03

Test Driven Design - TDD

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/laboratorio-java/stages/06/capitulo6.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Separando os candles

Agora que temos nosso leitor de XML que cria uma lista com as negociações representadas no arquivo passado, um novo problema surge: a BOVESPA permite fazer download de um arquivo XML contendo **todas** as negociações de um ativo desde a data especificada. Entretanto, nossa `CandlestickFactory` está preparada apenas para **construir candles de uma data específica**.

Dessa forma, precisamos ainda quebrar a lista que contém todas as negociações em partes menores, com negociações de um dia apenas, e usar o outro método para gerar cada `Candlestick` . Essas, devem ser armazenadas em uma nova lista para serem devolvidas.

Para fazer tal lógica, então, precisamos:

- Passar por cada negociações da lista original;
- Verificar **se continua no mesmo dia** e...
- ... se sim, adiciona na lista do dia;
- ... caso contrário:
  - Gera a *candle*;
  - Guarda numa lista de *candlesticks*;
  - Zera a lista de negociações do dia;
  - Indica que vai olhar o próximo dia, agora;
- Ao final, devolver a lista de *candles*.

O algoritmo não é trivial e, ainda, ele depende de uma verificação que o Java não nos dá prontamente: **se continua no mesmo dia**. Isto é, dado que sabemos qual a `dataAtual` , queremos verificar se a negociação pertence a esse mesmo dia.

Para verificar se uma negociação é do mesmo dia, já sabemos como o método `isMesmoDia` deverá se comportar em diversas situações:

- Se for exatamente o mesmo milissegundo => true;
- Se for no mesmo dia, mas em horários diferentes => true;
- Se for no mesmo dia, mas em meses diferentes => false;
- Se for no mesmo dia e mês, mas em anos diferentes => false.

Sempre que vamos começar a desenvolver uma lógica, intuitivamente, já pensamos em seu comportamento. Fazer os testes automatizados para tais casos é, portanto, apenas colocar nosso pensamento em forma de código. Mas fazê-lo incrementalmente, mesmo antes de seguir com a implementação, é o princípio do que chamamos de **Test Driven Design** (**TDD**).

Vantagens do TDD

TDD é uma técnica que consiste em pequenas iterações, em que novos casos de testes de funcionalidades desejadas são criados antes mesmo da implementação. Nesse momento, o teste escrito deve falhar, já que a funcionalidade implementada não existe.

Então, o código necessário para que os testes passem deve ser escrito e o teste deve passar. O ciclo se repete para o próximo teste mais simples que ainda não passa.

Um dos principais benefício dessa técnica é que, como os testes são escritos antes da implementação do trecho a ser testado, o programador não é influenciado pelo código já feito - assim, ele tende a escrever testes melhores, pensando no comportamento ao invés da implementação.

Lembremos: os testes devem mostrar (e documentar) o comportamento do sistema, e não o que uma implementação faz.

Além disso, nota-se que TDD traz baixo acoplamento, o que é ótimo, já que classes muito acopladas são difíceis de testar. Como criaremos os testes antes, desenvolveremos classes menos acopladas, isto é, menos dependentes de outras muitas, separando melhor as responsabilidades.

O TDD também é uma espécie de guia: como o teste é escrito antes, nenhum código do sistema é escrito por "acharmos" que vamos precisar dele. Em sistemas sem testes, é comum encontrarmos centenas de linhas que jamais serão invocadas, simplesmente porque o desenvolvedor "achou" que alguém um dia precisaria daquele determinado método.

Imagine que você já tenha um sistema com muitas classes e nenhum teste: provavelmente, para iniciar a criação de testes, muitas refatorações terão de ser feitas, mas como modificar seu sistema, garantindo o funcionamento dele após as mudanças, quando não existem testes que garantam que seu sistema tenha o comportamento desejado? Por isso, crie testes sempre e, de preferência, antes da implementação da funcionalidade.

TDD é uma disciplina difícil de se implantar, mas depois que você pega o jeito e o hábito é adquirido, podemos ver claramente as diversas vantagens dessa técnica.

Identificando negociações do mesmo dia

Poderíamos criar um método na classe `LeitorXML` que pega todo XML e converte em *candles*, mas ela teria muita responsabilidade. Vamos cuidar da lógica que separa as negociações em várias por datas em outro lugar.

Queremos então, em nossa classe de *factory*, pegar uma série de negociações e transformar em uma lista de *candles*. Para isso, vamos precisar que uma negociação saiba identificar se é do mesmo dia que a `dataAtual`.

Para saber, conforme percorremos todas as negociações, se a negociação atual aconteceu na mesma data que estamos procurando, vamos usar um método na classe `Negociacao` que faz tal verificação.

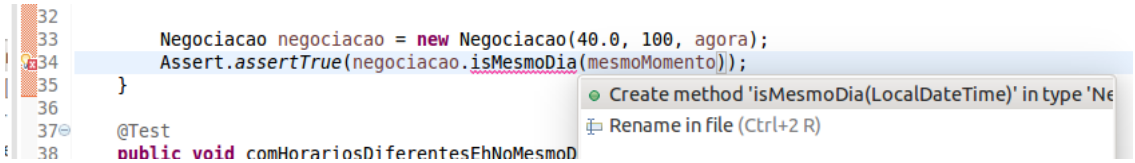
Seguindo os princípios do TDD, começamos escrevendo um teste na classe `NegociacaoTest`:

```
@Test
public void mesmoSegundoEhDoMesmoDia() {

    LocalDateTime agora = LocalDateTime.now();
    LocalDateTime mesmoMomento = agora;

    Negociacao negociacao = new Negociacao(40.0, 100, agora);
    Assert.assertTrue(negociacao.isMesmoDia(mesmoMomento));
}
```

Esse código não vai compilar de imediato, já que não temos esse método na nossa classe. No Eclipse, utilize o atalho **CTRL + 1** em cima do erro e escolha *Create method 'isMesmoDia(LocalDateTime)' in type 'Negociacao'*.



E qual será uma implementação interessante? Que tal simplificar usando o método `equals` de `LocalDateTime` ?

```
public boolean isMesmoDia(LocalDateTime outraData) {
    return this.data.equals(outraData);
}
```

Rode o teste, passa ? Nosso teste passou de primeira! Vamos tentar mais algum teste ? Vamos testar se datas iguais em horas diferentes são consideradas do mesmo dia. Crie o método a seguir na classe `NegociacaoTest` :

```
@Test
public void comHorariosDiferentesEhNoMesmoDia() {
    LocalDateTime manha =  LocalDateTime.of(2016, 02, 25, 8, 30);
    LocalDateTime tarde =  LocalDateTime.of(2016, 02, 25, 15, 30);

    Negociacao negociacao = new Negociacao(40.0, 100, manha);
    Assert.assertTrue(negociacao.isMesmoDia(tarde));
}
```

Rode o teste. Não passa!

Infelizmente, usar o `equals` não resolve nosso problema de comparação.

Para resolver esse problema, vamos comparar somente os dias do mês, invocando o método `getDayOfMonth` da classe `LocalDateTime` . A implementação que compara os dias será:

```
public boolean isMesmoDia(LocalDateTime outraData) {
    return this.data.getDayOfMonth() == outraData.getDayOfMonth();
}
```

Altere o método `isMesmoDia` na classe `Negociacao` e rode os testes anteriores. Passamos agora? Sim!

O próximo teste a implementarmos será o que garante que para dias iguais, mas meses diferentes, a data não será a mesma. Quer dizer: não basta comparar o campo referente ao dia do mês, ainda é necessário que seja o mesmo mês!

Crie o `mesmoDiaMasMesesDiferentesNaoSaoDoMesmoDia` na classe de testes `NegociacaoTest` , veja o teste falhar e, então, implemente o necessário para que ele passe. Note que, dessa vez, o valor esperado é o **false** e, portanto, utilizaremos o `Assert.assertFalse` .

```
@Test
public void mesmoDiaMasMesesDiferentesNaoSaoDoMesmoDia() {
```

```
    LocalDateTime manha =  LocalDateTime.of(2016, 02, 25, 8, 30);
    LocalDateTime tarde =  LocalDateTime.of(2016, 03, 25, 8, 30);

    Negociacao negociacao = new Negociacao(40.0, 100, manha);
    Assert.assertFalse(negociacao.isMesmoDia(tarde));

}
```

Vamos adicionar a comparação de meses no método `isMesmoDia` :

```
public boolean isMesmoDia(LocalDateTime outraData) {
    return this.data.getDayOfMonth() == outraData.getDayOfMonth()
            && this.data.getMonth() == outraData.getMonth();
}
```

Finalmente, o último teste a implementarmos será o que garante que para dia e meses iguais, mas anos diferentes, a data não é a mesma. Seguindo o procedimento de desenvolver com TDD , vamos criar o teste primeiro:

```
@Test
public void mesmoDiaEMesMasAnosDiferentesNaoSaoDoMesmoDia() {

    LocalDateTime manha =  LocalDateTime.of(2016, 02, 25, 8, 30);
    LocalDateTime tarde =  LocalDateTime.of(2017, 02, 25, 8, 30);

    Negociacao negociacao = new Negociacao(40.0, 100, manha);
    Assert.assertFalse(negociacao.isMesmoDia(tarde));
}
```

Agora com o teste implementado - e falhando, vamos alterar o método `isMesmoDia()` para incluir os anos também.

```
public boolean isMesmoDia(LocalDateTime outraData) {
    return this.data.getDayOfMonth() == outraData.getDayOfMonth()
            && this.data.getMonth() == outraData.getMonth()
            && this.data.getYear() == outraData.getYear();
}
```

Quebrando as Negociações em diferentes datas.

Nosso próximo passo é dado uma lista de negociações de várias datas diferentes mas ordenadas por data, quebrar em uma lista de *candles*, um para cada data.

Seguindo a disciplina do TDD: começamos pelo teste!

Vamos adicionar o método `paraNegociacoesDeTresDiasDistintosGeraTresCandles` na classe `CandlestickFactoryTest` . Ele vai criar diversas negociações de **três dias diferentes** e deve nos retornar **três candlesticks distintos**. Vamos lá:

```
@Test
public void paraNegociacoesDeTresDiasDistintosGeraTresCandles() {
```

```
        LocalDateTime hoje = LocalDateTime.now();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

        LocalDateTime amanha = hoje.plusDays(1);

        Negociacao negociacao5 = new Negociacao(48.8, 100, amanha);
        Negociacao negociacao6 = new Negociacao(49.3, 100, amanha);

        LocalDateTime depois = hoje.plusDays(2);

        Negociacao negociacao7 = new Negociacao(51.8, 100, depois);
        Negociacao negociacao8 = new Negociacao(52.3, 100, depois);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
            negociacao3, negociacao4, negociacao5, negociacao6, negociacao7,
            negociacao8);

        CandlestickFactory fabrica = new CandlestickFactory();

        List<Candlestick> candles = fabrica.constroiCandles(negociacoes);

        Assert.assertEquals(3, candles.size());
        Assert.assertEquals(40.5, candles.get(0).getAbertura(), 0.00001);
        Assert.assertEquals(42.3, candles.get(0).getFechamento(), 0.00001);
        Assert.assertEquals(48.8, candles.get(1).getAbertura(), 0.00001);
        Assert.assertEquals(49.3, candles.get(1).getFechamento(), 0.00001);
        Assert.assertEquals(51.8, candles.get(2).getAbertura(), 0.00001);
        Assert.assertEquals(52.3, candles.get(2).getFechamento(), 0.00001);
    }
```

O teste é grande mas não se assuste. Só estamos pedindo parar criar negociações de dias diferentes e pedindo à classe `CandlestickFactory` para que ela nos retorne uma lista com as negociações de cada dia. Como o método `constroiCandle` ainda não existe, o Eclipse nos dá um erro. Mas utilize o nosso famoso atalho **CTRL + 1** e peça para criá-lo!

Com o método criado, vamos pensar como implementá-lo. Precisamos:

- Criar a lista de *candlesticks*;
- Percorrer a lista de negociações, adicionando cada negociação no *candlestick* atual;
- Quando achar uma negociação de um novo dia, cria um *candlestick* novo e adiciona;
- Devolve a lista de *candles*;

O código deve ficar um pouco grande, mas ainda bem que já criamos o nosso teste para garantir sua corretude. Vamos lá:

```
    public List<Candlestick> constroiCandles(List<Negociacao> negociacoes) {

        // Cria a lista de negociações do dia e a lista de Candles que devemos retornar
        List<Candlestick> candlesticks = new ArrayList<>();
```

```
        List<Negociacao> negociacoesDoDia = new ArrayList<>();

        LocalDateTime dataAtual = negociacoes.get(0).getData();

        // Vamos percorrendo a lista com todas as negociacoes.
        for (Negociacao negociacao : negociacoes) {
            // se for mesmo dia, adiciona na lista de Negociacoes daquele dia
            if(negociacao.isMesmoDia(dataAtual)){
                negociacoesDoDia.add(negociacao);
            }else{
                // se não for mesmo dia, fecha o candle e reinicia variáveis
                Candlestick candle = geraCandleParaData(negociacoesDoDia, dataAtual);
                candlesticks.add(candle);

                negociacoesDoDia = new ArrayList<>();

             // Aqui precisamos adicionar o item que caiu no else na nova lista, caso constrário só
                negociacoesDoDia.add(negociacao);
                dataAtual = negociacao.getData();
            }

        }
        // adiciona último candle
        Candlestick candle = geraCandleParaData(negociacoesDoDia, dataAtual);
        candlesticks.add(candle);

        return candlesticks;
    }
```

Rode os testes e verifique que eles passam! Utilizando o TDD, conseguimos implementar esse método um pouco mais complicado com a garantia dos nossos testes por trás, para garantir que ele funciona como deveria.

Se você se interessou neste assunto de testes e TDD, aqui no Alura nós temos um curso completo focado nisso, que você pode conferir aqui (https://cursos.alura.com.br/course/tdd).

O que aprendemos?

- O que é o **Test Driven Design**.
- Quais as vantagens do TDD.
- A criar testes incrementais e simples antes mesmo da implementação.
- Como criar um *candlestick* por dia, a partir de uma lista grande de negociações.