

## CAPÍTULO 5

# Test Driven Design – TDD

*"Experiência sem teoria é cegueira, mas teoria sem experiência é mero jogo intelectual."*

— Immanuel Kant

## 5.1 – SEPARANDO AS 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 eu sei qual a `dataAtual`, quero verificar se a negociação pertence a esse mesmo dia.

Verificar uma negociação é do mesmo dia que um Calendar qualquer exige algumas linhas de código, mas veja que, mesmo antes de implementá-lo, 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).

## 5.2 – 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 em vez 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.

### Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/test-driven-design-tdd/)

## 5.3 – EXERCÍCIOS: IDENTIFICANDO NEGOCIAÇÕES DO MESMO DIA

Poderíamos criar uma classe `LeitorXML` que pega todo o XML e converte em candles, mas ela teria muita responsabilidade. Vamos cuidar da lógica que separa as negociações em vários candles por datas em outro lugar.

1. 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 ainda 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`:

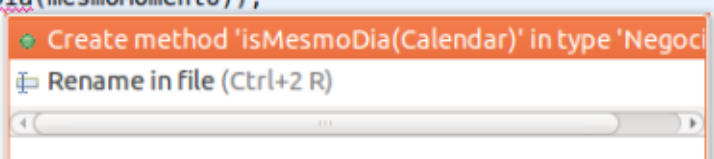
```
@Test
public void mesmoMilissegundoEhDoMesmoDia() {
    Calendar agora = Calendar.getInstance();
    Calendar mesmoMomento = (Calendar) agora.clone();

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

Esse código não vai compilar de imediato, já que não temos esse método na nossa classe. No Eclipse, aperte **Ctrl + 1** em cima do erro e escolha **Create method isMesmoDia**.

```
@Test
public void mesmoMilissegundoEhDoMesmoDia() {
    Calendar agora = Calendar.getInstance();
    Calendar mesmoMomento = (Calendar) agora.clone();

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



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

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

Rode o teste! Passa?

2. Nosso teste passou de primeira! Vamos tentar mais algum teste? Vamos testar datas iguais em horas diferentes, crie o método a seguir na classe `NegociacaoTest`:

```
@Test
public void comHorariosDiferentesEhNoMesmoDia() {
    // usando GregorianCalendar(ano, mes, dia, hora, minuto)
```

```
Calendar manha = new GregorianCalendar(2011, 10, 20, 8, 30);
Calendar tarde = new GregorianCalendar(2011, 10, 20, 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.

Lembre que um Calendar possui um *timestamp*, isso quer dizer que além do dia, do mês e do ano, há também informações de hora, segundos etc. A implementação que compara os dias será:

```
public boolean isMesmoDia(Calendar outraData) {
    return
        data.get(Calendar.DAY_OF_MONTH) == outraData.get(Calendar.DAY_OF_MONTH);
}
```

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

3. O próximo teste a implementarmos será o que garante que para dia igual, mas mês diferente, a data não é 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 do Negociacao, 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.

4. 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. Siga o mesmo procedimento para desenvolver com TDD:

- Escreva o teste mesmoDiaEMesMasAnosDiferentesNaoSaoDoMesmoDia;
- Rode e veja que falhou;
- Implemente o necessário para fazê-lo passar.

Feito esse processo, seu método isMesmoDia na classe Negociacao deve ter ficado bem parecido com isso:

```
public boolean isMesmoDia(Calendar outraData) {
    return
```

```

    this.data.get(Calendar.DAY_OF_MONTH) ==
    outraData.get(Calendar.DAY_OF_MONTH)
    && this.data.get(Calendar.MONTH) == outraData.get(Calendar.MONTH)
    && this.data.get(Calendar.YEAR) == outraData.get(Calendar.YEAR);
}

```

## 5.4 – EXERCÍCIOS: SEPARANDO OS CANDLES

1. Próximo passo: dada uma lista de negociações de várias datas diferentes mas ordenada por data, quebrar em uma lista de candles, uma para cada data.

Seguindo a disciplina do TDD: começamos pelo teste!

**Adicione o método** `paraNegociacoesDeTresDiasDistintosGeraTresCandles` na classe `CandlestickFactoryTest`:

```

@Test
public void paraNegociacoesDeTresDiasDistintosGeraTresCandles() {
    Calendar hoje = Calendar.getInstance();

    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);

    Calendar amanha = (Calendar) hoje.clone();
    amanha.add(Calendar.DAY_OF_MONTH, 1);

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

    Calendar depois = (Calendar) amanha.clone();
    depois.add(Calendar.DAY_OF_MONTH, 1);

    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);
}

```

```
}
```

A chamada ao método `constroiCandles` não compila pois o método não existe ainda. **Ctrl + 1** e **Create method**.

Como implementamos? Precisamos:

- Criar a `List<Candlestick>`;
- Percorrer a `List<Negociacao>` 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 talvez fique um pouco grande. Ainda bem que temos nosso teste!

```
public List<Candlestick> constroiCandles(List<Negociacao> todasNegociacoes) {  
    List<Candlestick> candles = new ArrayList<Candlestick>();  
  
    List<Negociacao> negociacoesDoDia = new ArrayList<Negociacao>();  
    Calendar dataAtual = todasNegociacoes.get(0).getData();  
  
    for (Negociacao negociacao : todasNegociacoes) {  
        // se não for mesmo dia, fecha candle e reinicia variáveis  
        if (!negociacao.isMesmoDia(dataAtual)) {  
            Candlestick candleDoDia = constroiCandleParaData(dataAtual,  
                                                                negociacoesDoDia);  
            candles.add(candleDoDia);  
            negociacoesDoDia = new ArrayList<Negociacao>();  
            dataAtual = negociacao.getData();  
        }  
        negociacoesDoDia.add(negociacao);  
    }  
    // adiciona último candle  
    Candlestick candleDoDia = constroiCandleParaData(dataAtual,  
                                                        negociacoesDoDia);  
    candles.add(candleDoDia);  
  
    return candles;  
}
```

Rode o teste!

## 5.5 – EXERCÍCIOS OPCIONAIS

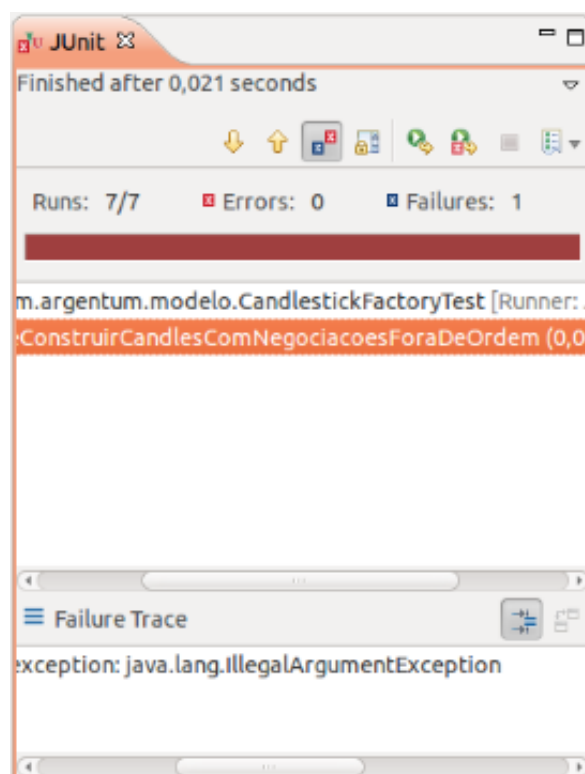
1. E se passarmos para o método `constroiCandles` da fábrica uma lista de

negociações que não está na ordem crescente? O resultado vai ser candles em ordem diferentes, e provavelmente com valores errados. Apesar da especificação dizer que os negociações vem ordenados pela data, é boa prática programar defensivamente em relação aos parâmetros recebidos.

Aqui temos diversas opções. Uma delas é, caso alguma Negociacao venha em ordem diferente da crescente, lançamos uma exception, a `IllegalStateException`.

Crie o `naoPermiteConstruirCandlesComNegociacoesForaDeOrdem` e configure o teste para verificar que uma `IllegalStateException` foi lançada. Basta usar como base o mesmo teste que tínhamos antes, mas adicionar as negociações com datas não crescentes.

Rode o teste e o veja falhar.



Pra isso, modificamos o código adicionando as linhas em negrito ao método `constroiCandles`:

```
for (Negociacao negociacao : todasNegociacoes) {  
    if (negociacao.getData().before(dataAtual)) {  
        throw new IllegalStateException("negociações em ordem errada");  
    }  
    // se não for mesmo dia, fecha candle e reinicia variáveis  
    ...  
}
```

2. Vamos criar um gerador automático de arquivos para testes da bolsa. Ele vai gerar 30 dias de candle e cada candle pode ser composto de 0 a 19 negociações. Esses preços podem variar.



```

public class GeradorAleatorioDeXML {
    public static void main(String[] args) throws IOException {
        Calendar data = Calendar.getInstance();
        Random random = new Random(123);
        List<Negociacao> negociacoes = new ArrayList<Negociacao>();

        double valor = 40;
        int quantidade = 1000;

        for (int dias = 0; dias < 30; dias++) {
            int quantidadeNegociacoesDoDia = random.nextInt(20);

            for (int negociacao = 0; negociacao < quantidadeNegociacoesDoDia;
                negociacao++){

                // no máximo sobe ou cai R$1,00 e não baixa além de R$5,00
                valor += (random.nextInt(200) - 100) / 100.0;
                if (valor < 5.0) {
                    valor = 5.0;
                }

                // quantidade: entre 500 e 1500
                quantidade += 1000 - random.nextInt(500);

                Negociacao n = new Negociacao(valor, quantidade, data);
                negociacoes.add(n);
            }
            data = (Calendar) data.clone();
            data.add(Calendar.DAY_OF_YEAR, 1);
        }

        XStream stream = new XStream(new DomDriver());
        stream.alias("negociacao", Negociacao.class);
        stream.setMode(XStream.NO_REFERENCES);

        PrintStream out = new PrintStream(new File("negociacao.xml"));
        out.println(stream.toXML(negociacoes));
    }
}

```

Se você olhar o resultado do XML, verá que, por usarmos o mesmo objeto Calendar em vários lugares, o XStream coloca referências no próprio XML evitando a cópia do mesmo dado. Mas talvez isso não seja tão interessante na prática, pois é mais comum na hora de integrar sistemas, passar um XML simples com todos os dados.

A opção `XStream.NO_REFERENCES` serve para indicar ao XStream que não queremos que ele crie *referências* a tags que já foram serializadas iguaizinhas. Você pode passar esse argumento para o método `setMode` do XStream. Faça o teste sem e com essa opção para entender a diferença.

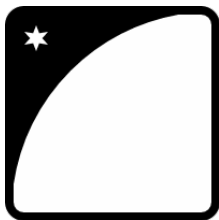
## Desafio – Ordene a lista on demand

1. Faça com que uma lista de `Negociacao` seja ordenável pela data das negociações.

Então poderemos, logo no início do método, ordenar todas as negociações com `Collections.sort` e não precisamos mais verificar se os negociações estão vindo em ordem crescente!

Perceba que mudamos uma regra de negócio, então teremos de refletir isso no nosso teste unitário que estava com `expected=IllegalStateException.class` no caso de vir em ordem errada. O resultado agora com essa modificação tem de dar o mesmo que com as datas crescentes.

**Você não está nessa página a toa**



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso \*Lab. Java com Testes, JSF e Design Patterns\*.](#)

CAPÍTULO ANTERIOR:

[Trabalhando com XML](#)

PRÓXIMO CAPÍTULO:

[Acessando um Web Service](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter