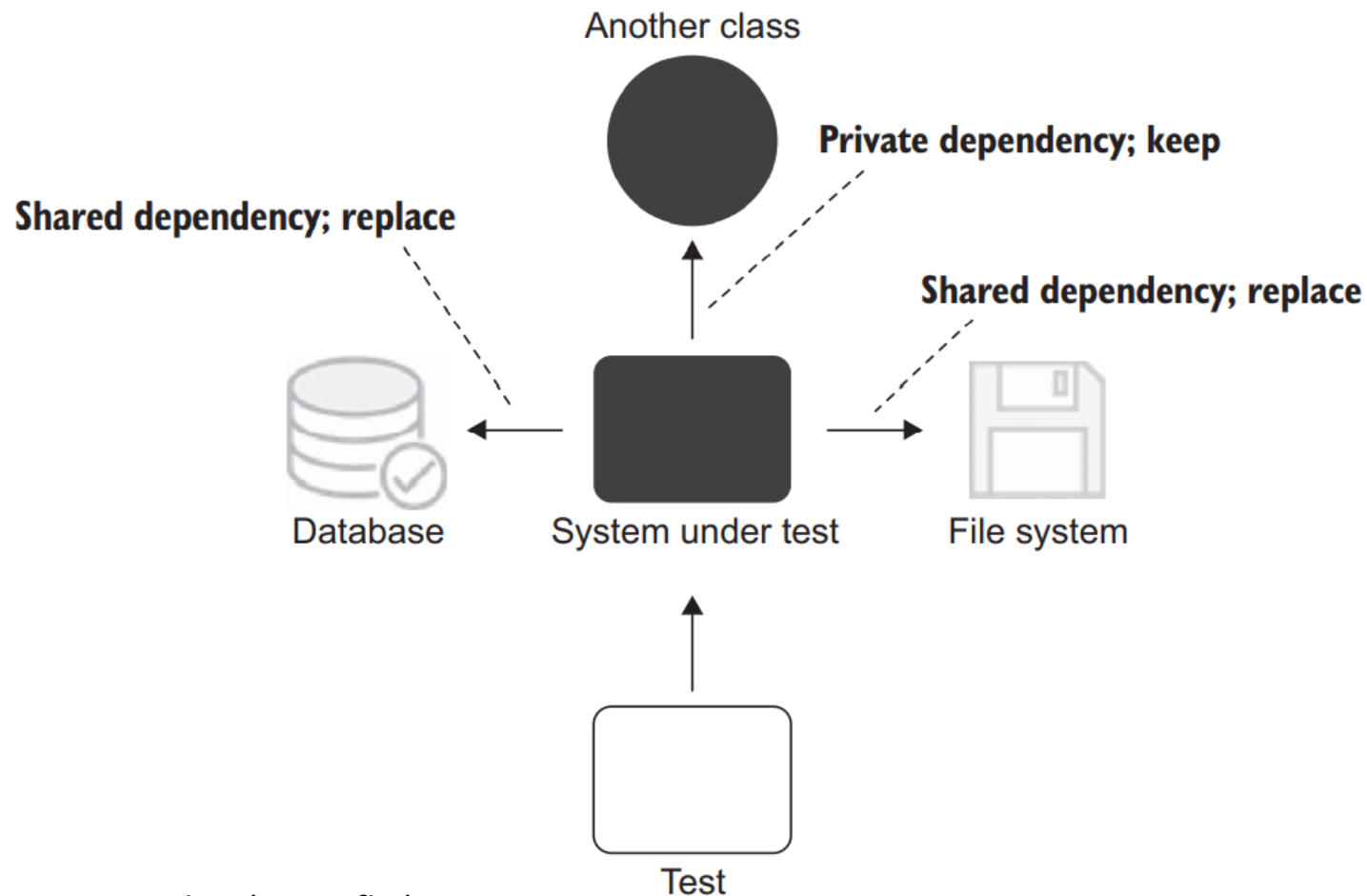
# Shared, Private e Out-of-Process

- Dependência Externa ao Processo (Out-of-Process)
  - Roda fora do processo de execução dos testes
  - Na maioria dos casos, é uma dependência compartilhada, mas nem sempre.
  - Exemplos:
    - DynamoDB é fora de processo. Tradicionalmente é compartilhada. Caso venha utilizar alguma versão estilo in-memory, como ocorre com uso do testcontainers, aí se torna privado.
    - Existem alguns DBs read-only ou até mesmo APIs.
      - InnoDB read-only: é fora de processo e privado, mesmo que seja utilizado por mais de um teste.
      - APIs read-only: fora de processo e privado



5

Tenho uma classe usada exclusivamente para um teste. Mantenha.



4

Message Broker (Ex. Kafka)

- É externo ao processo e compartilhado.
- O que fazer? Mock ou use in-memory.
  - Algumas vezes o in-memory também não é muito eficiente. Mock.

1

DB “normal”: caso utilize nos testes...

- Externo ao processo: roda num processo fora do processo de testes iniciado pelo SO
- Compartilhado: alteração do estado pode afetar o resultado dos testes
- O que podemos fazer? Mock ou use in-memory.

2

DB “in-memory”:

- Externo ao processo: roda num processo diferente dos testes
- Privado: caso cada instância de teste limpe a base de dados, como costuma ocorrer ao recriar um DB in-memory, um teste não irá afetar o outro
- O que podemos fazer? Utilize. Caso seja muito complexo e afetar a performance, Mock.

3

Sistema de Arquivos:

- Tem potencial de ser Compartilhado e Externo assim como DB tradicional.
- O que fazer? Mock ou use in-memory.

# Uso recursos da AWS. O que fazer?

- Recursos da AWS (sejam filas SQS, buckets do S3 e etc) são dependências externas ao processo e compartilhadas
  - Tem potencial de criar testes com resultados diferentes.
- O que fazer?
  - Há soluções “in-memory”: localstack com Docker. No entanto, devido a complexidade da simulação, consome muita memória.
  - A AWS oferece o Step Functions no modo “simulação de integração”. No entanto, incorremos ao custo do uso do serviço.
  - Em Java não existem projetos muito difundidos até o momento. Em Python existe o projeto ‘Moto’.

# Mock com moderação

- Está pensando em mockar uma chamada de API do DynamoDB?
  - Não seria melhor criar um interface como Repository Pattern e mockar sua implementação?
  - Mocks podem gerar muita complexidade. Serviços como o DynamoDB são pouco triviais.
  - Mockar um repository faz mais sentido porque repository abstrai o acesso direto a um banco de dados. Caso, no futuro, precise trocar a tecnologia de DB, seu teste não será afetado.

# O que é um Teste Integrado?

- Em resumo: é tudo o que não é unitário. End-to-end também é integração.
- As escolas de Londres e Clássica também divergem na definição do que seria “teste de integração”.
- A divergência surge na visão do problema do isolamento
- Escola de Londres considera que
  - Qualquer teste que usa um Collaborator production-ready seja um Teste Integrado
  - Logo, a maioria das abordagens clássicas são encaradas, pela Escola de Londres, como Testes de Integração
    - Teste de ‘Cliente faz compras’ é tratado pela clássica como teste unitário mas Londres encara como integração

# Recapitulando, o que são testes unitários?

- Para a Escola Clássica, deve reunir as características:
  - Verifica uma pequena porção do código
  - Executa rapidamente
  - Utiliza dependências compartilhadas
- Para a Escola de Londres:
  - Verifica um Unidade Simples de Comportamento
  - Executa rapidamente
  - Utiliza dependências privadas

# Dica para tomada de decisão (litmus test)

- A dependência usada no seu teste é mutável?
  - Substitua.
- É imutável?
  - Mantenha.

Caso a instância de Produto seja imutável, não precisa mockar.  
Caso típico: Value Objects são imutáveis. Não mocke.

```
@Test
void compraFalha_quandoEstoqueForInsuficiente() {
    // Arrange
    var lojaMockada : Loja = mock(Loja.class);
    var produtoAComprar : Produto = mock(Produto.class);

    when(lojaMockada.temEstoqueSuficiente(produtoAComprar, quantidade: 5)).thenReturn(value: false);

    var cliente = new Cliente();

    // Act
    var sucesso : boolean = cliente.compra(lojaMockada, produtoAComprar, quantidade: 5);

    // Assert
    assertFalse(sucesso);

    verify(lojaMockada, never()).removeProdutoEstoque(produtoAComprar, quantidadeARemover: 5);
}
```

# Visão geral de Testes Integrados

- Um teste integrado verifica duas ou mais unidades de comportamento.
- Também quando verificamos dois ou mais componentes criados por times separados que trabalham juntos.
- Em resumo, teste integrado é um teste que verifica se seu código funciona integrado com dependências compartilhadas, dependências out-of-process ou desenvolvido por outros times na organização

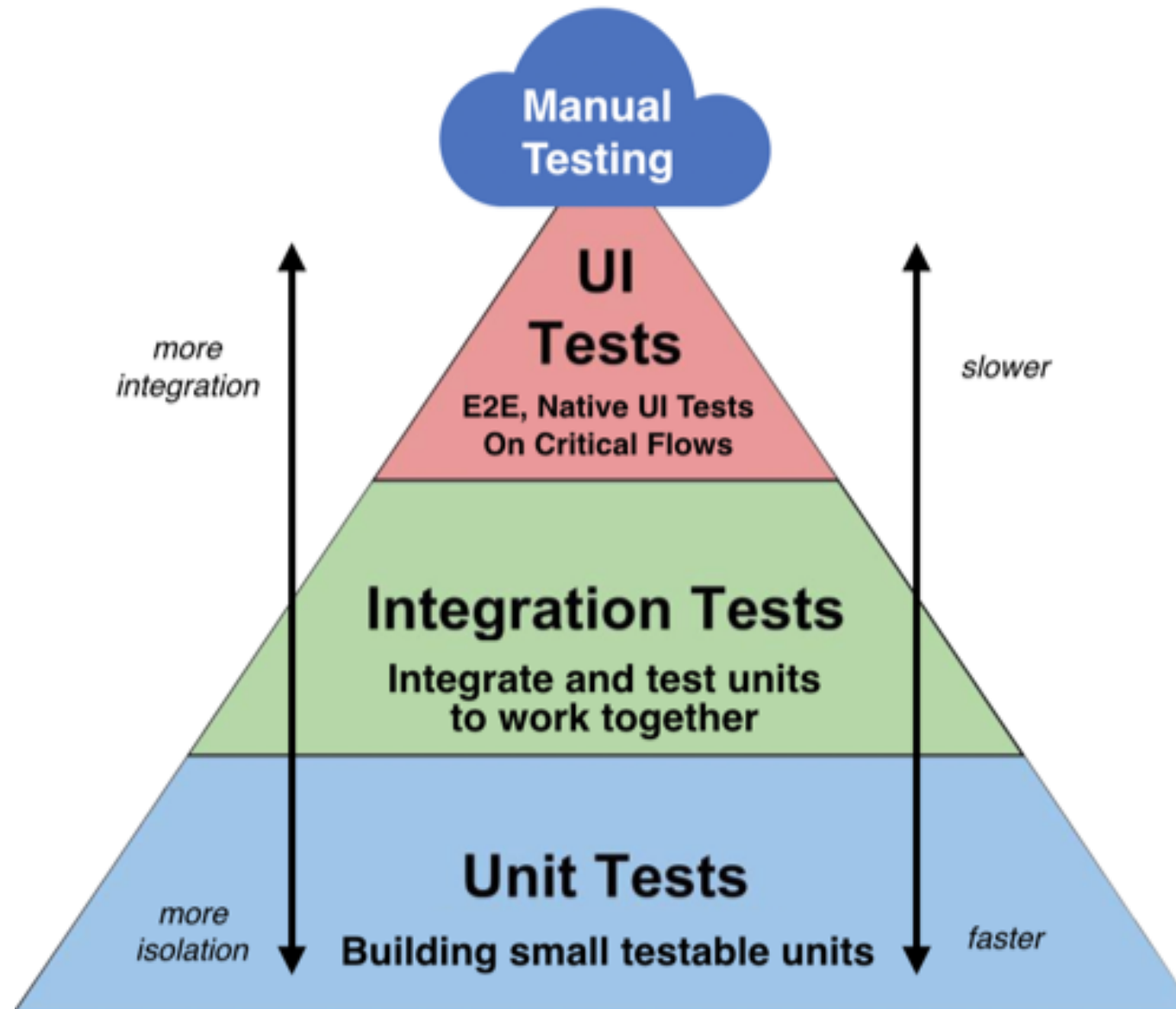


# Visão geral de Testes Integrados

- Um teste integrado verifica duas ou mais unidades de comportamento.
- Também quando verificamos dois ou mais componentes criados por times separados que trabalham juntos.
- Em resumo, teste integrado é um teste que verifica se seu código funciona integrado com dependências compartilhadas, dependências out-of-process ou desenvolvido por outros times na organização

# Visão geral de Testes Integrados

- Devido a possibilidade de uso de dependências compartilhadas, os testes integrados
  - Podem performar pior que os unitários
  - Mudanças externas afetam diretamente os testes
  - Tem custo elevado de manutenção
  - Precisam estar em menor quantidade



# Unidade II: Aspectos de Qualidade

# Referências

- <https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/november/async-programming-unit-testing-asynchronous-code>
- [https://medium.com/@kentbeck\\_7670/test-desiderata-94150638a4b3](https://medium.com/@kentbeck_7670/test-desiderata-94150638a4b3)

# Definições sobre Testes

O que é um Teste?

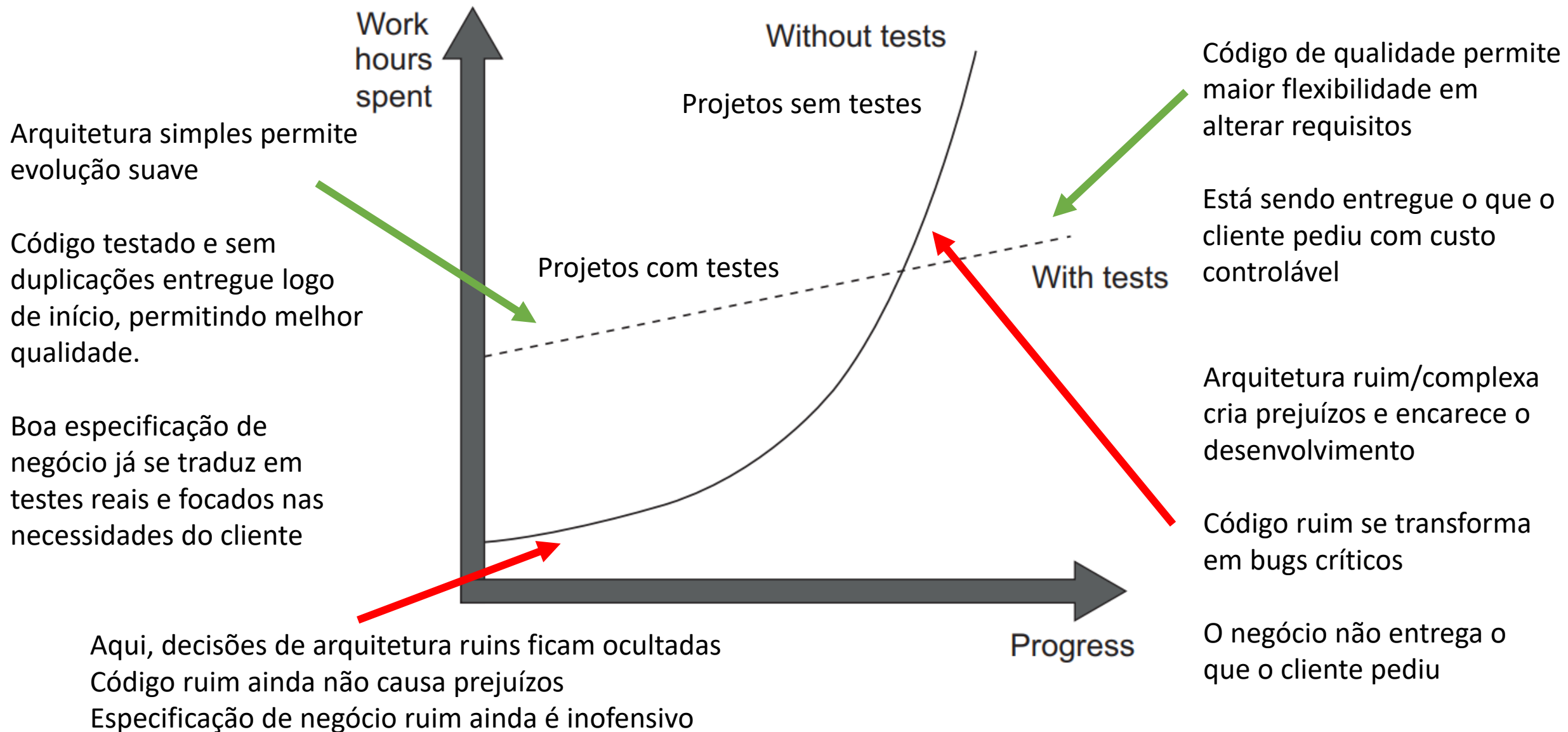
# A pergunta certa, na verdade, é o que um teste precisa alcançar?

Definição de teste de software é pouco clara. É comumente respondida como “uma maneira de avaliar a qualidade de uma aplicação e reduzir o risco de falha em operação”. Veremos o que pode definir melhor o conceito de teste de software.



“[...] Um teste habilita um crescimento sustentável de um projeto.”

(Khorikov, página 6)



“Testes ruins podem anular  
efeitos de testar. Logo, ter testes  
ruins é o mesmo que não testar.”

(Khorikov)

O que explica a redução da  
velocidade do desenvolvimento  
ao longo do tempo?

# Segunda Lei da Termodinâmica, a Entropia

O software sofre o mesmo fenômeno da física. Mais pessoas, mais requisitos, mais códigos vão aumentando o grau de desordem. O código começa a se deteriorar. Correção de bugs produzem mais bugs.

Testes ajudam a criar uma rede  
segura de alterações contra  
Regressões

O que é Regressão?

Quando uma feature para de funcionar depois de alguma alteração. É mesmo que bug.

Khorikov enfatiza que bug e regressão são sinônimos.



# Relação entre testes e retorno do esforço

- Hoje a principal discussão não é mais sobre fazer testes, mas sim sobre como encontrar o equilíbrio entre o esforço de testar e de implementar.
- A criação de teste é uma Ciência e saber se são ruins ou não precisa levar em consideração
  - Grau de experiência dos desenvolvedores
  - Consciência sobre Narrativa de Negócio com a prática do BDD e TDD
  - Cobertura de Código pode ser indicador de qualidade, mas com muitas ressalvas...

# Sobre Cobertura de Código

Conhecido como Code Coverage. A contagem de cobertura pode mudar ligeiramente dependendo dos critérios de contagem de linha.

```

@Test
void shouldStringBeLong() {
    var result : boolean = ProductionCode.isStringLong( input: "abc");
    assertEquals( expected: false, result);
}

```

```

4  @
5
6
7
8
9
public static boolean isStringLong(String input) {
    if (input.length() > 5)
        return true;

    return false;
}

```




Coverage: TestCode x			
Element ^	Class, %	Method, %	Line, %
org.example	100% (2/2)	100% (2/2)	66% (4/6)
ProductionCode	100% (1/1)	100% (1/1)	66% (2/3)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
ProductionCode()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
isStringLong(String)	<div><div></div></div>	75%	<div><div></div></div>	50%	1	2	1	3	0	1
Total	5 of 11	54%	1 of 2	50%	2	3	2	4	1	2

```
public static boolean isStringLong(String input)
{
    if (input.length() > 5)
        return true;
    return false;
}
```

} Branch não coberta pelo teste

} Branch coberta pelo teste (input.length() <= 5)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● <a href="#">ProductionCode()</a>		0%		n/a	1	1	1	1	1	1
● <a href="#">isStringLong(String)</a>		75%		50%	1	2	1	3	0	1
Total	5 of 11	54%	1 of 2	50%	2	3	2	4	1	2



JaCoCo considera 75%

Razão disso: Complexidade Ciclomática

[Doc - JaCoCo Coverage Counters](#)

Branches = Desvios de código produzidos por Condicionais (Ifs)

TOTAL DE BRANCHES COBERTAS / TOTAL DE BRANCHES

= 1 / 2 = 50%

Khorikov defende

TOTAL DE LINHAS COBERTAS / TOTAL DE LINHAS

4 / 5 = 80%

Obs.: Khorikov conta '{' e '}' como sendo linhas

Já que a branch “input.length() > 5” não estava sendo coberto, remover o if vai aumentar a cobertura...

```
public static boolean isStringLong(String input)
{
    return input.length() > 5;
}
```

### Cuidado!

- Para o IntelliJ e Khorikov, Cobertura vai ser 100%
- Para JaCoCo, cobertura continua intacta em 75%

Coverage: TestCode ×			
Element ▲			
org.example	100% (2/2)	100% (2/2)	100% (2/2)
ProductionCode	100% (1/1)	100% (1/1)	100% (1/1)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
ProductionCode()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
isStringLong(String)	<div><div></div></div>	75%	<div><div></div></div>	50%	1	2	0	1	0	1
Total	5 of 11	54%	1 of 2	50%	2	3	1	2	1	2

A quem dar ouvidos? IntelliJ ou  
JaCoCo?

JaCoCo usa uma contagem mais sofisticada baseada em Grafos



# JaCoCo pode nos ajudar

- Segundo Khorikov, é muito fácil manipular a cobertura de código
- Na contagem tradicional, apenas “cortando o if” já aumentou a cobertura de testes para 100%
- O JaCoCo é mais confiável devido a forma como conta. No entanto, no dia-a-dia é comum rejeitarmos um pouco a contagem dele devido a falta de praticidade ou percepção da presença do mecanismo.
- Dê uma chance ao JaCoCo de vez em quando.
- Se voltar aos testes e cobrir o caso onde string precisa ter comprimento maior do que 5, aí podemos dar robustez

```

public static boolean isStringLong(String input)
{
    return input.length() > 5;
}

```

@Test

```

void shouldStringBeLong() {

```

```

    var result : boolean = ProductionCode.isStringLong( input: "abc");
    assertEquals( expected: false, result);

```

```

    result = ProductionCode.isStringLong( input: "abcdef");
    assertEquals( expected: true, result);




```

```

}

```

## ProductionCode

Element	Missed Instructions	Cov.	Missed Branches	Cov.	N
● <a href="#">ProductionCode()</a>		0%		n/a	
● <a href="#">isStringLong(String)</a>		100%		100%	
Total	3 of 11	72%	0 of 2	100%	






# Não seja guiado pela Cobertura mas sim pelo Comportamento!

Um erro comum, de acordo com Khorikov, é seguir o resultado da cobertura para codificar os testes. Devemos seguir o comportamento desejado, para isso é necessário domínio da prática de TDD e BDD.

# Código que engana a cobertura (JaCoCo não consegue nos ajudar neste caso)

**Assertões é que efetivamente testam!  
O resultado SEMPRE precisa ser verificado**

```
@Test
void shouldStringBeLong() {
    var result1 : boolean = ProductionCode.isStringLong( input: "abc");
    var result2 : boolean = ProductionCode.isStringLong( input: "abcdef");
}
```

Element	Missed Instructions	Cov.	I
● <a href="#">ProductionCode()</a>		0%	
● <a href="#">setWasLastStringLong(boolean)</a>		0%	
● <a href="#">isWasLastStringLong()</a>		0%	
● <a href="#">isStringLong(String)</a>		100%	
Total	8 of 20	60%	(

# Código que engana a cobertura

(Nenhum verificador de cobertura vai ajudar)

```
var result :int = ProductionCode.parseToInt( input: "5");  
  
assertEquals( expected: 5, result);
```

```
public static int parseToInt(String input) {  
    return Integer.parseInt(input);  
}
```



parseToInt não está cobrindo

- Strings muito grandes
- Valores nulos
- Caracteres não-numéricos

Integer.parseInt pertence a uma biblioteca externa ao projeto. Não podemos garantir todos os casos que podem ocorrer.

Logo, a cobertura de código neste caso não serve de muita coisa.

JaCoCo diz que o método é coberto em 100%, mas sequer ele sabe que Strings muito grandes, valores não-numéricos e etc podem originar comportamentos inesperados.

Element	Missed Instructions	Cov.	
● <a href="#">ProductionCode()</a>		0%	
● <a href="#">parseToInt(String)</a>		100%	
Total	3 of 6	50%	(

# Código de Produção vs. Código de Teste

- Quanto mais código de teste, melhor? Não.
- Código é um passivo e não ativo. Quer dizer, não gera valor *per si*.
- Quanto mais código, maior a superfície para cobrir e mais bugs pode produzir. Quanto menos código, melhor.
- Testes são códigos, ou seja, são passivos também. Podem também estar infectados por bugs
  - Isso mostra a importância do Code Review nos próprios códigos de teste.
  - Chegar num projeto “olhando” para código de produção não tem valor. Código é passivo. Deve-se verificar integridade. Os testes estão realmente entregando valor ao negócio? Isso precisa ser analisado.
  - Testes nos ajudam a verificar a corretude da aplicação.

Agora voltemos ao assunto  
principal

# Implicações das definições

End-to-end



# Testes End-to-End (E2E) fazem parte dos Testes de Integração

- Testes E2E também verificam dependências out-of-process
- A diferença entre E2E para integração é que E2E normalmente inclui muito mais de dependências out-of-process
- Esclarecendo melhor
  - Teste de Integração: funciona com uma ou duas dependências out-of-process
  - Teste E2E: consiste exclusivamente de dependências out-of-process
- Origem do nome E2E: testes baseados no ponto de vista de um usuário, incluindo todas as aplicações externas que o sistema integra
- Sinônimos: UI Tests (User Interface), GUI Tests (Graphical User Interface) e Functional Tests

# Mapa Geral

Vamos ver como podemos tomar decisão para categorizar testes unitários, de integração e E2E

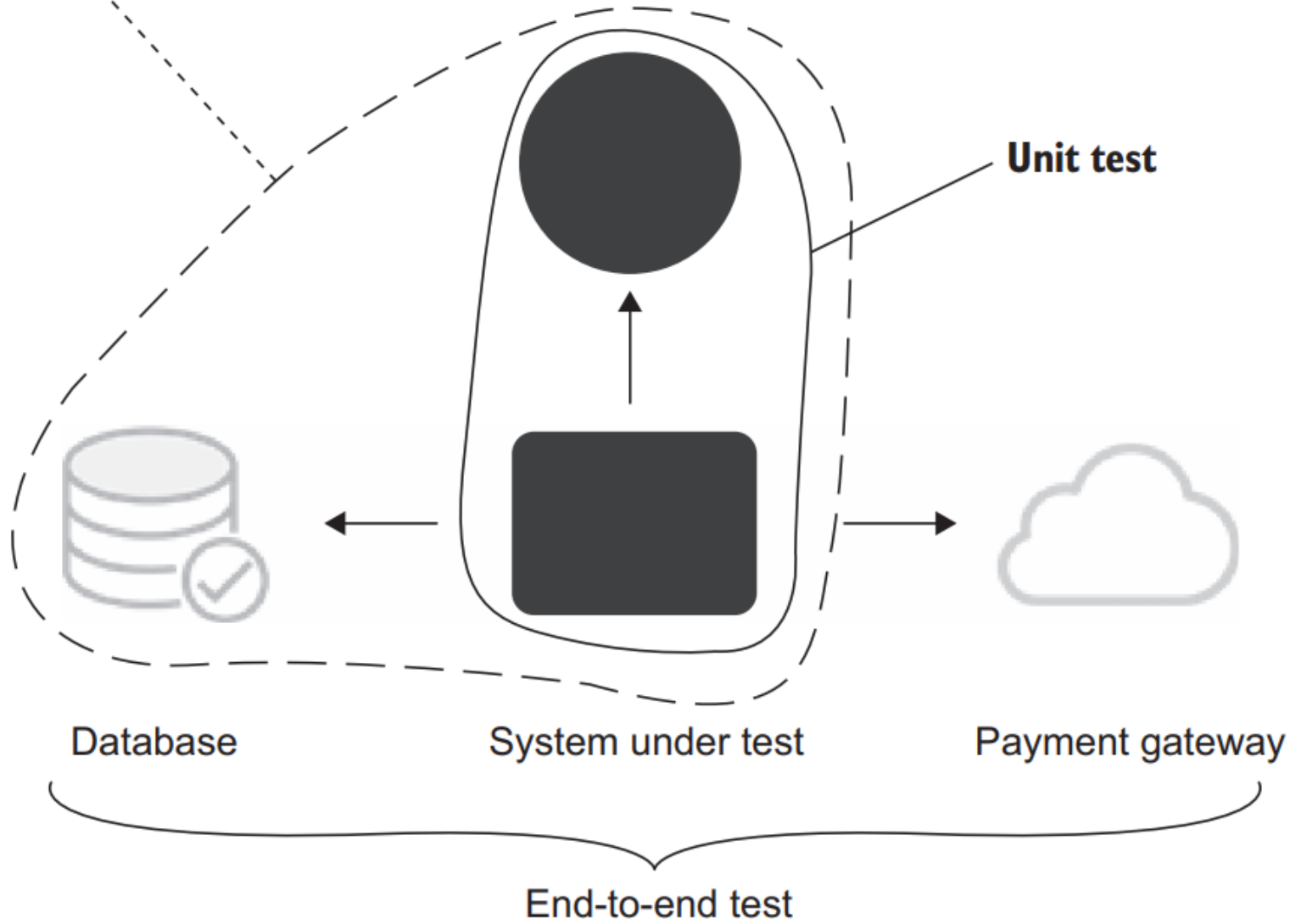
# Suponha uma Aplicação

- Possui suas classes que definem o comportamento de negócio
- Possui 3 dependências externas
  - Um banco de dados
  - Um sistema de arquivos
  - Um gateway de pagamento
- Por que?
  - Banco de dados e sistema de arquivos está sob total controle mas são externos ao processo de teste
  - Gateway de pagamento pode estar externo à organização, portanto fora do nosso controle.
  - Nos testes...
    - Unitários e Integração: mockamos. Na Integração “sanduíche” precisa ser mais independente caso o gateway seja um código de outra empresa ou equipe onde temos pouco controle para influenciar no seu design.
    - E2E: usamos o serviço real. Em geral, gateways de pagamento fornecem APIs de Sandbox.

**Integration test**

Another class

**Unit test**



Database

System under test

Payment gateway

End-to-end test

# Leituras Recomendadas por Khorikov

- Test-Driven Development: By Example. Kent Beck segue nesta obra a Escola Clássica.
- Growing Object-Oriented Software, Guided by Tests. Steve Freeman e Nat Pryce seguem a Escola de Londres.
- Principles, Practices, Patterns by Steven van Deursen e Mark Seemann. Explicam de forma detalhada a Injeção de Dependência.

Clássic vs. Londres: afeta a prática  
do TDD

<pagina 36>

# TDD e XP

Falar de Unitário ou Integração tem origem aqui...



Primeiro, o que devem ser  
Testes?

# Segundo Kent Beck (Test Desiderata)

- “Testes devem ser acoplados ao comportamento do código e desacoplados da estrutura do código. [...]”
- Kent Beck aponta que é muito comum códigos de testes repetindo o que está no código da implementação. Ou seja, os códigos de testes como “confirmadores” da verdade da implementação. Isto não é TDD.
- Beck relata as propriedades desejáveis que todo código de teste pode ter. Não é necessário ter todas as propriedades. Caso não queira uma determinada propriedade, é necessário substituí-la por outra de valor superior.

# Propriedades de um Teste

# Isolamento

- Testes devem retornar os mesmos resultados independente da ordem que são executados.
- O que podemos evitar na prática, com JUnit, por exemplo?

```
@Test
@Order(1)
void firstTest() {
    output.append("a");
}

@Test
@Order(2)
void secondTest() {
    output.append("b");
}

@Test
@Order(3)
void thirdTest() {
    output.append("c");
}
```

# Composable

- Podemos executar 1, 10, 100, 1 milhão de vezes... Temos que obter sempre os mesmos resultados.
- O que podemos fazer? Em relação aos Testes Unitários:
  - De acordo com Stephen Cleary (2014), devemos usar Métodos Síncronos em vez de Assíncronos. Dar await exige estabelecer sincronização, o que pode ocorrer alteração de comportamento da aplicação. É mais difícil estabelecer previsibilidade.

# Rápido

- Testes precisam executar rapidamente.
- O que podemos fazer?
  - Evitar o uso de tecnologias que agregam muita carga de inicialização. Exemplos:
    - @SpringBootTest : recomendado que se evite em teste unitário devido à quantidade de dependências envolvidas. São diversos beans injetados. Avalie a possibilidade de deixar para utilizar em testes integrados.
    - Localstack: operações com Docker envolvem uso excessivo de I/O e, executados repetidamente, penaliza em demasia os testes de um modo geral. Avalie possibilidade de não utilizá-lo ou, no máximo, deixar somente para testes integrados.
    - Evite ler arquivos ou base de dados baseado em disco. Opte por modelos baseados em RAM, como o SQLite ou criação de FileHandlers abertos somente em memória.
    - Nunca consulte serviços externos, como APIs. Geram I/O excessivo. É algo mais reservado para User Contract Tests e E2E, porque são executados em ocasiões mais especiais.

# Inspirador

- Testes precisam transmitir confiança.
- Como promover?
  - Code Review periódico pode nos ajudar a garantir que os testes reflitam a necessidade de negócio da aplicação.
  - O comportamento esperado é algo para ser avaliado por humanos. Stakeholders precisam estar envolvidos diretamente.

# Writable

- Testes precisam ser baratos para escrever.
- O que podemos fazer?
  - Evitar uso de hardware específico. Caso necessite, precisa dar acessibilidade ampla, bem documentada e testável em poucos passos.
  - Uso de ambientes precificados por execução, como em Cloud. Caso execução pode custar muito caro, com potencial de consumir recursos sem muito controle. Caso precise, que seja somente em testes E2E e User Contract Tests.



# Legível

- Testes precisam ser compreensíveis pelo leitor, dando-lhe motivação para contribuir no teste.
- O que podemos fazer?
  - Observar o uso do Clean Code
  - Eliminar duplicações o mais cedo possível para evitar “clones” capazes de executar atividades estranhas ou mal compreendidas ao negócio.
    - Exemplo: Cliente de Acesso a uma base de dados implementado de maneira diferente em locais muito próximos no código.

# Comportamental

- Testes precisam ser sensíveis a mudanças no comportamento do código de teste. Se o comportamento mudar, o resultado do teste deve mudar.
- O que podemos fazer?
  - Promover o BDD, procurando-se o comportamento de negócio a ser alcançado e o código que satisfaz. O risco de code smell é menor nesta prática.

# Insensível à estrutura

- Os testes não devem alterar seus resultados se a estrutura do código for alterada.
- O que quer dizer?
  - Refatorações, como melhoria da organização lógica e organização em pastas não devem alterar o resultado dos testes.

# Automatizado

- Os testes devem ser executados sem intervenção humana.
- O que podemos fazer?
  - Existe algum ponto manual? Já trate de escrever um script. Precisa ser acionável rapidamente sem login em algum sistema nem preenchimento de formulário. Nada.

# Específico

- Se um teste falhar, a causa da falha deve ser óbvia.
- Então?
  - Não insira dependências entre códigos de testes.
  - Observe a clareza no comportamento de negócio a ser testado.
  - Compreensão de negócio precisa ser mais clara possível, extraída diretamente do código, sem leituras de manuais ou documentos obscuros.

# Determinístico

- Se nada mudar, o resultado do teste não deve mudar.
- O que fazer?
  - Evitar ao máximo processamento assíncrono.
  - Evite de gerar valores aleatórios como entrada de dados.

# Preditivo

- Se todos os testes forem aprovados, o código de teste deve ser adequado para produção.
- O que podemos fazer?
  - Code Review
  - Escrita de boas histórias e cenários de negócio.