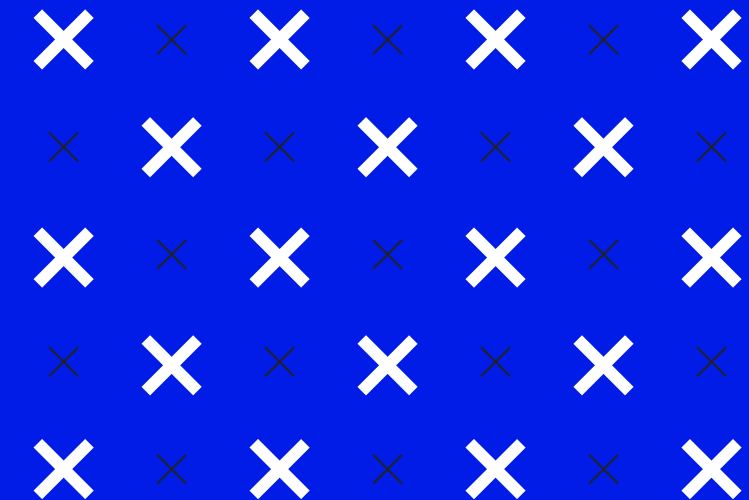


## MÓDULO 05

# Java Avançado

## Testes Automatizados



**mentorama.**

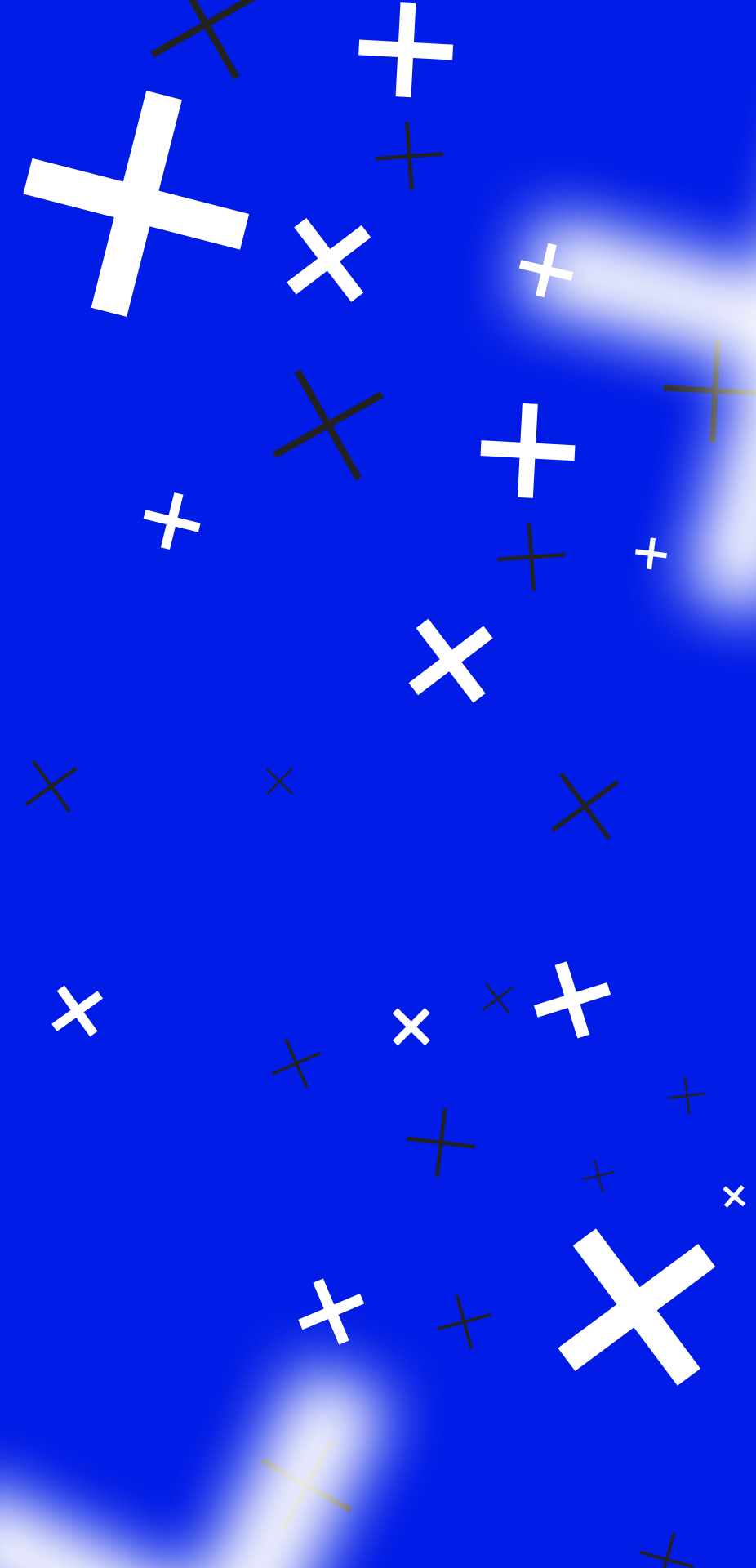
**mentorama.**

# Apresentação

Testes Automatizados

**mentorama.**

**mentorama.**

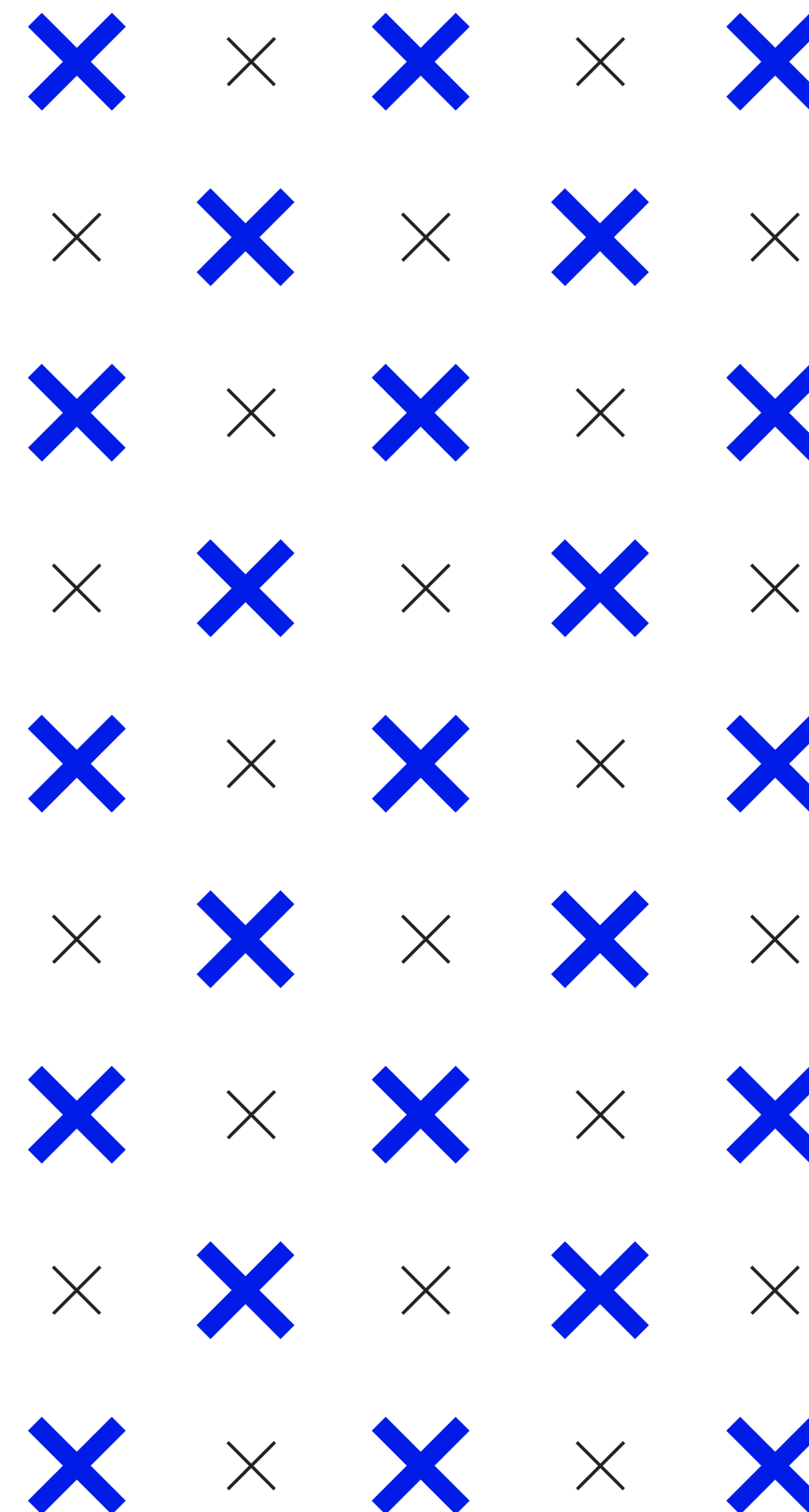


# O que veremos?

- ◆ Testes Unitários
- ◆ Testes de Integração
- ◆ Test Driven Development
- ◆ JUnit 5
- ◆ Mocks
- ◆ Métricas

**mentorama.**

**mentorama.**

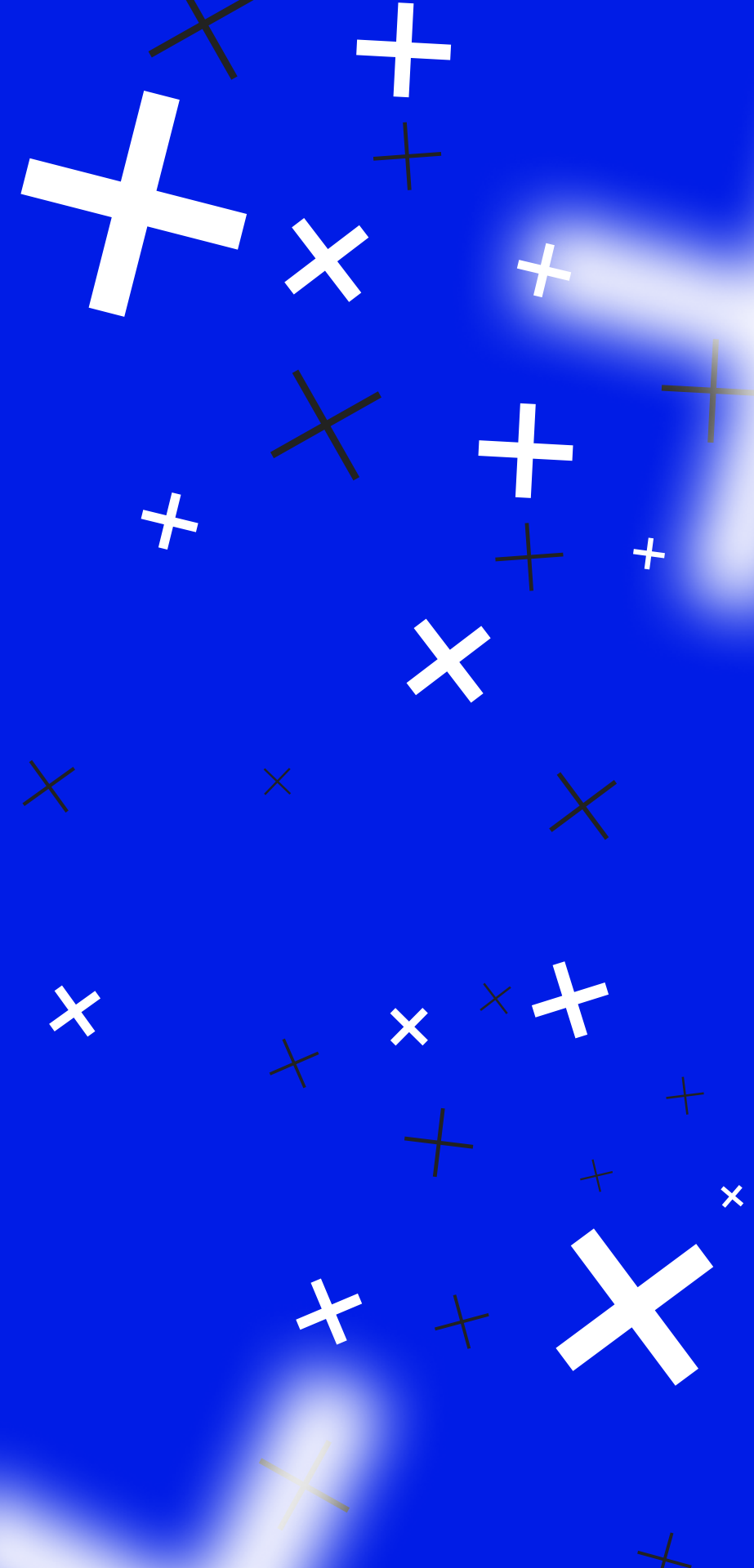


# Testes Automatizados

Testes Automatizados

**mentorama.**

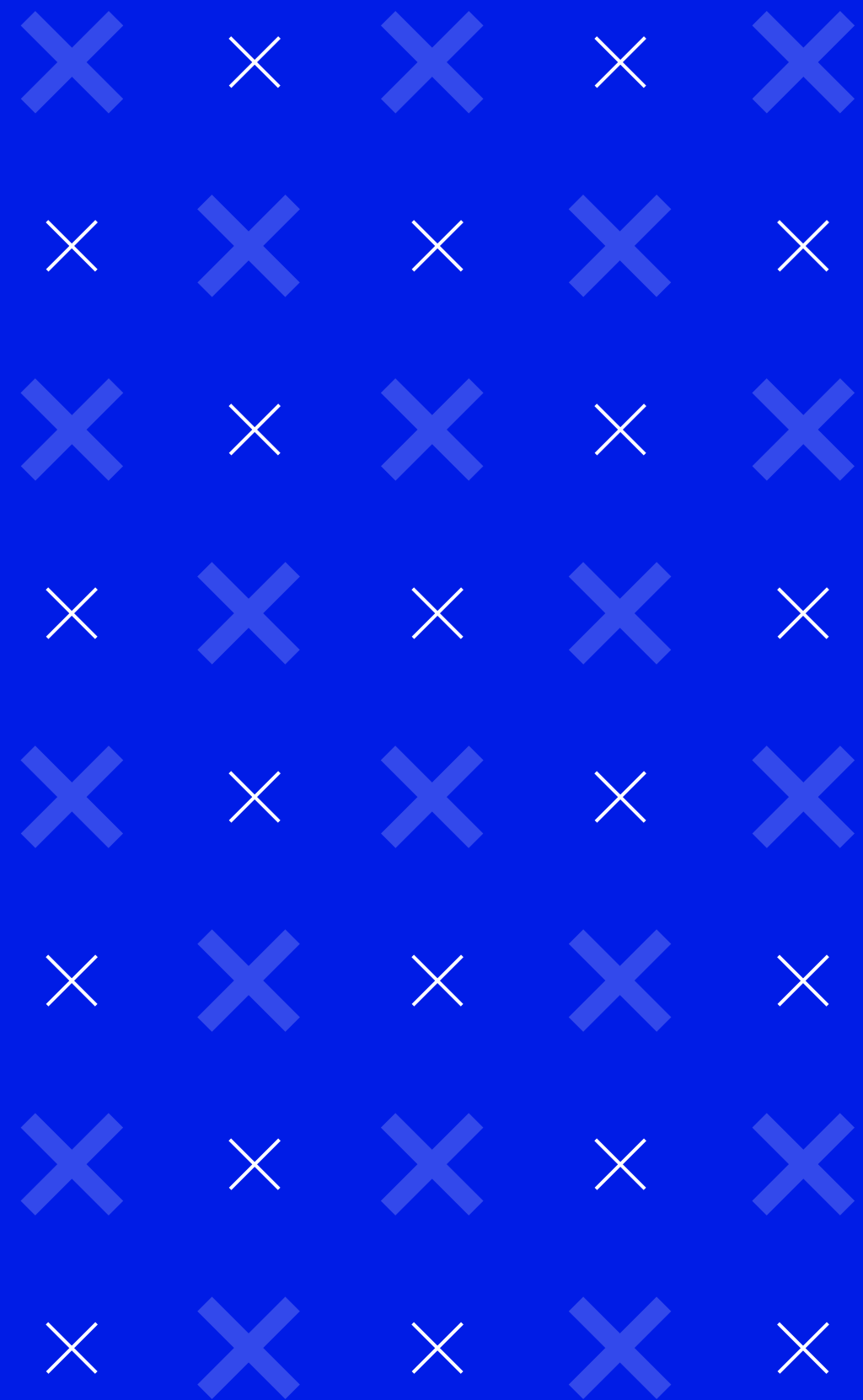
**mentorama.**



# Como testamos nossa aplicação até agora?

## ◆ Testes totalmente manuais

- ◆ Levantamos o servidor
- ◆ Verificamos se as funcionalidades estão funcionando conforme o esperado



# Qual o problema nisso?

- ◆ Quando fizemos uma refatoração, o correto seria testar se não mudamos o comportamento da aplicação a cada pequeno passo do processo
- ◆ Com testes manuais isso se torna bastante custoso e demorado
- ◆ Temos pouca segurança para executar refatorações ou corrigir bugs na aplicação



**mentorama.**

**mentorama.**

# Testes automatizados

- ◆ Código escrito separadamente do principal com o objetivo de executar o código principal e validar seus resultados
- ◆ Ou seja, você vai escrever código para testar se o código que você escreveu está funcionando conforme deveria

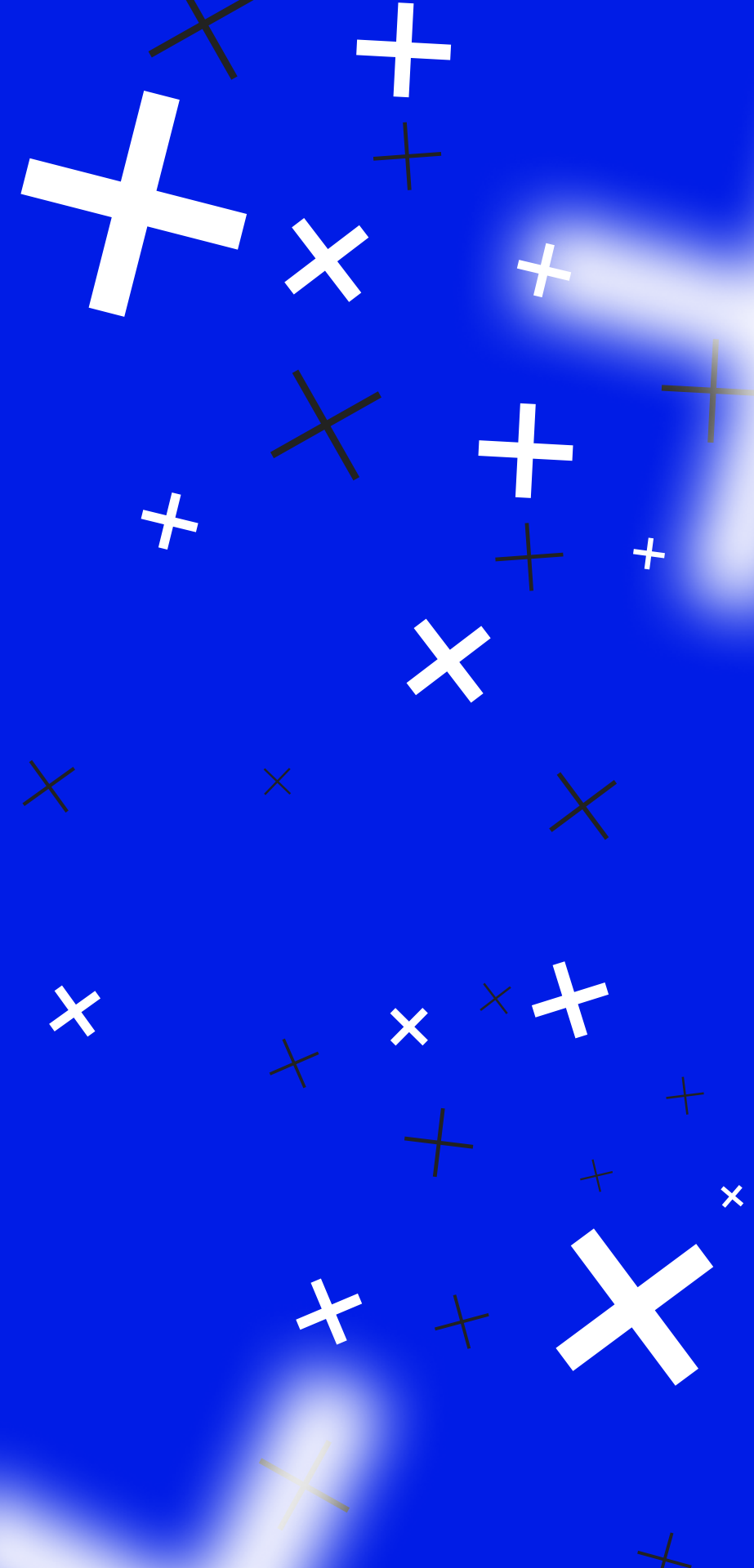
**mentorama.**

**mentorama.**

# Por que os testes são importantes?

mentorama.

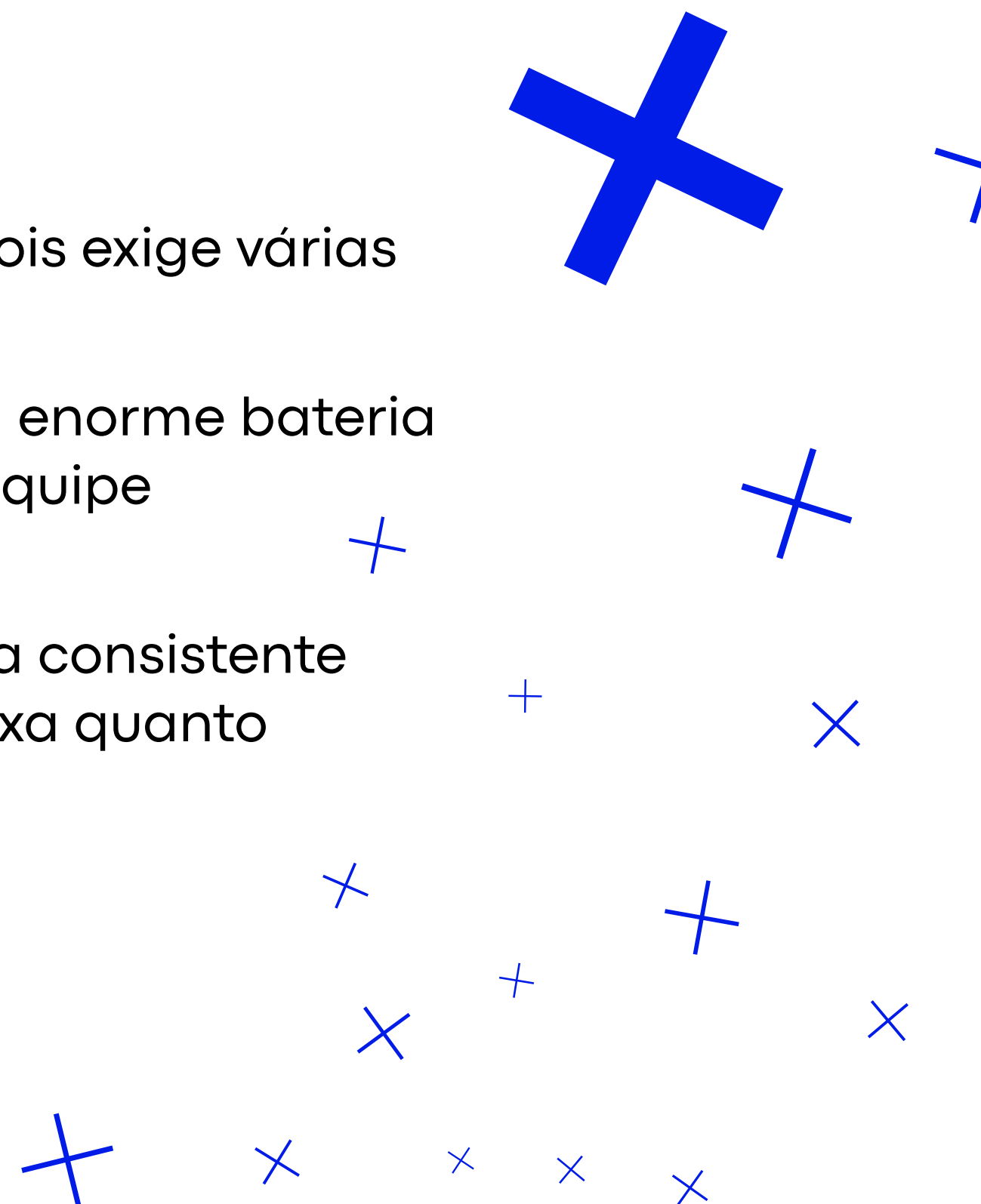
mentorama.





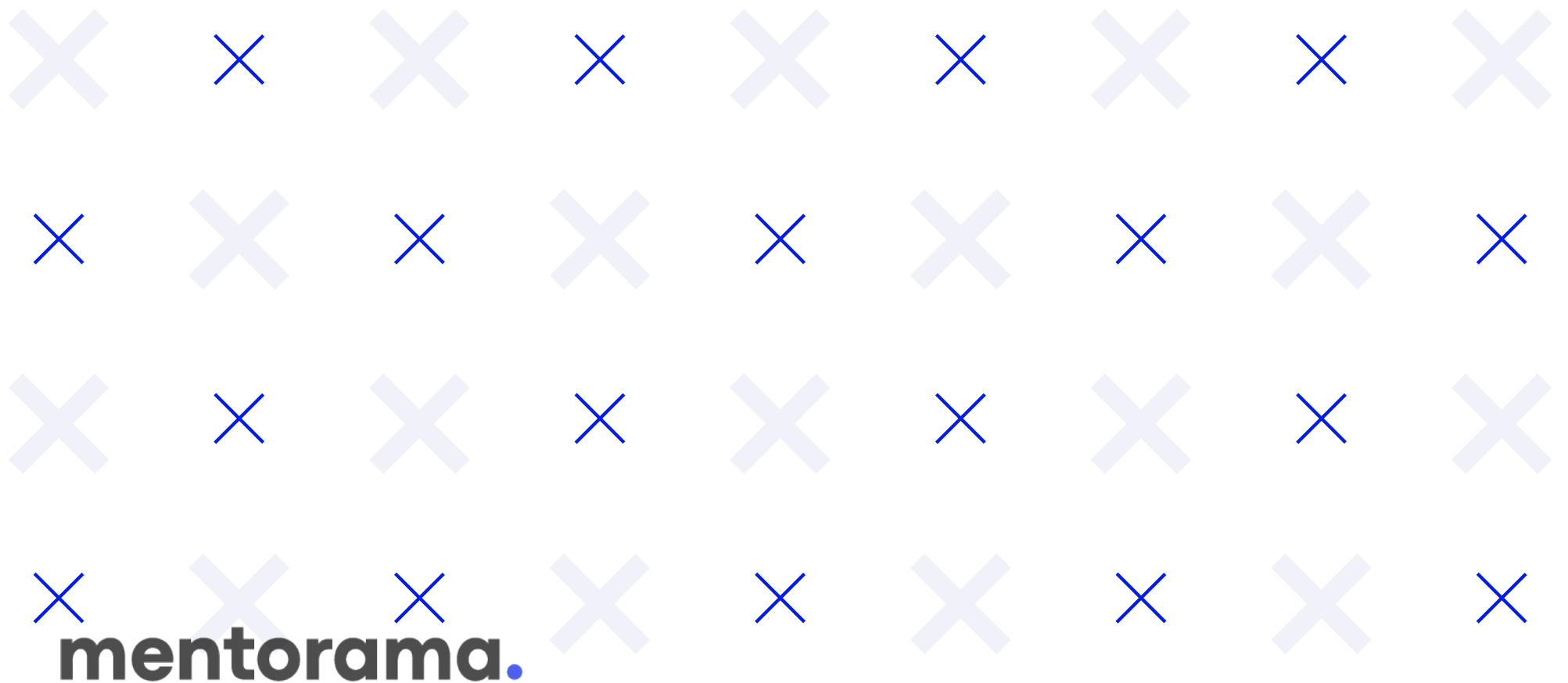
# Custo

- ◆ A execução manual de testes gera um alto custo pois exige várias horas de trabalho de stakeholders
- ◆ Imagine a cada nova release ter que executar uma enorme bateria de testes manuais, muitas vezes envolvendo uma equipe inteira no processo
- ◆ Esse processo pode ser substituído por uma bateria consistente de testes automatizados, que pode ser tão complexa quanto a aplicação precise



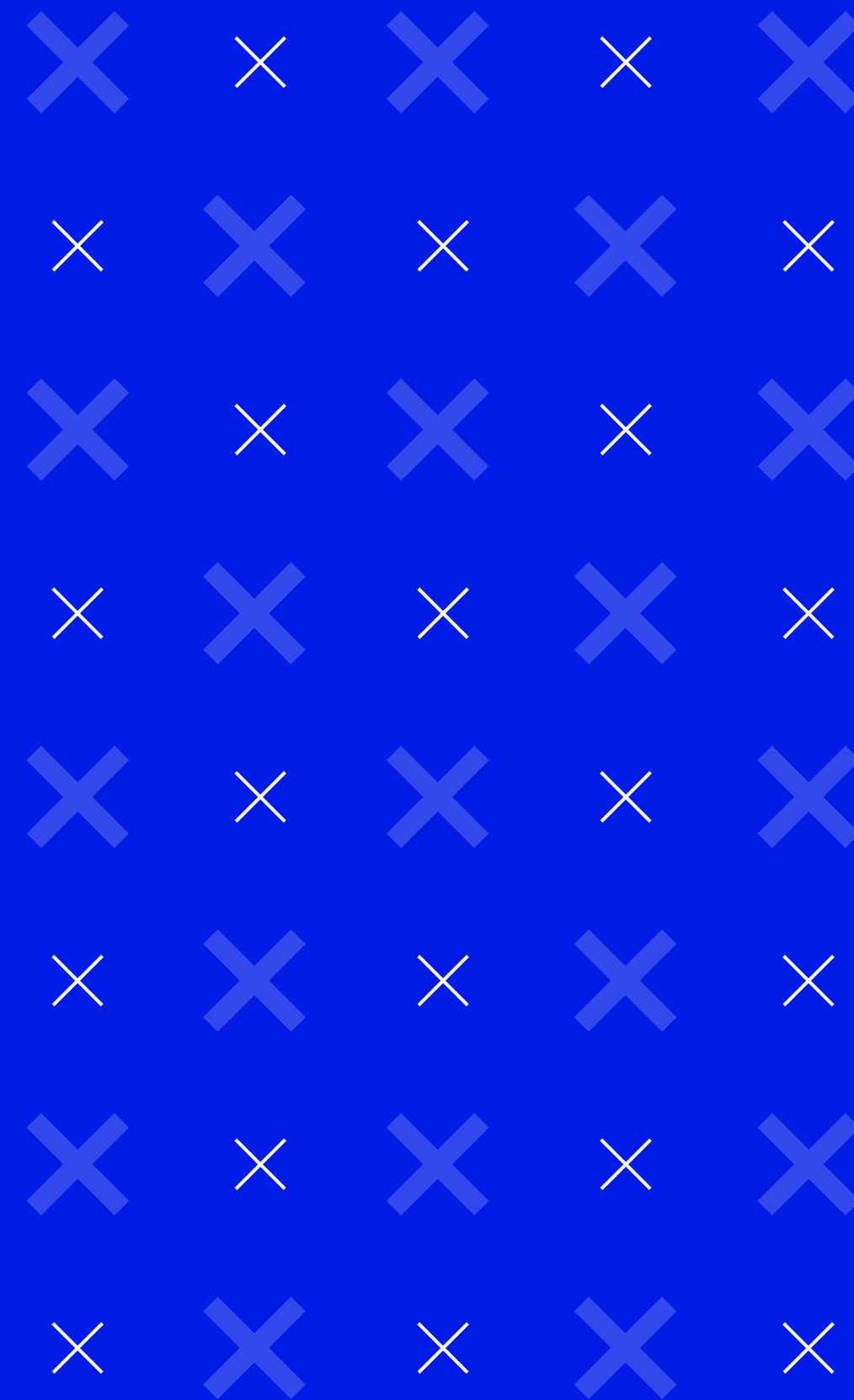
# Antecipação de bugs

- ◆ Bugs encontrados com antecedência custam muito menos para serem corrigidos
- ◆ Podemos executar uma bateria de testes que levam poucos segundos a cada pequena mudança na nossa aplicação para evitar que bugs avancem para produção



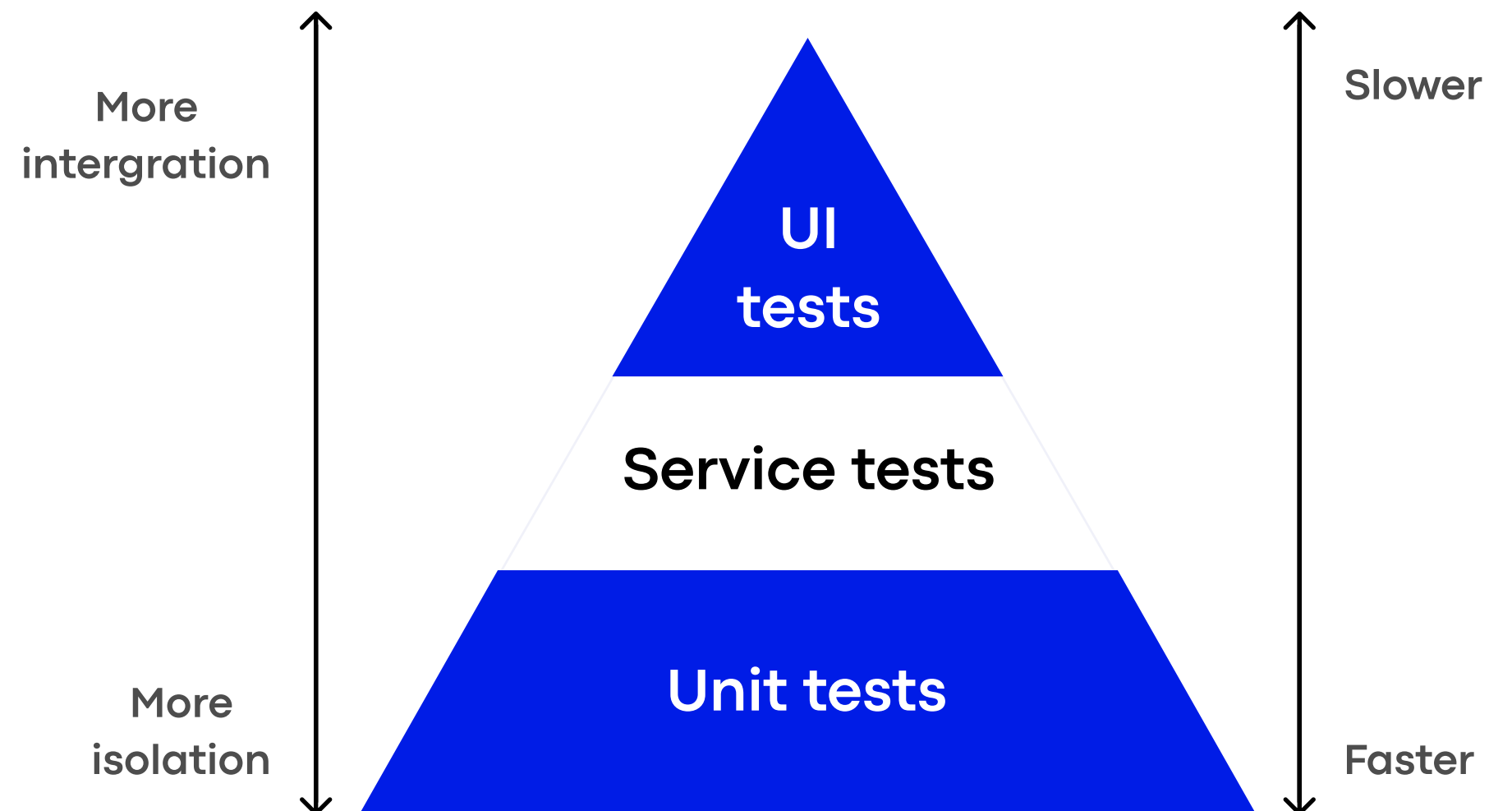
# Tempo de feedback

- ◆ Refatorações são uma realidade em nossos sistemas, ter um feedback rápido sobre o comportamento geral da aplicação após cada pequena mudança é essencial para executar esse processo de maneira segura e consistente



# Principais tipos de testes automatizados

- ◆ Testes de Unidade
- ◆ Testes de Integração
- ◆ Testes de UI

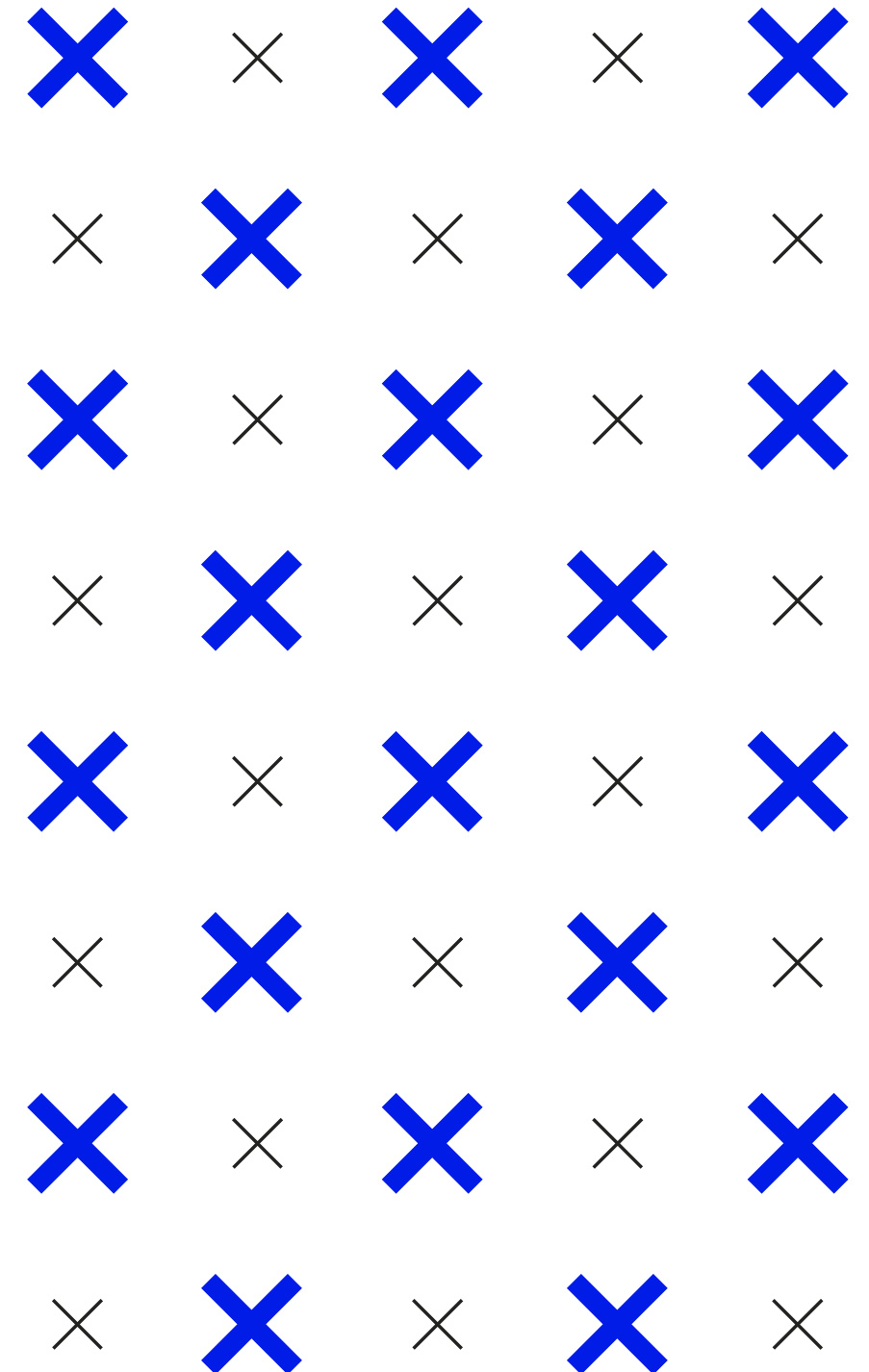


# E a produtividade?

- ◆ É comum que haja o questionamento de se haverá queda de produtividade no processo de desenvolvimento uma vez que será necessário desenvolver mais código
- ◆ Apesar da necessidade de escrever mais código do que se não houvessem testes, esse tempo é facilmente compensado com os benefícios de:
  - ◇ Não ter que executar as suítes manualmente
  - ◇ Menor chance de geração de bugs
  - ◇ Segurança na refatoração da aplicação

**mentorama.**

**mentorama.**

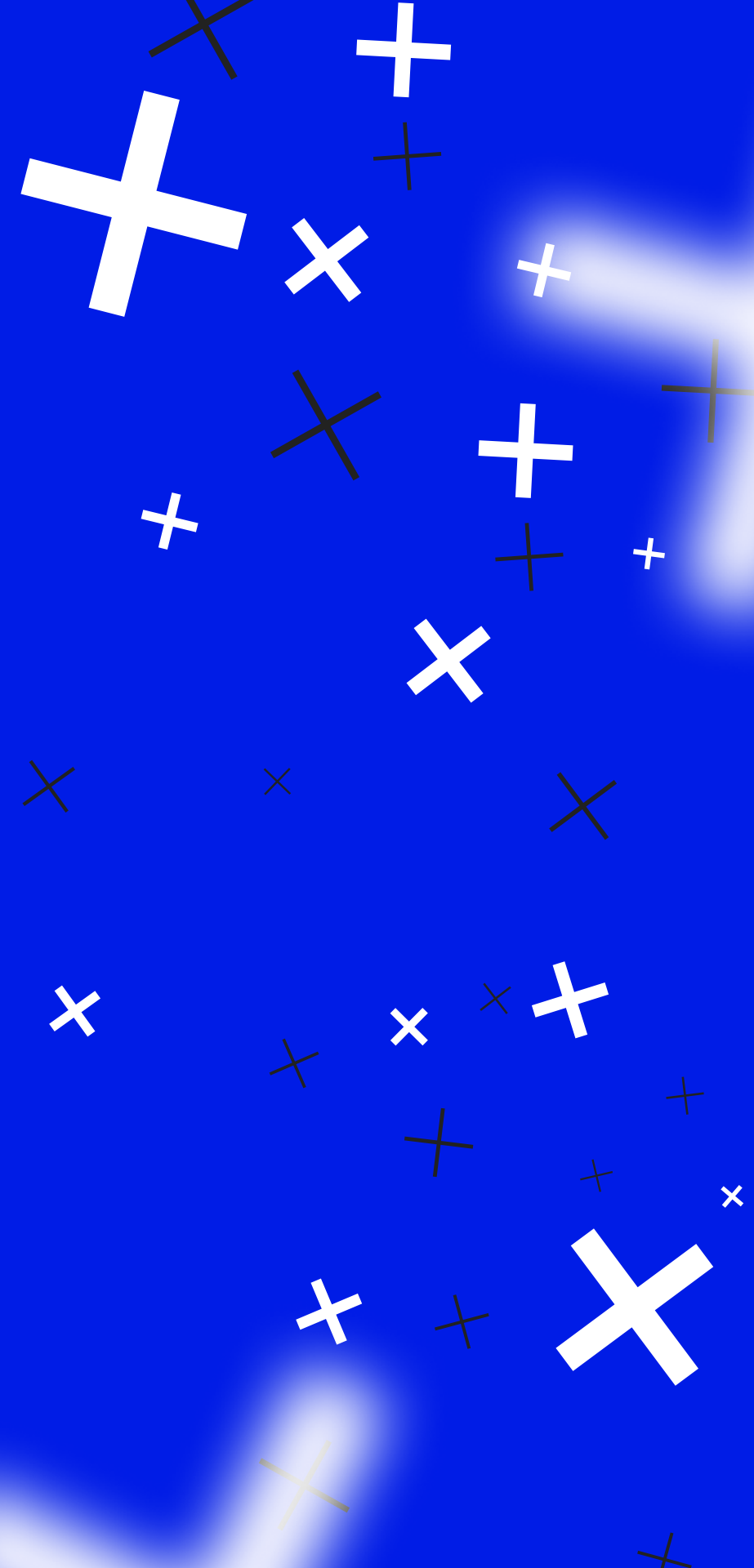


# Testes Unitários com JUnit

Testes Automatizados

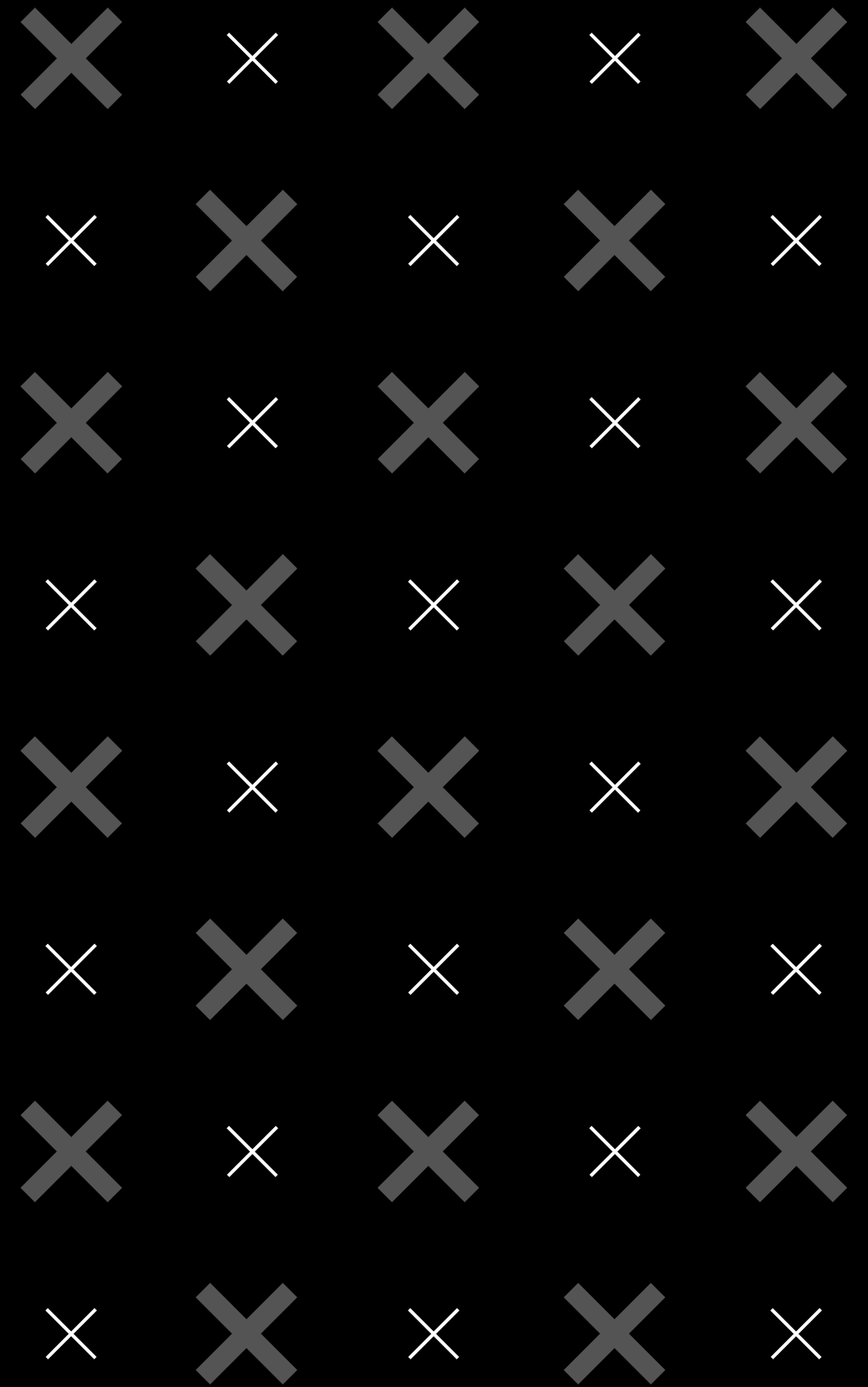
mentorama.

mentorama.



# Testes Unitários

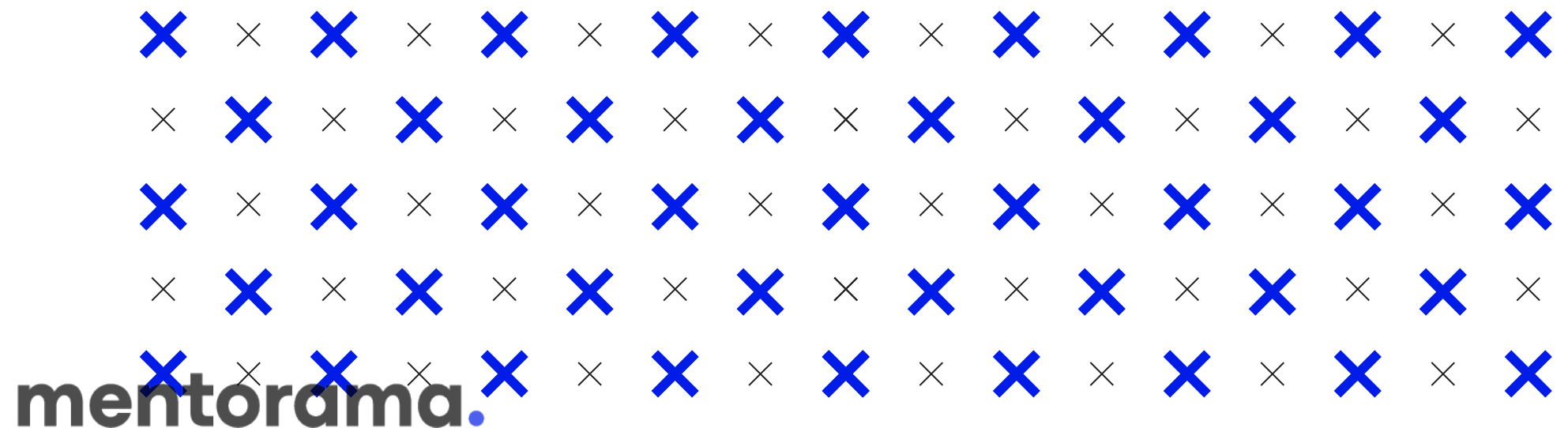
- ◆ Bem, como vimos na aula passada, testes Unitários consistem basicamente em testes que validam uma unidade de código de maneira isolada das demais partes do sistema
- ◆ São testes que devem ser isolados, simples e de rápida execução
- ◆ Tem a capacidade de nos fornecer um feedback rápido de onde pode estar o problema em nosso código



# JUnit

- ◆ JUnit é um framework que facilita o desenvolvimento e execução de testes automatizados em código Java, essa ferramenta foi criada pelo Erich Gamma e Kent Beck
- ◆ Podemos dizer que é a ferramenta padrão para execução de testes em Java da atualidade
- ◆ Apesar do nome, o JUnit não nos ajuda apenas com testes unitários, eles nos permite desenvolver vários outros tipos de testes

**mentorama.**





# JUnit 5

**mentorama.**

◆ JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

◆ JUnit Platform

◇ Serve como uma base para execução de frameworks de testes na JVM

◆ JUnit Jupiter

◇ Motor para rodar testes no padrão Jupiter

◆ JUnit Vintage

◇ Motor para rodar testes de JUnit 3 e 4

◆ JDKs suportadas

◇ O JUnit 5 funciona em versões a partir da 8 da JDK

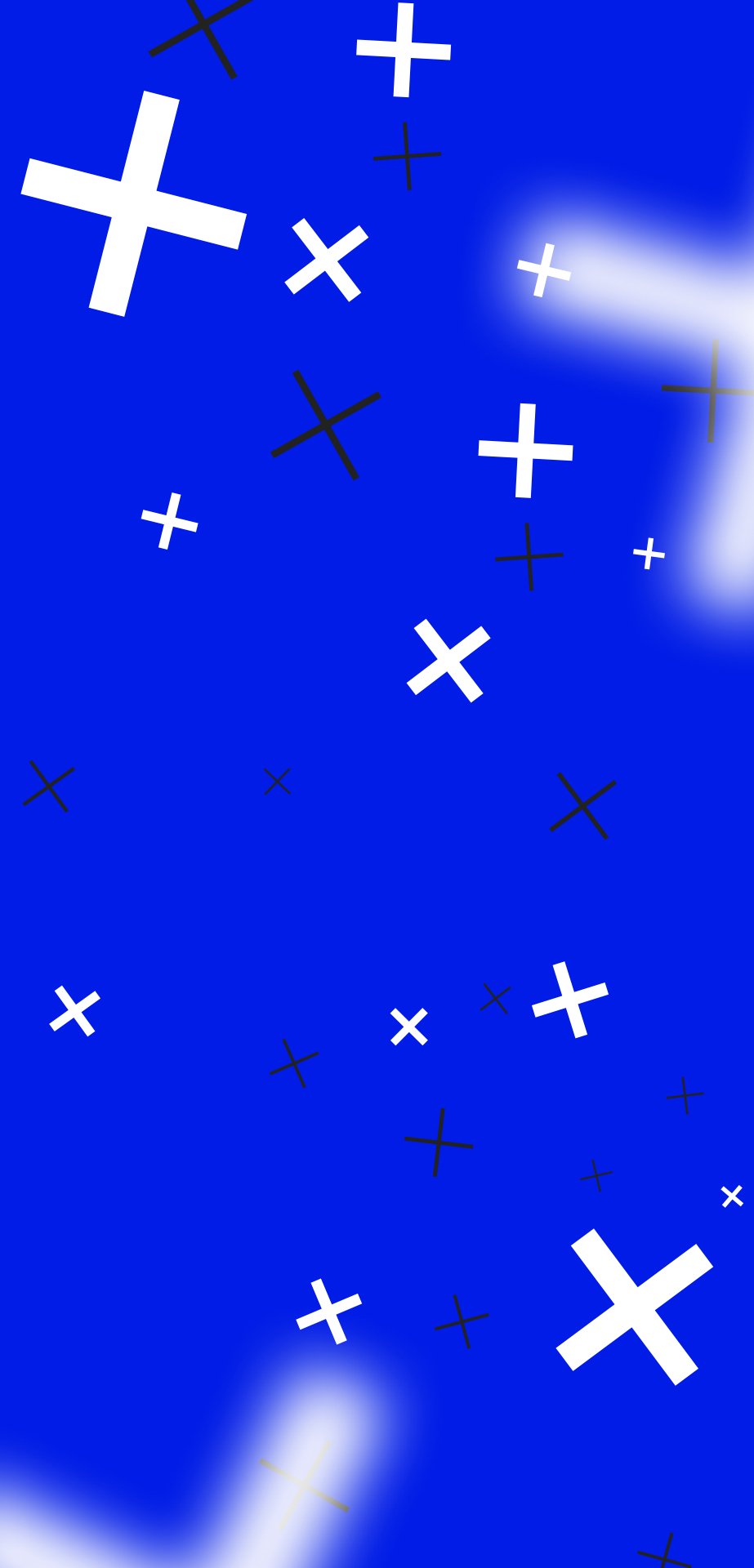
**mentorama.**



# Vamos desenvolver alguns testes?

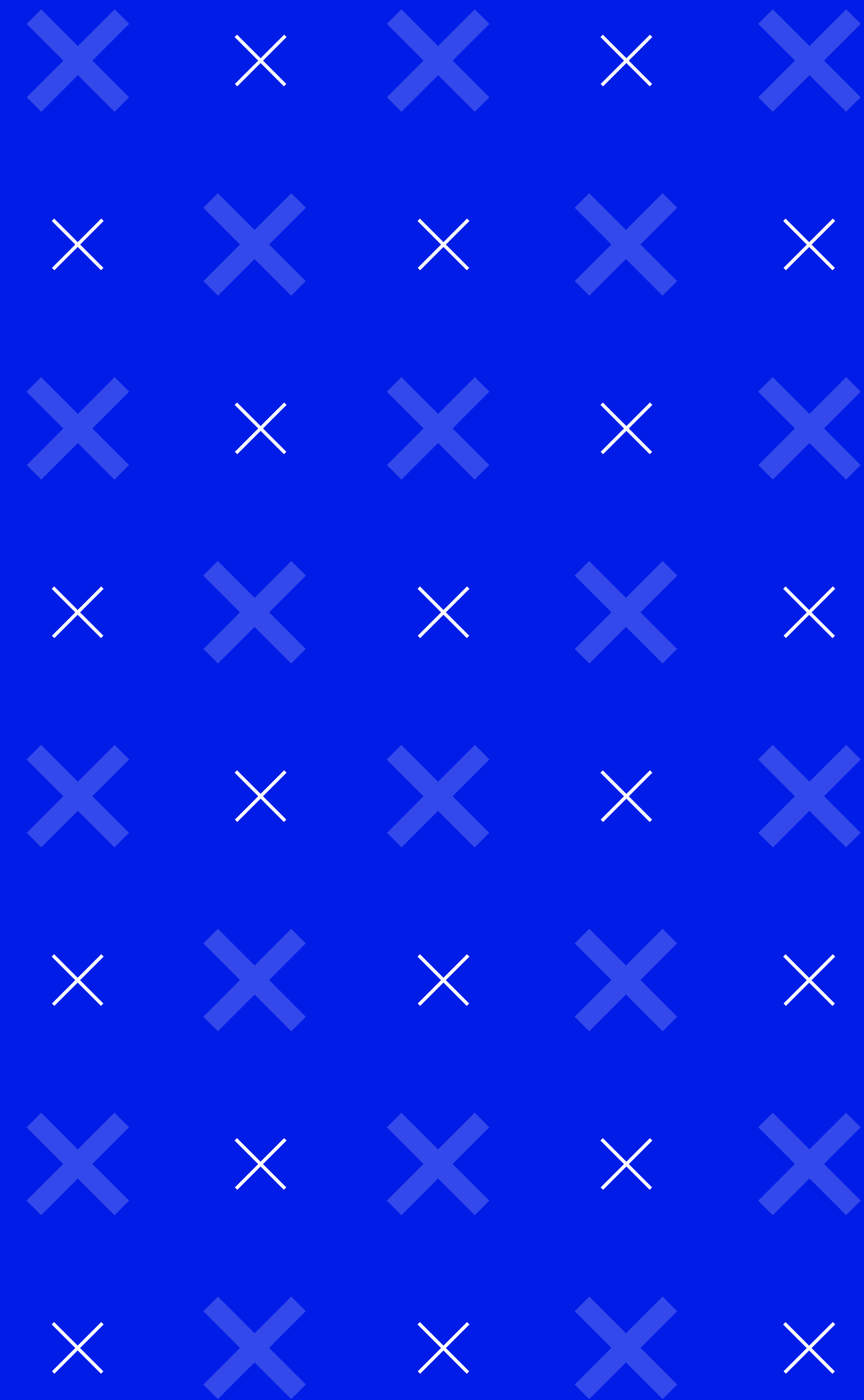
**mentorama.**

**mentorama.**



# Tópicos live code

- ◆ Escrevendo o nosso primeiro teste com Junit 5
- ◆ Annotations
- ◆ Assertions

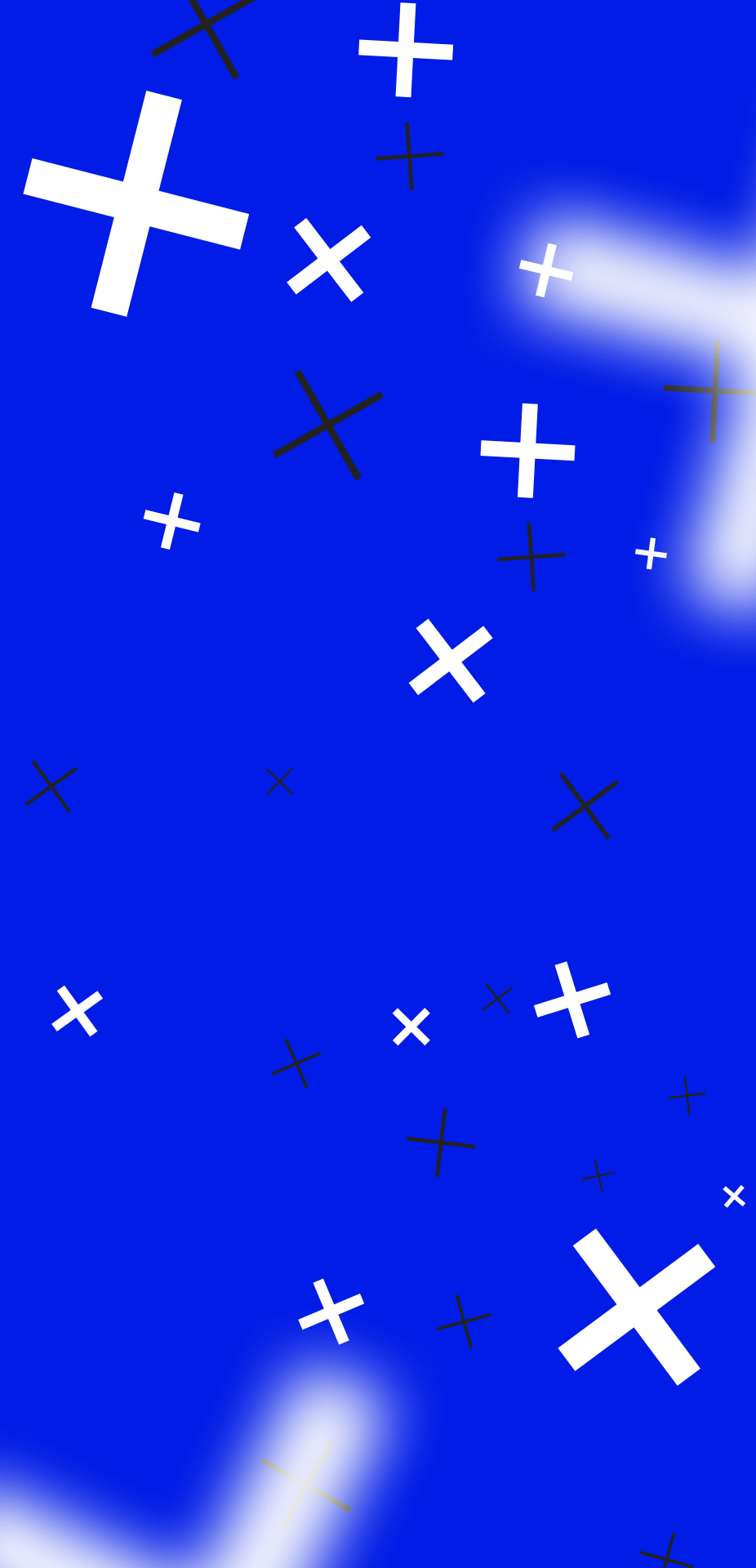


# Introdução ao TDD

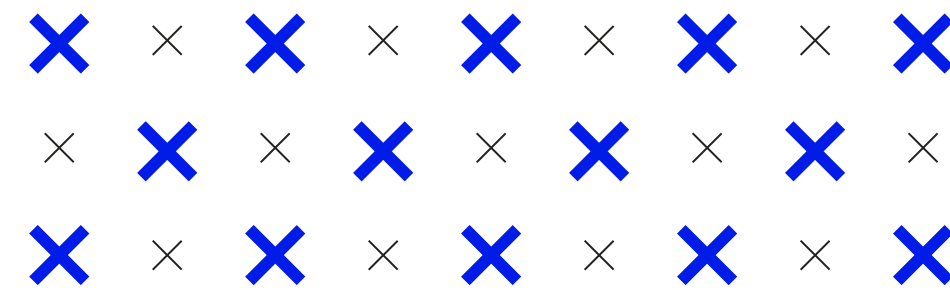
Testes Automatizados

**mentorama.**

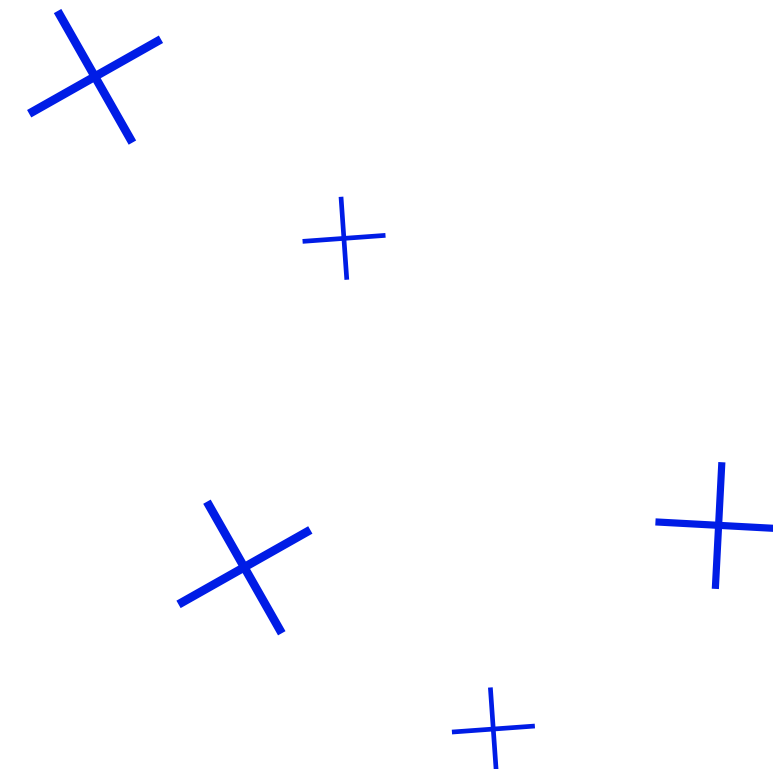
**mentorama**



# Test Driven Development

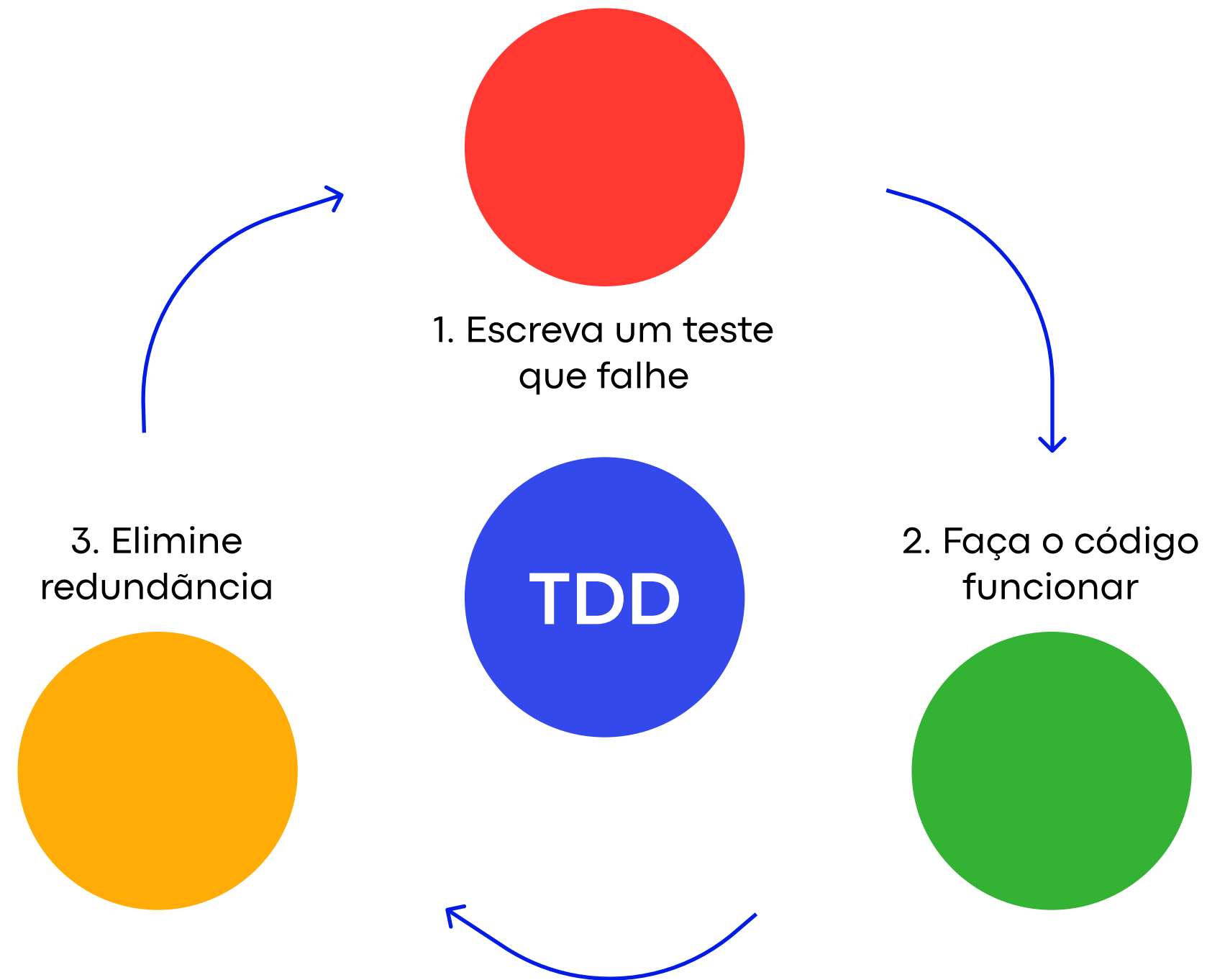


- ◆ Test-Driven Development (TDD) é uma técnica onde o desenvolvimento do software é guiado pela escrita de casos de teste, foi criada por Kent Beck no final dos anos 90.
- ◆ A essência do TDD está contida basicamente nos seguintes três passos
  - ◇ Escreva um teste para validar algum aspecto da próxima funcionalidade que você vai implementar
  - ◇ Escreva um código funcional apenas com o objetivo de fazer o teste passar
  - ◇ Refatore o código implementado para o deixar bem estruturado



# Ciclo do TDD

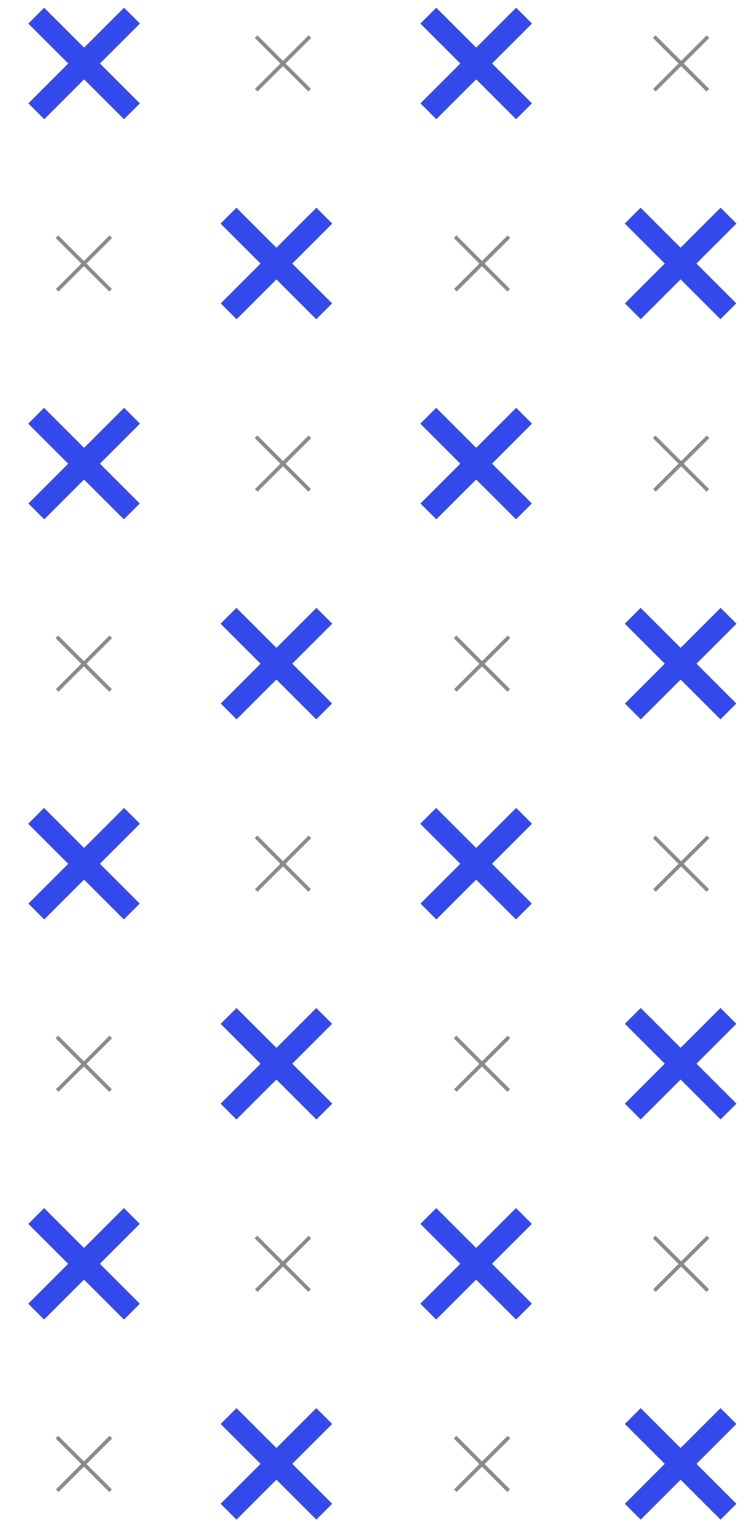
**mentorama.**



**mentorama.**

# Alguns benefícios

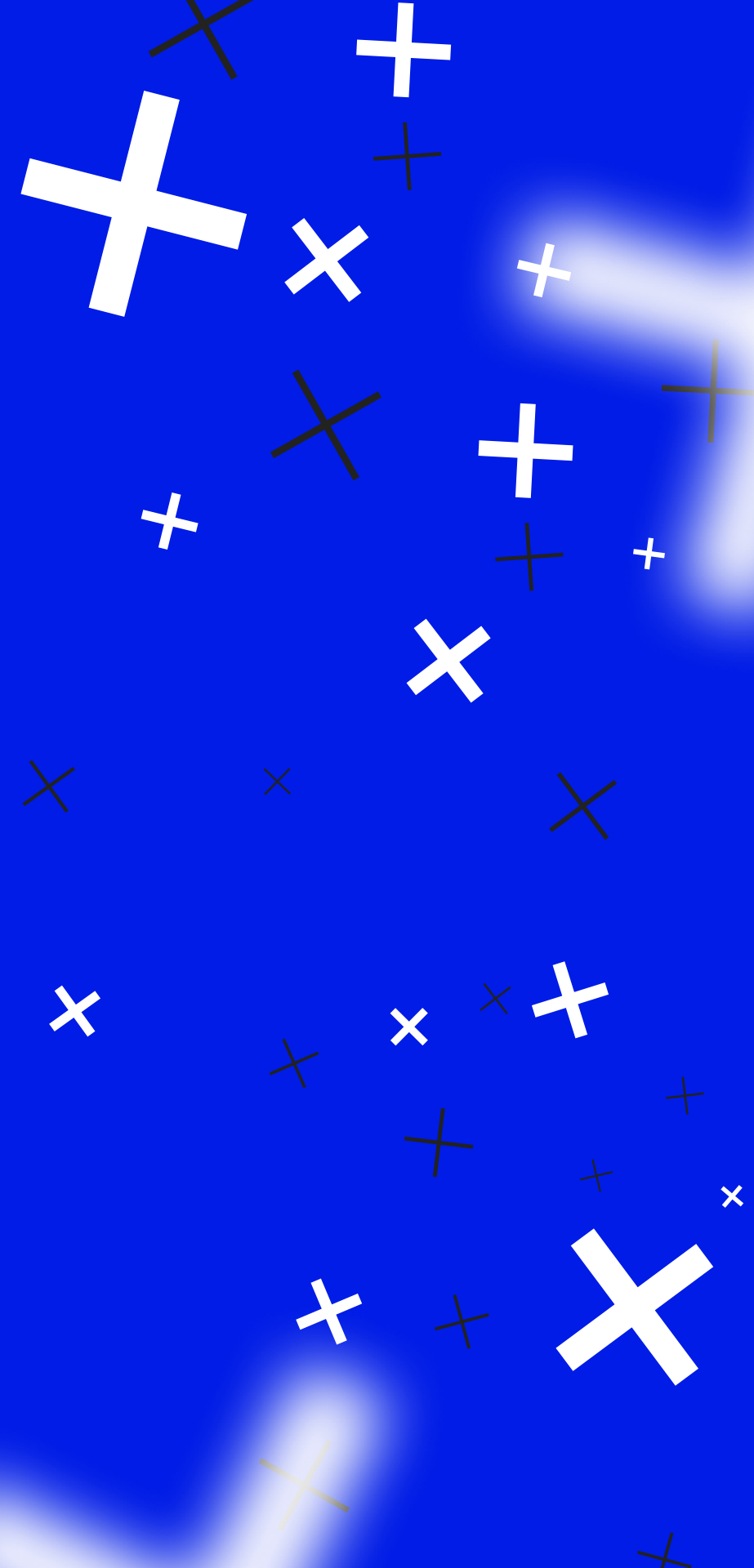
- ◆ De cara ganhamos todos os benefícios da escrita de testes automatizados, independente de utilizar ou não TDD
- ◆ Além disso:
  - ◇ Tendência natural de gerar um código fonte com mais coesão e menos acoplamento, uma vez que o código é pensado de maneira que seja fácil de ser testado
  - ◇ O teste se torna uma documentação viva do projeto, uma vez que cada caso de teste será uma espécie de “explicação” de como determinado método funciona
  - ◇ A Cultura do TDD ajuda os times a terem mais segurança e adotarem em seus processos individuais de desenvolvimento o hábito da refatoração continua



# Vamos codar?

mentorama.

mentorama.



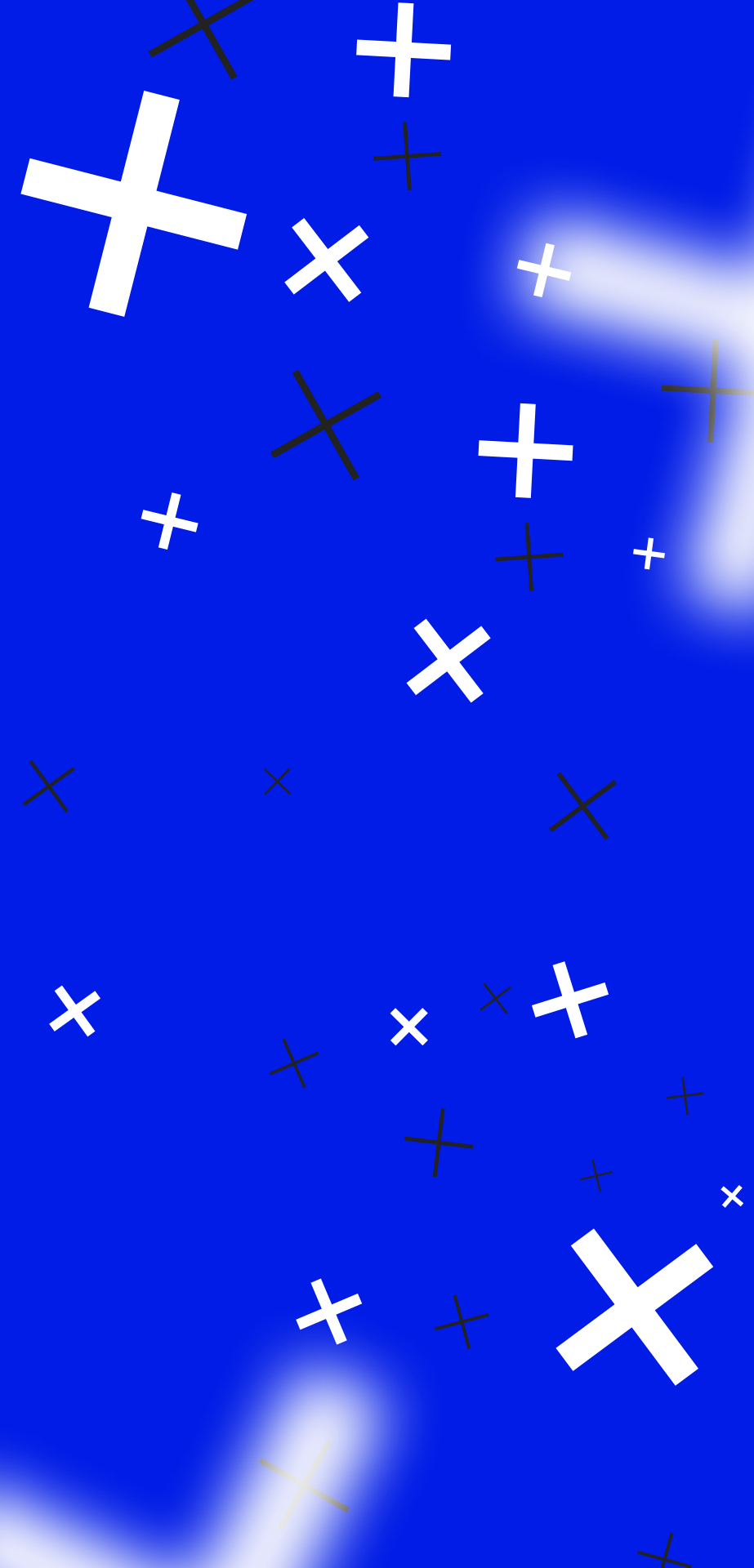


# Mockito

## Testes Automatizados

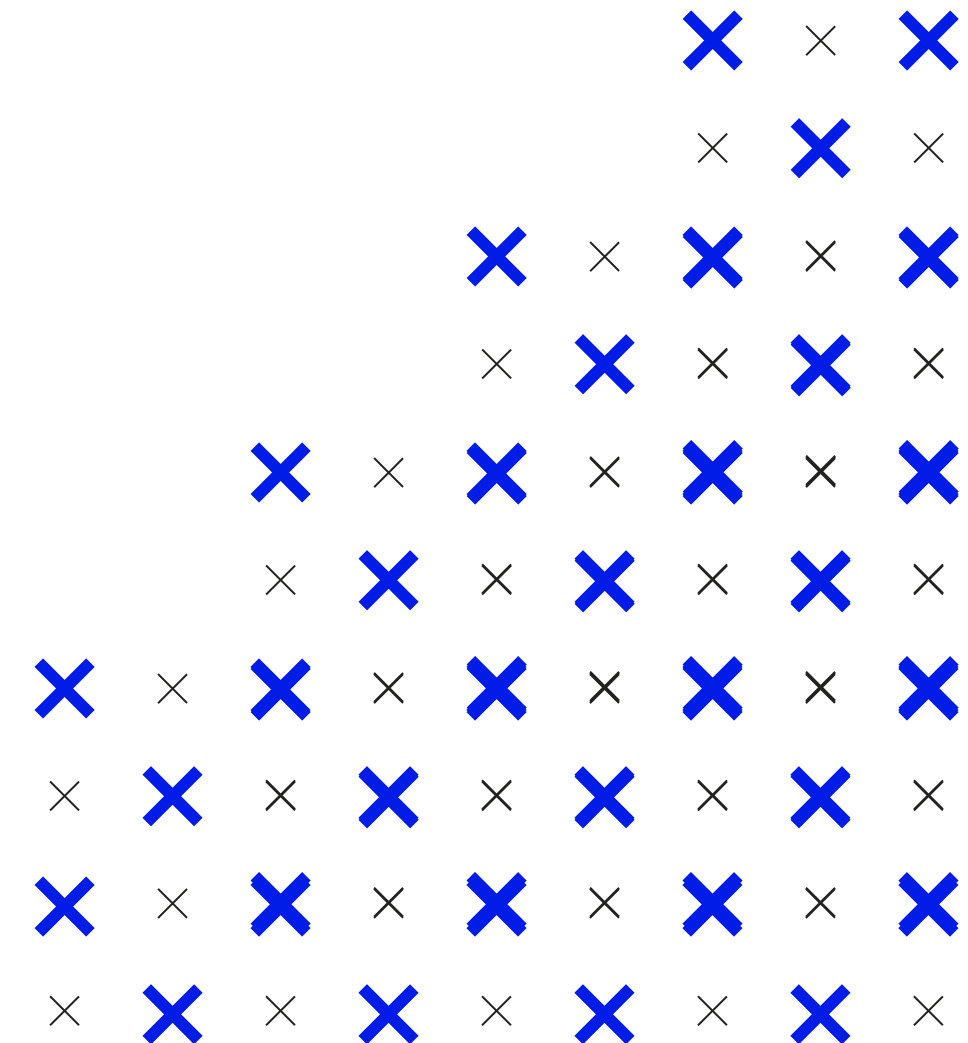
mentorama.

mentorama.



# Isolando teste unitários

- ◆ É natural que essa altura, você tenha tentado escrever um teste unitário para uma classe que possui dependências com outras classes e talvez você tenha percebido uma certa complexidade na criação desses objetos
- ◆ Um teste unitário, ou de unidade, deveria testar apenas uma unidade de código, normalmente apenas 1 método especificamente, em muitos casos realmente não podemos ou não vale a pena instanciar todas as dependências necessárias por uma classe para testar seu comportamento
- ◆ Para isso precisamos isolar a unidade que queremos testar, mas como fazer isso?

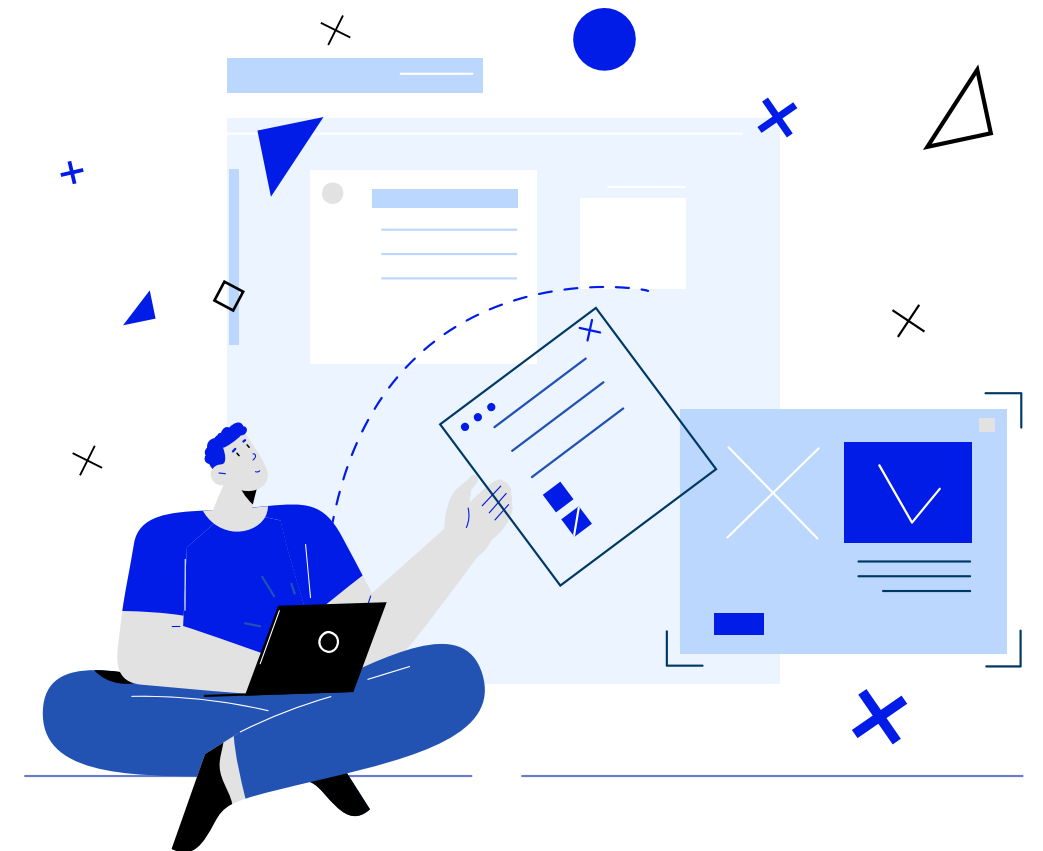
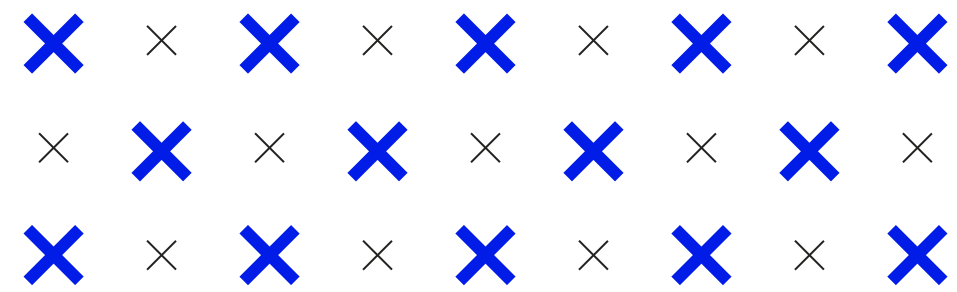


# Mock Objects

- ◆ Mock Objects ou simplesmente mocks são objetos que simulam o comportamento de objetos reais de forma controlada
- ◆ Com eles podemos simular e controlar todo o comportamento de objetos aos quais não queremos testar o comportamento em determinado teste

mentorama.

mentorama.



# Mockito

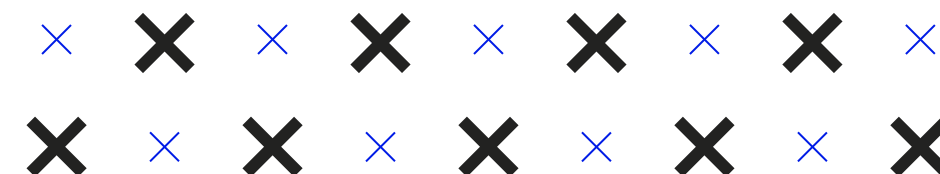
- ◆ Para nos ajudar nessa tarefa, temos um grande aliado do JUnit que é o Mockito
- ◆ O Mockito é basicamente um framework para criação de mocks que nos ajuda a mockar objetos de maneira simples e limpa
- ◆ Com ele podemos facilmente simular instâncias de objetos e controlar todo o comportamento de seus métodos



<https://site.mockito.org/>

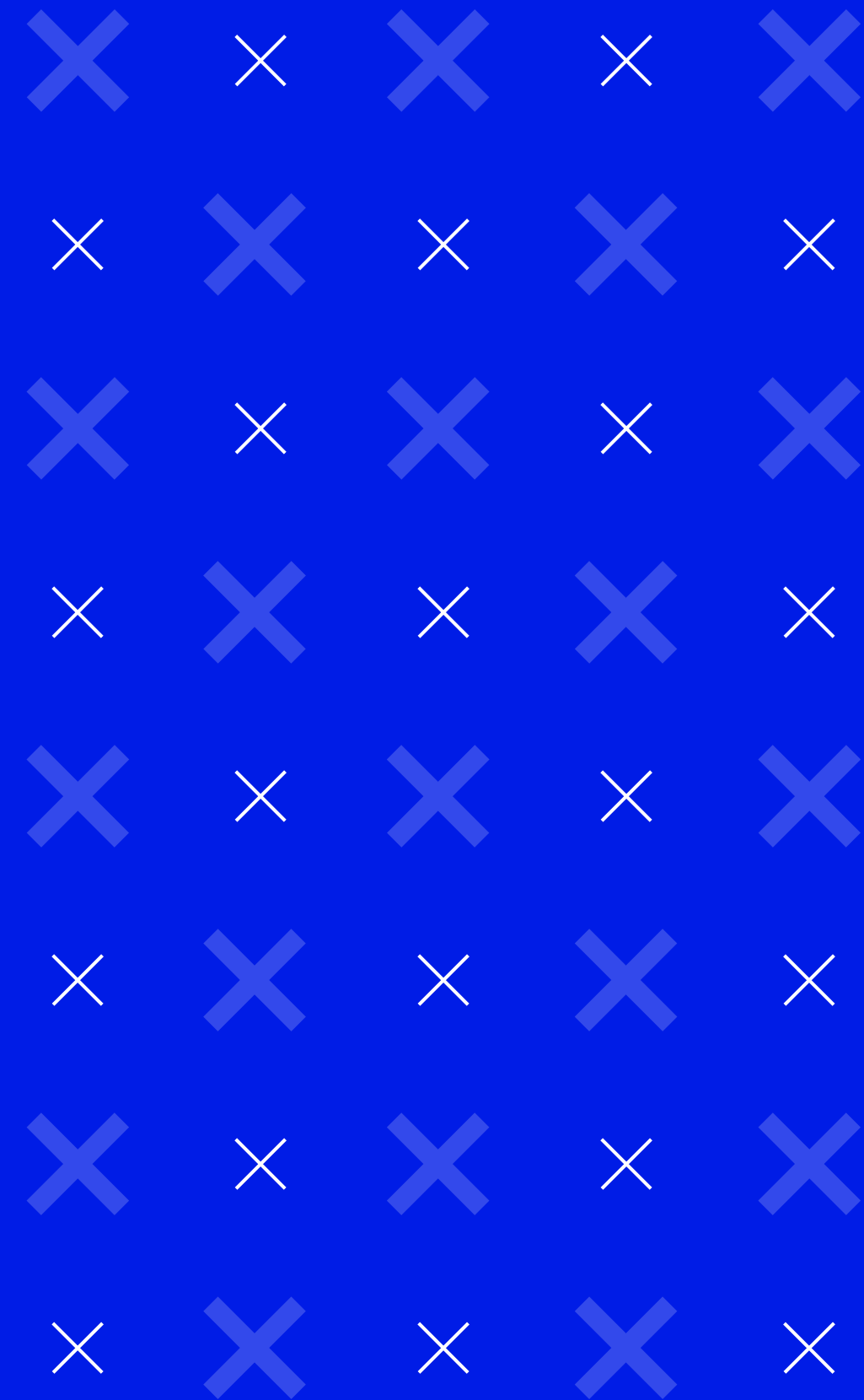
**mentorama.**

**mentorama.**



# Como utilizá-lo?

- ◆ Bem, vamos ver na prática como funciona o uso desse framework que facilita tanto a nossa vida ao escrever testes

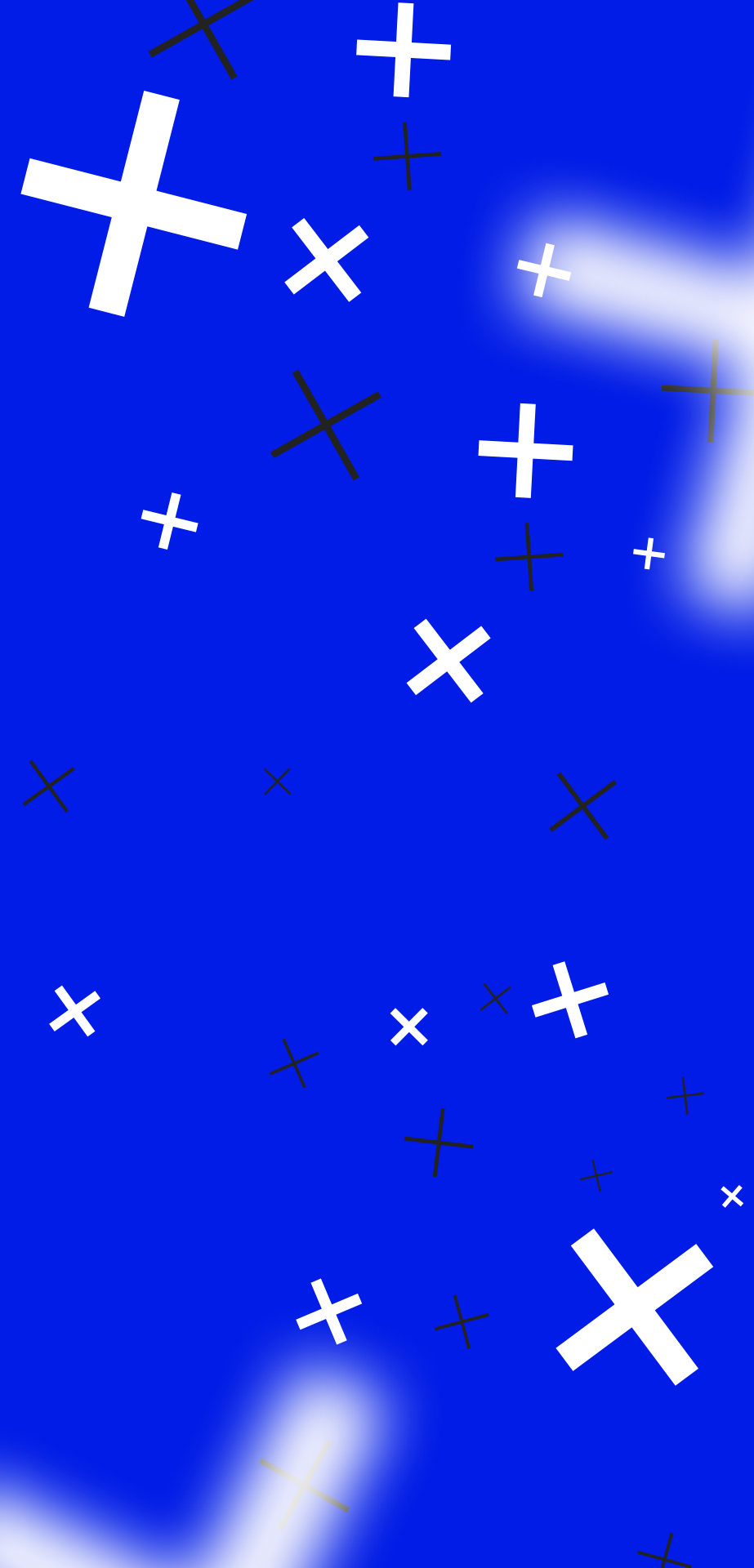


# Configurando o setUp dos nossos testes

Testes Automatizados

mentorama.

mentorama



# Setup dos testes

- ◆ Durante a codificação dos nossos testes, muitas vezes precisamos configurar todo o ambiente ao qual ele irá rodar
- ◆ Muitas vezes esse setup em cima do qual nossos testes irão rodar, se repete para vários testes diferentes, mesmo que unitários
- ◆ Logicamente que seria ótimo se pudéssemos ter uma maneira legal de aproveitar essas configurações não é mesmo?
- ◆ Então vamos ver no código como o JUnit pode nos ajudar nisso.

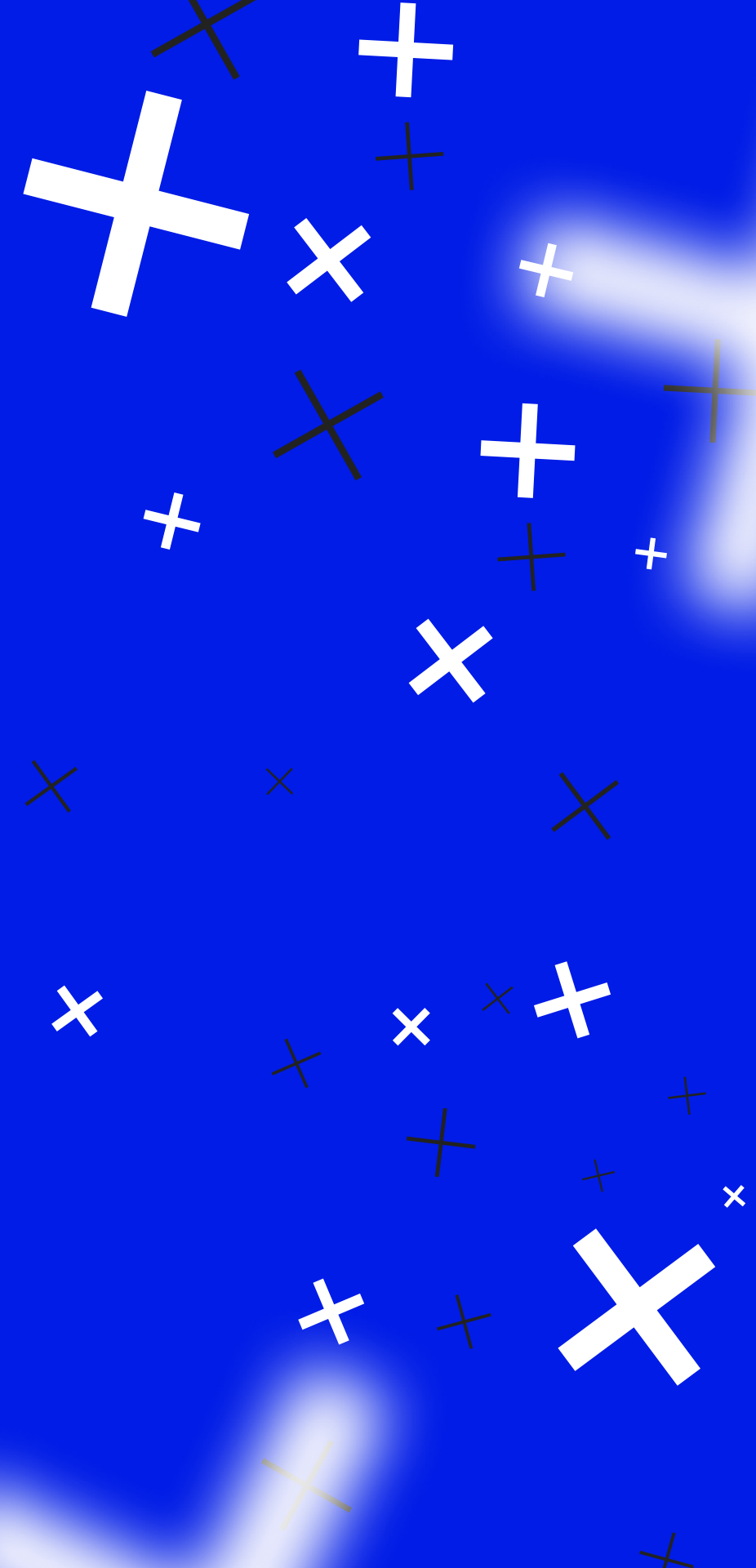


# Testes de Integração

Testes Automatizados

mentorama.

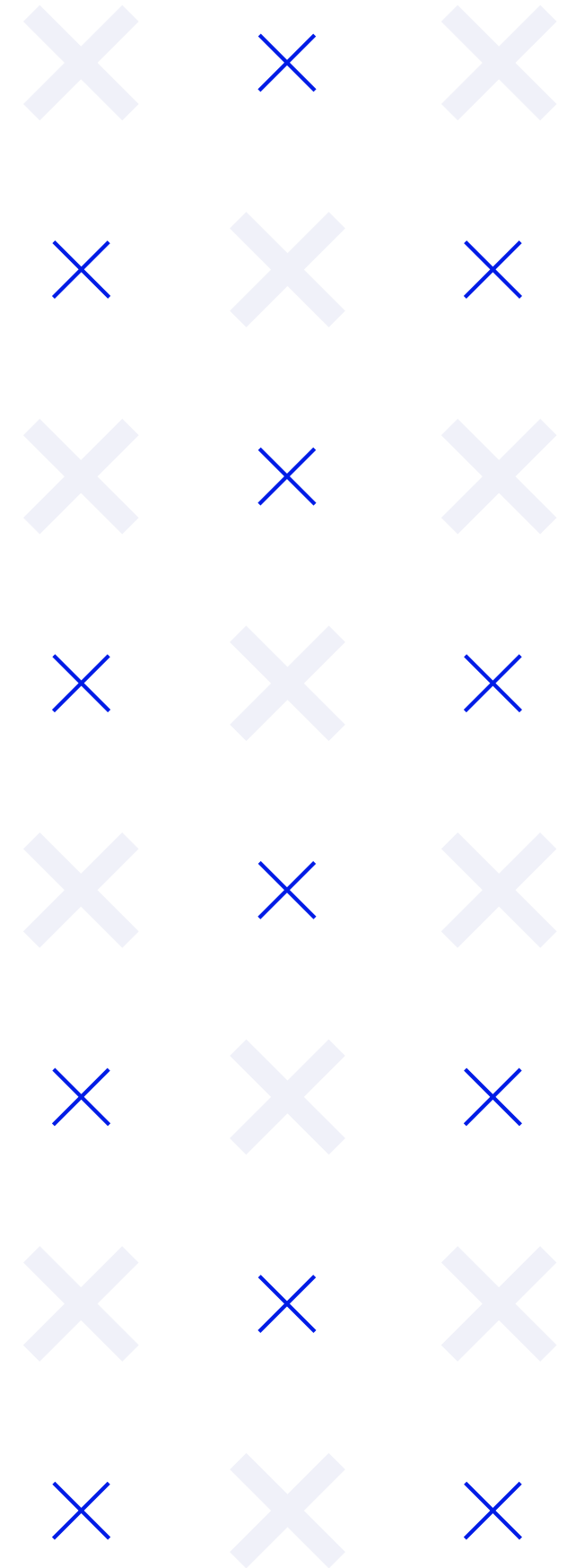
mentorama.





# Testes de Integração

- ◆ De maneira geral, testes de integração são aqueles que tem como objetivo testar a integração entre os componentes de sua aplicação, sejam internos ou externos
- ◆ **Exemplos**
  - ◇ Teste de comunicação entre classes
  - ◇ Integração com Banco de Dados
  - ◇ Integração com APIs Externas
  - ◇ Integração com Filas de Mensagens



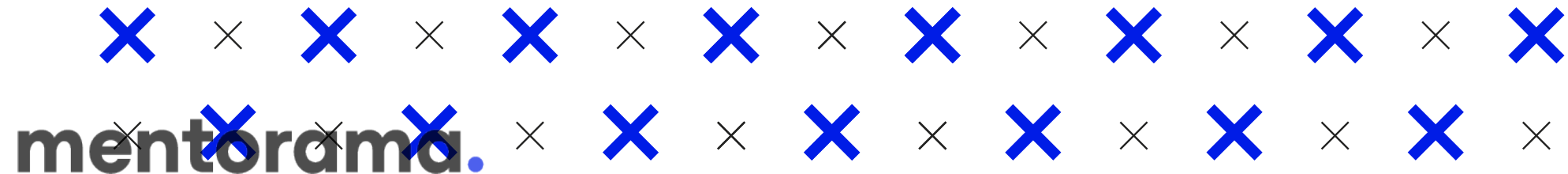
# Componentes Externos

- ◆ Como podemos realizar os testes com componentes externos como Banco de Dados, APIs Externas, precisamos levantar todo o serviço que desejamos testar para realizar esse tipo de teste?

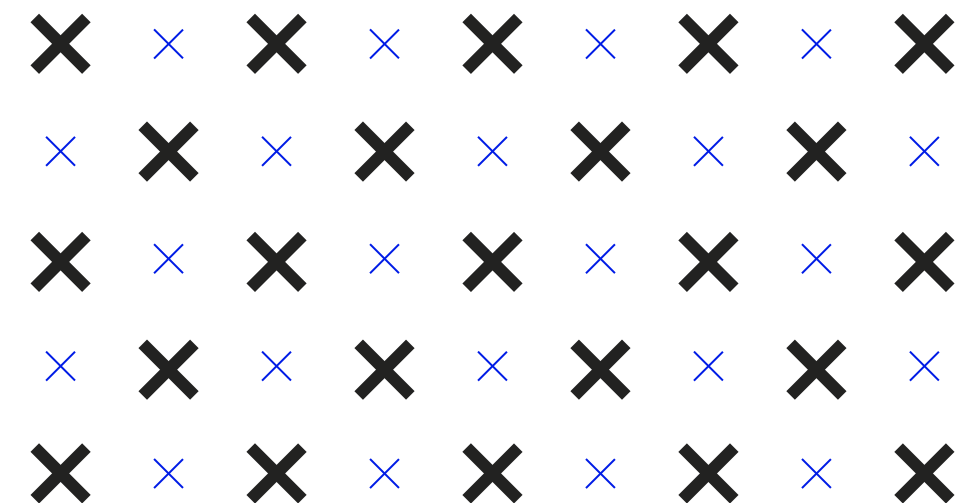


# Componentes Externos

- ◆ A resposta é NÃO!
- ◆ Em alguns casos até podemos utilizar serviços reais, como por exemplo um banco de dados de testes, mas isso não é estritamente necessário
- ◆ Sempre é importante ter claro em nossa mente o que exatamente estamos querendo testar com o teste que está sendo implementado
- ◆ No caso, queremos testar a INTEGRAÇÃO entre nossa aplicação e uma aplicação externa, nesse caso não precisamos garantir o funcionamento da aplicação externa
- ◆ Para isso, há várias maneiras de simularmos o comportamento dessas aplicações externas



# Algumas ferramentas bastante úteis



◆ [www.h2database.com](http://www.h2database.com)

◆ [www.testcontainers.org/](http://www.testcontainers.org/)

◆ [wiremock.org](http://wiremock.org)

◆ [localstack.cloud/](http://localstack.cloud/)



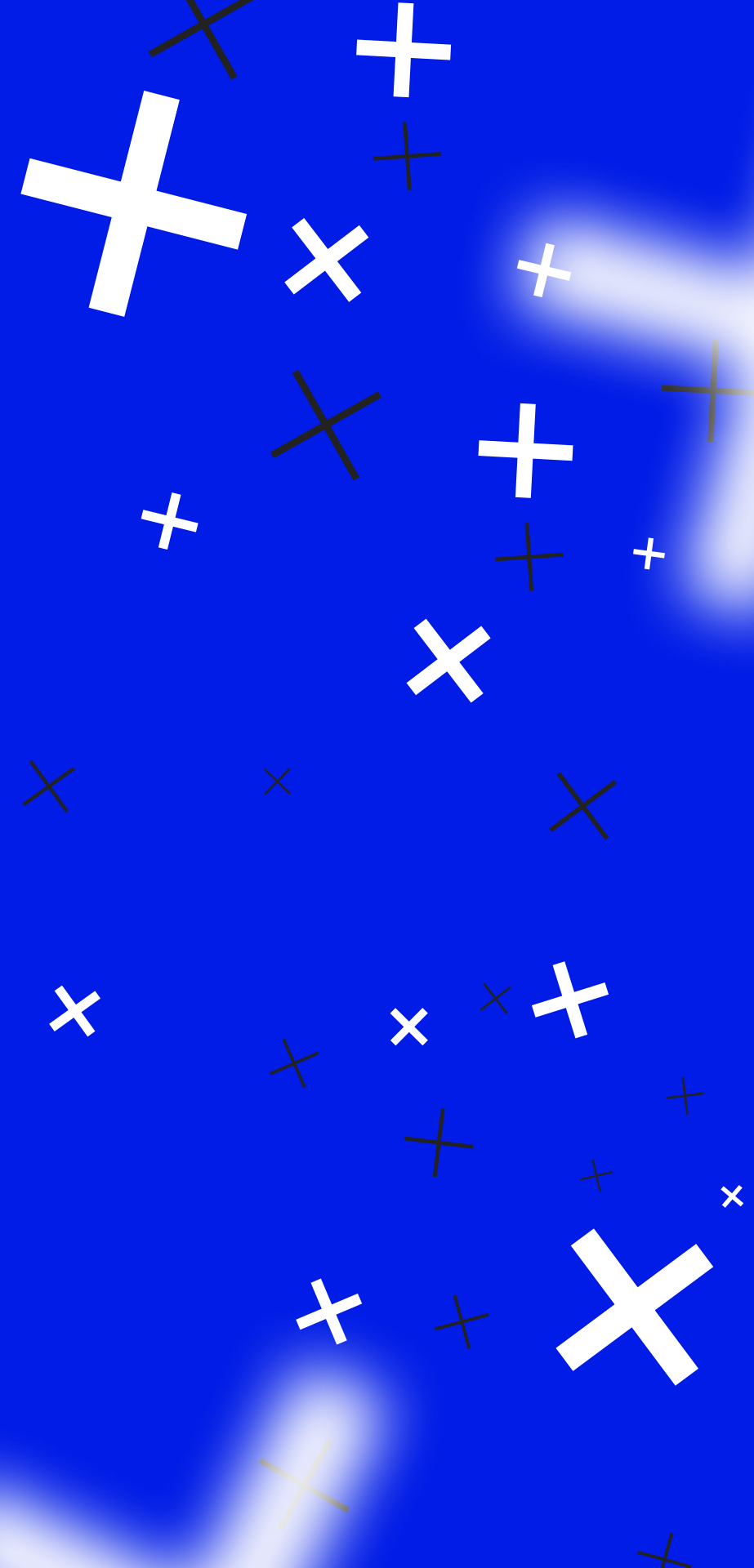
mentorama.

mentorama.

# Vamos codar?

mentorama.

mentorama.

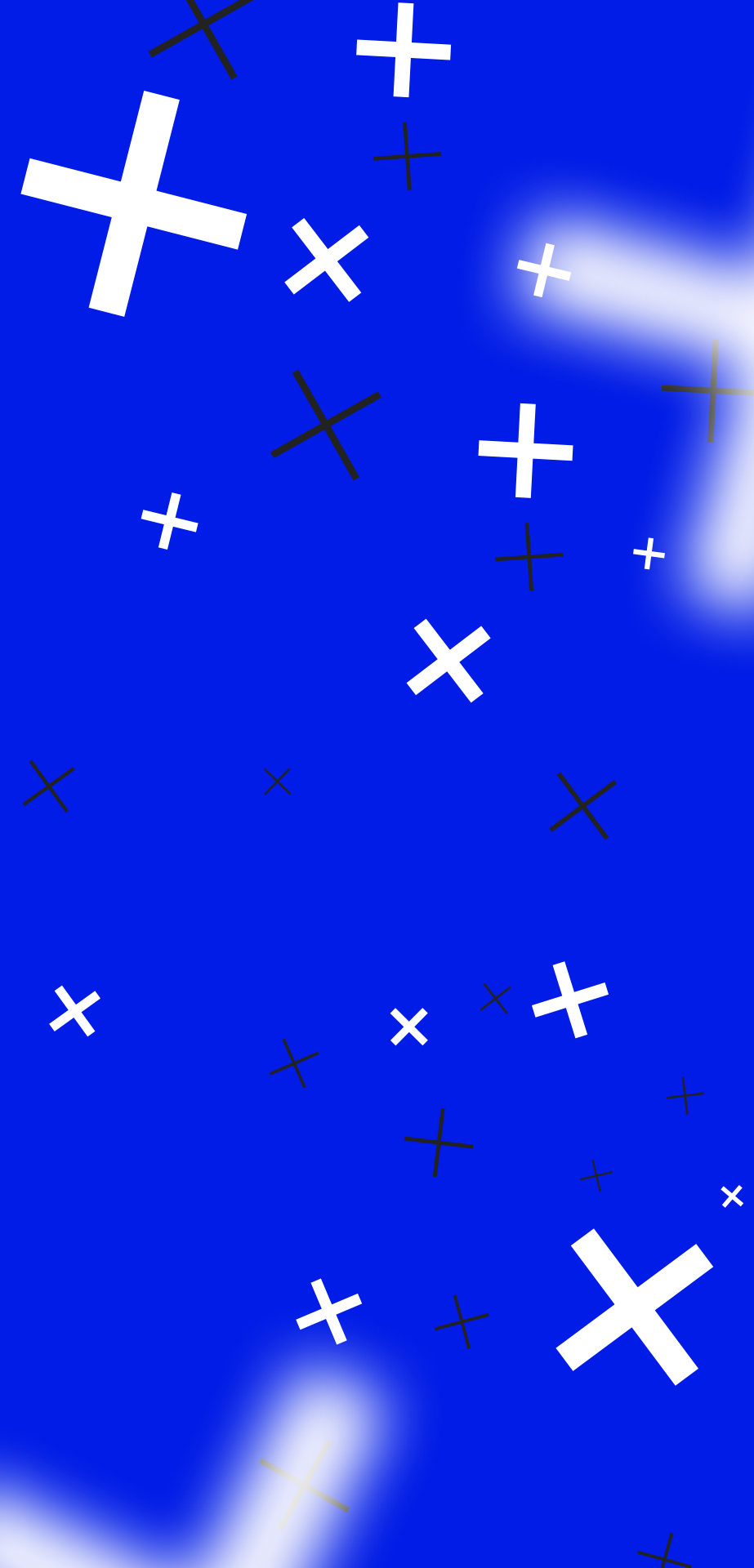


# Cobertura de Testes

Testes Automatizados

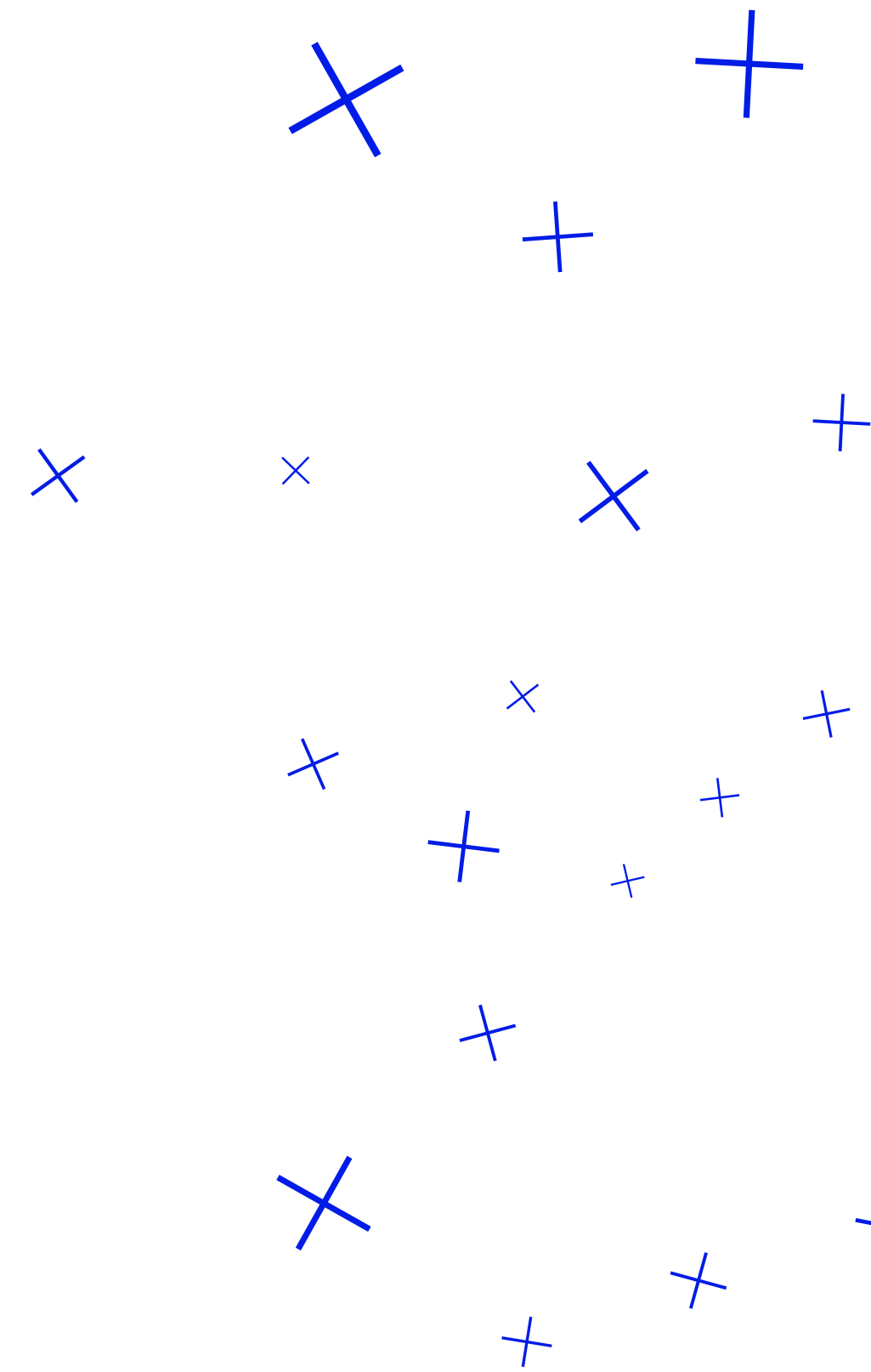
mentorama.

mentorama.



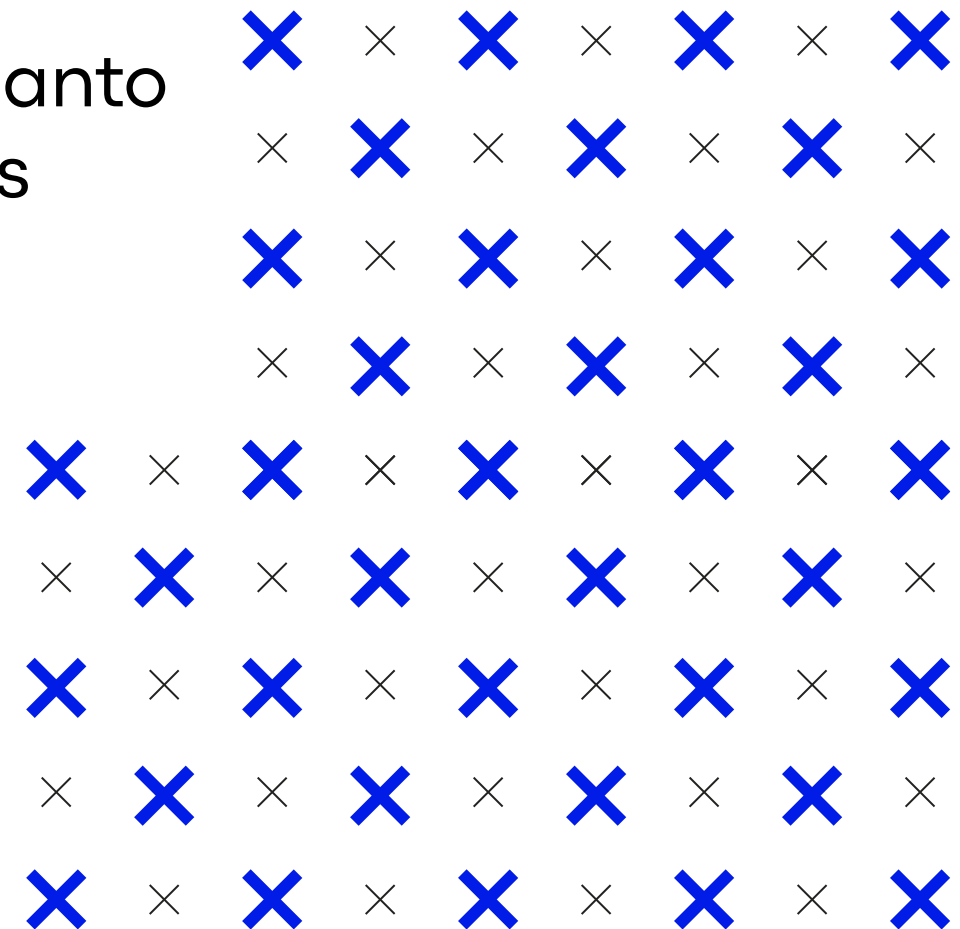
# Cobertura de Testes

- ◆ Após ter escrito testes tanto unitários quanto de integração, queremos saber o quanto nossa aplicação está coberta por testes, correto?
- ◆ Uma boa maneira de ter uma noção de quais partes de nosso código estão sendo executadas durante a execução da suíte de testes é extrair um relatório de cobertura de testes, ou em inglês, code coverage



# Cobertura de Testes

- ◆ Esse relatório basicamente analisa todo o nosso código e nos mostra a porcentagem do código que está sendo coberta pelos nossos testes e pode inclusive apontar exatamente quais as partes que estão ou não cobertas
- ◆ É realmente uma boa maneira de termos uma noção do quanto do nosso código é executado durante a execução dos testes





# Mas cuidado

- ◆ A cobertura de testes por si só não pode nos garantir efetivamente a qualidade do seu código e nem dos seus testes
- ◆ Você pode ter uma suíte de testes que apesar de ter um bom nível de cobertura, na prática pode não estar executando cenários de teste que realmente refletem a realidade, o que significa que sua aplicação pode não estar sendo corretamente testada e bugs poderão surgir em produção



# Algumas ferramentas

# Cobertura


## Cobertura



# Jacoco



## IntelliJ



✗ ✗

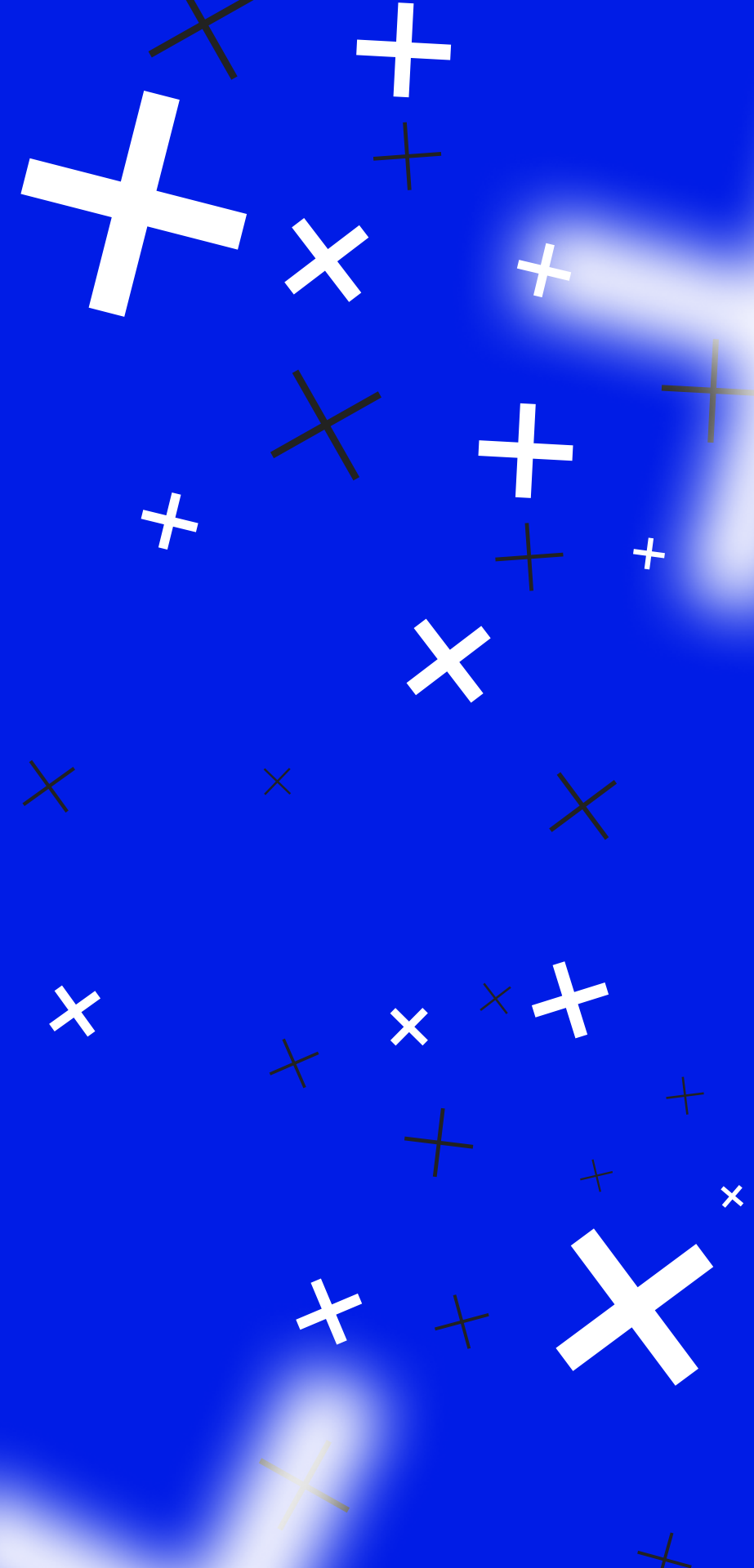
**mentorama.**

**mentorama.**

Vamos analisar  
nosso código?

mentorama.

mentorama.

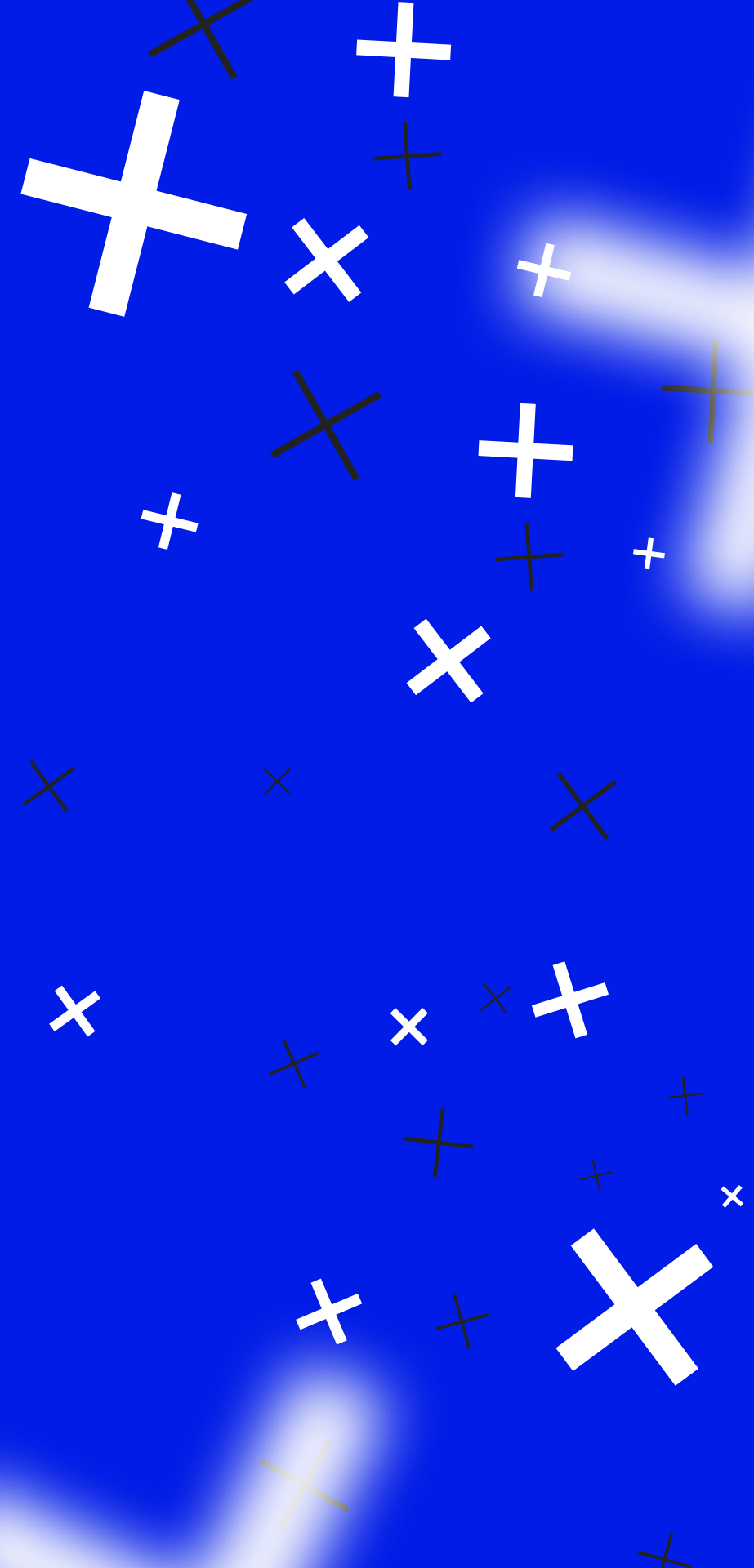


# Boas Práticas

Testes Automatizados

mentorama.

mentorama.



# Boas práticas para escrita de testes automatizados

- ◆ Assim como todo código que escrevemos de funcionalidades, o código que escrevemos para testes também possuem uma série de boas práticas que são interessantes serem seguidas
- ◆ Vamos passar por algumas delas e entender como elas podem nos ajudar a escrever um código de teste mais limpo e coeso

**mentorama.**

**mentorama.**

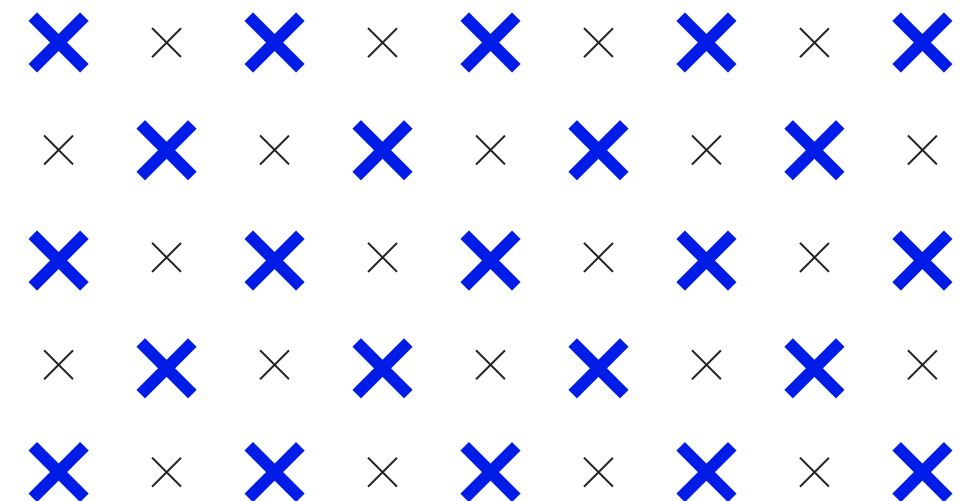


# 1 - Testes unitários devem ser realmente unitários

- ◆ Parece óbvio, mas é comum que ao testarmos unidades de código, como por exemplo, um método, tenhamos vários cenários diferentes para testes
- ◆ Por exemplo, um mesmo método pode ter fluxos diferentes dependendo do valor de seus parâmetros
- ◆ Para cada situação dessa, deve ser escrito um teste separado

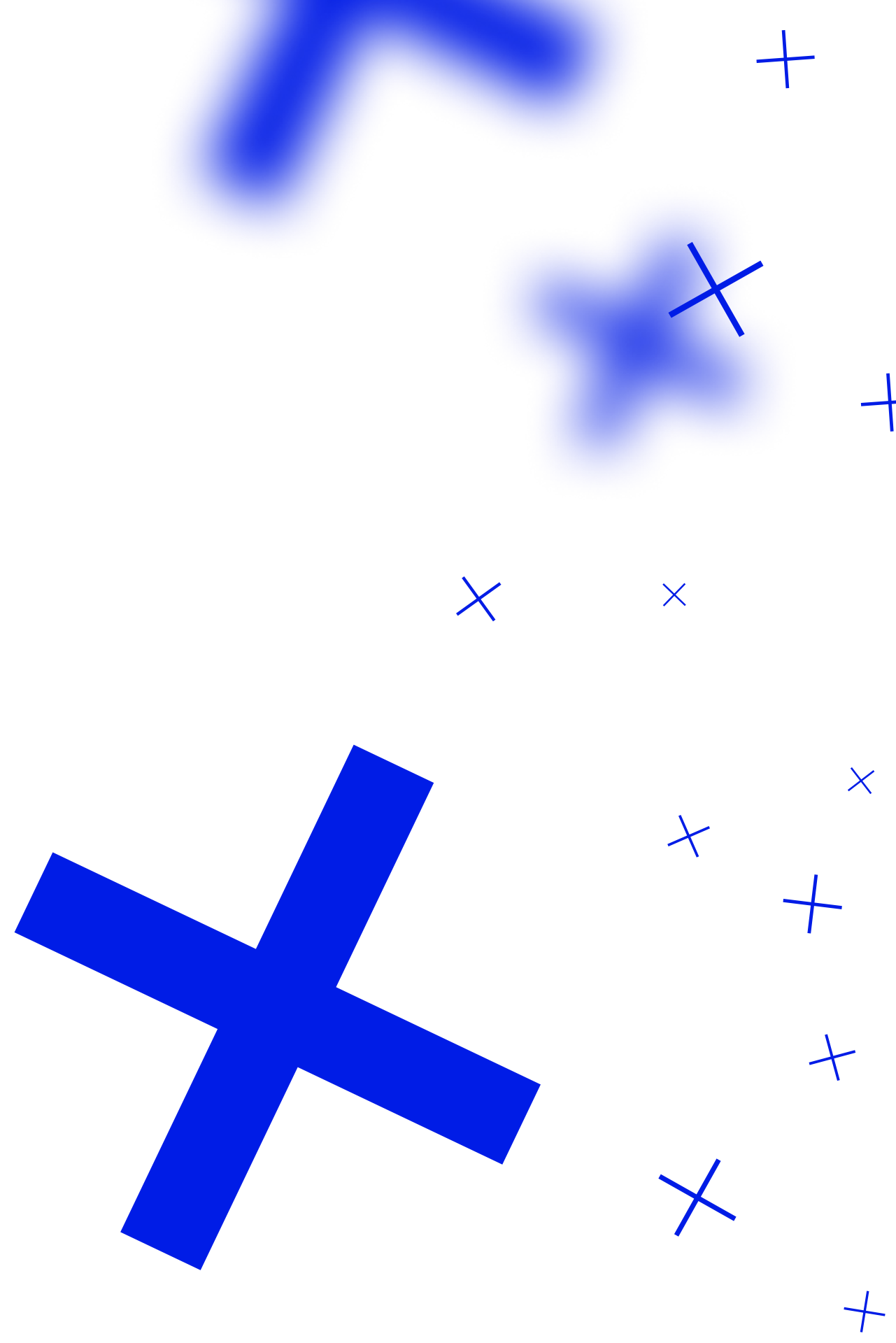
**mentorama.**

**mentorama.**



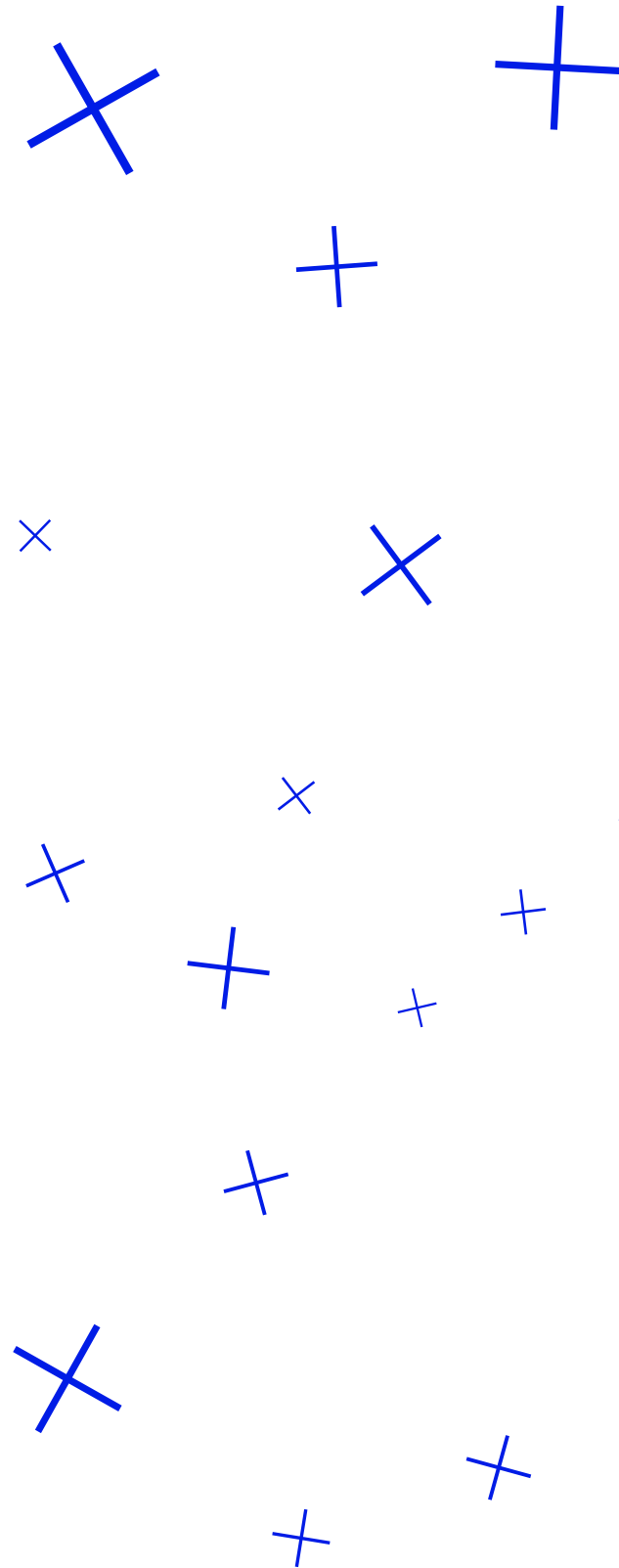
## 2 - Não faça assertions desnecessárias

- ◆ Testes bem escritos, se tornam a documentação viva de uma base de código, por isso mantenha coesão em suas assertions
- ◆ Apenas faça assertions do que for essencial para o caso de teste em questão



# 3 - Faça cada teste independente dos outros

- ◆ Não escreva testes que dependam um do outro, inclusive em relação a ordem de execução
- ◆ Além de trazer dificuldades ao identificar a causa raiz de um erro, ainda dificulta a execução de casos de testes em paralelo o que pode aumentar o tempo de execução da sua suíte de testes, e lembre, testes devem ser um feedback rápido sobre a consistência de sua aplicação



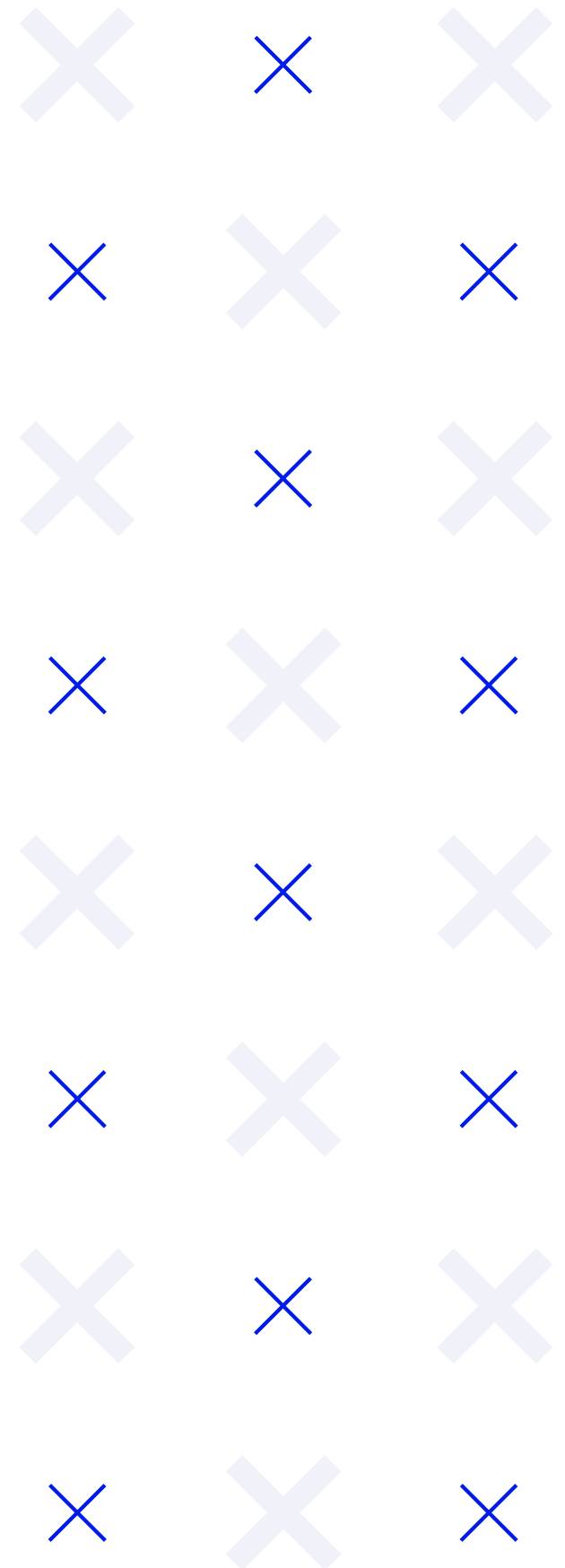


# 4 - Mock todo serviço ou dependência externa

- ◆ Salvo cenários específicos, queremos que nossos testes validem o comportamento de nossa aplicação de maneira isolada
- ◆ Não queremos que aplicações externas possa influenciar no resultado de nossos testes
- ◆ Por isso sempre que possível, mock o resultado de chamadas a aplicações externas para que elas não atrapalhem as validações de seus testes

**mentorama.**

**mentorama.**

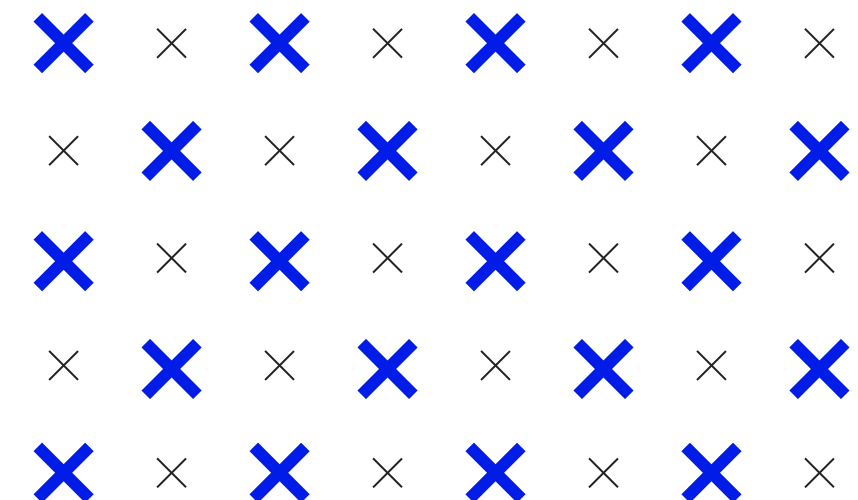


# 5 - Dê nomes consistentes para seus testes

- ◆ Esse apesar de simples, é um ponto bastante importante
- ◆ Sempre tenha em mente que o código que você escreve vai ser lido por outras pessoas, e é importante facilitar ao máximo a compreensão de tudo
- ◆ Para testes isso é especialmente importante pois como já falamos, são uma verdadeira documentação de todo o código

- ◆ O nome do teste deve deixar claro sua intenção, exemplos:

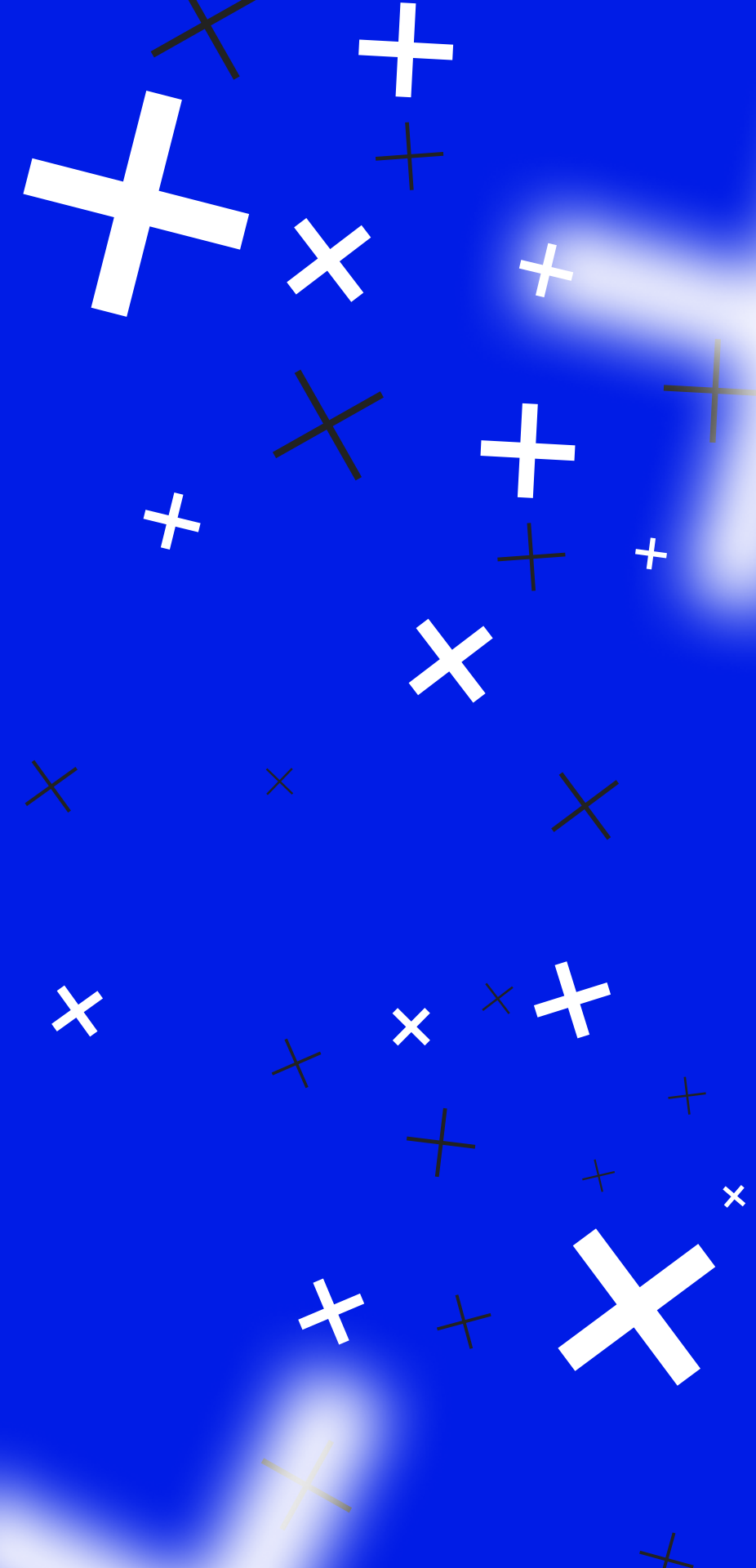
- ◇ deveRetornarVazioAoPassarCódigoDeUsuárioInválido
- ◇ deveListarTodosOsFilmes
- ◇ deveLançarExceptionParaFilmeInválido



Essas são algumas  
boas práticas...

mentorama.

mentorama.



# Home Task

**mentorama.**

- ◆ Desenvolva uma API de venda de Produtos com suporte aos seguintes endpoints:
  - ◇ Listar produtos com seus respectivos valores
  - ◇ Dar entrada em produtos
    - Deve conter o id do produto e a quantidade em cada requisição
  - ◇ Vender produtos
    - O endpoint de venda de produtos deve passar uma lista de Items onde cada item deve possuir os seguintes atributos:
      - Código do produto
      - Quantidade
      - Desconto
    - O endpoint deve retornar o valor total final da venda considerando as quantidade, descontos e aplicação das regras de negócio

# Home Task

**mentorama.**

- ◆ Os produtos vendidos devem estar pré-cadastrados no sistema e devem possuir os seguintes atributos:
  - ◆ Quantidade em estoque
  - ◆ Id(único)
  - ◆ Valor
  - ◆ Desconto máximo permitido
  - ◆ Nome
- ◆ As seguintes regras de negócio devem ser aplicadas
  - ◆ Ao tentar dar um desconto maior do que o permitido para o produto, deve ser considerado o desconto máximo
  - ◆ Ao tentar realizar uma venda de uma quantidade maior do que a disponível em estoque, deve ser vendido apenas a quantidade de produtos disponíveis
- ◆ Todas as regras de negócio devem ser validadas através de testes unitários
- ◆ Os endpoints devem ser testados utilizando testes de integração

**mentorama.**