02

Testando o que importa em um projeto real

Nosso próximo passo agora é começar a testar um sistema maior. Nosso exemplo aqui será um software, feito em Javascript, que lida com pacientes e consultas. Algo como um sistema de controle de consultas para clínicas médicas. Um paciente é representado também por uma classe em Javascript, e contém nome, idade, peso e altura. Ele também possui algumas funções que calculam seu IMC (índice de massa corpórea) e quantidade de batimentos aproximados que seu coração já deu na vida.

Veja o código:

```
function Paciente(nome, idade,
    peso, altura) {

    var clazz = {

        imprime : function() {
            alert(nome + " tem " + idade);
        },

        batimentos : function() {
            return idade * 365 * 24 * 60 * 80;
        },

        imc : function() {
            return peso/(altura*altura);
        }
    };

    return clazz;
}
```

Já uma consulta contém um paciente, uma lista de procedimentos, e booleanos que guardam se essa é uma consulta particular, e se é um retorno. O código que implementa isso é algo como:

```
function Consulta(paciente, procedimentos, particular, retorno) {

    var clazz = {
        preco : function() {
            if(retorno) return 0;

            var precoFinal = 0;

            procedimentos.forEach(function(procedimento) {
                if("raio-x" == procedimento) precoFinal += 55;
                else if("gesso" == procedimento) precoFinal += 32;
                else precoFinal += 25;
            });

            if(particular) precoFinal *= 2;

            return precoFinal;
```

```
            }
        }

        return clazz;
    };
```

Vamos começar com o Paciente. Vamos criar o `PacienteSpec.js` e fazer um teste para o método `imc()`. Veja que o método contém apenas uma conta matemática, e retorna um número. Dado que não existem caminhos diferentes nesse método (ou seja, nenhum `if` ou `for` para mudar a execução), um único teste é suficiente. O teste, como você já sabe, deve criar um cenário, invocar o método e verificar o resultado por meio de um `expect`:

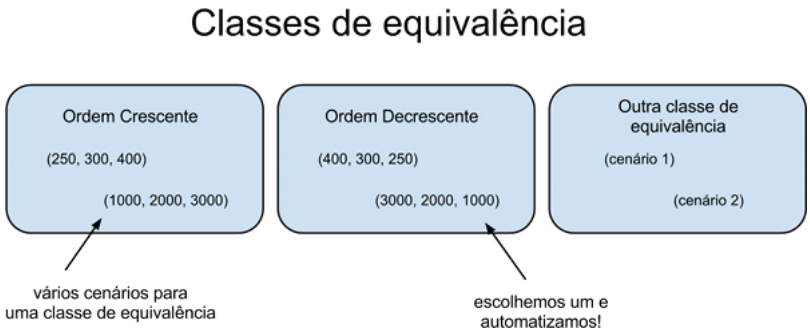
```
describe("Paciente", function() {
    it("deve calcular o IMC", function() {
        var guilherme = new Paciente("Guilherme", 28, 72, 1.82);

        expect(guilherme.imc()).toEqual(72 / (1.82*1.82));
    });
});
```

Veja que no caso acima, usamos um paciente com 72 quilos e 1.82 de altura. Será que vale a pena fazermos outro teste, com um paciente com outra altura e peso?

```
describe("Paciente", function() {
    it("deve calcular o IMC", function() {
        var guilherme = new Paciente("Guilherme", 28, 72, 1.82);

        expect(guilherme.imc()).toEqual(72 / (1.82*1.82));
    });

    it("deve calcular o IMC 2", function() {
        var guilherme = new Paciente("Guilherme", 28, 82, 1.77);

        expect(guilherme.imc()).toEqual(82 / (1.77*1.77));
    });
});
```

Agora, temos mais segurança. Afinal, temos 2 testes para o método IMC. O problema é que isso é uma afirmação falsa. Repare que, se colocarmos propositalmente um erro na implementação, os dois testes quebrarão. Se corrigirmos, os dois testes voltarão a passar. Ou seja, temos dois testes que ficam verde e vermelhos juntos. Se isso sempre acontece, podemos jogar fora um deles; afinal, quanto menos código tivermos, melhor.

Quando temos testes que passam juntos e quebram juntos, dizemos que ambos pertencem a mesma **classe de equivalência**. Seu código irá buscar por classes de equivalência diferentes, ou seja, cenários que exercitem trechos diferentes de código.



Vamos exercitar isso agora na funcionalidade de Consultas. Dê uma olhada no método `preco()`. Ele é bem complicado, e possui diversos caminhos diferentes, dependendo dos parâmetros. Por exemplo, se `retorno` for verdadeiro, o método retorna 0 de primeira. Caso contrário, ele faz um loop em todos os procedimentos, e de acordo com o nome do procedimento, ele calcula de uma maneira diferente. A consulta particular também, por sua vez, dobra o valor final.

Ou seja, temos diferentes cenários para testar. E essa é a graça. É conseguir pensar nesse conjunto de cenários, para garantir que nosso algoritmo esteja realmente testado. Vamos começar com o mais fácil. Ou seja, uma consulta do tipo retorno. Esperamos que o resultado seja 0:

```
describe("Consulta", function() {

    it("nao deve cobrar nada se a consulta for um retorno", function() {
        var guilherme = new Paciente("Guilherme", 28, 72, 1.80);
        var consulta = new Consulta(guilherme, [], true, true);

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

O próximo passo é começar a testar aquela parte mais complexa dos loops. Veja que cada procedimento comum é cobrado 25 reais. Vamos fazer um teste então para garantir que o preço cresce conforme a quantidade de ítens:

```
    it("deve cobrar 25 por cada procedimento comum", function() {
        var guilherme = new Paciente("Guilherme", 28, 72, 1.80);
        var consulta = new Consulta(guilherme, ["proc1", "proc2"], false, false);

        expect(consulta.preco()).toEqual(50);
    });
```

Veja que não faz diferença passarmos uma lista com 5, 6 ou 7 procedimentos. O processamento é o mesmo.

Podemos agora testar o mesmo para consultas particulares. Afinal, o valor deve dobrar:

```
    it("deve dobrar o preco da consulta particular", function() {
        var guilherme = new Paciente("Guilherme", 28, 72, 1.80);
        var consulta = new Consulta(guilherme, ["proc1", "proc2"], true, false);

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

Ainda temos o caso de testar os procedimentos que tem valores específicos, como é o caso do **raio-x** e do **gesso**. Podemos fazer um cenário onde misturamos ambos procedimentos comuns e especiais, e ver o resultado. Por exemplo:

```
it("deve cobrar preco especifico dependendo do procedimento", function() {
    var guilherme = new Paciente("Guilherme", 28, 72, 1.80);
    var consulta = new Consulta(guilherme, ["procedimento-comum", "raio-x", "procedimento-c

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

Ainda poderíamos testar outros cenários, como uma consulta particular com os procedimentos especiais, e garantir que o valor sairia dobrado ao final. Repare que a ideia aqui é criar o menor conjunto de cenários possíveis que cubram todas as possibilidades de execução daquele método.

Dessa forma, ganhamos segurança. Podemos refatorar o código de produção quantas vezes quisermos, que nossos testes nos darão sempre segurança sobre o funcionamento dele.

Escrever testes é fundamental, e como estamos mostrando até agora, mais fácil do que parece.