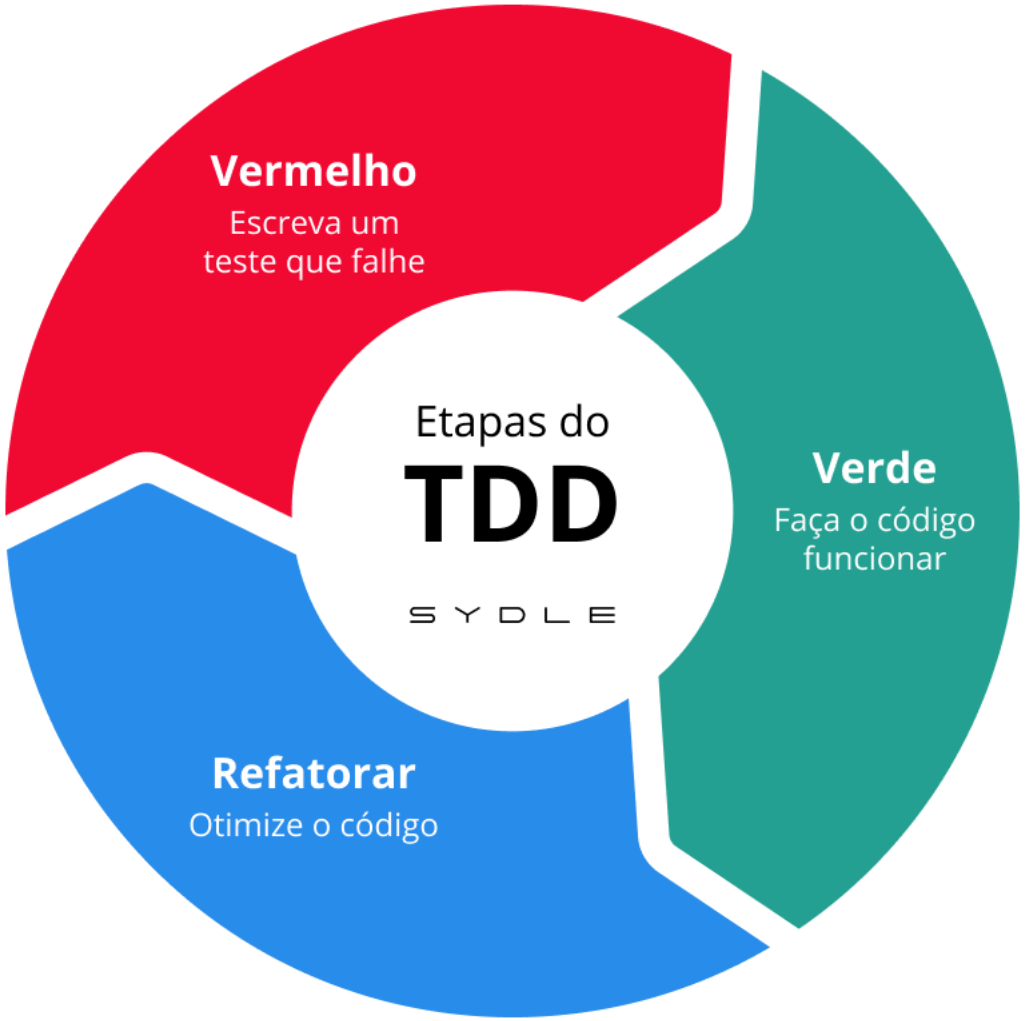
TÓPICO 10 - TESTES UNITÁRIOS

Clean Code - Professor Ramon Venson - SATC 2025

Testes Unitários

Testes unitários são testes que verificam o comportamento de uma única unidade de código.

São testes aplicados especificamente a funções ou métodos para validar seu funcionamento.



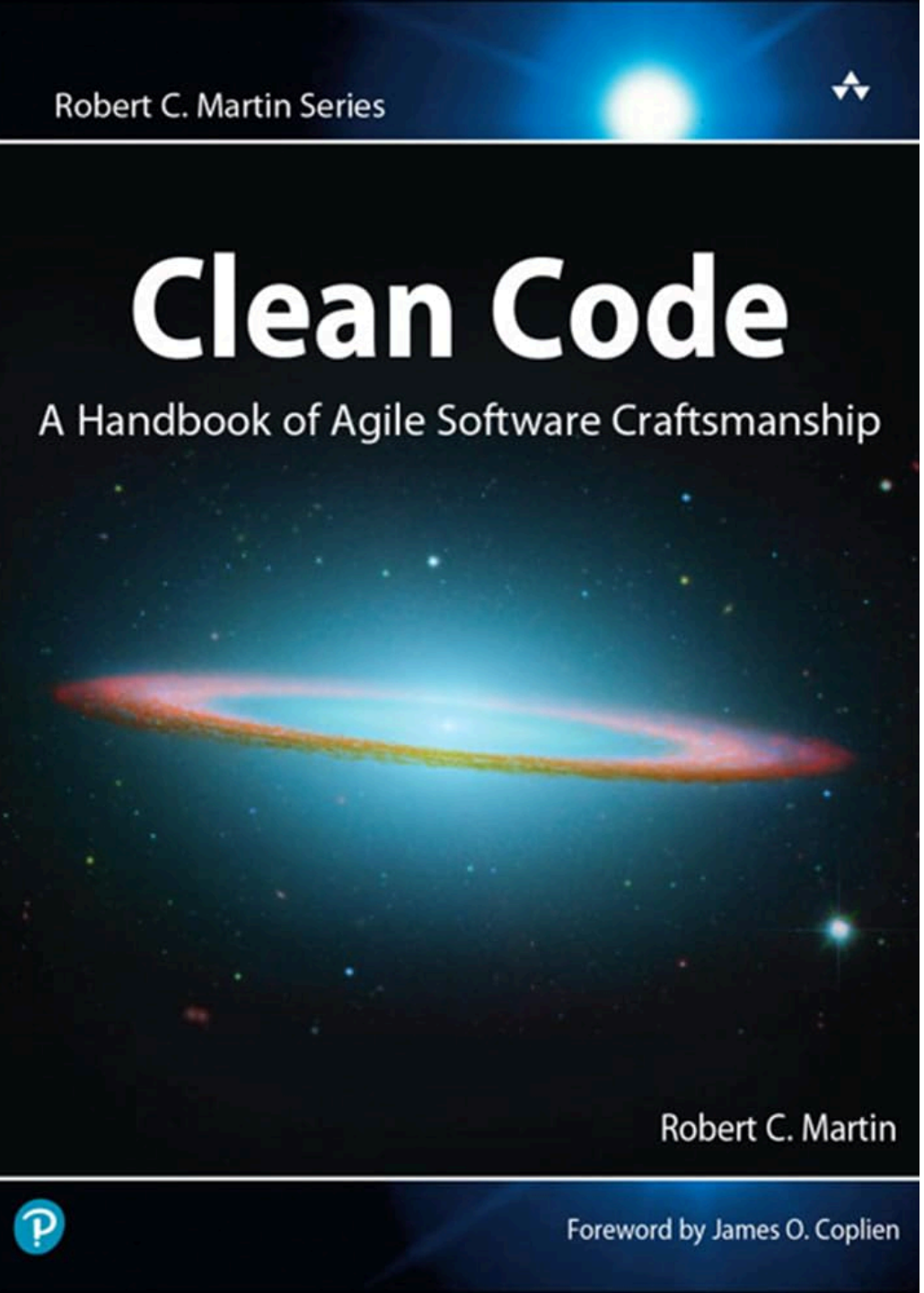
TDD

Test-Driven Development (TDD) é uma prática de desenvolvimento de software que consiste em escrever testes antes de escrever o código de produção.

As leis do TDD

As três leis do TDD são:

- Você não pode escrever qualquer código de produção sem escrever um teste de falha primeiro;
- Você não pode escrever mais de um teste de unidade do que o necessário para falhar;
- Não se deve escrever mais código do que o necessário para passar no teste.



Dicas do Livro

- Como manter os testes limpos
- Os testes habilitam as mudanças
- Testes Limpos
- Linguagem de testes específica ao domínio
- Um padrão duplo
- Uma confirmação por teste
- Um único conceito por teste
- FIRST

Como manter os testes limpos

Os testes precisam seguir o mesmo padrão de qualidade do código de produção.

Testes feitos à "Go Horse" são piores do que nenhum teste.

Uma breve história

Imagine que uma equipe de desenvolvimento está trabalhando em um projeto de um novo software.

O projeto está crescendo e a equipe está trabalhando em várias funcionalidades diferentes.

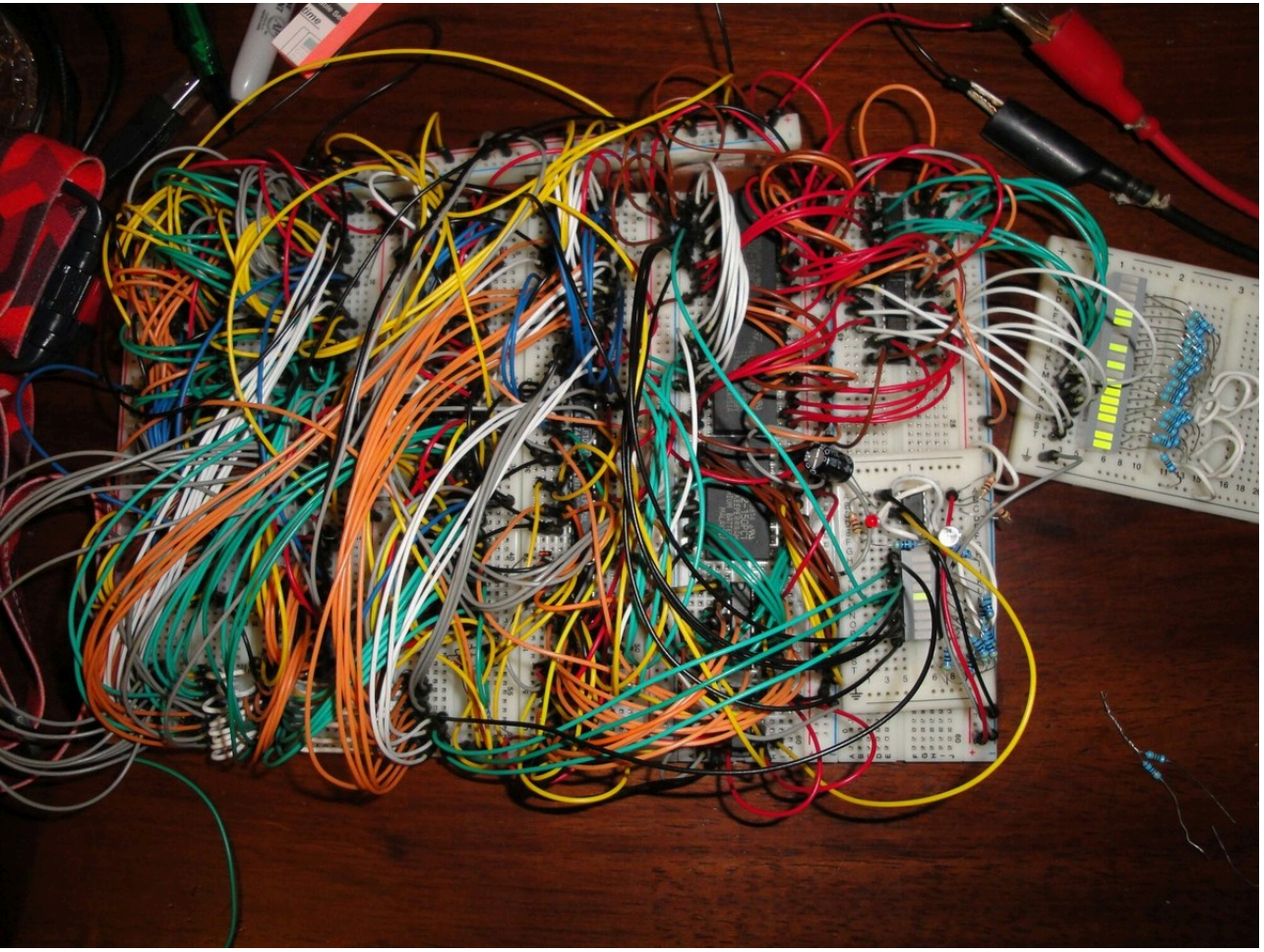
Os problemas com código imprevisível crescem dia após dia e a equipe toma a decisão:

> "Precisamos de uma base de testes unitários".

Para garantir a eficiência nas entregas, a equipe decidiu que os testes não deviam ser preservados segundo os mesmos padrões de qualidade do código de produção.

> "Nós implantamos pra produção, então testamos"

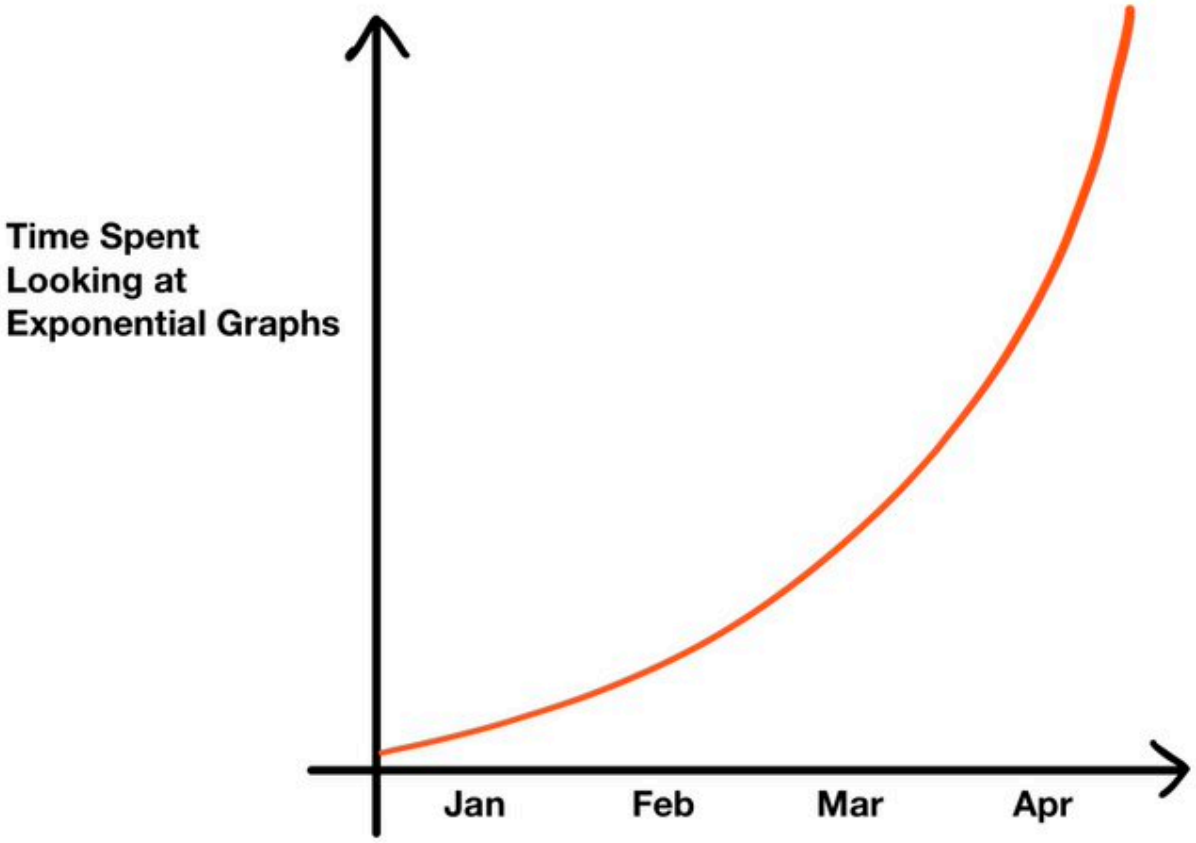
No início, os testes eram simples e fáceis de rodar.



Mas a base de testes unitários rapidamente se tornou uma *macarronada*, espremendo novos testes dentro de um código já difícil de manter.

Os testes antigos começam a falhar.

Rapidamente os testes começam a ser vistos como um problema em constante crescimento.



O custo da manutenção dos testes faz com que a equipe aumente as estimativas para finalização de novas features.

> Tempo gasto olhando para gráficos exponenciais

Por fim, a equipe é forçada a descartar toda a coleção de testes

Sem uma coleção de testes, a equipe não consegue garantir a qualidade do código.

Os problemas com código imprevisível crescem dia após dia...

Dicas gerais

- Os testes devem ser alterados com a mesma frequência que o código de produção.
- Os desenvolvedores são os responsáveis por escrever testes e manter os testes limpos.
- Assim como o código de produção, os testes não devem adicionar uma complexidade não-linear ao código.

Os testes habilitam as mudanças

Testes são responsáveis por manter a flexibilidade do código.

Sem testes, você ficará relutante em fazer mudanças no código e vai correr o risco de quebrá-lo quando fizer.

> E se eu dissesse que você pode ter velocidade e segurança?

Testes Limpos

O que torna um teste limpo:

- Legibilidade
- Legibilidade
- Legibilidade

O que torna um código legível:

- Clareza
- Simplicidade
- Consistência

O que os códigos a seguir possuem em comum?

Código 01

```
public void testGetPageHierarchyAsXml() throws Exception {
   makePages("PageOne", "PageOne.ChildOne", "PageTwo");
   submitRequest("root", "type:pages");
   assertResponseIsXML();
   assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
   );
}
```

Código 02

```
public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");
    addLinkTo(page, "PageTwo", "PageTwo");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    assertResponseDoesNotContain("PageTwo");
}
```

Código 03

```
public void testGetDataAsXml() throws Exception {
    magePageWithContent("Test Page One", "test page");
    submitRequest("TestPageOne", "type:data");
    assertResponseIsXML();
    assertResponseContains("test page");
}
```

Linguagem de testes específica ao domínio

Os exemplos anteriores utilizam um conjunto de funções e utilitários que foram construídos especialmente para os testes.

Essas funções representam uma API construída especialmente para ajudar a escrever e entender os testes unitários.

Um padrão duplo

De fato, testes e código de produção possuem padrões distintos.

Testes unitários rodam em ambiente de testes, não de produção, logo possuem diferentes requisitos e dependências.

Uma única confirmação por teste

Um teste deve fazer apenas uma única confirmação.

É uma regra bastante limitante, mas garante que testes falhem rápido se forem bem estruturados.

Exemplo:

```
public void testGetPageHierarchyAsXml() throws Exception {
   makePages("PageOne", "PageOne.ChildOne", "PageTwo");
   submitRequest("root", "type:pages");
   assertResponseIsXML();
   assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
   );
}
```

O que acontece se o teste falhar? Será preciso analisar os detalhes para descobrir o que deu errado.

Ao invés, use:

```
public void testPageHierarchyAsXml() throws Exception {
   makePages("PageOne", "PageOne.ChildOne", "PageTwo");
   submitRequest("root", "type:pages");
   assertResponseIsXML();
}

public void testPageHierarchyHasRightTags() throws Exception {
   makePages("PageOne", "PageOne.ChildOne", "PageTwo");
   submitRequest("root", "type:pages");
   assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
   );
}
```

Cada teste será responsável por uma única falha, facilitando a localização do problema.

Um único conceito por teste

Caso a regra de "Uma única confirmação" seja limitante demais, uma opção é dividir o teste em conceitos.

Um conceito pode ser, por exemplo, testar se todo o conteúdo da página está correta.

F.I.R.S.T

Testes limpos seguem as regras do acrônimo: `F.I.R.S.T`

Rápido

Fast: Os testes devem ser rápidos de forma que rodá-los não seja um problema.

Independente

Independecy: Os testes não devem depender uns dos outros. Todo teste deve poder ser rodado independentemente.

Reprodutível

Repeatable: Os testes devem ser repetíveis e sob quaisquer circunstâncias resultarem no mesmo resultado.

Autovalidável

Self-validating: Os testes devem ser auto-validáveis, respondendo apenas `true` ou `false`.

Oportuno

Timely: Os testes devem ser escritos imediatamente antes do código de produção para o qual será aplicado.