02

Códigos de Teste Legíveis

Já temos uma grande quantidade de código de teste, e isso é ótimo: temos cada vez mais segurança para mexermos nos nossos códigos de produção. Mas muito código, seja de testes ou seja de produção, é sempre um problema: se for mal escrito, pode dar problemas de manutenção. Do mesmo jeito que nós desenvolvedores nos preocupamos ao máximo com a qualidade do código de produção, é preciso também se preocupar com o código de testes. Nesse capítulo, vamos discutir algumas das boas práticas que temos para códigos de teste, e como fazer para que eles sejam mais fáceis de serem mantidos.

Veja, por exemplo, nosso `ConsultaSpec.js`. Em todos os nossos testes, repetimos a linha que instancia o Paciente. O que aconteceria se, por exemplo, precisássemos mudar essa linha? E se o construtor da classe mudasse? Precisaríamos mudar em todos os métodos de teste, e isso nos daria um trabalho muito grande. Precisamos então isolar esse processo de criação do objeto `guilherme` em algum método, para reutilizá-lo:

```
var guilherme;
function setUp() {
  guilherme = new Paciente("Guilherme", 28, 72, 1.80);
}
```

Agora precisamos invocar esse método antes de cada teste. Veja que até deixamos a variável `guilherme` como global, para que todos os testes consigam acessá-la. Dado que isso é um procedimento bastante comum em testes (executar um conjunto de código antes de cada teste), o próprio Jasmine já dá a opção para isso. O nome desse método é `beforeEach`, e ele é executado automaticamente pelo framework antes de cada teste. Veja o código. Repare também que a variável `guilherme` é global dentro do escopo dos testes, ou seja, todos eles acessam a mesma variável.

```
describe("Consulta", function() {
    var guilherme;

    beforeEach(function() {
        guilherme = new Paciente("Guilherme", 28, 72, 1.80);
    });

  // testes aqui...
}
```

Se colocar um simples `console.log` dentro do `beforeEach`, vemos que ele é executado antes de cada teste! Analogamente, temos o `afterEach`, que é executado após cada teste.

Podemos também separar melhor nossos testes. Veja, por exemplo, a `ConsultaSpec`. Temos testes ali para consultas que são retorno, consultas comuns, consultas particulares, e etc. Para facilitar a legibilidade dos mesmos, poderíamos agrupá-los. Para isso, podemos usar a instrução que já conhecemos: a `describe`. Podemos ter describes dentro de describes, para facilitar a legibilidade. Veja:

```
describe("Consulta", function() {
    var guilherme;

    beforeEach(function() {
```

```
        guilherme = new PacienteBuilder("Guilherme", 28, 72, 1.80);
    });

    describe("Consultas que são retornos", function() {

        it("nao deve cobrar nada se a consulta for um retorno", function() {
            var consulta = new Consulta(guilherme, [], true, true);

            expect(consulta.preco()).toEqual(0);
        });

    });

    describe("Consultas que são particulares", function() {

        it("deve dobrar o preco da consulta particular", function() {
            var consulta = new Consulta(guilherme, ["proc1", "proc2"], true, false);

            expect(consulta.preco()).toEqual(50 * 2);
        });

        it("deve dobrar o preco da consulta particular mesmo com procedimentos especiais", funct
            var consulta = new Consulta(guilherme, ["raio-x"], true, false);

            expect(consulta.preco()).toEqual(55 * 2);
        });
    });

    describe("Consultas por um convênio", function() {
        it("deve cobrar preco especifico dependendo do procedimento", function() {
            var consulta = new Consulta(guilherme, ["procedimento-comum", "raio-x", "procediment

            expect(consulta.preco()).toEqual(25 + 55 + 25 + 32);
        });
    });

});
```

Veja que temos **consultas que são particulares**, **consultas que são retornos**, e assim por diante. Os testes ficam agrupados e fáceis de serem lidos. Sempre que tiver testes que são de alguma forma agrupados/similares, descreva-os e junte-os. Veja só como que isso é exibido na hora que o executamos:

```
Paciente
  deve calcular o IMC
  deve calcular batimentos cardiacos
Consulta
  Consultas que são retornos
    nao deve cobrar nada se a consulta for um retorno
  Consultas que são particulares
    deve dobrar o preco da consulta particular
    deve dobrar o preco da consulta particular mesmo com procedimentos especiais
  Consultas por um convênio
    deve cobrar preco especifico dependendo do procedimento
Maior e Menor
  deve entender numeros em ordem nao sequencial
  deve entender numeros em ordem decrescente
  deve entender array com um elemento
```

O próximo passo agora é facilitar ainda mais o processo de criação desses objetos. Veja só nossa classe `Paciente`. Sempre que precisamos de um, precisamos passar muitas informações: nome, idade, peso, altura. Imagine se essa

classe fosse mais complexa, tendo, por exemplo, endereço. Quanto mais complicado é a entidade, mais complicado é criar o cenário pro teste. E, muitas vezes, algumas informações, apesar de obrigatórias, não são úteis para o teste. Veja o nome, por exemplo: não importa qual é, contanto que tenha um nome.

Agora pense novamente em mudanças. Se a classe `Paciente` mudar seu construtor, precisaremos passar por todos os testes (ou por todos `beforeEach`) e fazer as modificações. E paciente é uma classe importante, ela é usada em muitos pontos do sistema. Daria mais um trabalho.

Vamos então criar uma classe cujas tarefas serão facilitar o processo de criação de pacientes, e ao mesmo tempo, caso algo mude no processo de criação do objeto, a mudança será feita só lá. Veja:

```
function PacienteBuilder() {

    var nome = "Guilherme";
    var idade = 28;
    var peso = 72;
    var altura = 1.80;

    var clazz = {
        comNome : function(valor) {
            nome=valor;
            return this;
        },

        comIdade : function(valor) {
            idade=valor;
            return this;
        },

        comPeso : function(valor) {
            peso=valor;
            return this;
        },
        comAltura : function(valor) {
            altura = valor;
            return this;
        },
        constroi : function() {
            return new Paciente(nome, idade, peso, altura);
        }

    };

    return clazz;
}
```

Repare que esse builder cria Pacientes. Mas ele já tem valores padrões para o paciente. Ou seja, se o teste não precisar mudar nenhum desses valores, criar um paciente é simples: `new PacienteBuilder().constroi()`. Veja nosso teste de consultas, modificado para usá-lo:

```
var guilherme;

beforeEach(function() {
```

```
    guilherme = new PacienteBuilder().constroi();
});
```

Muito mais fácil. Se precisássemos de um paciente com uma altura específica, faríamos:

```
var joao = new PacienteBuilder().comAltura(1.72).constroi();
```

O nome desse padrão que tomamos para facilitar a criação dos nossos testes é **Test Data Builders**, ou seja, construtores de dados para testes. Esse é um padrão muito comum, usado em testes escritos em qualquer linguagem de programação, para facilitar a criação dos cenários dos testes. E veja só, se qualquer coisa mudar no processo de criação de um paciente, mudaremos em um único lugar.

Sempre que você tiver objetos complicados de serem criados, ou que sofrem modificações constantes, crie Builders pra eles. Aqui demos uma possível implementação de builders, mas você pode implementar da maneira que preferir, contanto que ele facilite a criação do objeto.

Pronto, agora diminuímos um pouco o problema que tínhamos com a manutenção dos testes. Afinal, mudar agora é fácil. **Sempre que tiver código repetido em seus testes, isole-os de alguma forma. Use o beforeEach, crie métodos privados, use test data builders e etc. Escreva código de teste de qualidade.**