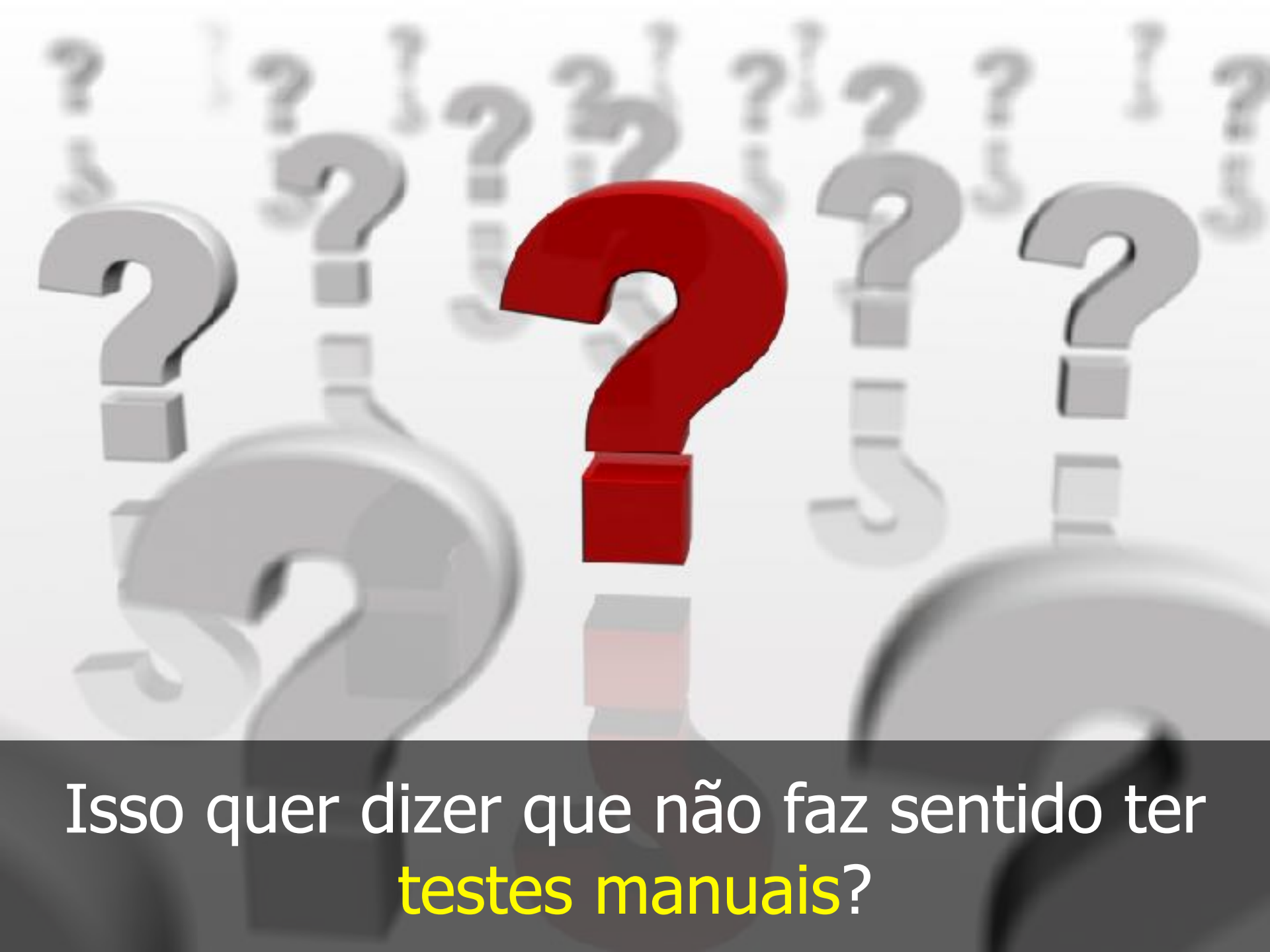
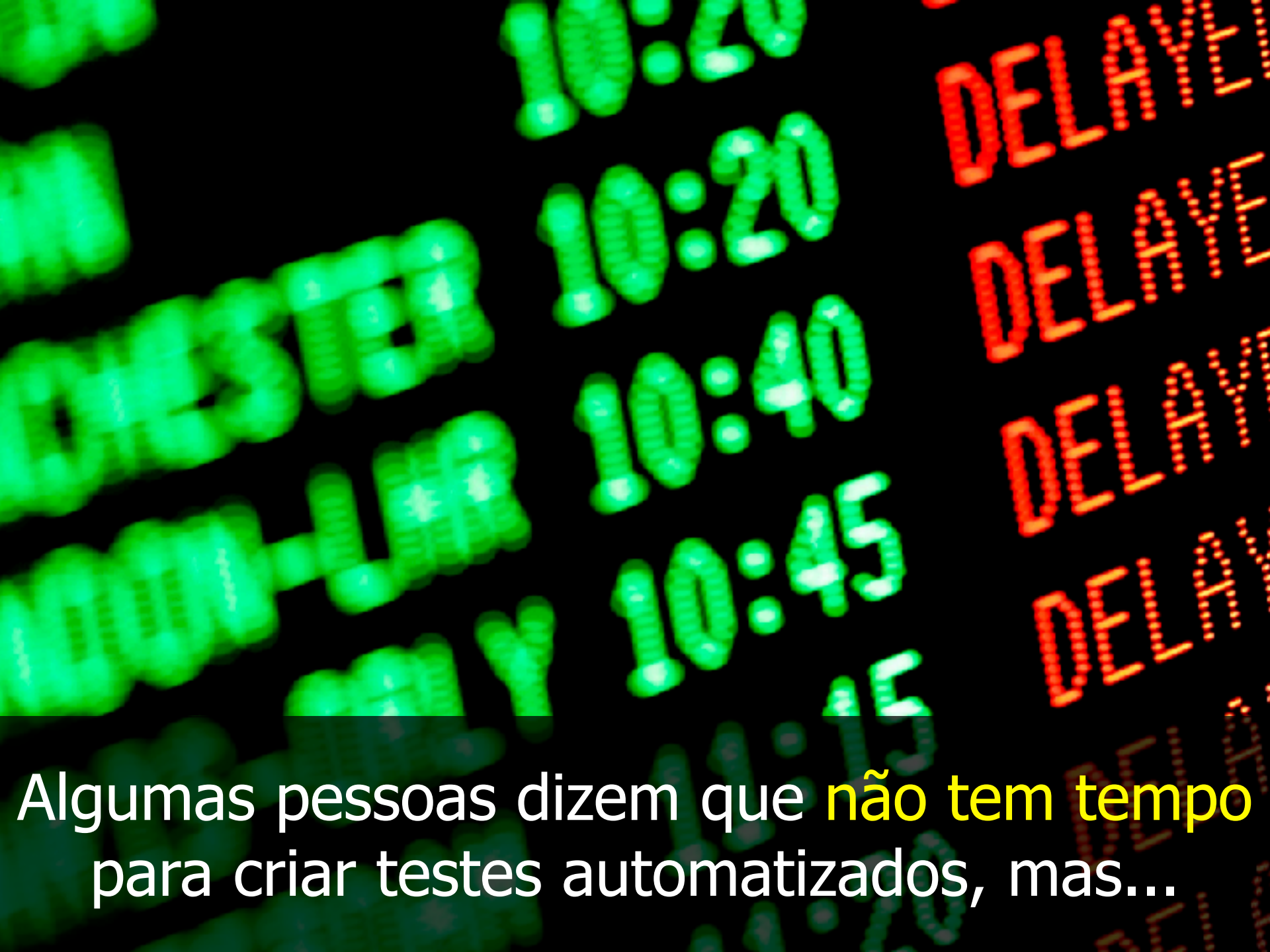
# Test-Driven Development


Os testes automatizados são a única forma que temos para garantir que o código funciona



Isso quer dizer que não faz sentido ter testes manuais?

Os testes manuais são importantes, principalmente para a aceitação por parte de usuários-chave, mas devem ser complementares e não garantem que não existirá a regressão ao longo do tempo




Algumas pessoas dizem que não tem tempo para criar testes automatizados, mas...

Tem tempo para ouvir reclamação dos clientes que descobrem erros em produção

Tem tempo para corrigir bugs

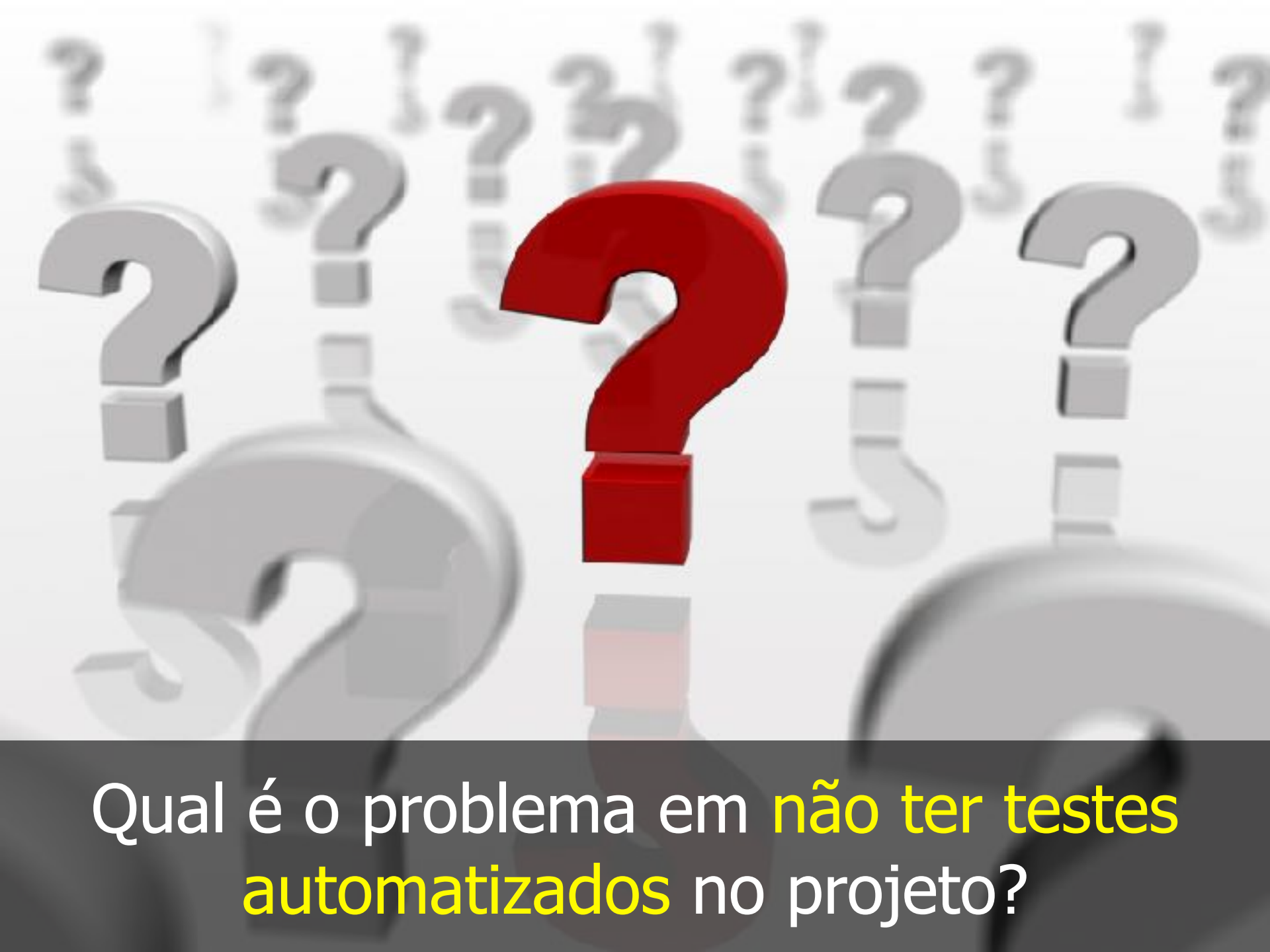
Tem tempo para lidar com código de baixa qualidade que não foi refatorado

Tem tempo para testar manualmente

Tem tempo para treinar pessoas novas que entram no time toda hora

Tem tempo para ficar reclamando do projeto


Ter testes automatizados não é uma garantia de que não vão existir defeitos


Qual é o problema em não ter testes automatizados no projeto?

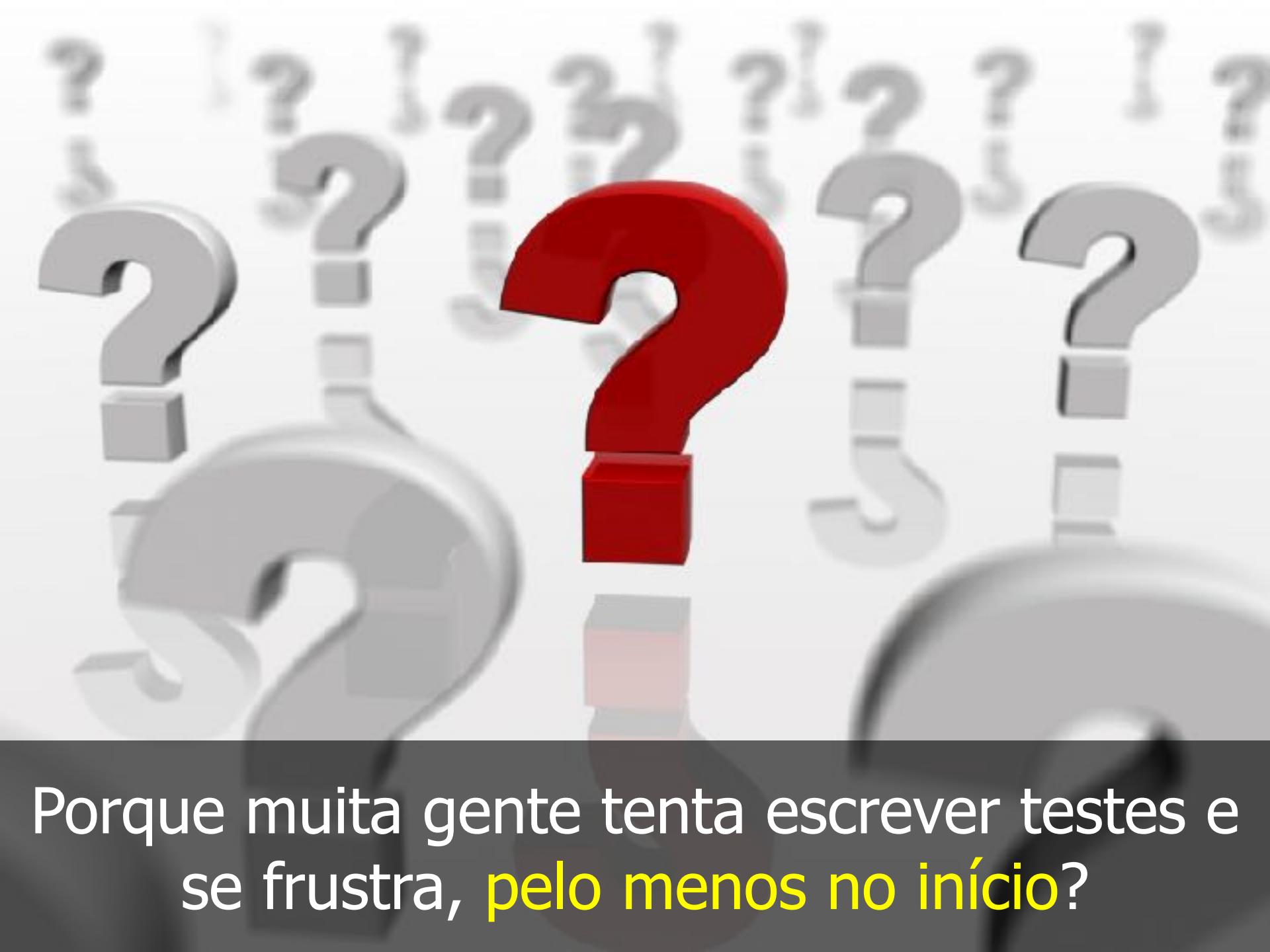
Você fica torcendo para que o código-fonte funcione durante o desenvolvimento ou até mesmo em produção

Já decorou os dados que precisam ser digitados na tela para testar uma determinada funcionalidade

É bem menos produtivo do que poderia ser

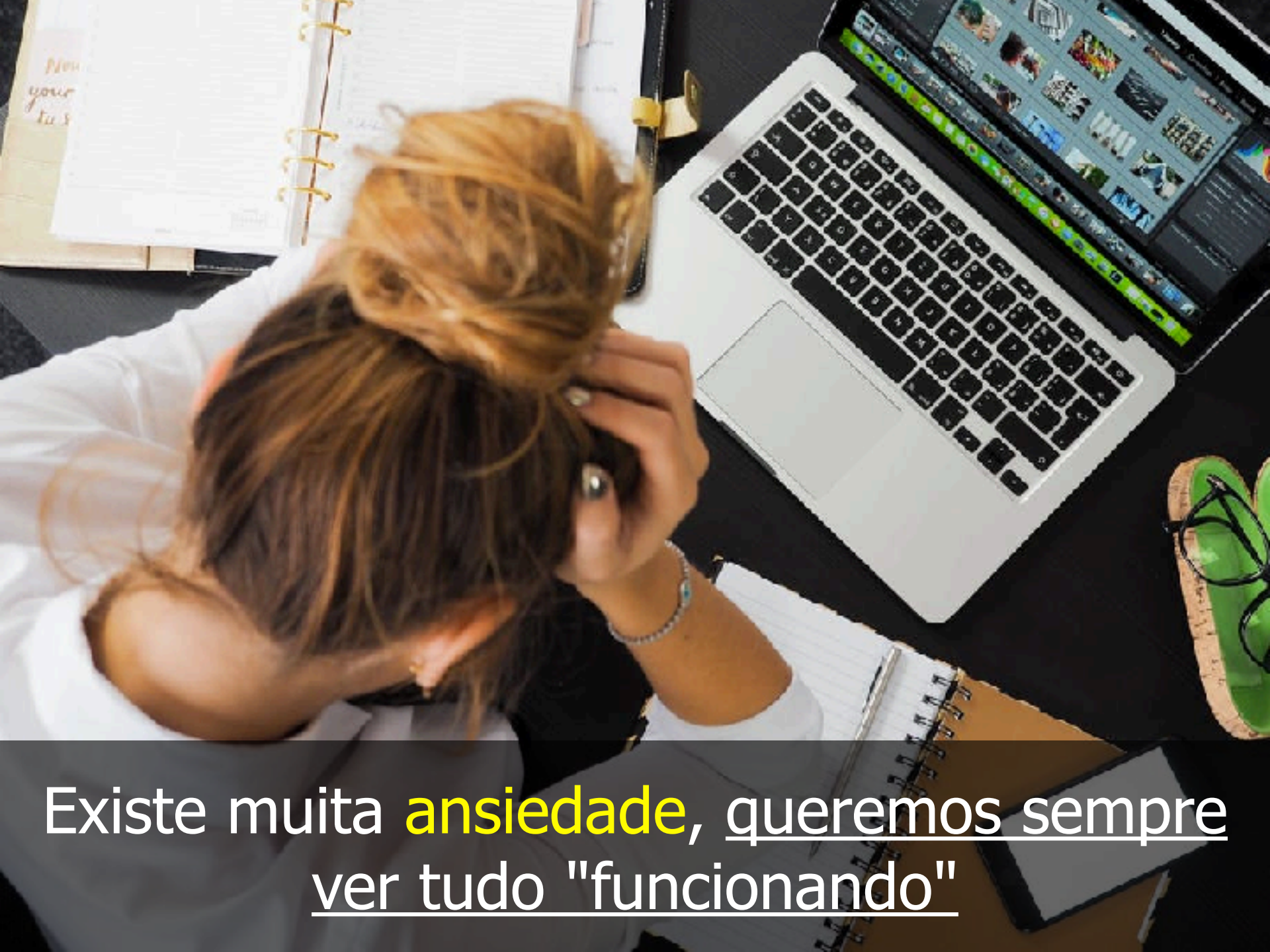
Eventualmente mexe em uma coisa e estragou a outra

Fica com medo de mexer em alguma funcionalidade

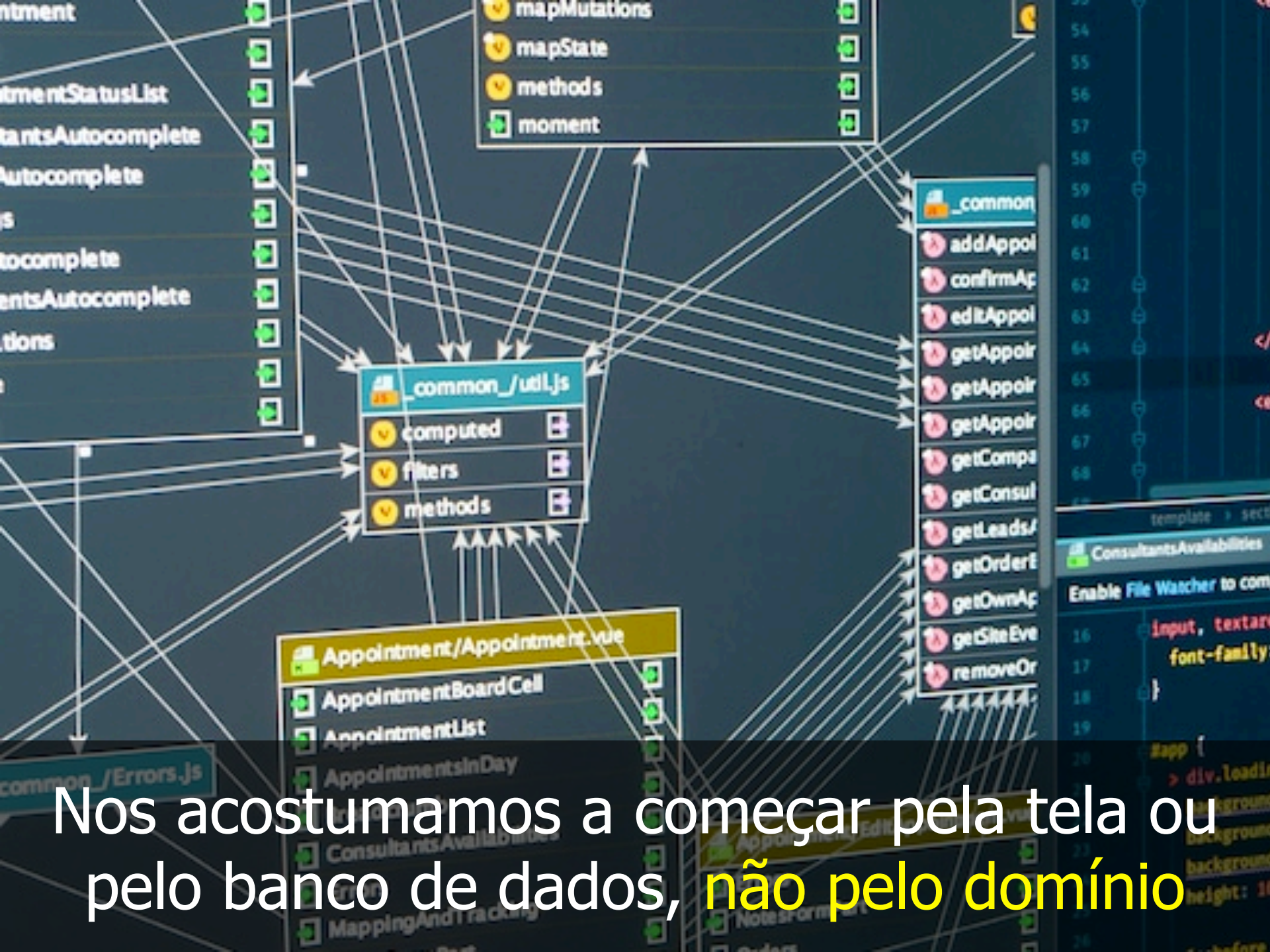


Porque muita gente tenta escrever testes e se frustra, pelo menos no início?

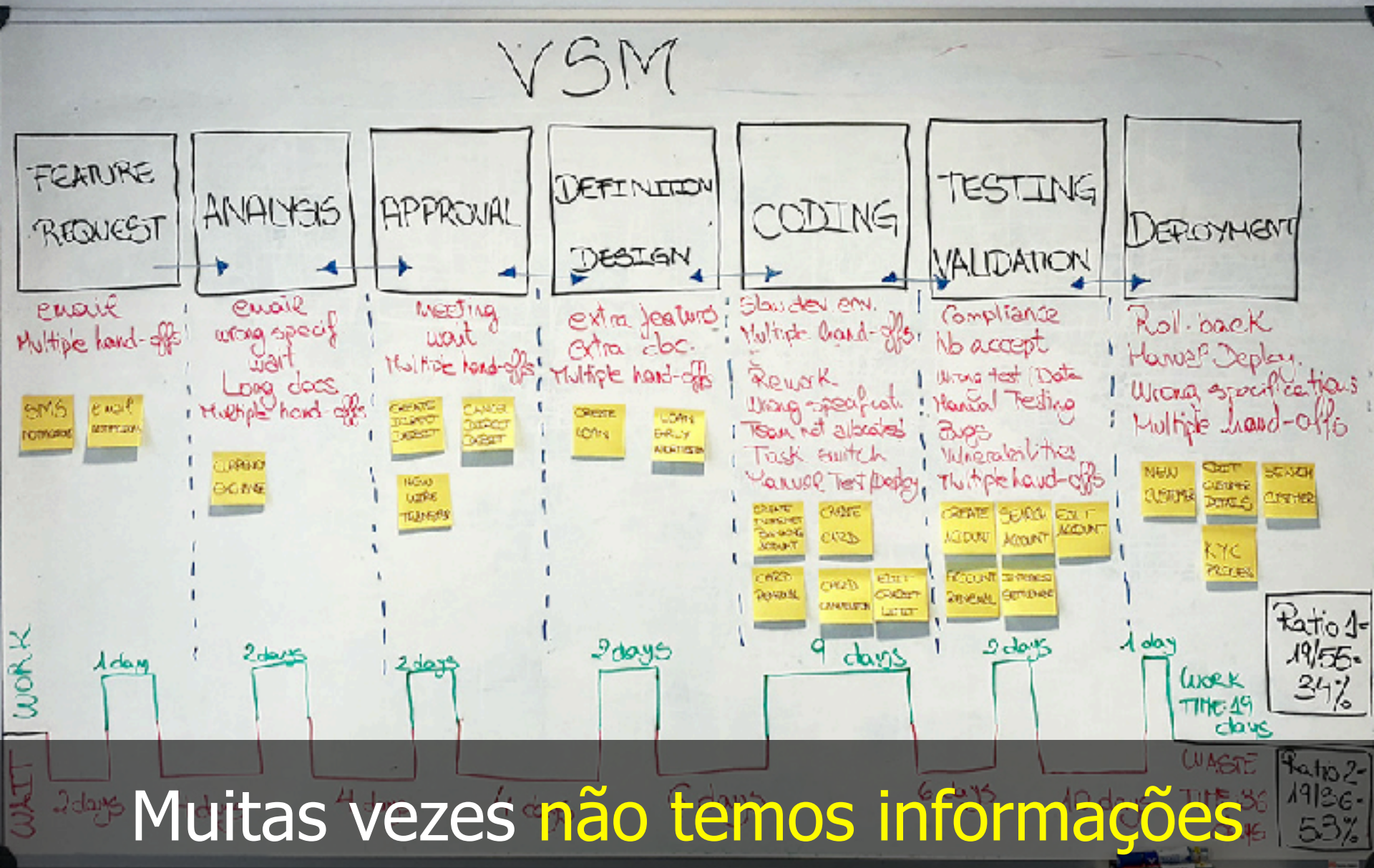

Escrever os testes requer muita disciplina



Existe muita ansiedade, queremos sempre ver tudo "funcionando"

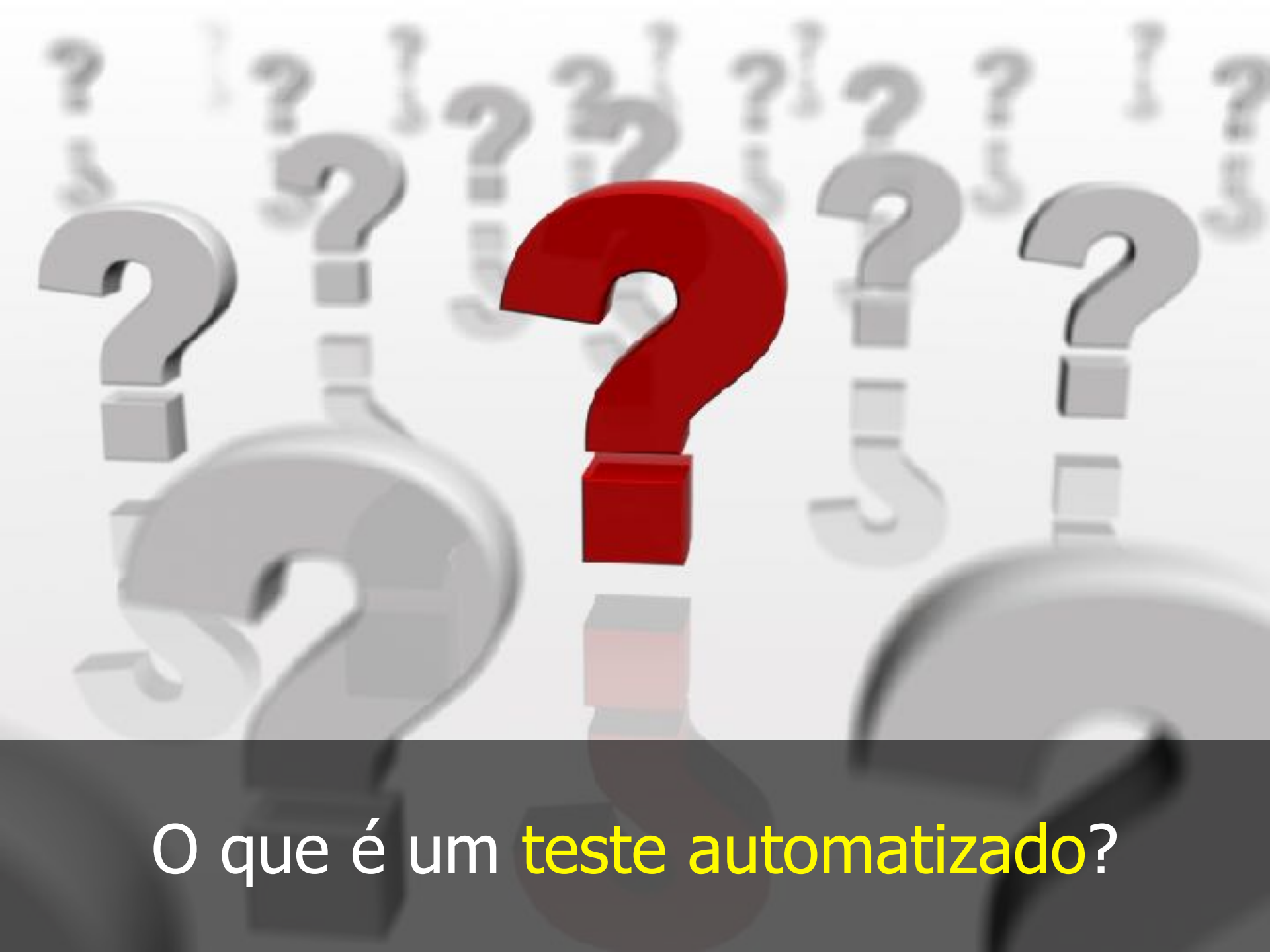



Nos acostumamos a começar pela tela ou pelo banco de dados, não pelo domínio




Muitas vezes não temos informações suficientes para começar a desenvolver

Muitas vezes o design e a arquitetura não favorecem a automação dos testes



# O que é um teste automatizado?

# Dado um conjunto de entradas, quando algo acontecer a saída deve suprir as expectativas

Given/Arrange: Definição de todas as informações necessárias para executar o comportamento que será testado

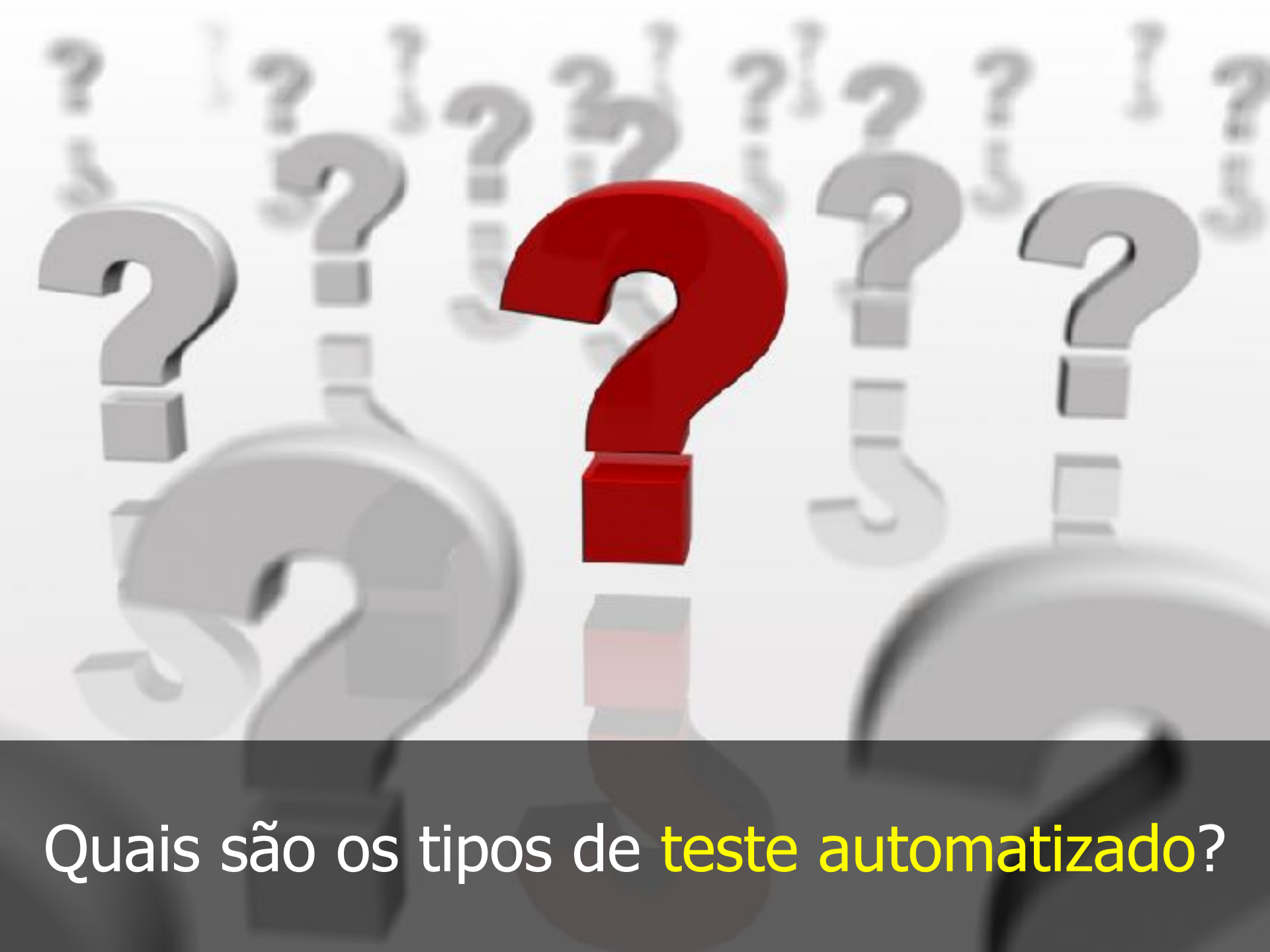
When/Act: Executar o comportamento

Then/Assert: Verificar o que aconteceu após a execução, comparando as informações retornadas com a expectativa que foi criada

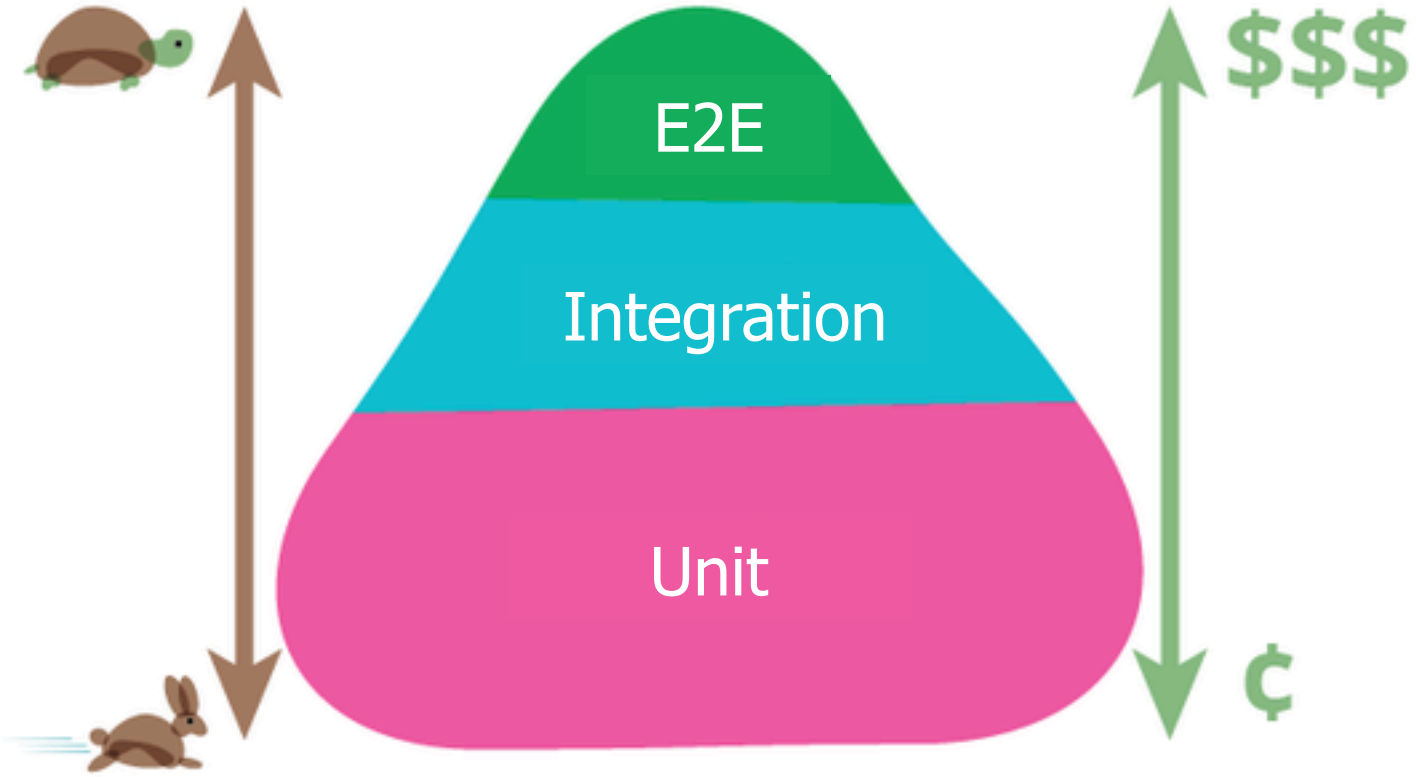
# Exemplo

Deve fazer um pedido com 3 itens

Dado um novo pedido com 3 itens associados, um Livro de R$50,00, um CD de R$20,00 e um DVD de R$30,00
Quando o pedido for realizado
Então deve ser retornado uma confirmação do pedido contendo o código, juntamente com o total do pedido de R$100,00 e o status aguardando pagamento



# Quais são os tipos de teste automatizado?

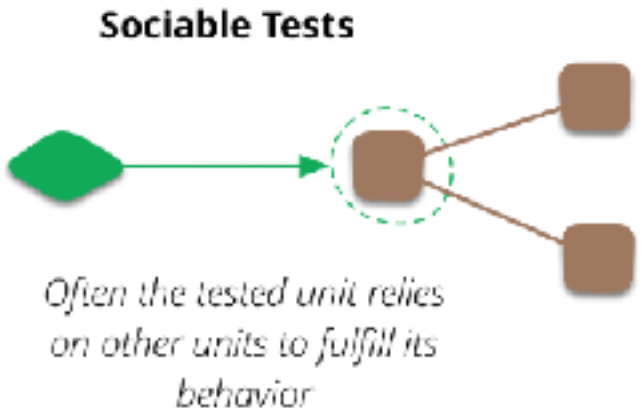

E2E
Integration
Unit
$$$
¢

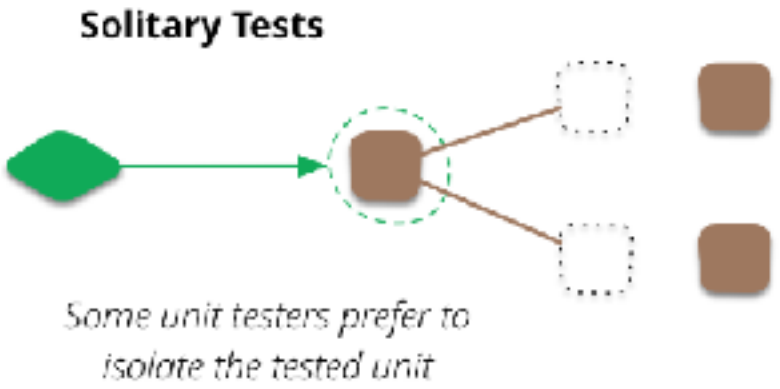

Qual é a diferença entre eles?

# Unit Tests

São testes de unidade, não necessariamente unitários, que podem ou não envolver vários componentes pertencentes à mesma camada e sem qualquer interação com recursos externos como um banco de dados, uma API ou o sistema de arquivos

Por não consumirem recursos externos são muito rápidos, sendo executados em poucos milisegundos




But not all unit testers use solitary unit tests. Indeed when xunit testing began in the 90's we made no attempt to go solitary unless communicating with the collaborators was awkward (such as a remote credit card verification system). We didn't find it difficult to track down the actual fault, even if it caused neighboring tests to fail. So we felt allowing our tests to be sociable didn't lead to problems in practice.

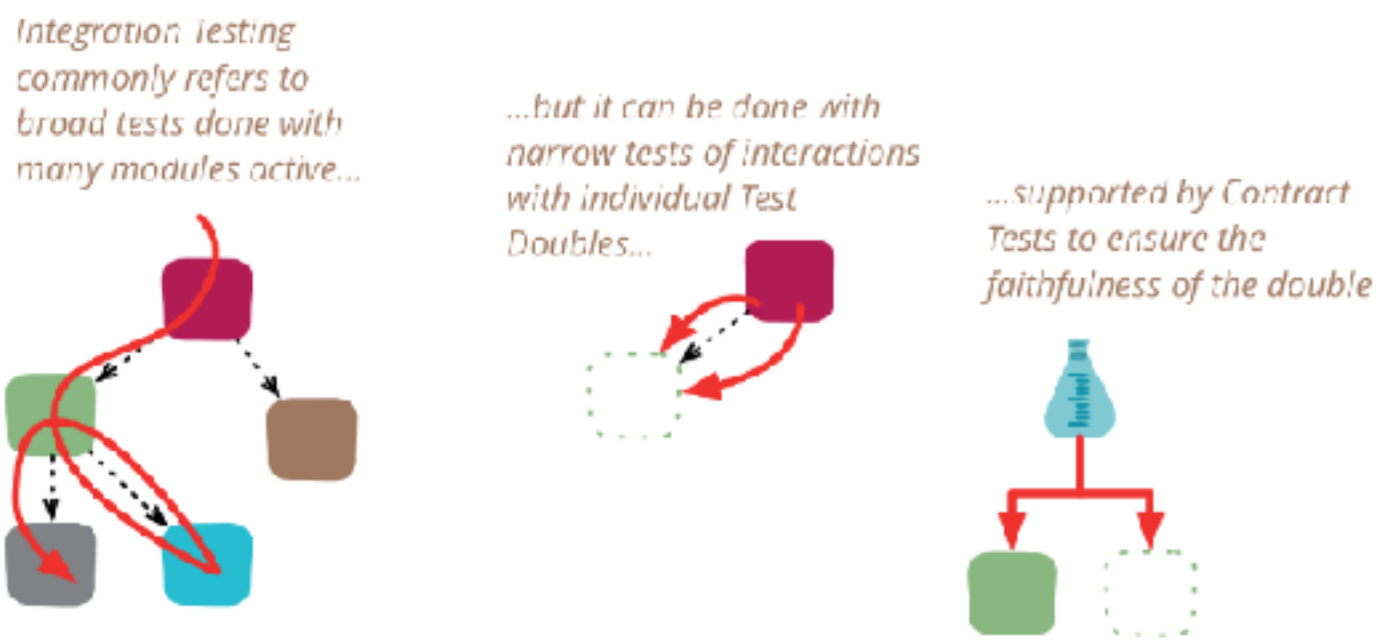
Indeed using sociable unit tests was one of the reasons we were criticized for our use of the term "unit testing". I think that the term "unit testing" is appropriate because these tests are tests of the behavior of a single unit. We write the tests assuming everything other than that unit is working correctly.

https://martinfowler.com/bliki/UnitTest.html

# Integration Tests

Testam componentes pertencentes à múltiplas camadas e normalmente envolvem recursos externos, sejam eles reais ou não, ou seja, o fato de utilizar um Test Pattern como um stub ou mock não torna o teste de unidade.

Em geral são mais lentos por fazerem I/O



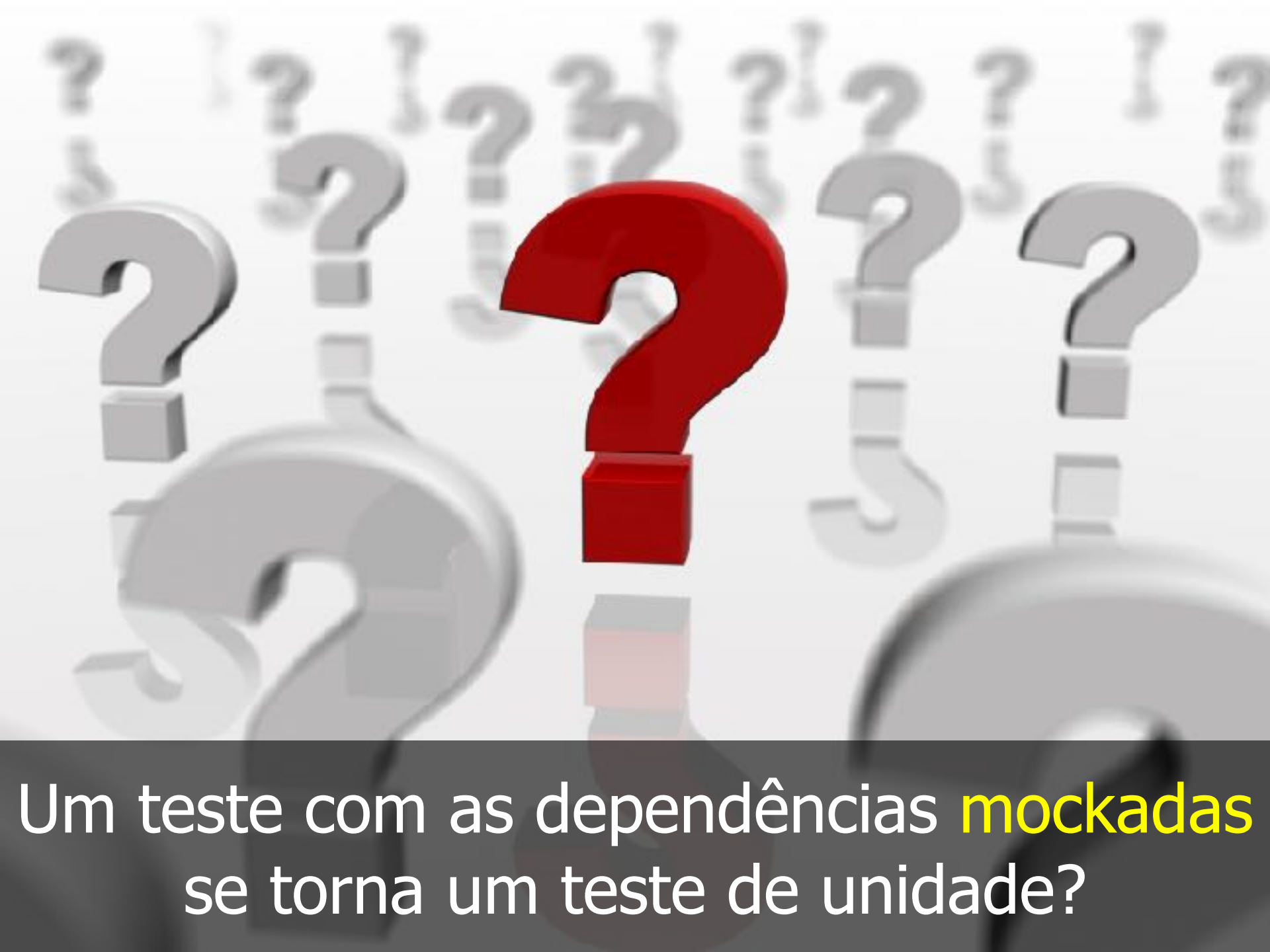

narrow integration tests

- exercise only that portion of the code in my service that talks to a separate service
- uses test doubles of those services, either in process or remote
- thus consist of many narrowly scoped tests, often no larger in scope than a unit test (and usually run with the same test framework that's used for unit tests)

broad integration tests

- require live versions of all services, requiring substantial test environment and network access
- exercise code paths through all services, not just code responsible for interactions

https://martinfowler.com/bliki/IntegrationTest.html

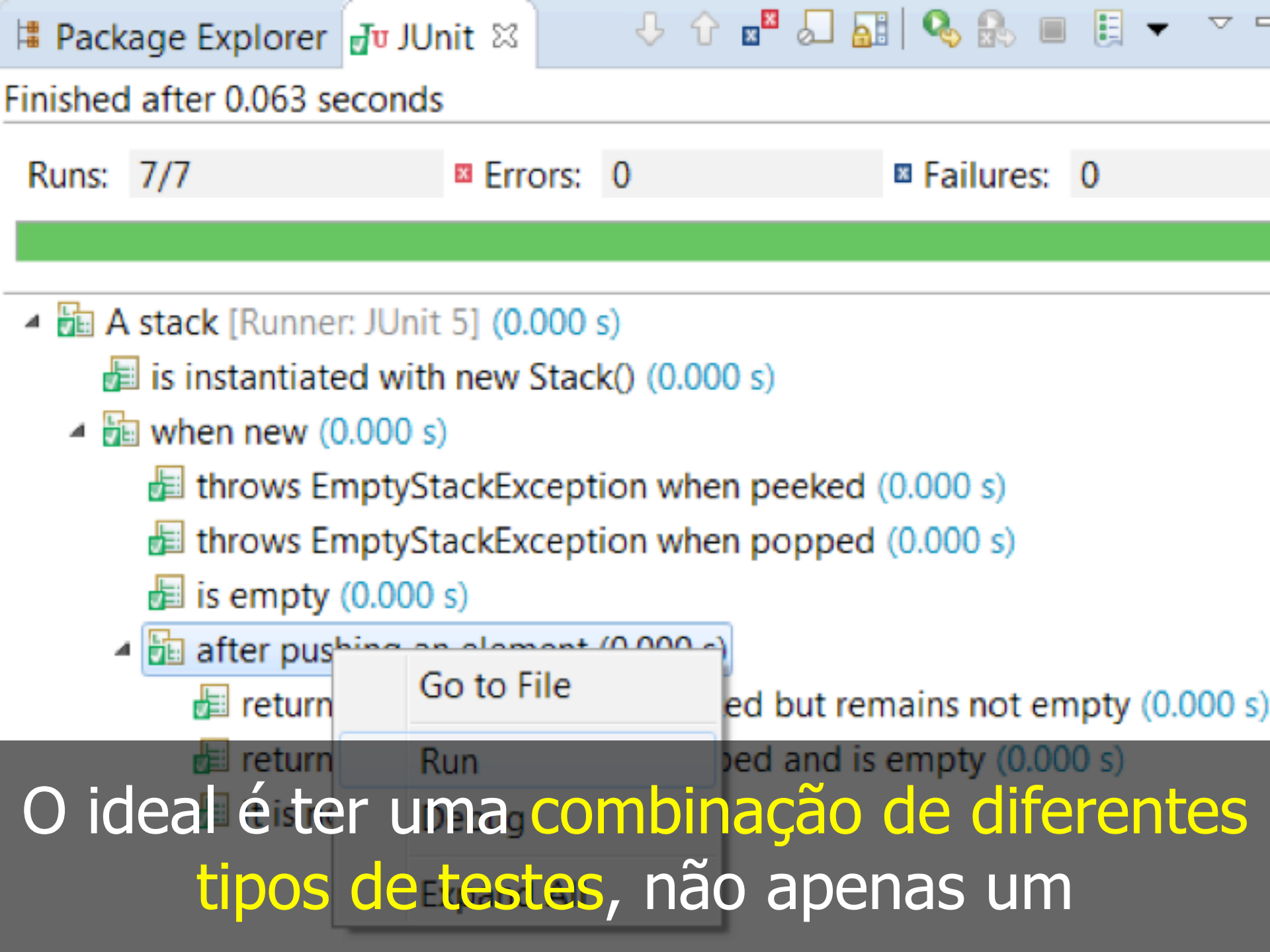

Um teste com as dependências mockadas
se torna um teste de unidade?

# E2E Tests

Replicam o ambiente o usuário final, ou seja, são testes executados de ponta a ponta

Pelas suas características acabam sendo mais lentos e frágeis sendo quebrados facilmente caso a interface com o usuário final seja modificada

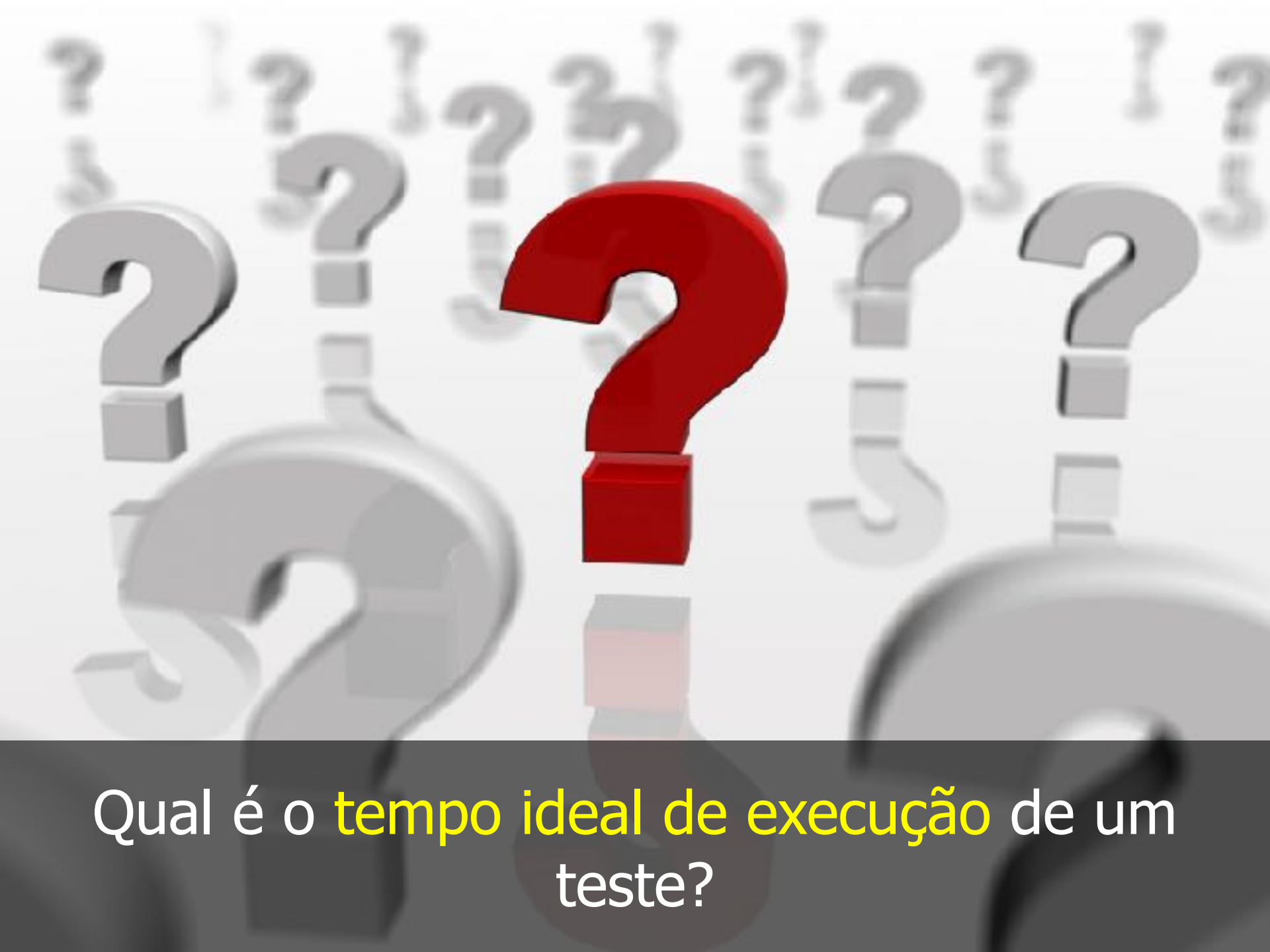

Existem dezenas de tipos de testes, muitos deles são muito similares


Package Explorer
JUnit
Finished after 0.063 seconds
Runs: 7/7
Errors: 0
Failures: 0
A stack [Runner: JUnit 5] (0.000 s)
is instantiated with new Stack() (0.000 s)
when new (0.000 s)
throws EmptyStackException when peeked (0.000 s)
throws EmptyStackException when popped (0.000 s)
is empty (0.000 s)
Go to File
Run
O ideal é ter uma combinação de diferentes tipos de testes, não apenas um


Marlboro
Formula 1™
FIRST

# FIRST

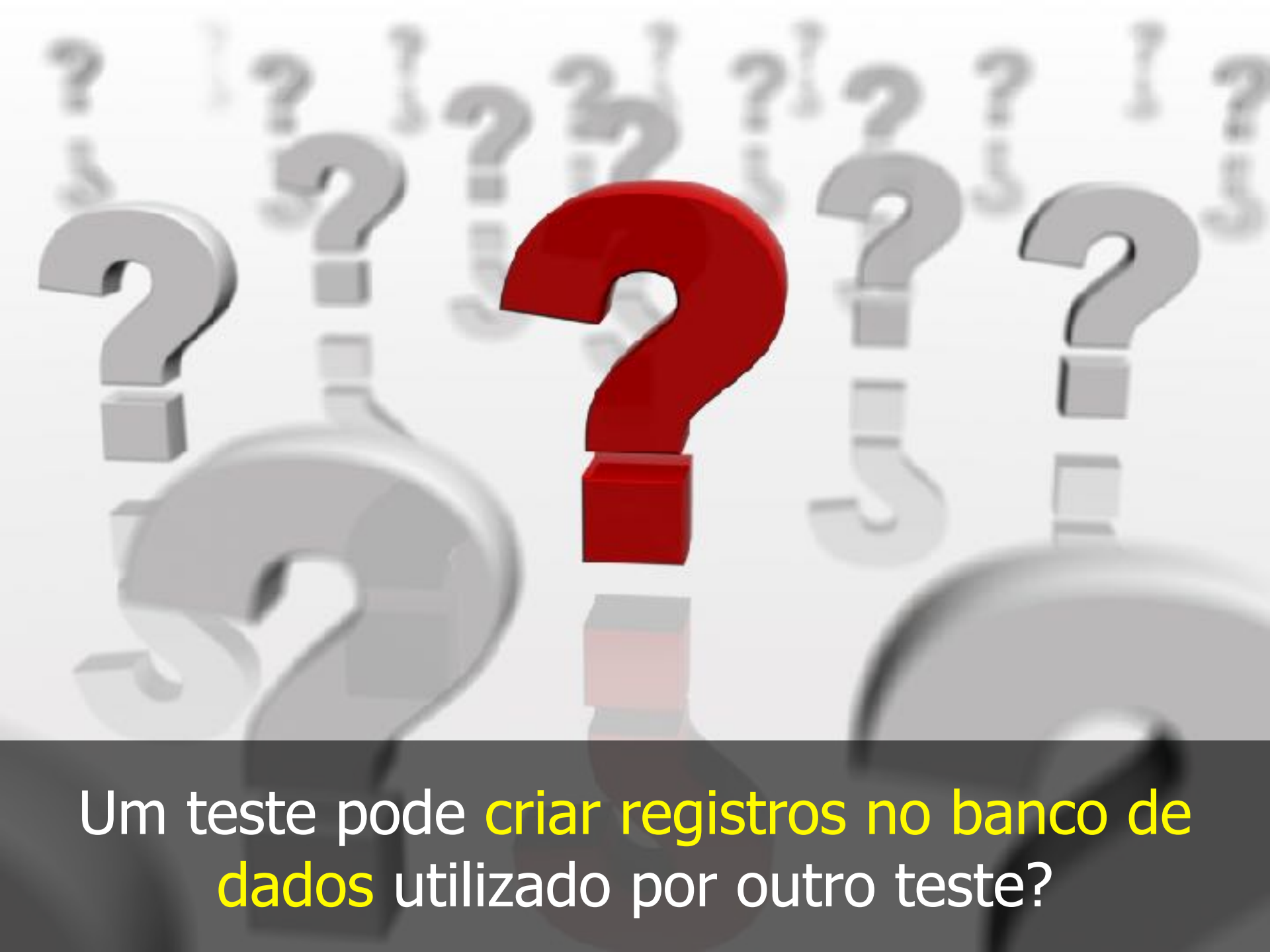
- Fast: Os testes devem rodar rápido



Qual é o tempo ideal de execução de um teste?

# FIRST

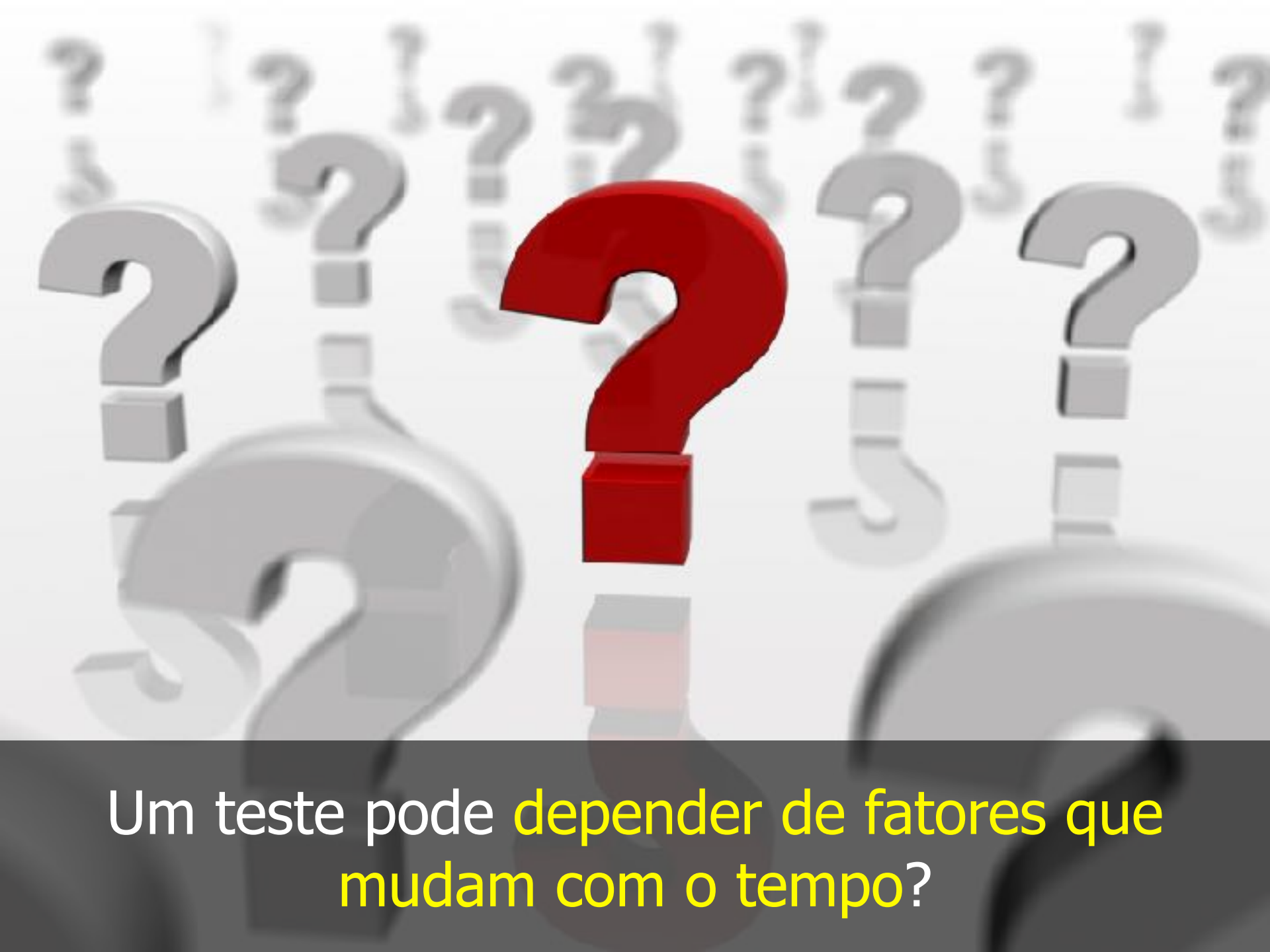
- Fast: Os testes devem rodar rápido
- Independent: Não deve existir dependência entre os testes, eles devem poder ser executados de forma isolada



Um teste pode criar registros no banco de dados utilizado por outro teste?

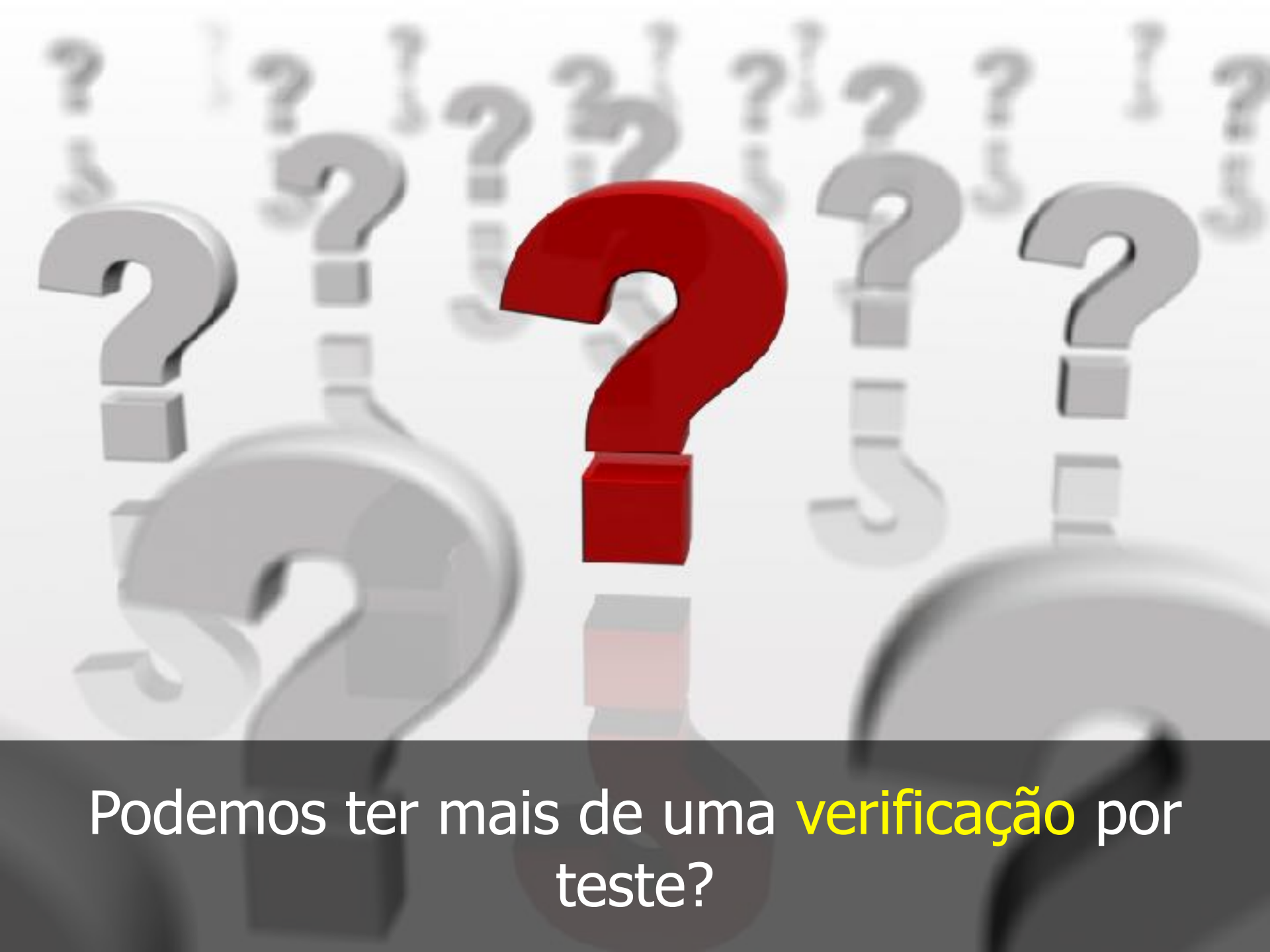
# FIRST

- Fast: Os testes devem rodar rápido
- Independent: Não deve existir dependência entre os testes, eles devem poder ser executados de forma isolada
- Repeatable: O resultado deve ser o mesmo independente da quantidade de vezes que seja executado


Um teste pode depender de fatores que mudam com o tempo?

# FIRST

- Fast: Os testes devem rodar rápido
- Independent: Não deve existir dependência entre os testes, eles devem poder ser executados de forma isolada
- Repeatable: O resultado deve ser o mesmo independente da quantidade de vezes que seja executado
- Self-validating: O próprio teste deve ter uma saída bem definida que é válida ou não fazendo com que ele passe ou falhe

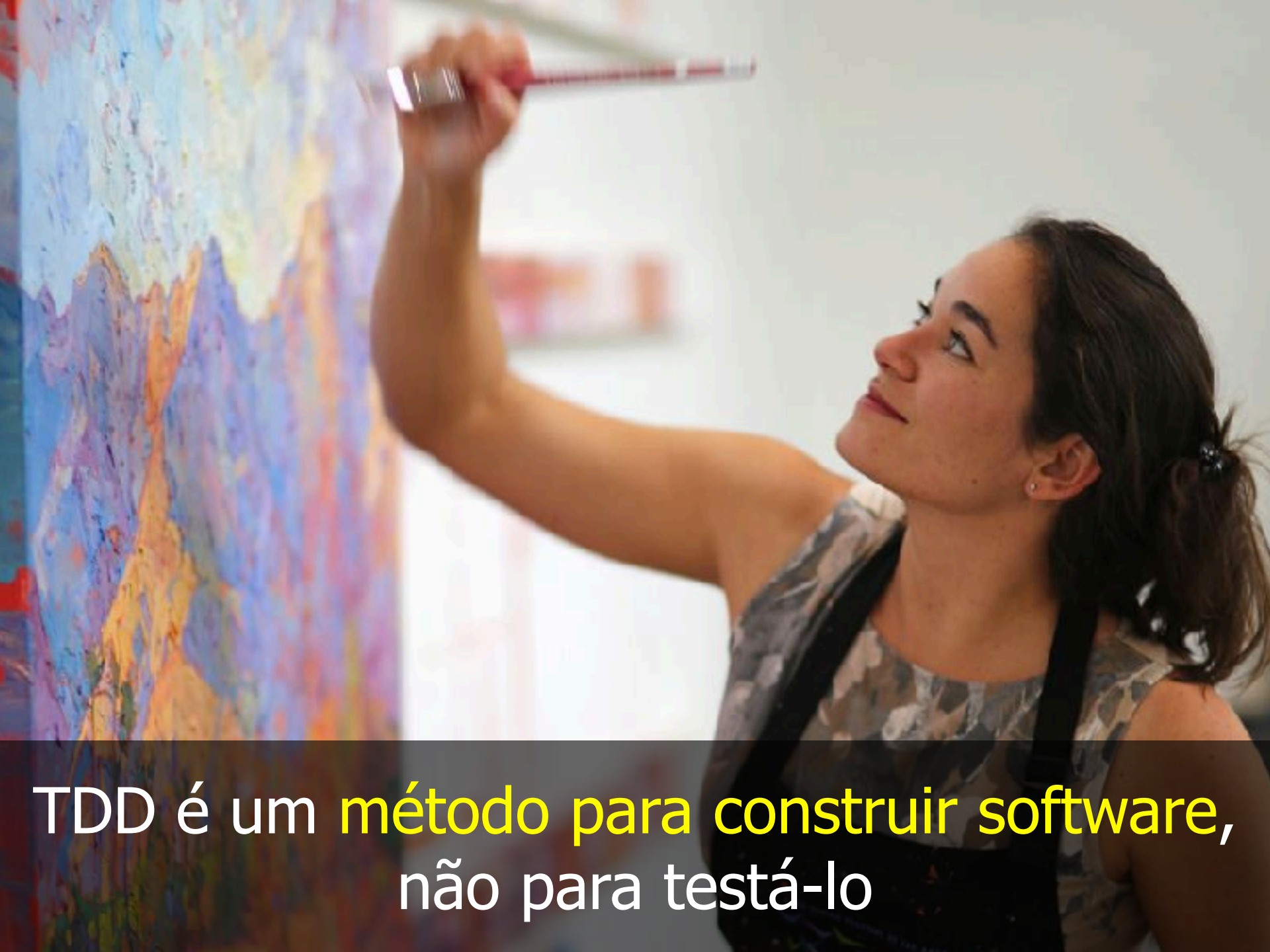


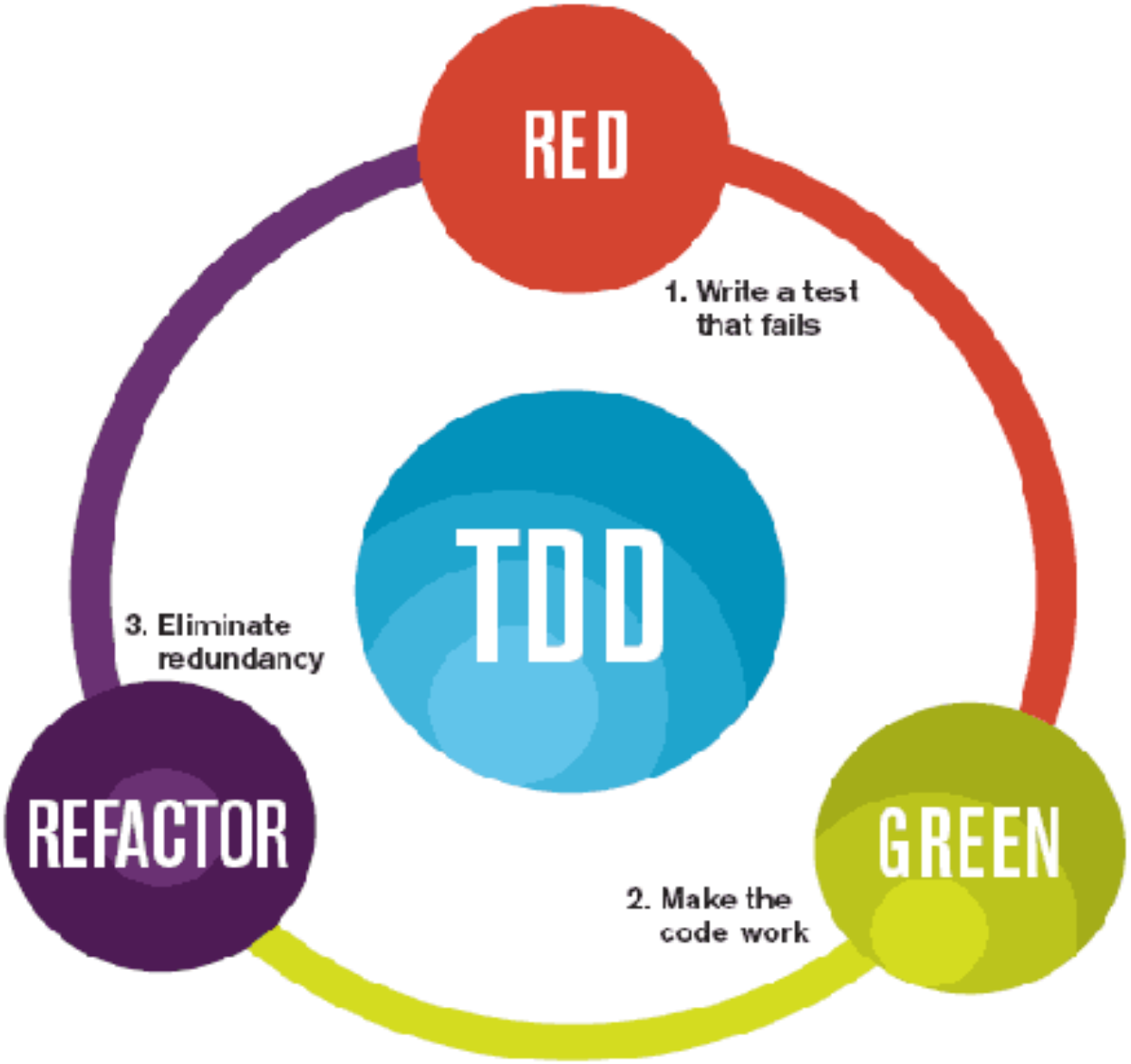
Podemos ter mais de uma verificação por teste?

# FIRST

- Fast: Os testes devem rodar rápido
- Independent: Não deve existir dependência entre os testes, eles devem poder ser executados de forma isolada
- Repeatable: O resultado deve ser o mesmo independente da quantidade de vezes que seja executado
- Self-validating: O próprio teste deve ter uma saída bem definida que é válida ou não fazendo com que ele passe ou falhe
- Timely: Os testes devem ser escritos antes do código-fonte

# Como funciona o Test-Driven Development?


TDD é um método para construir software,
não para testá-lo


RED
1. Write a test that fails
TDD
GREEN
2. Make the code work
REFACTOR
3. Eliminate redundancy

"TDD is a way of managing fear during programming"

Kent Beck


butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
butUncleBob.com
Edit
Properties
Refactor
Where Used
Search
Files
Versions
Recent Changes
User Guide
Test History
ArticleS UncleBob
TheThreeRulesOfTdd
THE THREE LAWS OF TDD.
Over the years I have come to describe Test Driven Development in terms of three simple rules. They are:
1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.
You must begin by writing a unit test for the functionality that you intend to write. But by rule 2, you can't write very much of that unit test. As soon as the unit test code fails to compile, or fails an assertion, you must stop and write production code. But by rule 3 you can only write the production code that makes the test compile or pass, and no more.
If you think about this you will realize that you simply cannot write very much code at all without compiling and executing something. Indeed, this is really the point. In everything we do, whether writing tests, writing production code, or refactoring, we keep the system executing at all times. The time between running tests is on the order of seconds, or minutes. Even 10 minutes is too long.
Too see this in operation, take a look at The Bowling Game Kata.
Now most programmers, when they first hear about this technique, think: "This is stupid!" "It's going to slow me down, it's a waste of time and effort, It will keep me from thinking, it will keep me from designing, it will just break my flow." However, think about what would happen if you walked in a room full of people working this way. Pick any random person at any random time. A minute ago, all their code worked.
Let me repeat that: A minute ago all their code worked! And it doesn't matter who you pick, and it doesn't matter when you pick. A minute ago all their code worked!
If all your code works every minute, how often will you use a debugger? Answer, not very often. It's easier to simply hit ^Z a bunch of times to get the code back to a working state, and then try to write the last minutes worth again. And if you aren't debugging very much, how much time will you be saving? How much time do you spend debugging now? How much time do you spend fixing bugs once you've debugged them? What if you could decrease that time by a significant fraction?
But the benefit goes far beyond that. If you work this way, then every hour you are producing several tests. Every day dozens of tests. Every month hundreds of tests. Over the course of a year you will write thousands of tests. You can keep all these tests and run them any time you like. When would you run them? All the time! Any time you made any kind of change at all!
Why don't we clean up code that we know is messy? We're afraid we'll break it. But if we have the tests, we can be reasonably sure that the code is not broken, or that we'll detect the breakage immediately. If we have the tests we become fearless about making changes. If we see messy code, or an unclean structure, we can clean it without fear. Because of the tests, the code becomes malleable again. Because of the tests, software becomes soft again.
But the benefits go beyond that. If you want to know how to call a certain API, there is a test that does it. If you want to know how to create a certain object, there is a test that does it. Anything you want to know about the existing system, there is a test that demonstrates it. The tests are like little design documents, little coding examples, that describe how the system works and how to use it.
Have you ever integrated a third party library into your project? You got a big manual full of nice documentation. At the end there was a thin appendix of examples. Which of the two did you read? The examples of course! That's what the unit tests are! They are the most useful part of the documentation. They are the living examples of how to use the code. They are design documents that are hideously detailed, utterly unambiguous, so formal that they execute, and they cannot get out of sync with the production code.
But the benefits go beyond that. If you have ever tried to add unit tests to a system that was already working, you probably found that it wasn't much fun. You likely found that you either had to change portions of the design of the system, or cheat on the tests: because the system you were trying to write tests for was not designed to be testable. For example, you'd like to test some function f'. However, f' calls another function that deletes a record from the database. In your test, you don't want the record deleted, but you don't have any way to stop it. The system wasn't

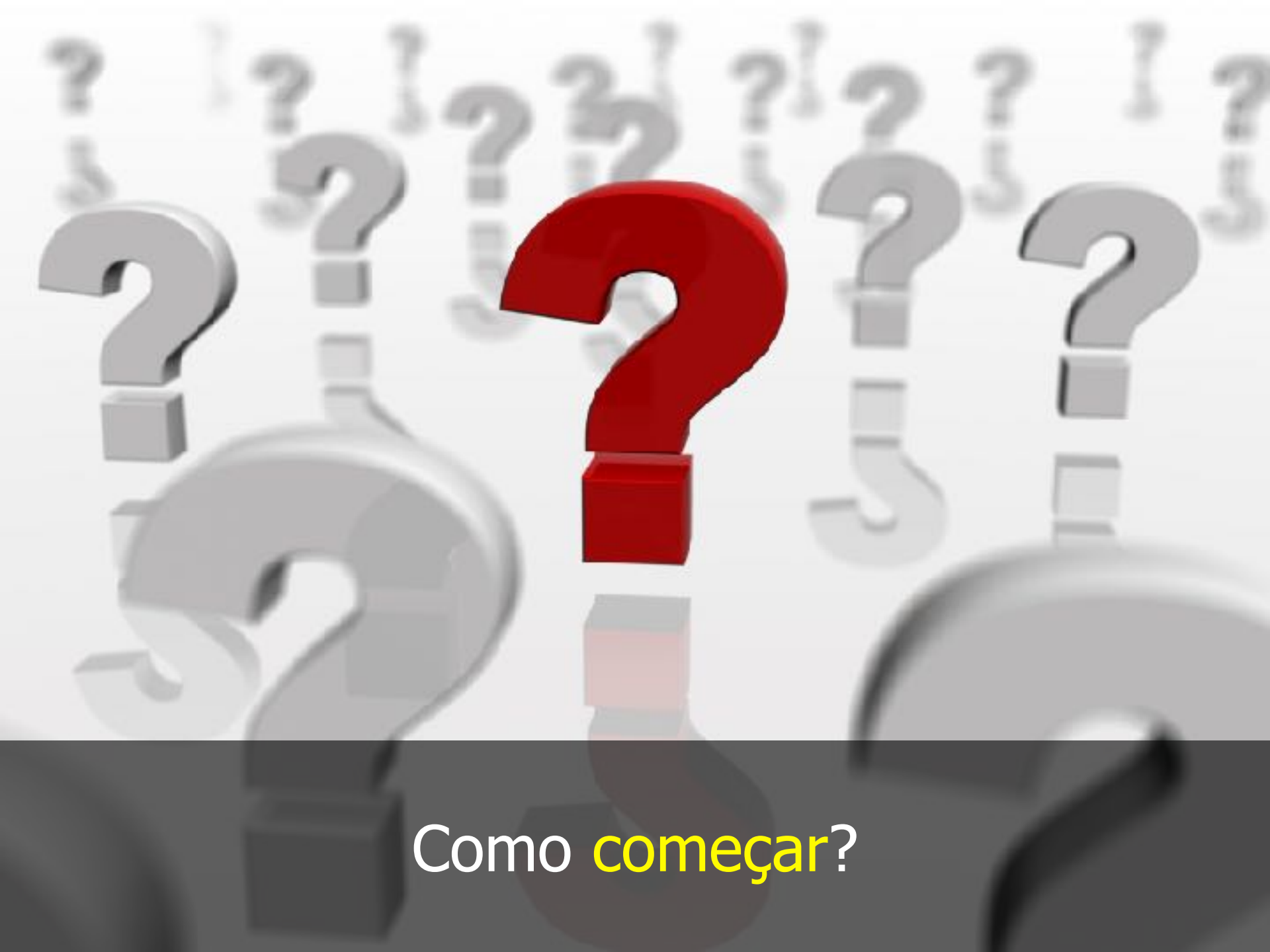
# Three Laws of TDD

Você não pode escrever nenhum código até ter escrito um teste que detecte uma possível falha.

Você não pode escrever mais testes de unidade do que o suficiente para detectar a falha.

Você não pode escrever mais código do que o suficiente para passar nos testes.

(Robert C. Martin)


Como começar?

Dê o primeiro passo, ou seja, crie um exemplo que as pessoas na equipe possam seguir e utilizar como base

Pra chegar nos 100% de cobertura você precisa passar pelo 5%, 10%, 50% e assim por diante...

Foque no que tem mais risco e muda com mais frequência, o sucesso não está em ter 100% de cobertura mas sim em automatizar os testes daquilo que dá mais retorno

Usar test patterns como um stub ou mock não é necessariamente ruim, em muitos casos é algo necessário, mas só conseguir testar se utilizar esses recursos pode indicar que o design precisa melhorar

A disciplina do TDD pode parecer muito complicada no início, mas certamente faz sentido definir o cenário desejado antes e formalizá-lo no código


Os testes automatizados são a única forma que temos para garantir que o código funciona