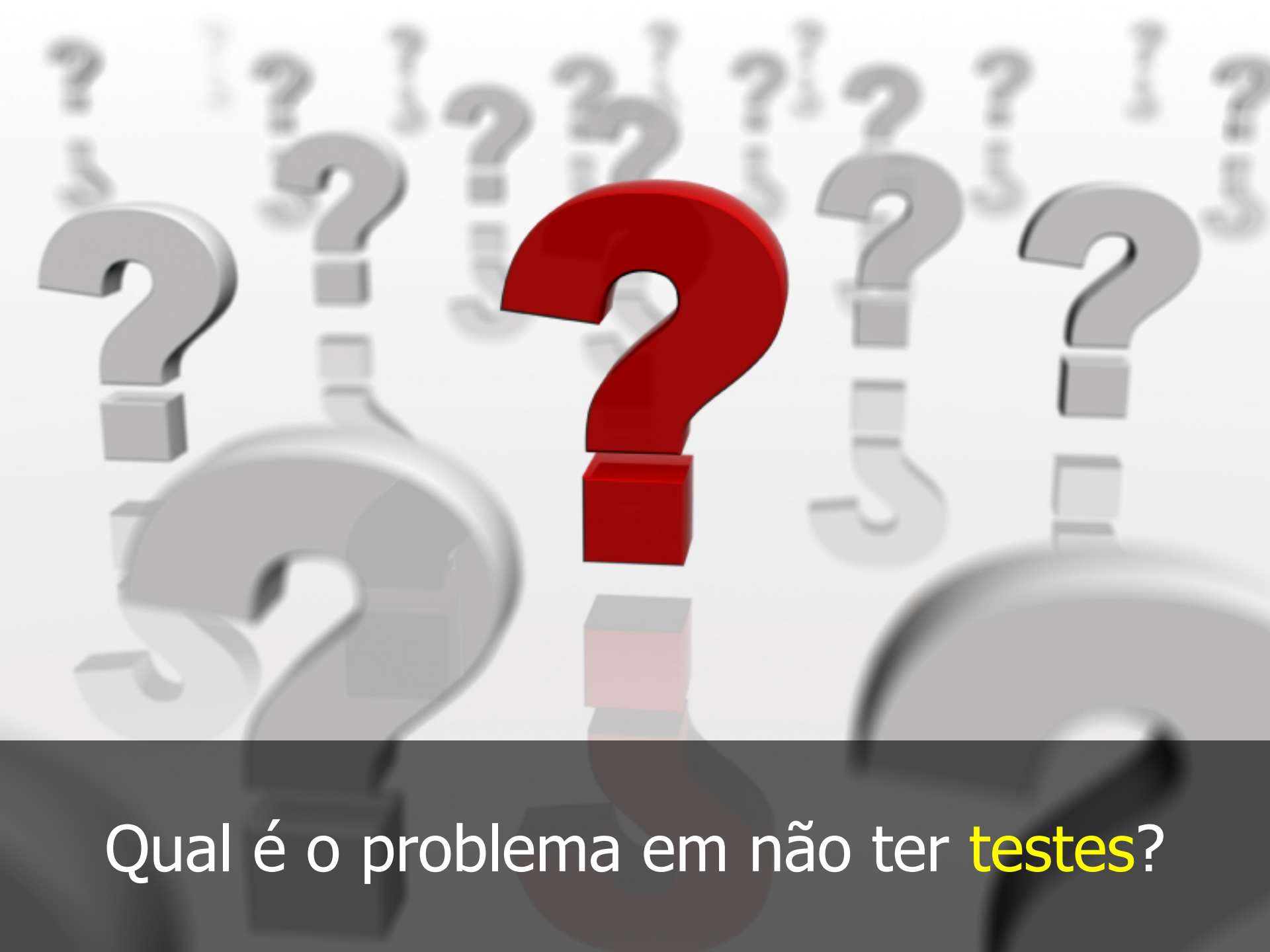
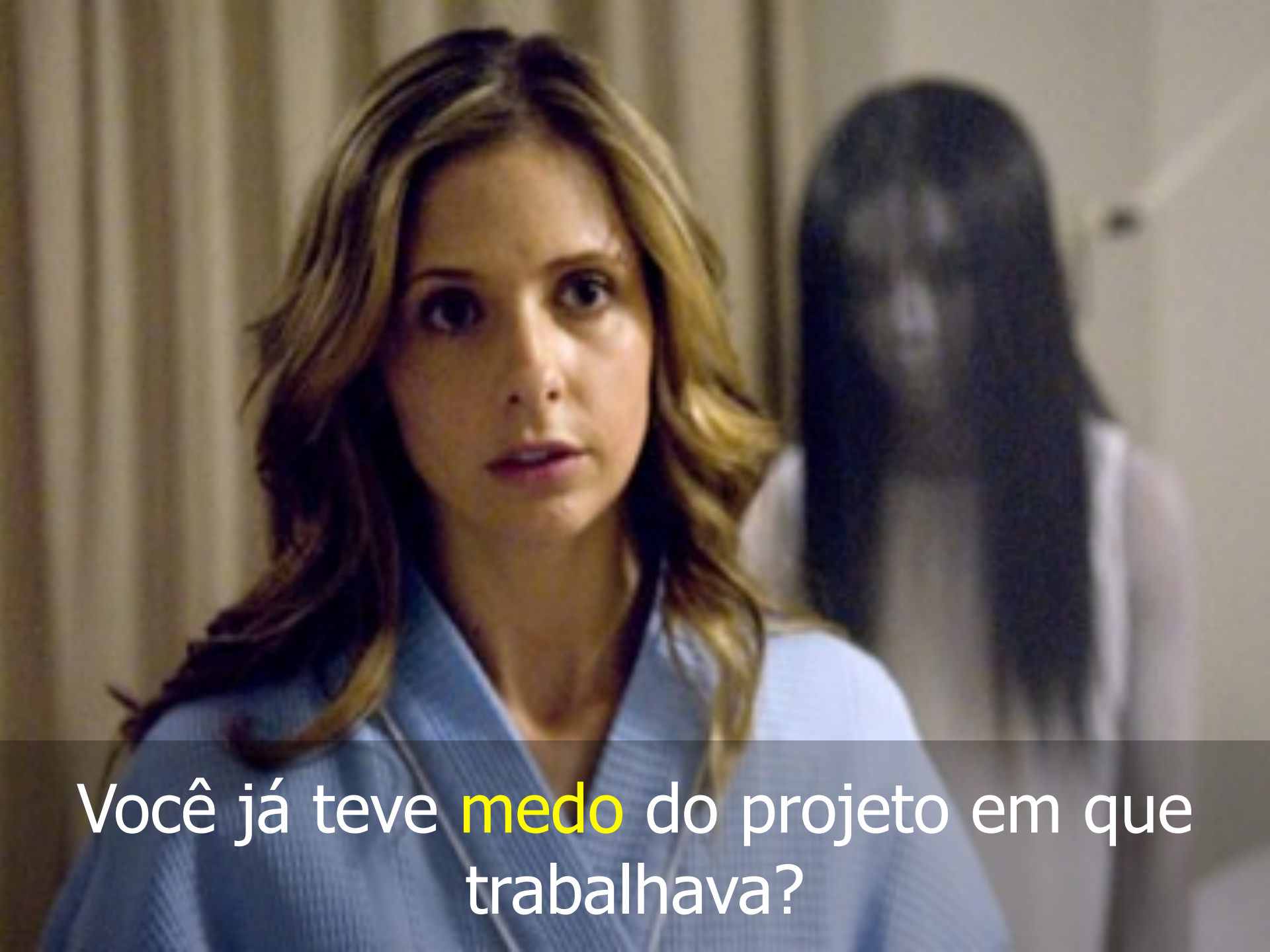
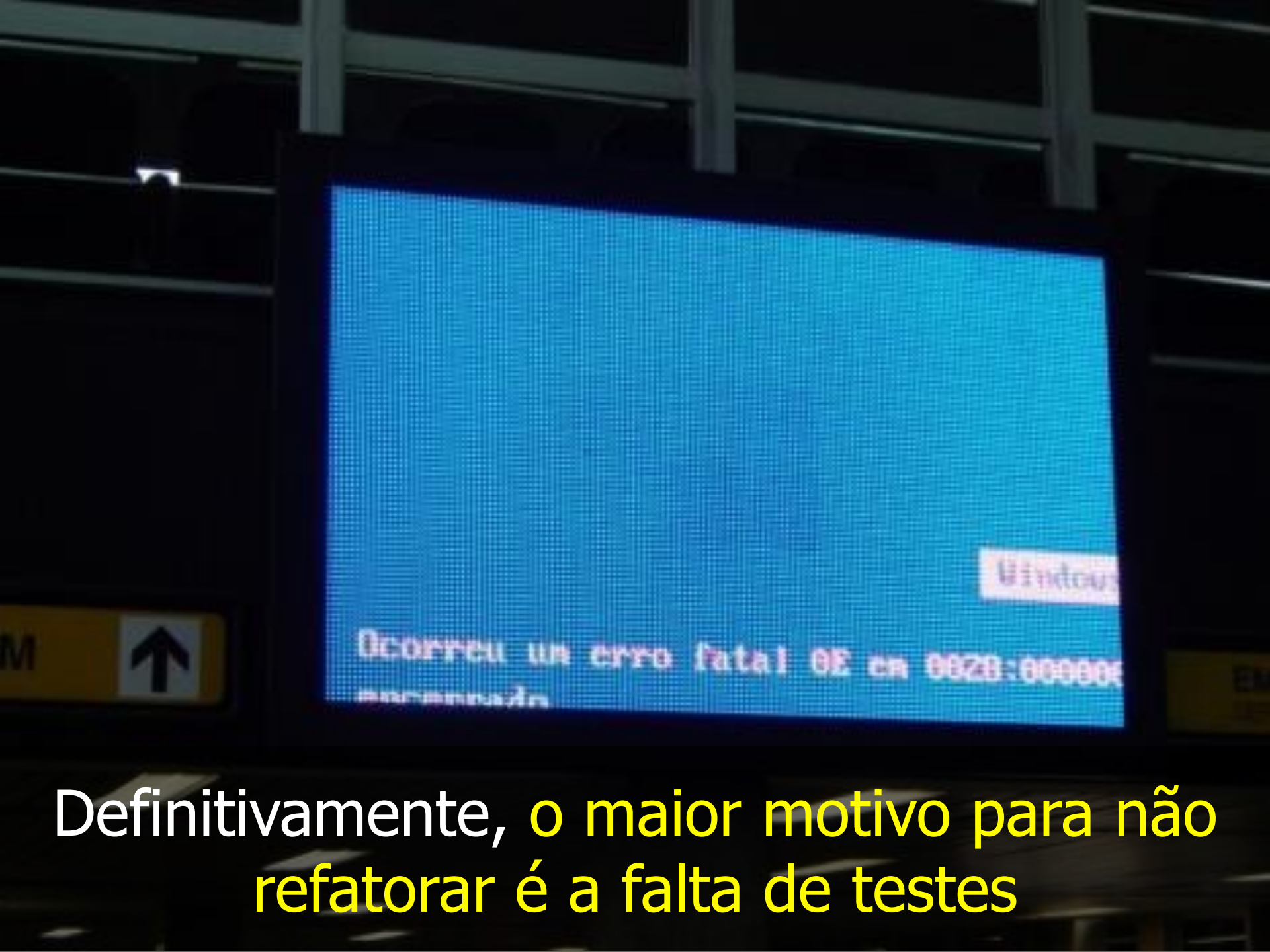

Qual é o problema em não ter testes?


Estamos sempre torcendo para que o código que escrevemos funcione

Nos tornamos preenchedores ágeis e profissionais das telas do sistema

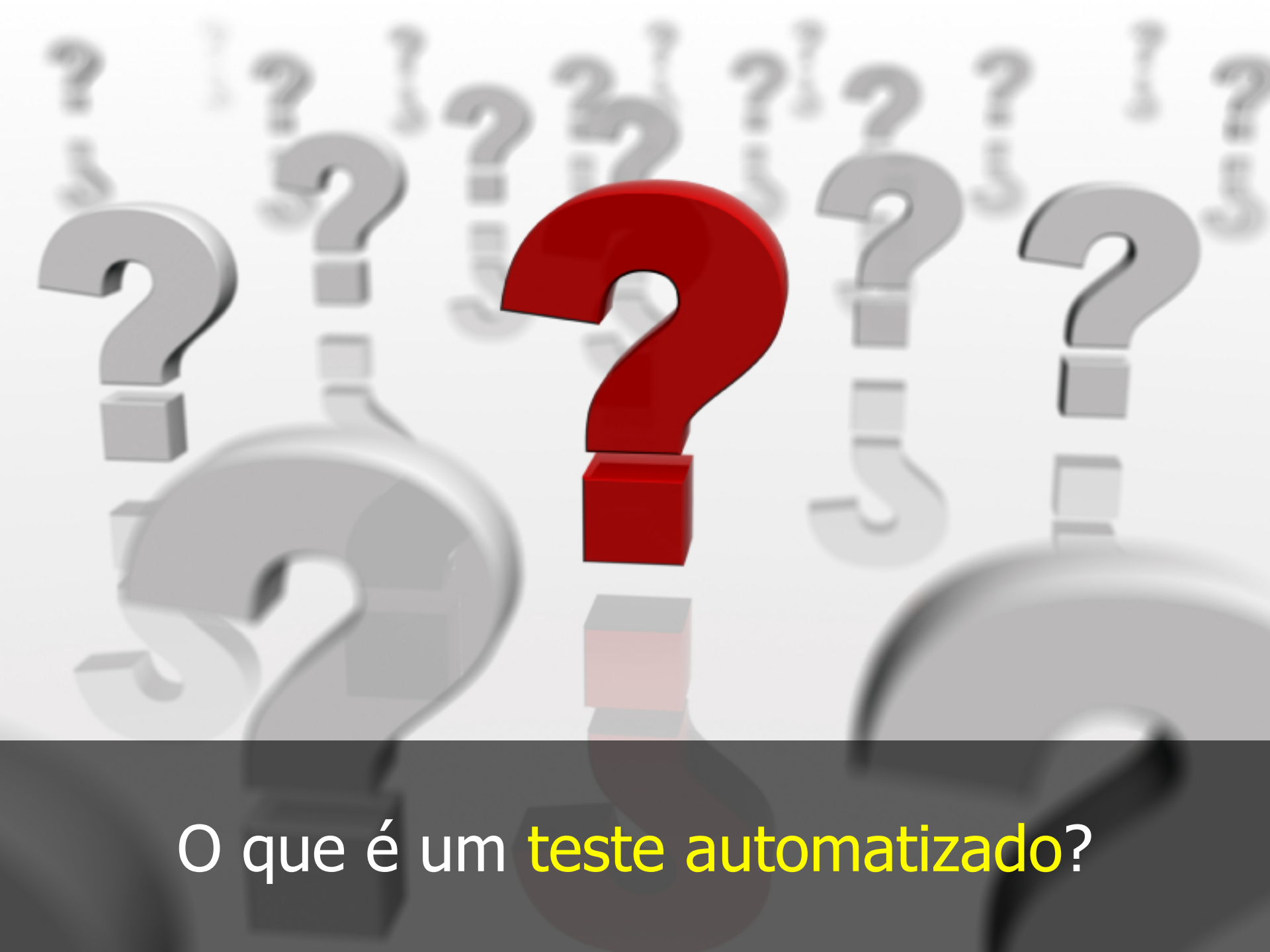
Temos a sensação de estar em um castelo de cartas que pode cair a qualquer momento


Você já teve medo do projeto em que trabalhava?



Definitivamente, o maior motivo para não refatorar é a falta de testes

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



O que é um teste automatizado?

Estrutura de um teste automatizado

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

When: Executar o comportamento

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

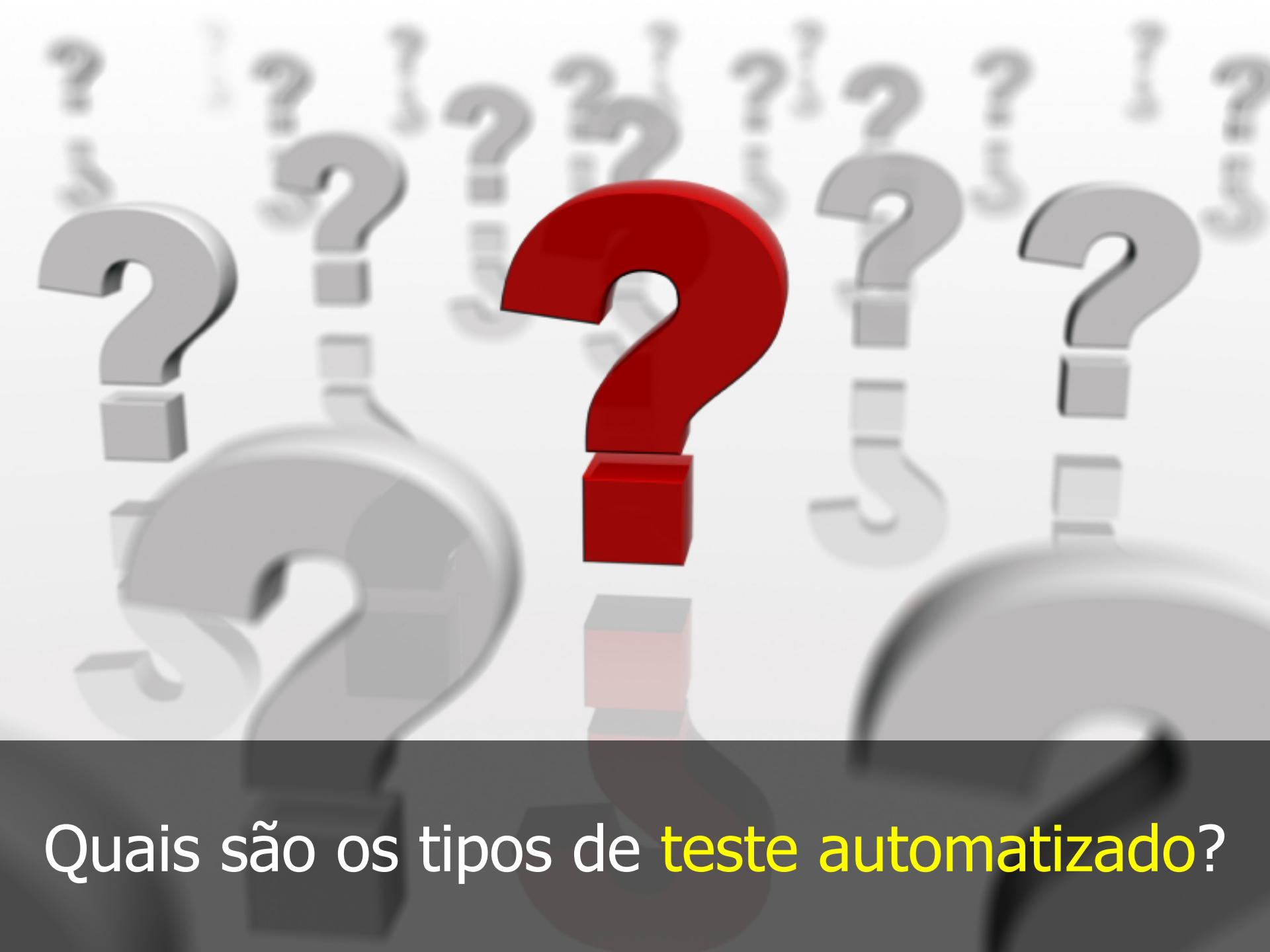
Exemplos

Fazer um pedido

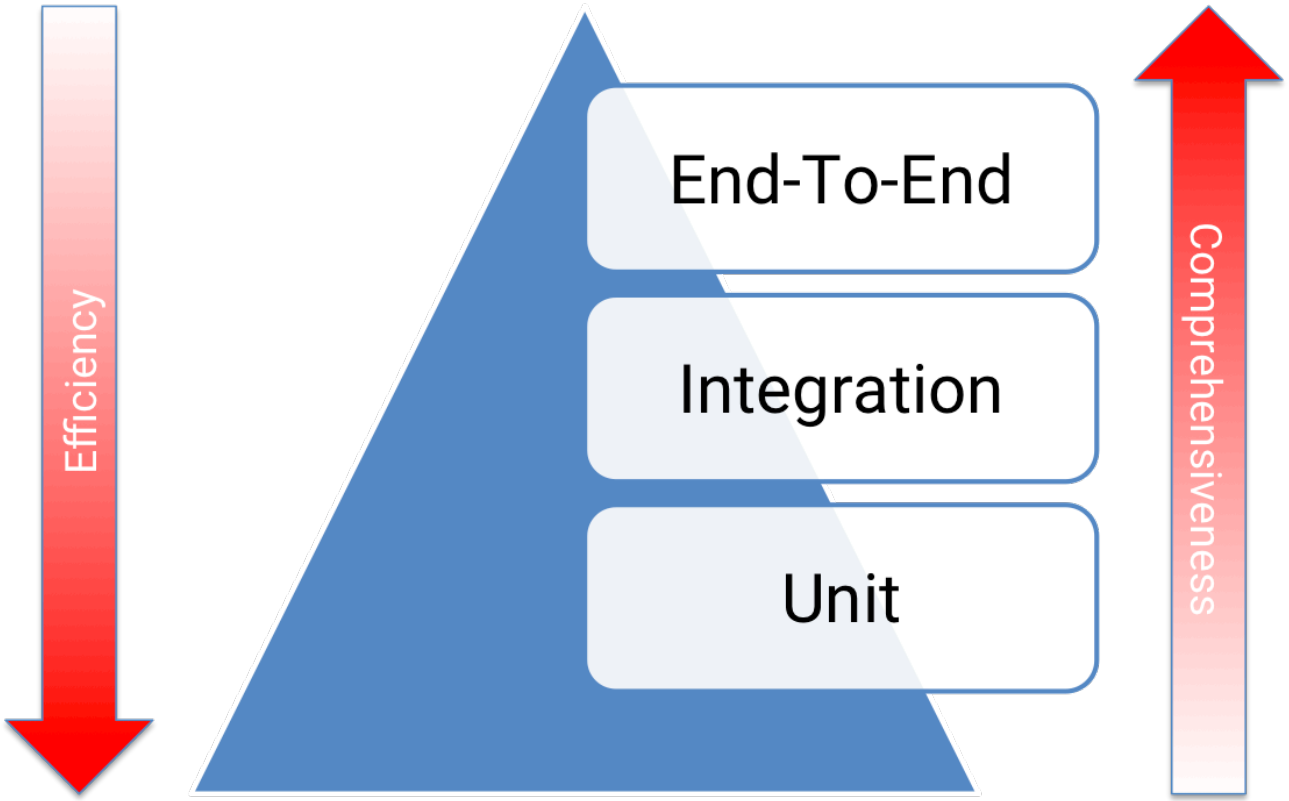
Dado um novo pedido com 3 itens associados, um Livro de R$50,00, um CD de R$20,00 e um DVD de R$30,00
Quando o pedido for realizado
Então deve ser retornado uma confirmação do pedido contendo o código, juntamente com o total do pedido de R$100,00 e o status aguardando pagamento, além disso os itens devem ter sido bloqueados no estoque

Cancelar um pedido existente

Dado um pedido existente, contendo apenas o código
Quando o pedido for cancelado
Então deve ser retornado uma confirmação do cancelamento do pedido, caso ele ainda não tenha sido pago, além disso os itens devem ser liberados no estoque

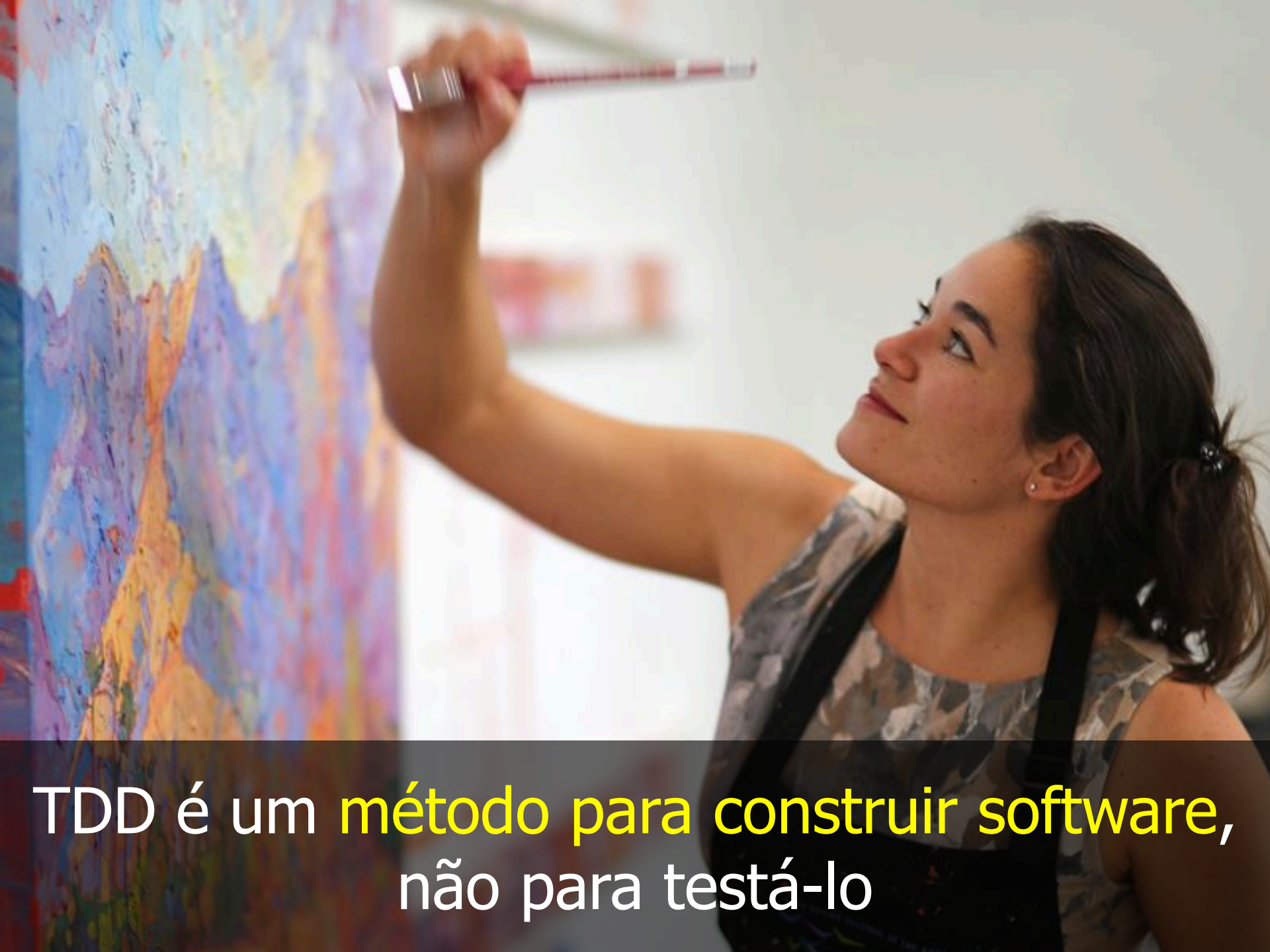


Quais são os tipos de teste automatizado?

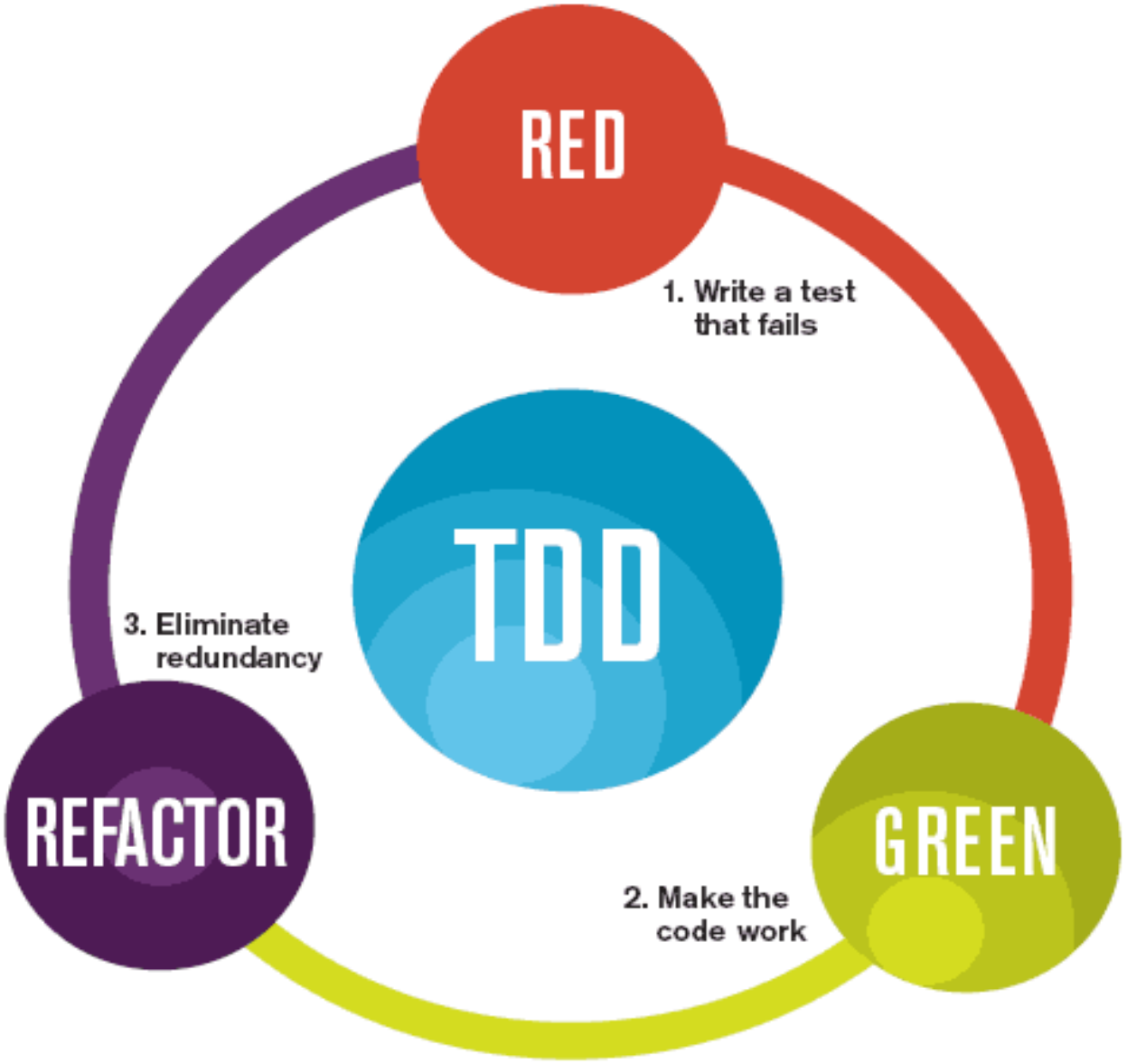

Efficiency
End-To-End
Integration
Unit
Comprehensiveness

O ideal é sempre ter uma combinação de diversos tipos de testes, somente testes de unidade não garantem que um sistema funciona... Quem aqui anda de bicicleta?

Como funciona o Test-Driven Development?

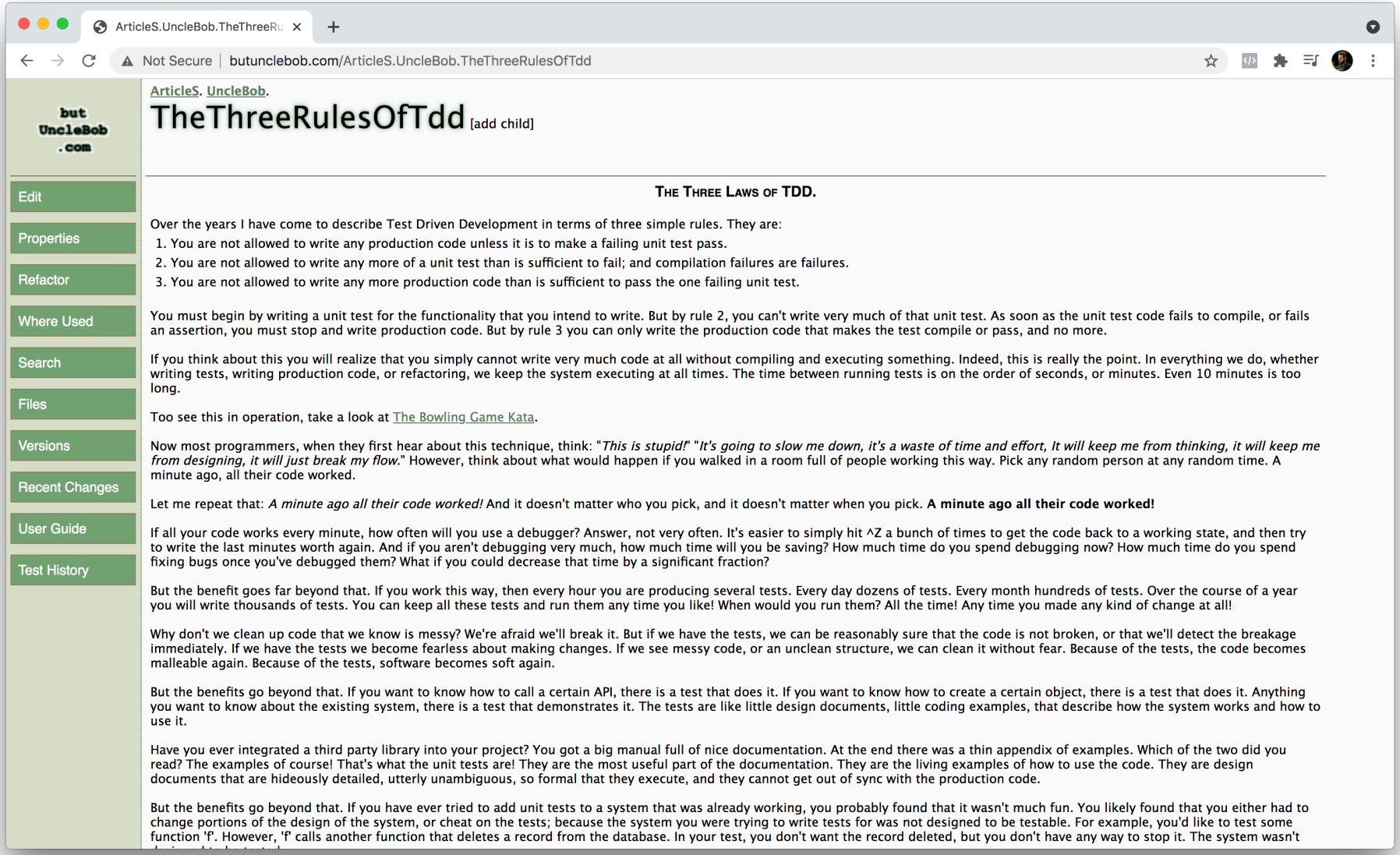


TDD é um método para construir software,
não para testá-lo


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


ArticleS.UncleBob.TheThreeRu
Not Secure
butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
but UncleBob .com
Edit
Properties
Refactor
Where Used
Search
Files
Versions
Recent Changes
User Guide
Test History
ArticleS. UncleBob.
TheThreeRulesOfTdd [add child]
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

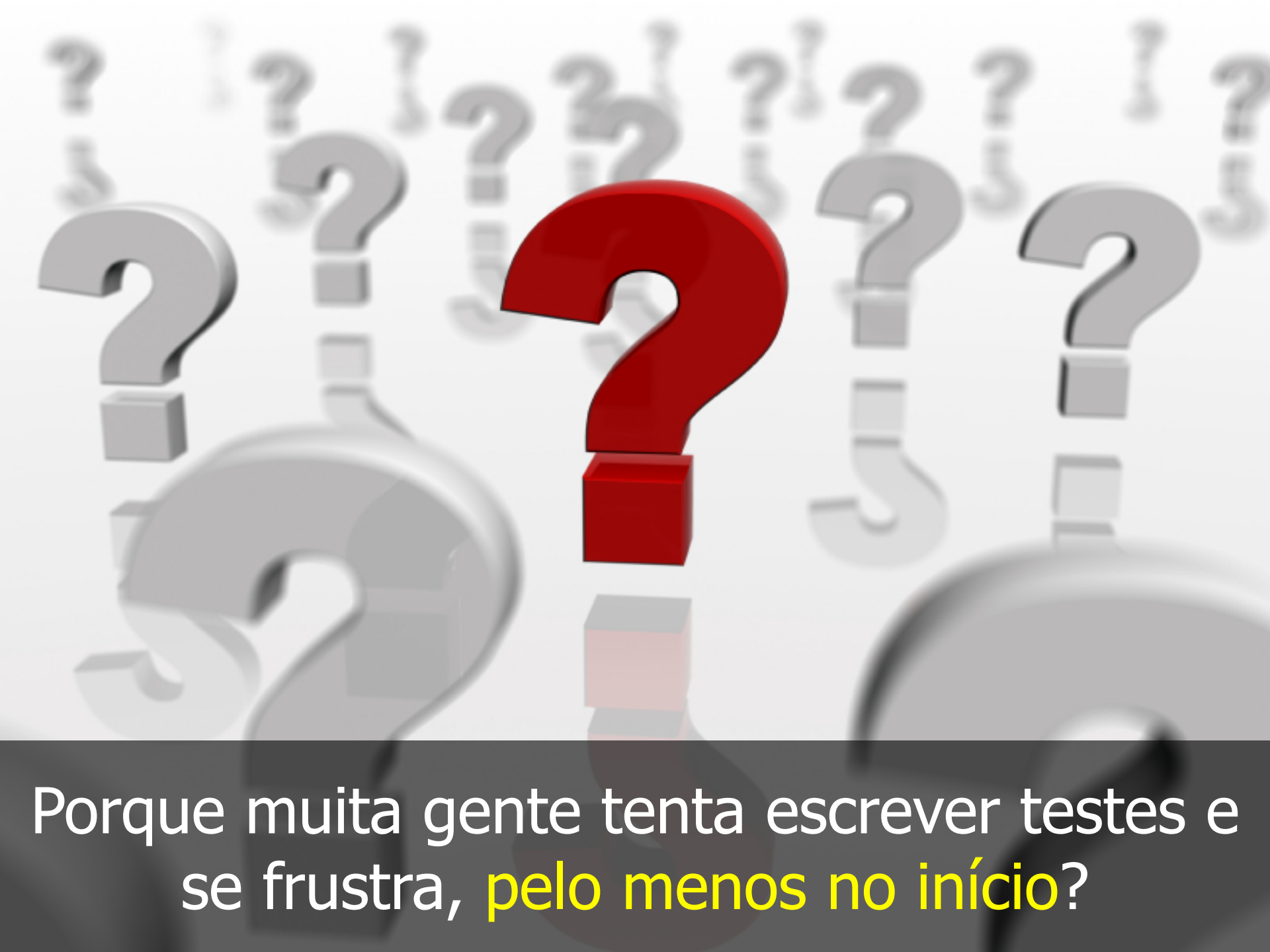
Three Laws of TDD

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.

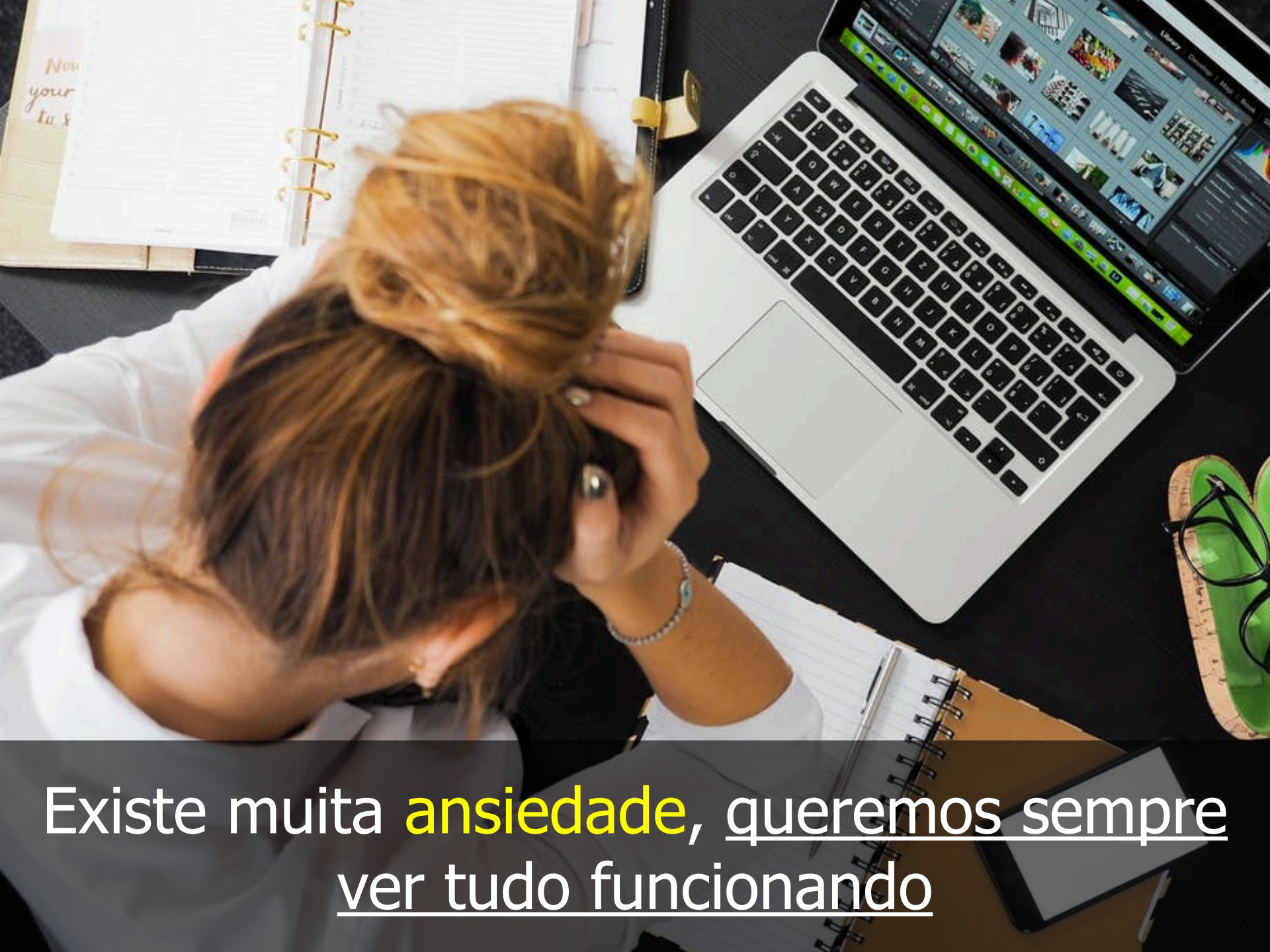
Você não pode escrever mais código do que o suficiente para passar nos testes.

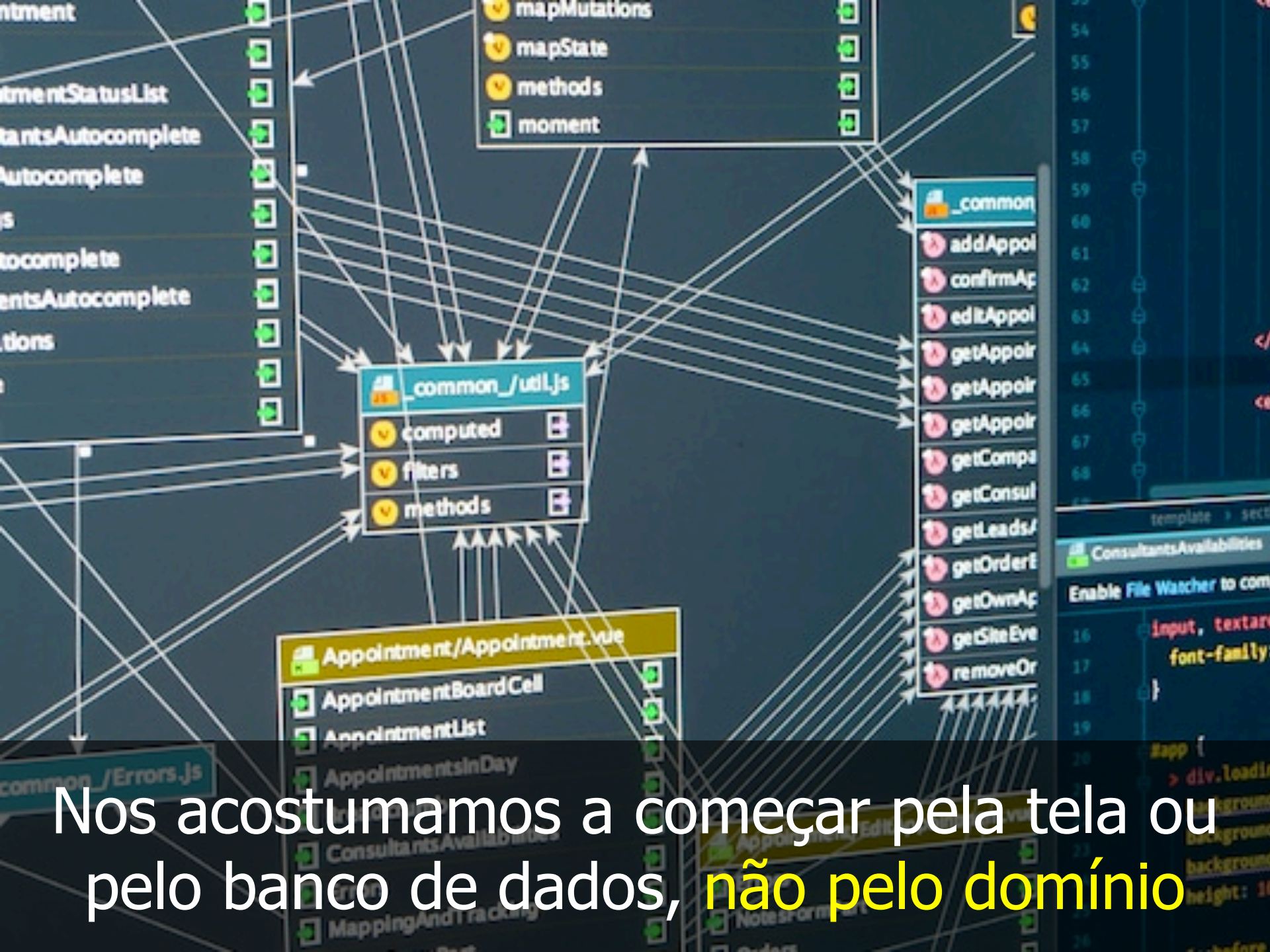
(Robert C. Martin)



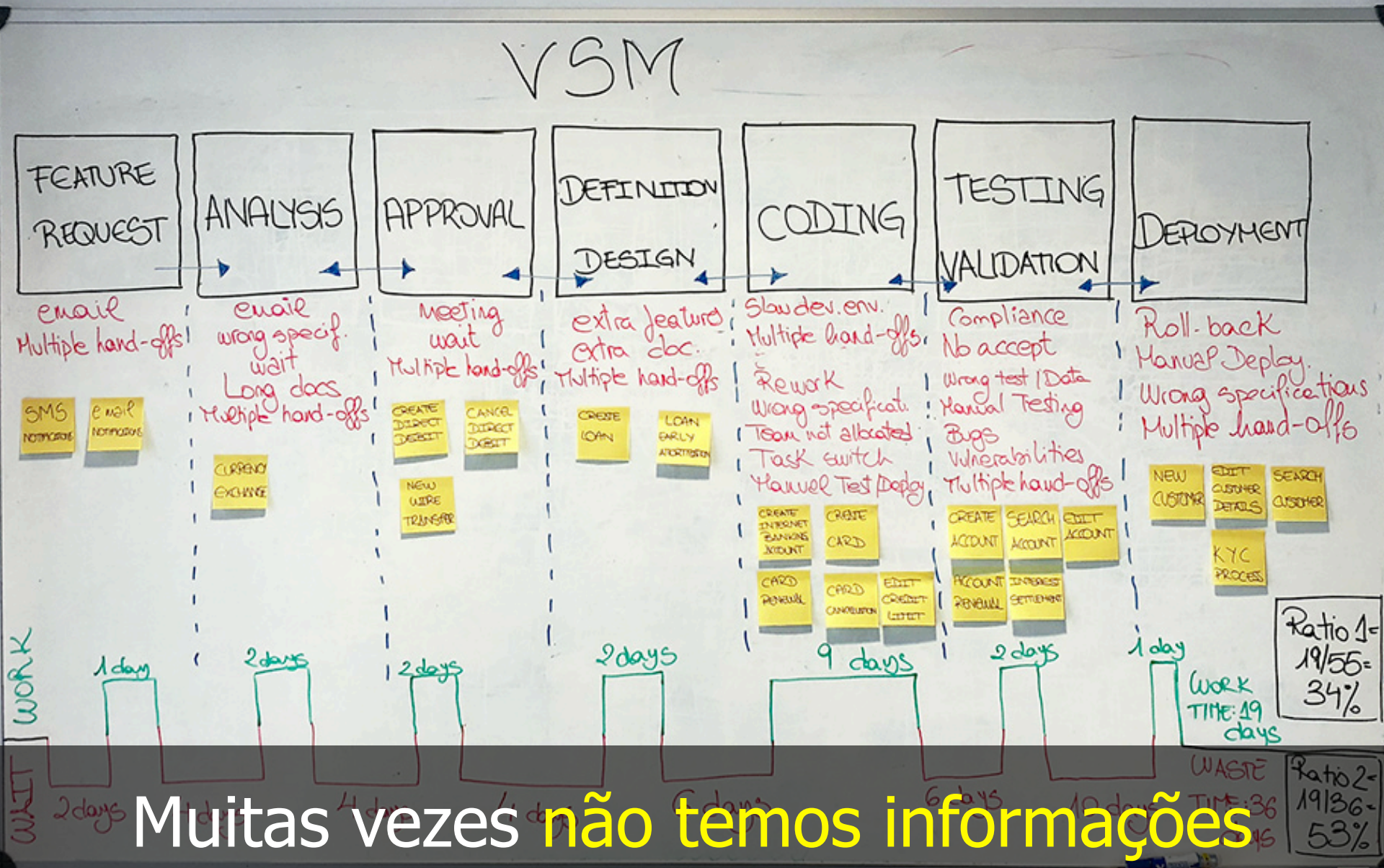
Porque muita gente tenta escrever testes e se frustra, pelo menos no início?


Escrever os testes requer muita disciplina


Existe muita ansiedade, queremos sempre ver tudo funcionando

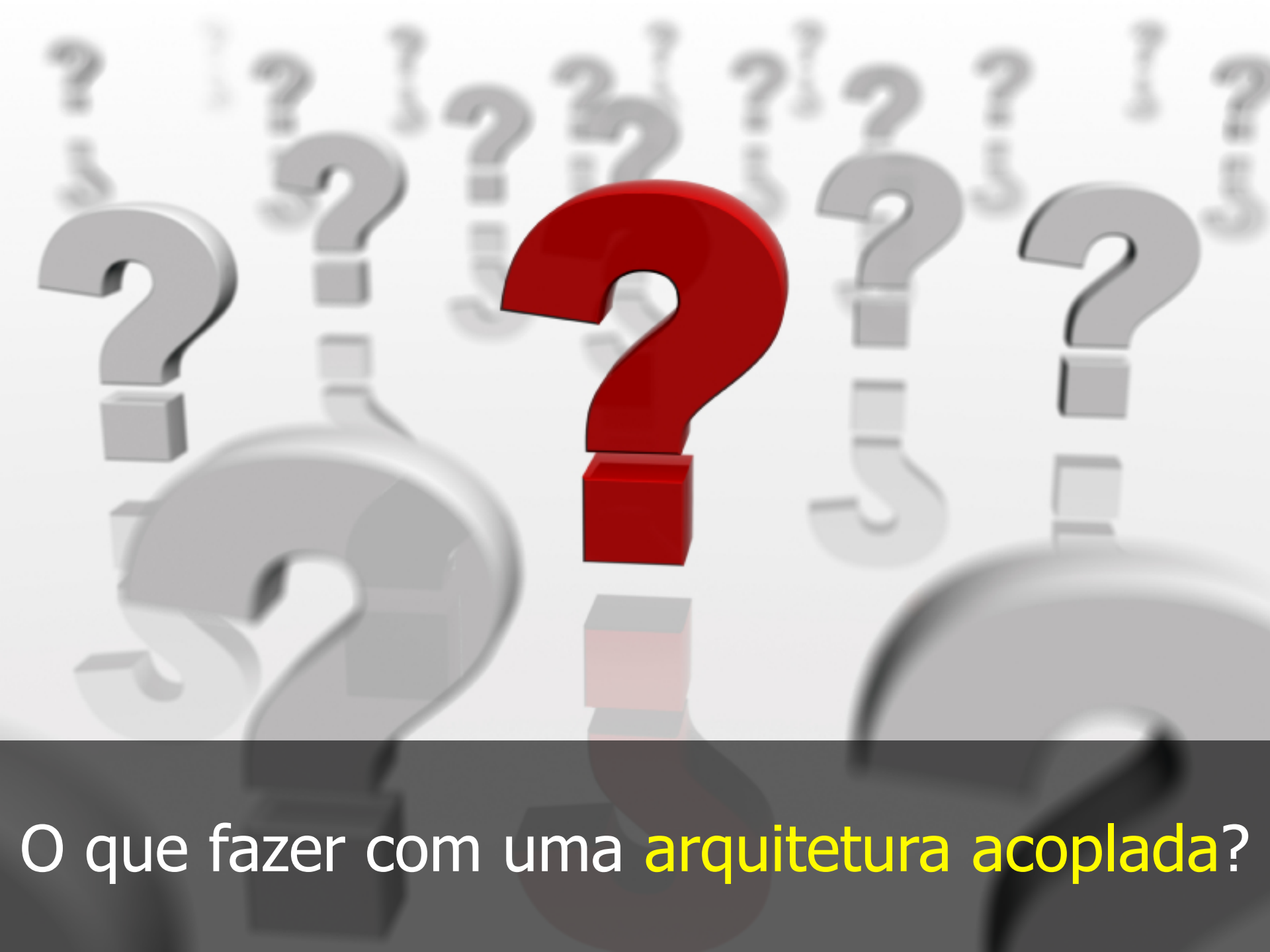


Nos acostumamos a começar pela tela ou pelo banco de dados, não pelo domínio



Muitas vezes não temos informações suficientes para começar a desenvolver

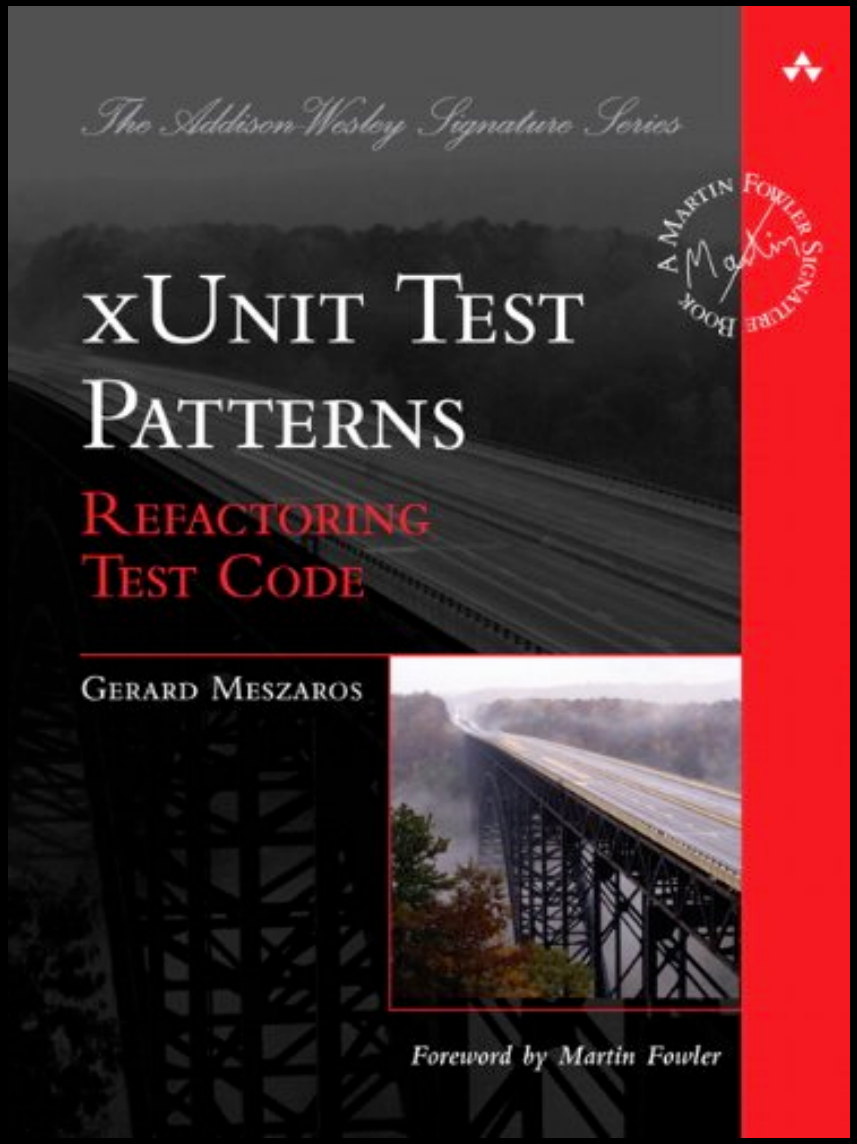

Muitas vezes a arquitetura não é testável



O que fazer com uma arquitetura acoplada?

Um test double é um padrão que tem o objetivo de substituir um DOC (depended-on component) em um determinado tipo de teste por motivos de performance ou segurança


Nem tudo é mock


The Addison Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

Dummy: Objetos que criamos apenas para completar a lista de parâmetros que precisamos passar para invocar um determinado método

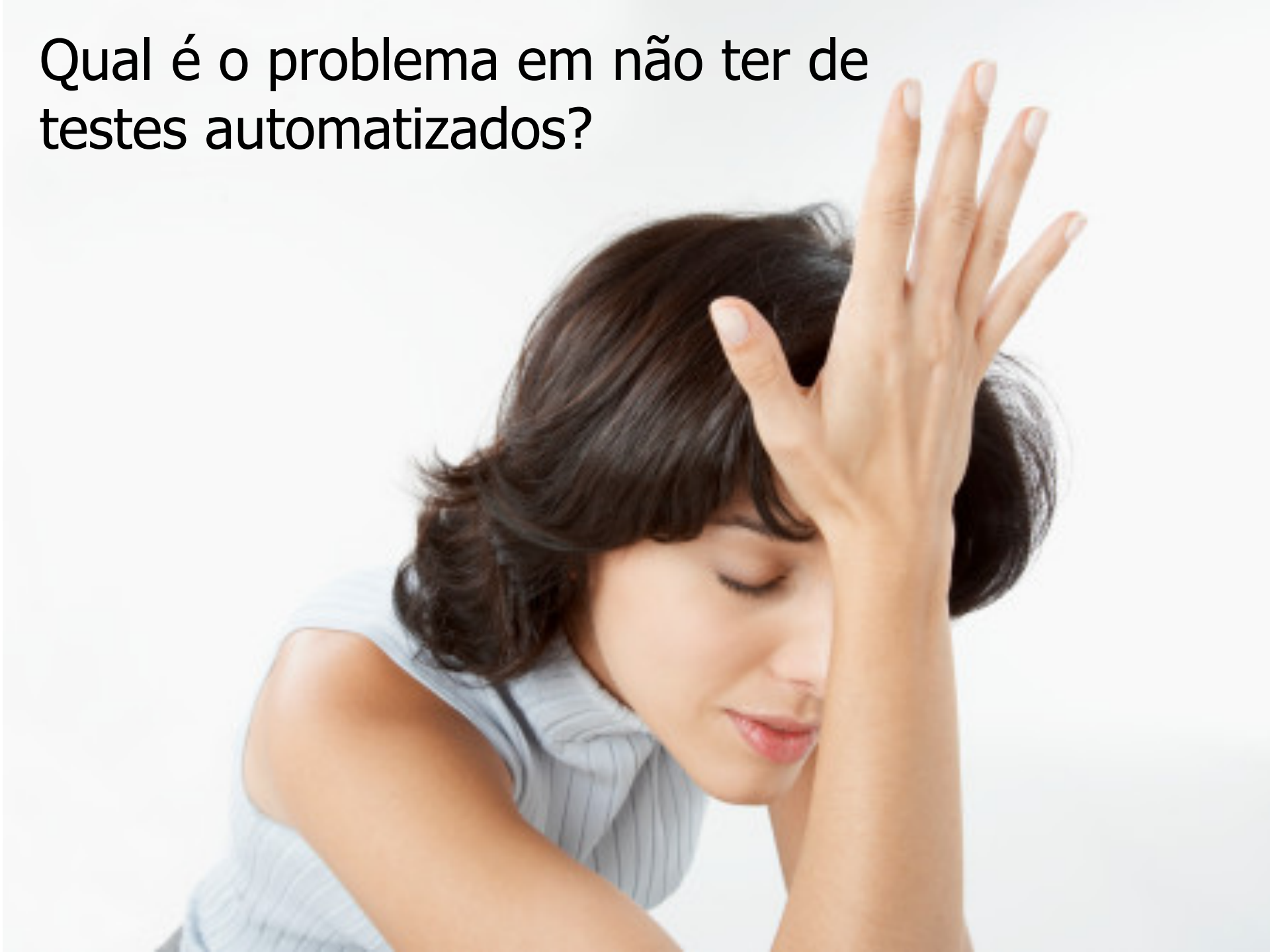
Stubs: Objetos que retornam respostas prontas, definidas para um determinado teste, por questão de performance ou segurança (exemplo: quando eu executar o método fazer pedido preciso que o método pegar cotação do dólar retorne R$3,00)

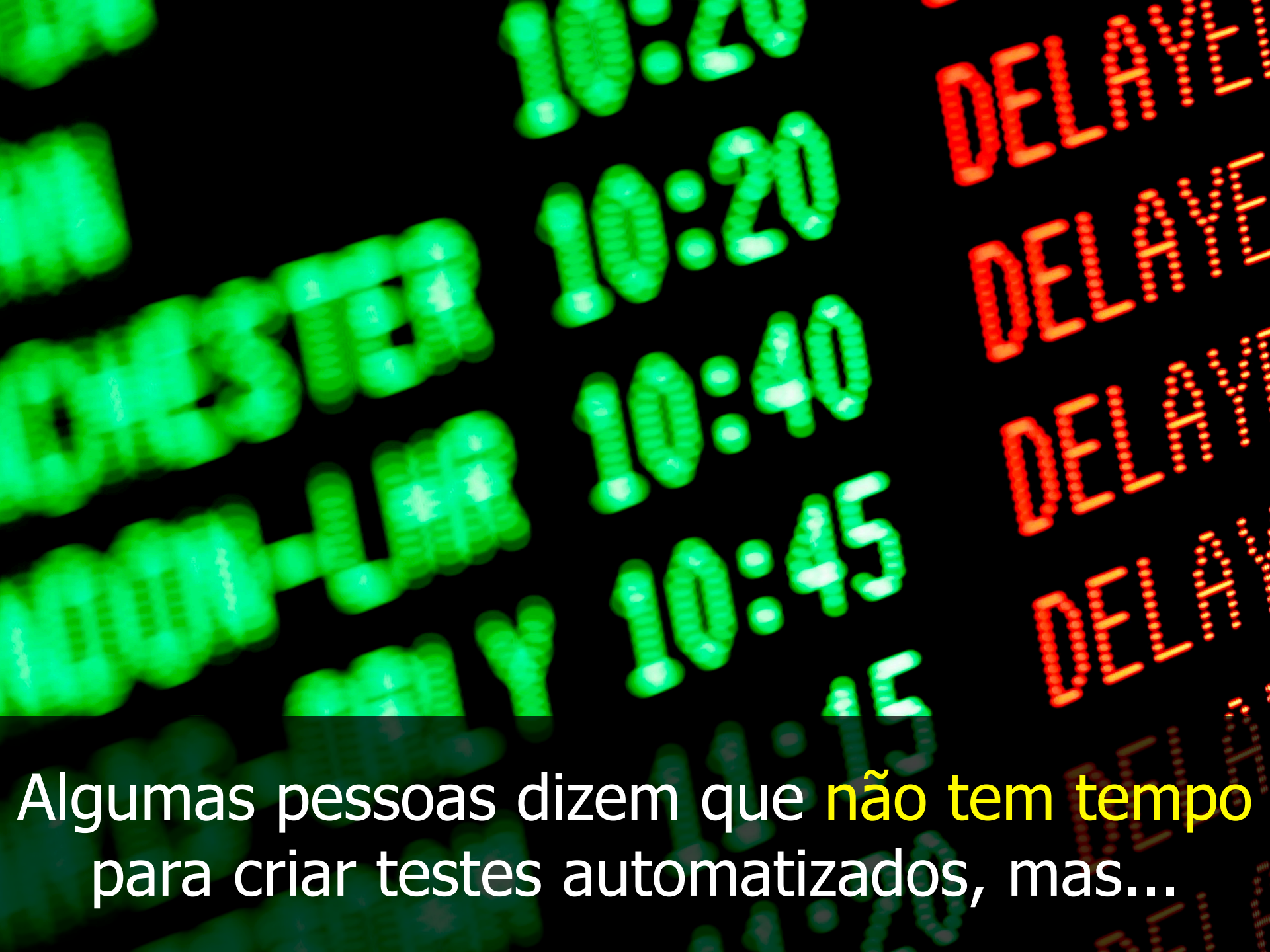
Spies: Objetos que "espionam" a execução do método e armazenam os resultados para verificação posterior (exemplo: quando eu executar o método fazer pedido preciso saber se o método enviar email foi invocado internamente e com quais parâmetros)

Mocks: Objetos similares a stubs e spies, permitem que você diga exatamente o que quer que ele faça e o teste vai quebrar se isso não acontecer

Fake: Objetos que tem implementações que simulam o funcionamento da instância real, que seria utilizada em produção (exemplo: uma base de dados em memória)

Qual é o problema em não ter de testes automatizados?





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

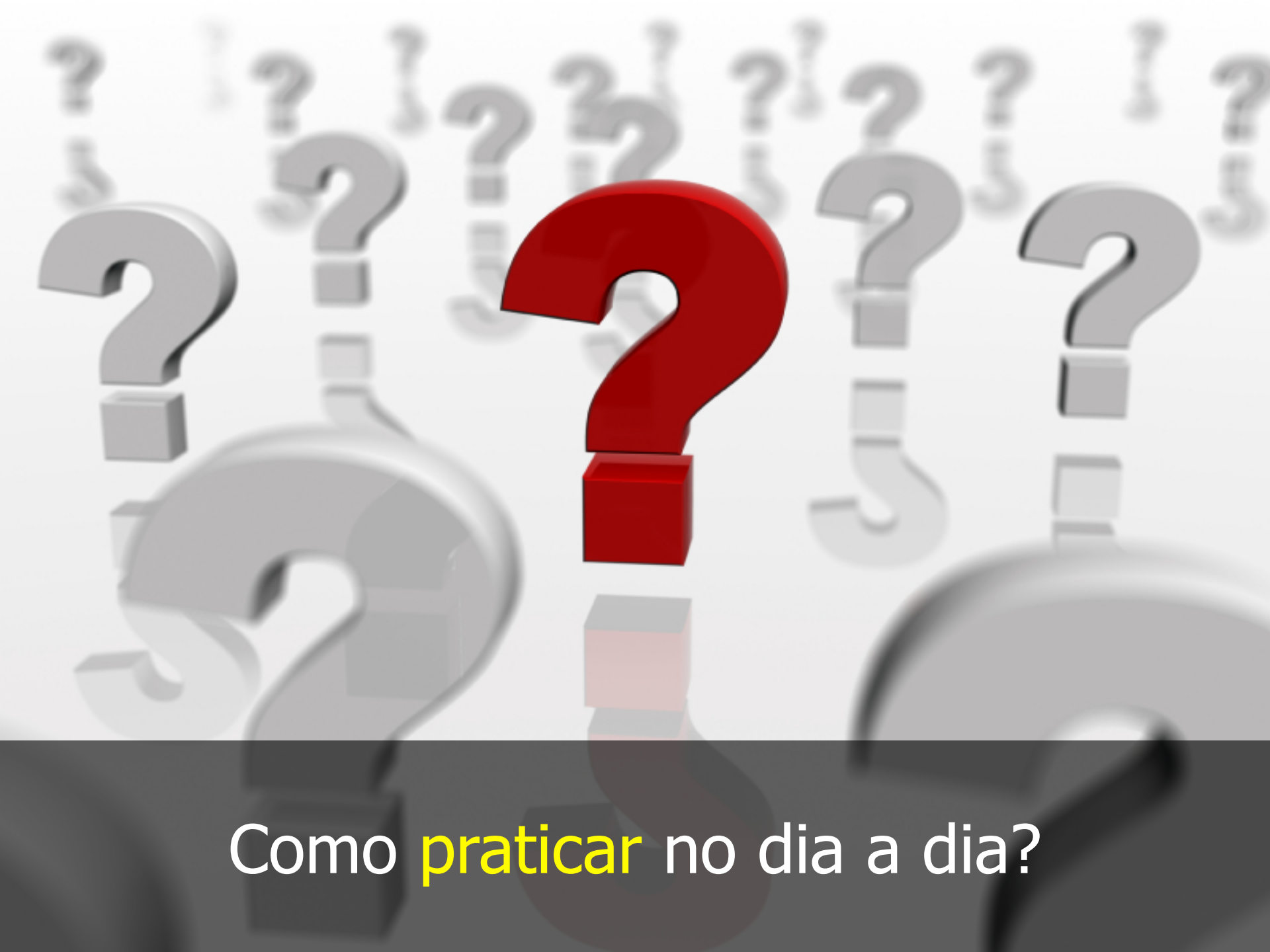
Tem tempo para lidar com código de baixa qualidade que não foi refatorado

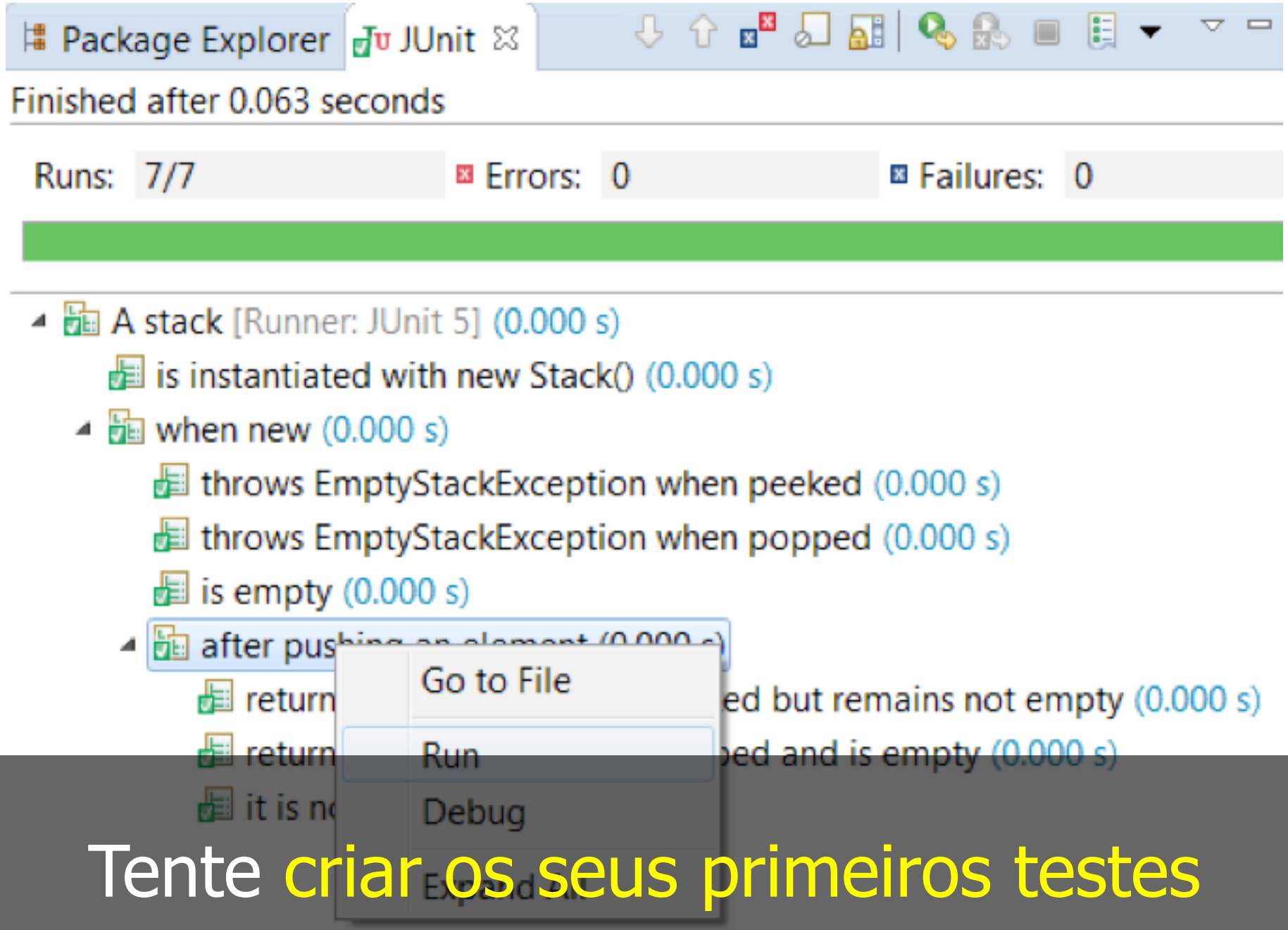
Tem tempo para entender um código sem qualquer tipo de documentação

Tem tempo para testar manualmente

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

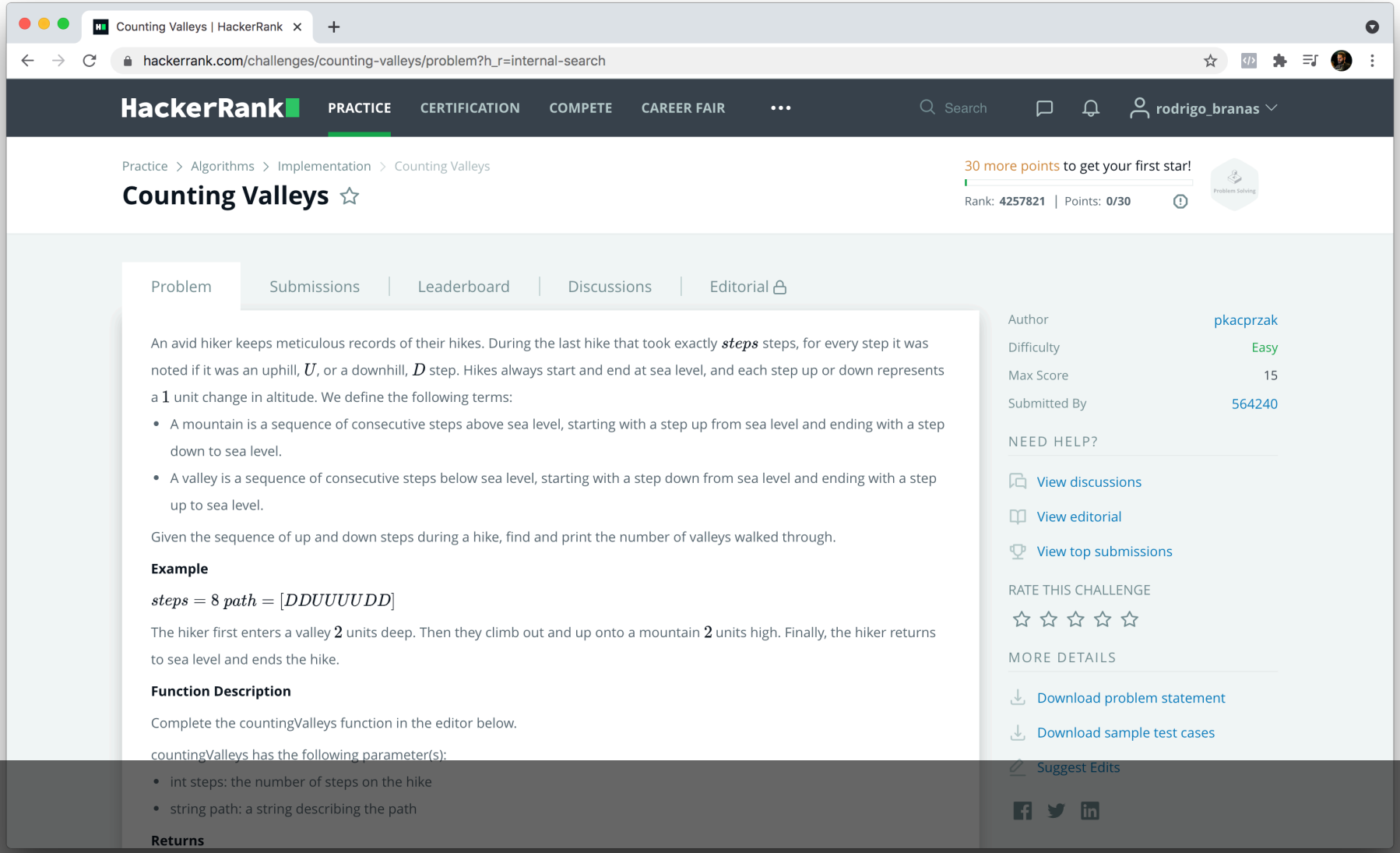
Tem tempo para ficar reclamando do projeto


Como praticar no dia a dia?



Tente criar os seus primeiros testes


Promova eventos tipo um Coding Dojo



Pratique online por meio de desafios

Plataformas como a Hackerrank oferecem desafios e testes de uma forma interativa e são utilizados por muitas empresas na hora de contratar um novo profissional