02

Test-Driven Development (TDD)

Nosso próximo passo agora é implementar uma nova funcionalidade. Sempre que um paciente passa por uma consulta, ele precisa marcar um retorno. O retorno deve ser marcado para 20 dias após a data da consulta. E, no caso, de 20 dias pra frente, ser um fim de semana, então o sistema deve empurrar para o próximo dia útil livre.

Nosso primeiro passo será então acrescentar uma data na consulta. Além disso, já que as variáveis que guardam as informações da consulta são "privados", vamos criar métodos para que seja possível consultá-los:

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

    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;
        },

        getNome : function() { return paciente; },
        getProcedimentos : function() { return procedimentos; },
        isParticular : function() { return particular; },
        isRetorno : function() { return retorno; },
        getData : function() { return data; }

    }

    return clazz;
};
```

Implementamos essa funcionalidade e a testaremos, claro. Mas, dessa vez, faremos ao contrário. Começaremos pelo teste. Mas será possível? Claro que é. Só é questão de se acostumar. Vamos lá, imagine que a funcionalidade já exista e você deve apenas testá-la. O primeiro teste então verificará que o agendamento é marcado para 20 dias depois:

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

    var agenda;

    beforeEach(function() {
        agenda = new Agendamento();
        gui = new PacienteBuilder().constroi();
    });
```

```
    it("deve agendar para 20 dias depois", function() {

        var consulta = new Consulta(gui, [], false, false, new Date(2014, 7, 1));
        var novaConsulta = agenda.para(consulta);

        expect(novaConsulta.getData().toString()).toEqual(new Date(2014,7,21).toString());

    });

});
```

O teste, claramente, não passa. Porque o `Agendamento` nem existe. Vamos lá, então, começar a implementação dessa classe. E mais ainda: vamos fazer a mais simples implementação possível pra isso:

```
function Agendamento() {

    var clazz = {

        para : function(consulta) {

            var novaData = new Date(2014, 7, 21);
            var novaConsulta = new Consulta(consulta.getNome(), consulta.getProcedimentos(),
                consulta.isParticular(), consulta.isRetorno(), novaData);
            return novaConsulta;
        }
    };

    return clazz;
}
```

Ótimo. O teste passa. Mas essa implementação não está boa, afinal a data está fixa ali. Hora de refatorar, e melhorar:

```
function Agendamento() {

    var clazz = {

        para : function(consulta) {

            var vinteDiasEmMillisegundos = 1000 * 60 * 60 * 24 * 20;
            var novaData = new Date(consulta.getData().getTime() + vinteDiasEmMillisegundos);

            var novaConsulta = new Consulta(consulta.getNome(), consulta.getProcedimentos(),
                consulta.isParticular(), consulta.isRetorno(), novaData);
            return novaConsulta;
        }
    };

    return clazz;
}
```

Excelente. O teste continua passando. Vamos para a próxima funcionalidade. Ele deve pular fins de semana. Para isso, vamos passar uma data na qual saibamos que 20 dias depois, cai em um sábado. Novamente, vamos começar pelo teste:

```
    it("deve pular fins de semana", function() {

        var consulta = new Consulta(gui, [], false, false, new Date(2014, 5, 30));
        var novaConsulta = agenda.para(consulta);

        expect(novaConsulta.getData().toString()).toEqual(new Date(2014,6,21).toString());

    });
```

Com o teste vermelho, vamos para a implementação:

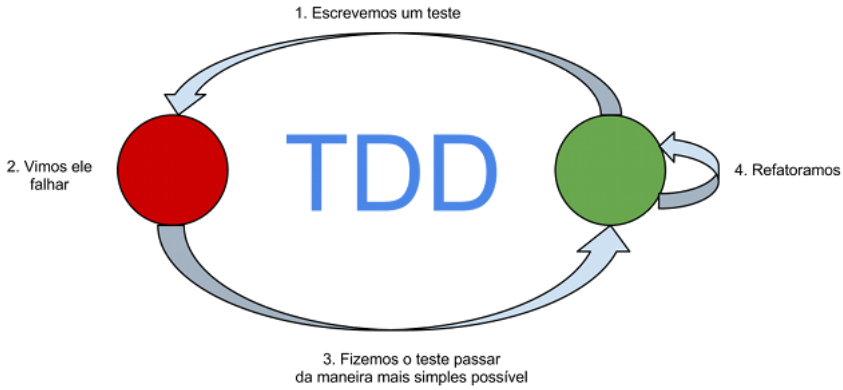
```
        para : function(consulta) {

            var vinteDiasEmMillisegundos = 1000 * 60 * 60 * 24 * 20;
            var umDiaEmMilissegundo = 1000 * 60 * 60 * 24;

            var novaData = new Date(consulta.getData().getTime() + vinteDiasEmMillisegundos);
            while(novaData.getDay()==0 || novaData.getDay()==6) novaData = new Date(novaData.get

            var novaConsulta = new Consulta(consulta.getNome(), consulta.getProcedimentos(),
                consulta.isParticular(), consulta.isRetorno(), novaData);
            return novaConsulta;
        }
```

Teste verde! Momento para melhorar nosso código. Veja que podemos escrever um código mais legível ali na parte dos milissegundos:

```
        para : function(consulta) {

            var umDiaEmMilissegundo = 1000 * 60 * 60 * 24;
            var vinteDiasEmMillisegundos = umDiaEmMilissegundo * 20;

            var novaData = new Date(consulta.getData().getTime() + vinteDiasEmMillisegundos);
            while(novaData.getDay()==0 || novaData.getDay()==6) {
                novaData = new Date(novaData.getTime() + umDiaEmMilissegundo);
            }

            var novaConsulta = new Consulta(consulta.getNome(), consulta.getProcedimentos(),
                consulta.isParticular(), consulta.isRetorno(), novaData);
            return novaConsulta;
        }
```

Pronto! Os testes continuam passando! Poderíamos continuar escrevendo testes antes do código, mas vamos agora pensar um pouco sobre o que acabamos de fazer.

Em primeiro lugar, escrevemos um teste; o rodamos e o vimos falhar; em seguida, escrevemos o código mais simples para passar o teste; rodamos-o novamente, e dessa vez ele passou; por fim, refatoramos nosso código para que ele fique melhor e mais claro. A figura abaixo mostra o ciclo que acabamos de fazer:



Esse ciclo de desenvolvimento, onde escrevemos um teste antes do código, é conhecido por Test-Driven Development. A popularidade da prática de TDD tem crescido cada vez mais entre os desenvolvedores, uma vez que ela traz diversas vantagens para o desenvolvedor:

- Se sempre escrevermos o teste antes, garantimos que todo nosso código já "nasce" testado;
- Temos segurança para refatorar nosso código, afinal sempre refatoraremos com uma bateria de testes que garante que não quebraremos o comportamento já existente;
- Como o teste é a primeira classe que usa o seu código, você naturalmente tende a escrever código mais fácil de ser usado e, por consequência, mais fácil de ser mantido.

Vamos praticar!