CAPÍTULO 3

Testes Automatizados

"Apenas duas coisas são infinitas: o universo e a estupidez humana. E eu não tenho certeza do primeiro."
— Albert Einstein

3.1 - Nosso código está funcionando corretamente?

Escrevemos uma quantidade razoável de código no capítulo anterior, meia dúzia de classes. Elas funcionam corretamente? Tudo indica que sim, até criamos um pequeno `main` para verificar isso e fazer as perguntas corretas.

Pode parecer que o código funciona, mas ele tem **muitas** falhas. Olhemos com mais cuidado.

3.2 - Exercícios: testando nosso modelo sem frameworks

1. Será que nosso programa funciona para um determinado dia que ocorrer apenas uma única negociação? Vamos escrever o teste e ver o que acontece:

```
public class TestaCandlestickFactoryComUmaNegociacaoApenas {

  public static void main(String[] args) {
    Calendar hoje = Calendar.getInstance();

    Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);

    List<Negociacao> negociacoes = Arrays.asList(negociacao1);

    CandlestickFactory fabrica = new CandlestickFactory();
    Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

    System.out.println(candle.getAbertura());
    System.out.println(candle.getFechamento());
    System.out.println(candle.getMinimo());
    System.out.println(candle.getMaximo());
```

```
    System.out.println(candle.getVolume());
  }
}
```

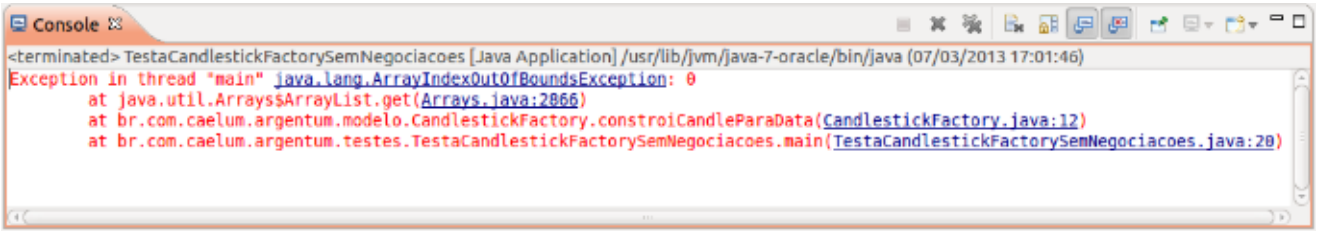
A saída deve indicar `40.5` como todos os valores, e `4050.0` como volume. Tudo parece bem?

2. Mais um teste: as ações menos negociadas podem ficar dias sem nenhuma operação acontecer. O que nosso sistema gera nesse caso?

Vamos ao teste:

```
public class TestaCandlestickFactorySemNegociacoes {

  public static void main(String[] args) {
    Calendar hoje = Calendar.getInstance();

    List<Negociacao> negociacoes = Arrays.asList();

    CandlestickFactory fabrica = new CandlestickFactory();
    Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

    System.out.println(candle.getAbertura());
    System.out.println(candle.getFechamento());
    System.out.println(candle.getMinimo());
    System.out.println(candle.getMaximo());
    System.out.println(candle.getVolume());
  }
}
```

Rodando o que acontece? Você acha essa saída satisfatória? Indica bem o problema?



3. `ArrayIndexOutOfBoundsException` certamente é uma péssima exceção para indicar que não teremos *Candle*.

Qual decisão vamos tomar? Podemos lançar nossa própria *exception*, podemos retornar `null` ou ainda podemos devolver um `Candlestick` que possui um significado especial. Devolver `null` deve ser sempre a última opção.

Vamos retornar um `Candlestick` que possui um volume zero. Para corrigir o erro, vamos alterar o código do nosso `CandlestickFactory`.

Poderíamos usar um `if` logo de cara para verificar se `negociacoes.isEmpty()`, porém podemos tentar algo mais sutil, sem ter que criar vários pontos de `return`.

Vamos então iniciar os valores de `minimo` e `maximo` sem usar a lista, que pode estar vazia. Mas, para nosso algoritmo funcionar, precisaríamos iniciar o `maximo` com um valor **bem pequeno**, isto é, menor do que o da ação mais barata. Assim, quando percorrermos o `for`, qualquer valor que encontremos vai substituir o `maximo`.

No caso do máximo, é fácil pensar nesse valor! Qual é o limite inferior para o preço de uma ação? Será que ele chega a ser negativo? A resposta é não. Ninguém vai vender uma ação por um preço negativo! Portanto, se inicializarmos o máximo com **zero**, é certeza que ele será substituido na primeira iteração do `for`.

Similarmente, queremos inicializar o `minimo` com um valor **bem grande**, maior do que o maior valor possível para o valor de uma ação. Esse é um problema mais complexo, já que não existe um limitante superior tão claro para o preço de uma ação!

Qual valor colocaremos, então? Quanto é um número grande o suficiente? Podemos apelar para a limitação do tipo que estamos usando! Se o preço é um double, certamente não poderemos colocar nenhum valor que estoure o tamanho do double. Felizmente, a classe `Double` conta com a constante que representa o maior double válido! É o `Double.MAX_VALUE`.

Altere o método `constroiCandleParaData` da classe `CandlestickFactory`:

```
double maximo = 0;
double minimo = Double.MAX_VALUE;
```

Além disso, devemos verificar se `negociacoes` está vazia na hora de calcular o preço de `abertura` e `fechamento`. **Altere** novamente o método:

```
double abertura = negociacoes.isEmpty() ? 0 : negociacoes.get(0).getPreco();
double fechamento = negociacoes.isEmpty() ? 0 :
          negociacoes.get(negociacoes.size() - 1).getPreco();
```

Pronto! Rode o teste, deve vir tudo zero e números estranhos para máximo e mínimo!

4. Será que tudo está bem? Rode novamente os outros dois testes, o que acontece?

Incrível! Consertamos um bug, mas adicionamos outro. A situação lhe parece familiar? Nós desenvolvedores vivemos com isso o tempo todo: tentando fugir dos

velhos bugs que continuam a reaparecer!

O teste com apenas uma negociação retorna 1.7976931348623157E308 como valor mínimo agora! Mas deveria ser 40.5. Ainda bem que *lembramos* de rodar essa classe, e que percebemos que esse número está diferente do que deveria ser.

Vamos sempre confiar em nossa memória?

5. (opcional) Será que esse erro está ligado a ter apenas uma negociação? Vamos tentar com mais negociações? Crie e rode um teste com as seguintes negociações:

```
Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
Negociacao negociacao3 = new Negociacao(49.8, 100, hoje);
Negociacao negociacao4 = new Negociacao(53.3, 100, hoje);
```

E com uma sequência decrescente, funciona? Por quê?

3.3 - Definindo melhor o sistema e descobrindo mais bugs

Segue uma lista de dúvidas pertinentes ao Argentum. Algumas dessas perguntas você não saberá responder, porque não definimos muito bem o comportamento de alguns métodos e classes. Outras você saberá responder.

De qualquer maneira, crie um código curto para testar cada uma das situações, em um `main` apropriado.

1. Uma negociação da Petrobras a 30 reais, com uma quantidade negativa de negociações é válida? E com número zero de negociações?

Em outras palavras, posso dar `new` em uma `Negociacao` com esses dados?

2. Uma negociação com data nula é válida? Posso dar `new Negociacao(10, 5, null)`? Deveria poder?

3. Um candle é realmente imutável? Não podemos mudar a data de um candle de maneira alguma?

4. Um candle em que o preço de abertura é igual ao preço de fechamento, é um candle de alta ou de baixa? O que o sistema diz? O que o sistema deveria dizer?

5. Como geramos um candle de um dia que não houve negociação? O que acontece?

6. E se a ordem das negociações passadas ao `CandlestickFactory` não estiver na ordem crescente das datas? Devemos aceitar? Não devemos?

7. E se essas `Negociações` forem de dias diferentes que a data passada como argumento para a factory?

3.4 - Testes de Unidade

Testes de unidade são testes que testam apenas uma classe ou método, verificando se seu comportamento está de acordo com o desejado. Em testes de unidade, verificamos a funcionalidade da classe e/ou método em questão passando o mínimo possível por outras classes ou dependências do nosso sistema.

> **Unidade**
>
> Unidade é a menor parte testável de uma aplicação. Em uma linguagem de programação orientada a objetos como o Java, a menor unidade é um método.
>
> O termo correto para esses testes é **testes de unidade**, porém o termo *teste unitário* propagou-se e é o mais encontrado nas traduções.

Em testes de unidade, não estamos interessados no comportamento real das dependências da classe, mas em como a classe em questão se comporta diante das possíveis respostas das dependências, ou então se a classe modificou as dependências da maneira esperada.

Para isso, quando criamos um teste de unidade, simulamos a execução de

métodos da classe a ser testada. Fazemos isso passando parâmetros (no caso de ser necessário) ao método testado e definimos o resultado que esperamos. Se o resultado for igual ao que definimos como esperado, o teste passa. Caso contrário, falha.

> **Atenção**
>
> Muitas vezes, principalmente quando estamos iniciando no mundo dos testes, é comum criarmos alguns testes que testam muito mais do que o necessário, mais do que apenas a unidade. Tais testes se transformam em verdadeiros **testes de integração** (esses sim são responsáveis por testar o sistemas como um todo).
>
> Portanto, lembre-se sempre: testes de unidade testam **apenas** unidades!

3.5 - JUnit

O **JUnit** (junit.org) é um framework muito simples para facilitar a criação destes testes de unidade e em especial sua execução. Ele possui alguns métodos que tornam seu código de teste bem legível e fácil de fazer as **asserções**.

Uma **asserção** é uma afirmação: alguma invariante que em determinado ponto de execução você quer garantir que é verdadeira. Se aquilo não for verdade, o teste deve indicar uma falha, a ser reportada para o programador, indicando um possível bug.

À medida que você mexe no seu código, você roda novamente toda aquela bateria de testes com um comando apenas. Com isso você ganha a confiança de que novos bugs não estão sendo introduzidos (ou reintroduzidos) conforme você cria novas funcionalidades e conserta antigos bugs. Mais fácil do que ocorre quando fazemos os testes dentro do `main`, executando um por vez.

O JUnit possui integração com todas as grandes IDEs, além das ferramentas de build, que vamos conhecer mais a frente. Vamos agora entender um pouco mais sobre anotações e o `import` estático, que vão facilitar muito o nosso trabalho com o JUnit.

Usando o JUnit - configurando Classpath e seu JAR no Eclipse

O JUnit é uma biblioteca escrita por terceiros que vamos usar no nosso projeto. Precisamos das classes do JUnit para escrever nossos testes. E, como sabemos, o formato de distribuição de bibliotecas Java é o JAR, muito similar a um ZIP com as classes daquela biblioteca.

Precisamos então do JAR do JUnit no nosso projeto. Mas quando rodarmos nossa aplicação, como o Java vai saber que deve incluir as classes daquele determinado JAR junto com nosso programa? (dependência)

É aqui que o **Classpath** entra história: é nele que definimos qual o "*caminho para buscar as classes que vamos usar*". Temos que indicar onde a JVM deve buscar as classes para compilar e rodar nossa aplicação.

Há algumas formas de configurarmos o *classpath*:

- Configurando uma variável de ambiente (**desaconselhado**);
- Passando como argumento em linha de comando (**trabalhoso**);
- Utilizando ferramentas como Ant e Maven (veremos mais a frente);
- Deixando o eclipse configurar por você.

No Eclipse, é muito simples:

1. Clique com o botão direito em cima do nome do seu projeto.
2. Escolha a opção *Properties*.
3. Na parte esquerda da tela, selecione a opção "*Java Build Path*".

E, nessa tela:

1. "*Java Build Path*" é onde você configura o *classpath* do seu projeto: lista de locais definidos que, por padrão, só vêm com a máquina virtual configurada;

2. Opções para adicionar mais caminhos, "Add JARs..." adiciona Jar's que estejam no seu projeto; "Add External JARs" adiciona Jar's que estejam em qualquer outro lugar da máquina, porém guardará uma referência para aquele caminho (então seu projeto poderá não funcionar corretamente quando colocado em outro micro, mas existe como utilizar variáveis para isso);

No caso do JUnit, por existir integração direta com Eclipse, o processo é ainda mais fácil, como veremos no exercício. Mas para todas as outras bibliotecas que formos usar, basta copiar o JAR e adicioná-lo ao *Build Path*. Vamos ver esse

procedimento com detalhes quando usarmos as bibliotecas que trabalham com XML e gráficos em capítulos posteriores.

3.6 - Anotações

Anotação é a maneira de escrever metadados na própria classe, isto é, configurações ou outras informações pertinentes a essa classe. Esse recurso foi introduzido no Java 5.0. Algumas anotações podem ser mantidas (*retained*) no `.class`, permitindo que possamos reaver essas informações, se necessário.

É utilizada, por exemplo, para indicar que determinada classe deve ser processada por um framework de uma certa maneira, evitando assim as clássicas configurações através de centenas de linhas de XML.

Apesar dessa propriedade interessante, algumas anotações servem apenas para indicar algo ao compilador. `@Override` é o exemplo disso. Caso você use essa anotação em um método que não foi reescrito, vai haver um erro de compilação! A vantagem de usá-la é apenas para facilitar a legibilidade.

`@Deprecated` indica que um método não deve ser mais utilizado por algum motivo e decidiram não retirá-lo da API para não quebrar programas que já funcionavam anteriormente.

`@SurpressWarnings` indica para o compilador que ele não deve dar warnings a respeito de determinado problema, indicando que o programador sabe o que está fazendo. Um exemplo é o warning que o compilador do Eclipse dá quando você não usa determinada variável. Você vai ver que um dos quick fixes é a sugestão de usar o `@SupressWarnings`.

Anotações podem receber parâmetros. Existem muitas delas na API do java 5, mas realmente é ainda mais utilizada em frameworks, como o Hibernate 3, o EJB 3 e o JUnit4.

3.7 - JUnit4, convenções e anotação

Para cada classe, teremos uma classe correspondente (por convenção, com o sufixo `Test`) que contará todos os testes relativos aos métodos dessa classe. Essa classe ficará no pacote de mesmo nome, mas na *Source Folder* de testes (`src/test/java`).

Por exemplo, para a nossa `CandlestickFactory`, teremos a `CandlestickFactoryTest`:

```
package br.com.caelum.argentum.modelo;

public class CandlestickFactoryTest {

  public void sequenciaSimplesDeNegociacoes() {
    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);

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

    CandlestickFactory fabrica = new CandlestickFactory();
    Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);
  }
}
```

Em vez de um `main`, criamos um método com nome expressivo para descrever a situação que ele está testando. Mas... como o JUnit saberá que deve executar aquele método? Para isso **anotamos** este método com `@Test`, que fará com que o JUnit saiba no momento de execução, por reflection, de que aquele método deva ser executado:

```
public class CandlestickFactoryTest {

  @Test
  public void sequenciaSimplesDeNegociacoes() {
    // ...
  }
}
```

Pronto! Quando rodarmos essa classe como sendo um teste do JUnit, esse método será executado e a View do JUnit no Eclipse mostrará se tudo ocorreu bem. Tudo ocorre bem quando o método é executado sem lançar exceções inesperadas e se todas as **asserções** passarem.

Uma asserção é uma verificação. Ela é realizada através dos métodos estáticos da classe `Assert`, importada do `org.junit`. Por exemplo, podemos verificar se o valor de abertura desse candle é 40.5:

```
Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
```

O primeiro argumento é o que chamamos de *expected*, e ele representa o valor que esperamos para argumento seguinte (chamado de *actual*). Se o valor real for diferente do esperado, o teste não passará e uma barrinha vermelha será mostrada, juntamente com uma mensagem que diz:

expected <valor esperado> but was <o que realmente deu>

Double é inexato

Logo na primeira discussão desse curso, conversamos sobre a inexatidão do `double` ao trabalhar com arredondamentos. Porém, diversas vezes, gostaríamos de comparar o double esperado e o valor real, sem nos preocupar com diferenças de arredondamento quando elas são **muito** pequenas.

O JUnit trata esse caso adicionando um terceiro argumento, que só é necessário quando comparamos valores **double** ou **float**. Ele é um delta que se aceita para o erro de comparação entre o valor esperado e o real.

Por exemplo, quando lidamos com dinheiro, o que nos importa são as duas primeiras casas decimais e, portanto, não há problemas se o erro for na quinta casa decimal. Em softwares de engenharia, no entanto, um erro na quarta casa decimal pode ser um grande problema e, portanto, o delta deve ser ainda menor.

Nosso código final do teste, agora com as devidas asserções, ficará assim:

```
public class CandlestickFactoryTest {

  @Test
  public void sequenciaSimplesDeNegociacoes() {
    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);
```

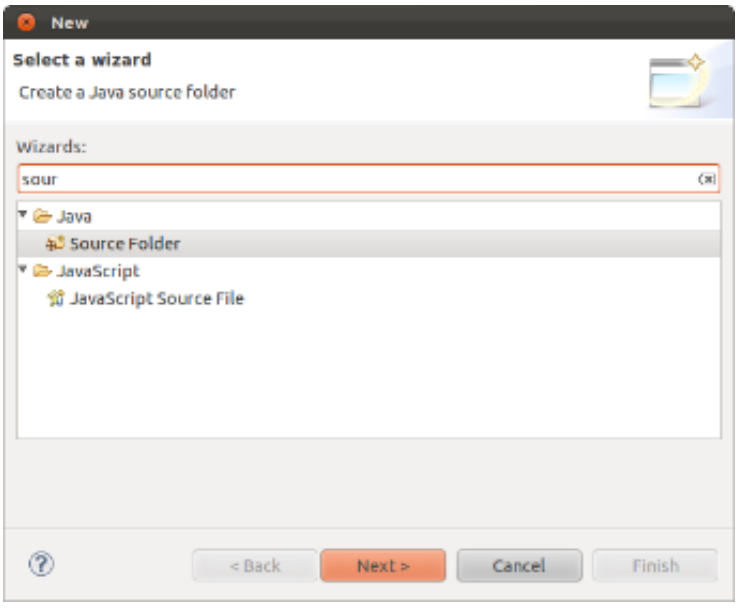
```
        Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

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

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

        Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
        Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
        Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
        Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
        Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
    }
}
```

Existem ainda outras anotações principais e métodos importantes da classe `Assert`, que conheceremos no decorrer da construção do projeto.

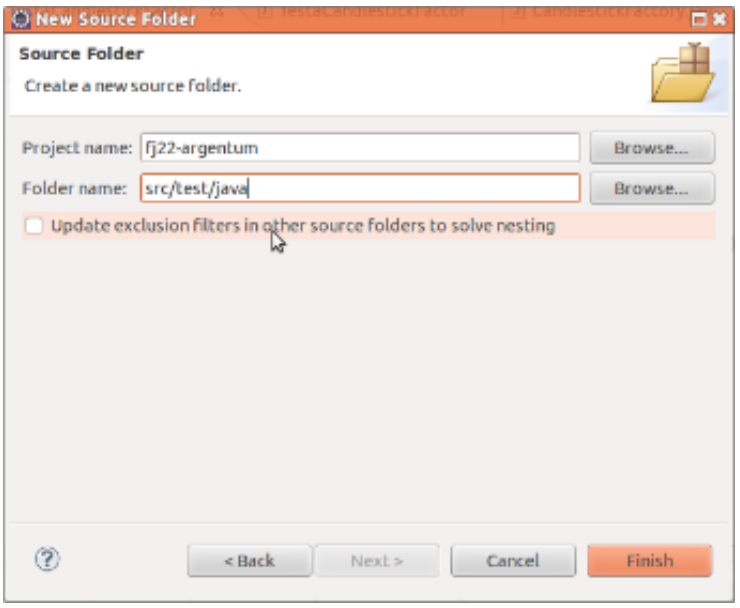
3.8 – Exercícios: migrando os testes do main para JUnit

1. É considerada boa prática separar as classes de testes das classes principais. Para isso, normalmente se cria um novo *source folder* apenas para os testes. Vamos fazer isso:

a. **Ctrl + N** e comece a digitar "Source Folder" até que o filtro a encontre:



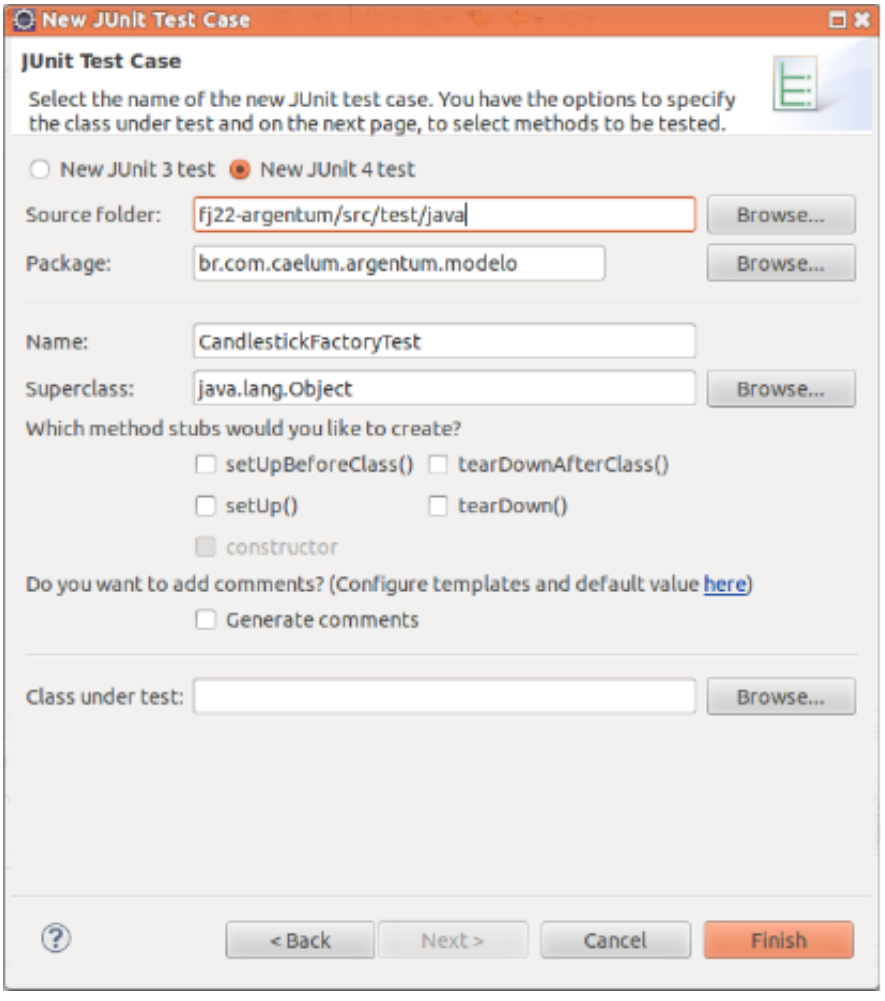
b. Preencha com **src/test/java** e clique **Finish**:



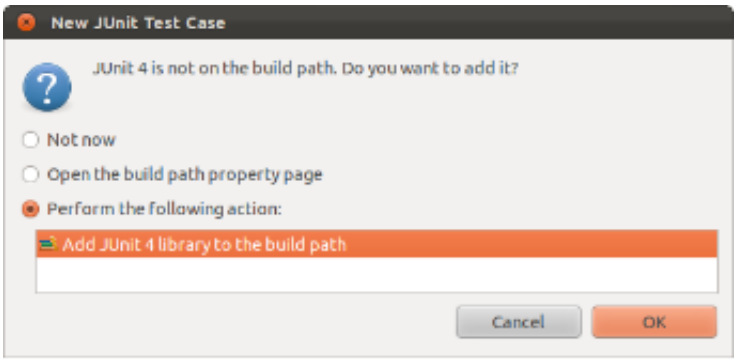
É nesse novo diretório em que você colocará todos seus testes de unidade.

2. Vamos criar um novo *unit test* em cima da classe `CandlestickFactory`. O Eclipse já ajuda bastante: com o editor na CandlestickFactory, crie um novo (**ctrl + N**) JUnit Test Case.

Na janela seguinte, selecione o *source folder* como **src/test/java**. Não esqueça, também, de **selecionar JUnit4**.



Ao clicar em *Finish*, o Eclipse te perguntará se pode adicionar os jars do JUnit no projeto.



A anotação `@Test` indica que aquele método deve ser executado na bateria de testes, e a classe `Assert` possui uma série de métodos estáticos que realizam comparações, e no caso de algum problema um exceção é lançada.

Vamos colocar primeiro o teste inicial:

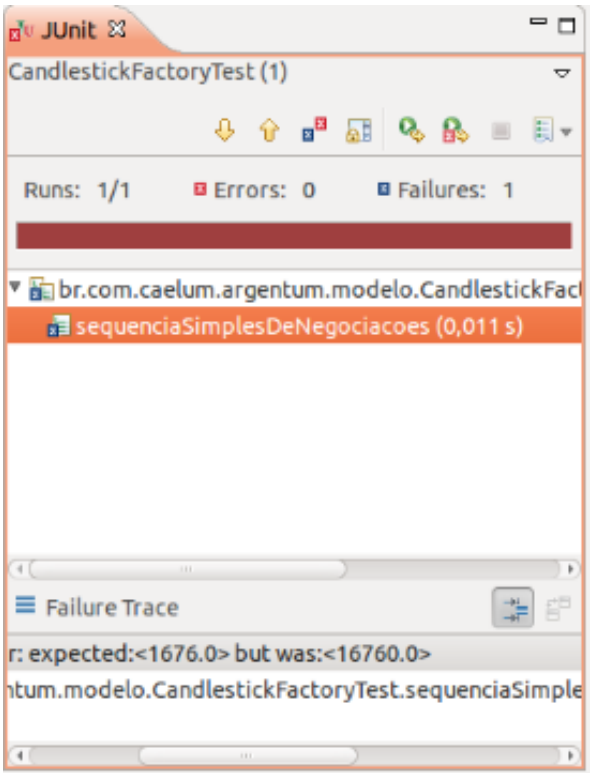
```
public class CandlestickFactoryTest {

  @Test
  public void sequenciaSimplesDeNegociacoes() {
    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);

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

    CandlestickFactory fabrica = new CandlestickFactory();
    Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

    Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
    Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
    Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
    Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
    Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
  }
}
```

Para rodar, use qualquer um dos seguintes atalhos:

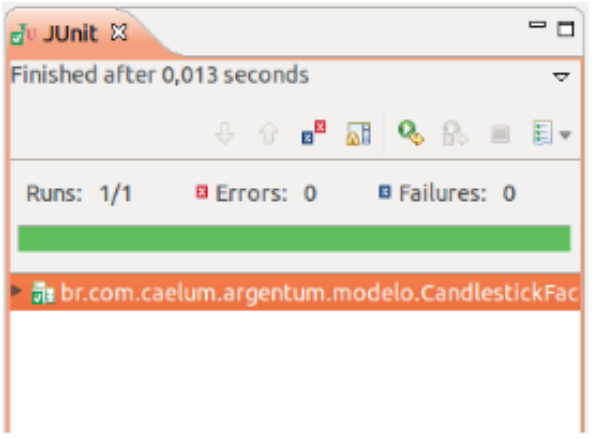
- **ctrl + F11**: roda o que estiver aberto no editor;
- **alt + shift + X** (solte) **T**: roda testes do JUnit.

Não se assuste! **Houve a falha porque o número esperado do volume está errado**

no teste. Repare que o Eclipse já associa a falha para a linha exata da asserção e explica porque falhou:



O número correto é mesmo **16760.0**. Adicione esse zero na classe de teste e rode-o novamente:



É comum digitarmos errado no teste e o teste falhar, por isso, é importante sempre verificar a corretude do teste, também!

3. Vamos **adicionar** outro método de teste à mesma classe `CandlestickFactoryTest`, dessa vez para testar o método no caso de não haver nenhuma negociação:

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

  List<Negociacao> negociacoes = Arrays.asList();
```

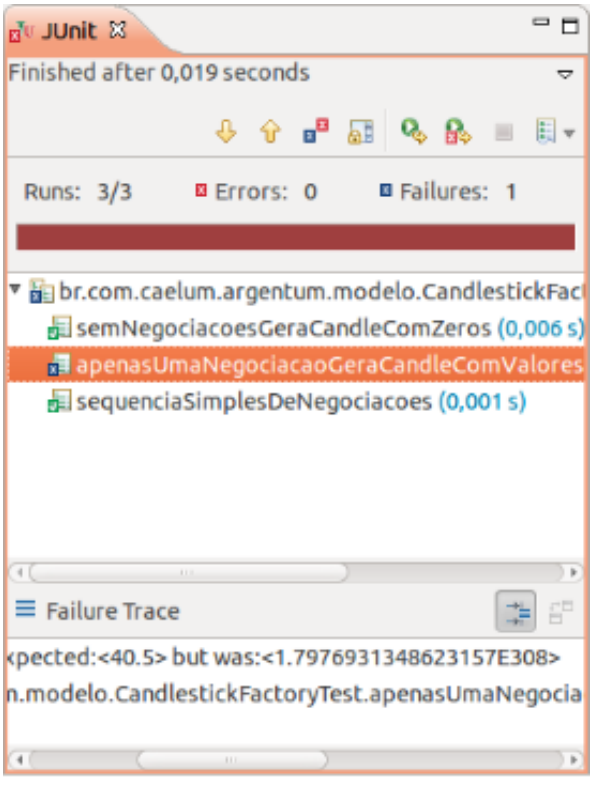
```
  CandlestickFactory fabrica = new CandlestickFactory();
  Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

  Assert.assertEquals(0.0, candle.getVolume(), 0.00001);
}
```

Rode o teste com o mesmo atalho.

4. E, agora, vamos para o que tem apenas uma negociação e estava falhando. Ainda na classe `CandlestickFactoryTest` **adicione** o método: (repare que cada classe de teste possui vários métodos com vários casos diferentes)

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

  Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);

  List<Negociacao> negociacoes = Arrays.asList(negociacao1);

  CandlestickFactory fabrica = new CandlestickFactory();
  Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

  Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
  Assert.assertEquals(40.5, candle.getFechamento(), 0.00001);
  Assert.assertEquals(40.5, candle.getMinimo(), 0.00001);
  Assert.assertEquals(40.5, candle.getMaximo(), 0.00001);
  Assert.assertEquals(4050.0, candle.getVolume(), 0.00001);
}
```

Rode o teste. Repare no erro:



Como consertar?

3.9 - Vale a pena testar classes de modelo?

Faz todo sentido testar classes como a `CandlestickFactory`, já que existe um algoritmo nela, alguma lógica que deve ser executada e há uma grande chance de termos esquecido algum comportamento para casos incomuns - como vimos nos testes anteriores.

Mas as classes de modelo, Negociacao e Candlestick, também precisam ser testadas?

A resposta para essa pergunta é um grande e sonoro **sim**! Apesar de serem classes mais simples elas também têm comportamentos específicos como:

1. as classes `Negociacao` e `Candlestick` devem ser **imutáveis**, isto é, não devemos ser capazes de alterar nenhuma de suas informações depois que o objeto é criado;

2. valores negativos também não deveriam estar presentes nas negociações e candles;

3. se você fez o opcional CandleBuilder, ele não deveria gerar a candle se os valores não tiverem sido preenchidos;

4. etc...

Por essa razão, ainda que sejam classes mais simples, elas merecem ter sua integridade testada - mesmo porque são os objetos que representam nosso modelo de negócios, o coração do sistema que estamos desenvolvendo.

3.10 - Exercícios: novos testes

1. A classe `Negociacao` é realmente imutável?

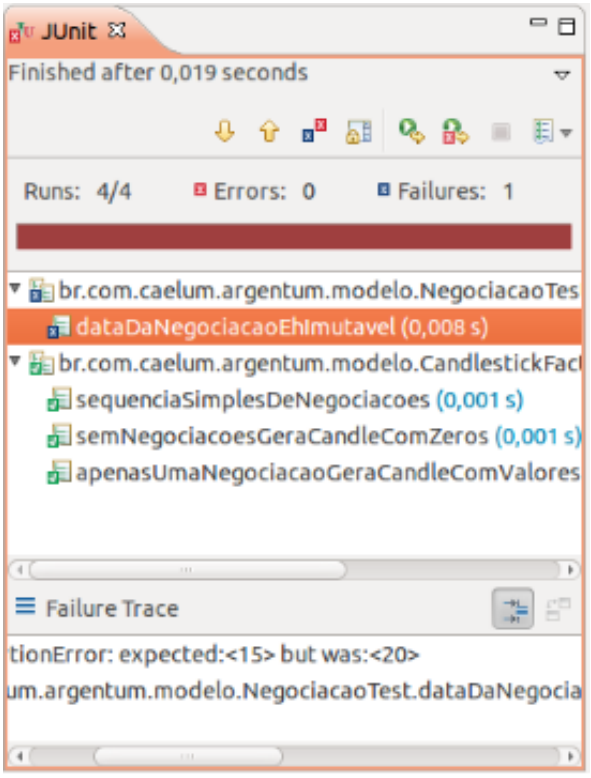
Vamos criar um novo *Unit Test* para a classe `Negociacao`. O processo é o mesmo que fizemos para o teste da `CandlestickFactory`: abra a classe `Negociacao` no editor e faça **Ctrl + N** JUnit Test Case.

Lembre-se de alterar a Source Folder para `src/test/java` e selecionar o JUnit 4.

```
public class NegociacaoTest {

  @Test
  public void dataDaNegociacaoEhImutavel() {
    // se criar um negocio no dia 15...
    Calendar c = Calendar.getInstance();
    c.set(Calendar.DAY_OF_MONTH, 15);
    Negociacao n = new Negociacao(10, 5, c);


    // ainda que eu tente mudar a data para 20...
    n.getData().set(Calendar.DAY_OF_MONTH, 20);

    // ele continua no dia 15.
    Assert.assertEquals(15, n.getData().get(Calendar.DAY_OF_MONTH));
  }
}
```

Você pode rodar esse teste apenas, usando o atalho (alt + shift + X T) ou pode fazer melhor e o que é mais comum, rodar **todos** os testes de unidade de um projeto.

Basta selecionar o projeto na *View* Package Explorer e mandar rodar os testes: para essa ação, o único atalho possível é o `alt + shift + X T`.



Esse teste falha porque devolvemos um objeto mutável através de um *getter*. Deveríamos ter retornado uma cópia desse objeto para nos assegurarmos que o original permanece intacto.

> **Effective Java**
>
> Item 39: Faça cópias defensivas quando necessário.

Basta **alterar** a classe `Negociacao` e utilizar o método `clone` que todos os objetos têm (mas só quem implementa `Cloneable` executará com êxito):

```
public Calendar getData() {
  return (Calendar) this.data.clone();
}
```

Sem `clone`, precisaríamos fazer esse processo na mão. Com `Calendar` é relativamente fácil:

```
public Calendar getData() {
  Calendar copia = Calendar.getInstance();
  copia.setTimeInMillis(this.data.getTimeInMillis());
  return copia;
}
```

Com outras classes, em especial as que tem vários objetos conectados, isso pode ser mais complicado.

Listas e arrays

Esse também é um problema que ocorre muito com coleções e arrays: se você retorna uma *List* que é um atributo seu, qualquer um pode adicionar ou remover um elemento de lá, causando estrago nos seus atributos internos.

Os métodos `Collections.unmodifiableList(List)` e outros ajudam bastante nesse trabalho.

2. Podemos criar uma `Negociacao` com data nula? Por enquanto, podemos, mas não deveríamos. Para que outras partes do meu sistema não se surpreendam mais tarde, vamos impedir que a Negociacao seja criada se sua data estiver nula, isto é, vamos lançar uma exceção.

Mas qual exceção? Vale a pena criar uma nova exceção para isso?

A exceção padrão no Java para cuidar de parâmetros indesejáveis é a `IllegalArgumentException` então, em vez de criar uma nova com a mesma semântica, vamos usá-la para isso.

Effective Java

Item 60: Favoreça o uso das exceções padrões!

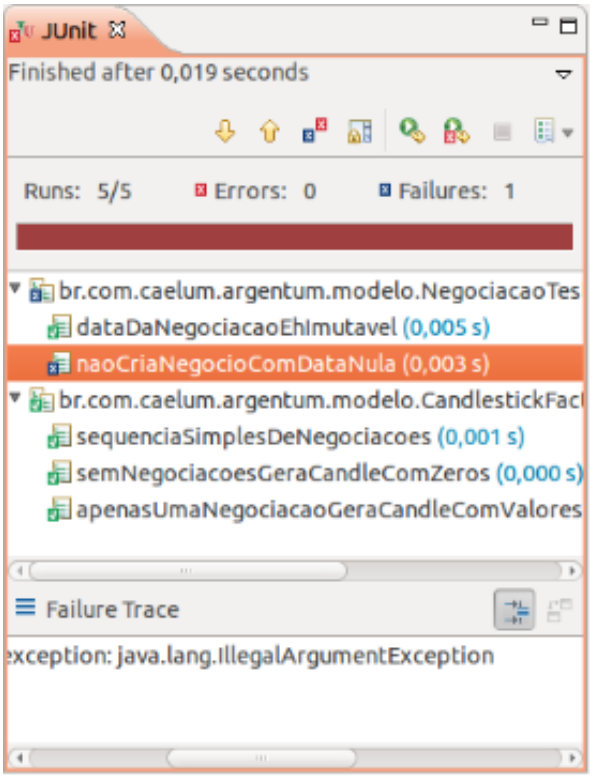
Antes de fazer a modificação na classe, vamos preparar o teste. Mas o que queremos testar? Queremos saber se nossa classe `Negociacao` não permite a criação do objeto e lança uma `IllegalArgumentException` quando passamos `null` no construtor.

Ou seja, *esperamos* que uma exceção aconteça! Para o teste *passar*, ele precisa dar a exceção (parece meio contraditório). É fácil fazer isso com JUnit.

Adicione um novo método `naoCriaNegociacaoComDataNula` na classe `NegociacaoTest`. Repare que agora temos um argumento na anotação `expected=IllegalArgumentException.class`. Isso indica que, para esse teste ser considerado um sucesso, uma exceção deve ser lançada daquele tipo. Caso contrário será uma falha:

```
@Test(expected=IllegalArgumentException.class)
public void naoCriaNegociacaoComDataNula() {
  new Negociacao(10, 5, null);
}
```

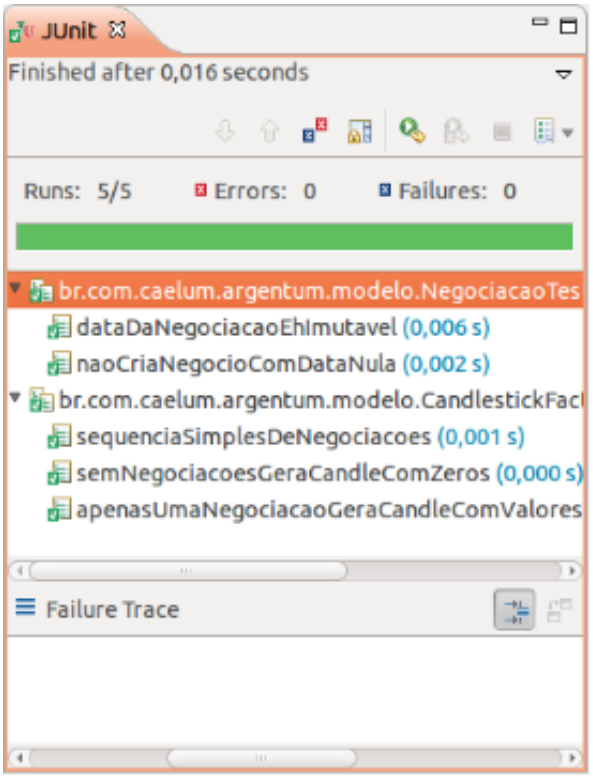
Rode os testes. Barrinha vermelha! Já que ainda não verificamos o argumento na classe `Negociacao` e ainda não lançamos a exceção:



Vamos **alterar** a classe `Negociacao`, para que ela lance a exceção no caso de data nula. No construtor, adicione o seguinte `if`:

```
public Negociacao(double preco, int quantidade, Calendar data) {
  if (data == null) {
    throw new IllegalArgumentException("data nao pode ser nula");
  }
  this.preco = preco;
  this.quantidade = quantidade;
  this.data = data;
}
```

Rode novamente os testes.



3. (opcional) Nosso teste para quando não há negociações na `CandlestickFactory` está verificando apenas se o volume é zero. Ele também poderia verificar que os outros valores dessa candle são zero.

Modifique o método `semNegociacoesGeraCandleComZeros` e adicione os asserts faltantes de abertura, fechamento, mínimo e máximo.

O teste vai parar de passar!

Corrija ele da mesma forma que resolvemos o problema para as variáveis `abertura` e `fechamento`.

4. (opcional) Um Candlestick pode ter preço máximo menor que o preço mínimo? Não deveria.

Crie um novo teste, o `CandlestickTest`, da maneira que fizemos com o `Negociacao`. É boa prática que todos os testes da classe `X` se encontrem em `XTest`.

Dentro dele, crie o `precoMaximoNaoPodeSerMenorQueMinimo` e faça um `new` passando argumentos que quebrem isso. O teste deve esperar pela `IllegalArgumentException`.

A ideia é testar se o construtor de Candlestick faz as validações necessárias. Lembre-se que o construtor recebe como argumento `Candlestick(abertura, fechamento, minimo, maximo, volume, data)`, portanto queremos testar se algo assim gera uma exceção (e deveria gerar):

```
new Candlestick(10, 20, 20, 10, 10000, Calendar.getInstance());
```

5. (opcional) Um `Candlestick` pode ter data nula? Pode ter algum valor negativo?

Teste, verifique o que está errado, altere código para que os testes passem! Pegue o ritmo, essa será sua rotina daqui para a frente.

6. (opcional) Crie mais dois testes na `CandlestickFactoryTest`: o `negociacoesEmOrdemCrescenteDeValor` e `negociacoesEmOrdemDecrescenteDeValor`, que devem fazer o que o próprio nome diz.

Agora eles funcionam?

3.11 - Para saber mais: Import Estático

Algumas vezes, escrevemos classes que contém muitos métodos e atributos estáticos (finais, como constantes). Essas classes são classes utilitárias e precisamos sempre nos referir a elas antes de chamar um método ou utilizar um atributo:

```
import pacote.ClasseComMetodosEstaticos;
class UsandoMetodosEstaticos {
  void metodo() {
    ClasseComMetodosEstaticos.metodo1();
    ClasseComMetodosEstaticos.metodo2();
  }
}
```

Começa a ficar muito chato escrever, toda hora, o nome da classe. Para resolver esse problema, no Java 5.0 foi introduzido o `static import`, que importa métodos e atributos estáticos de qualquer classe. Usando essa nova técnica, você pode importar os métodos do exemplo anterior e usá-los diretamente:

```
import static pacote.ClasseComMetodosEstaticos.*;
class UsandoMetodosEstaticos {
  void metodo() {
    metodo1();
    metodo2();
  }
}
```

Apesar de você ter importado todos os métodos e atributos estáticos da classe `ClasseComMetodosEstaticos`, a classe em si não foi importada e, se você tentasse dar `new`, por exemplo, ele não conseguiria encontrá-la, precisando de um `import` normal à parte.

Um bom exemplo de uso são os métodos e atributos estáticos da classe `Assert` do JUnit:

```
import static org.junit.Assert.*;

class TesteMatematico {

  @Test
  void doisMaisDois() {
    assertEquals(4, 2 + 2);
  }
}
```

Use os imports estáticos dos métodos de `Assert` nos testes de unidade que você escreveu.

3.12 - Mais exercícios opcionais

1. Crie um teste para o `CandleBuilder`. Ele possui um grande erro: se só chamarmos alguns dos métodos, e não todos, ele construirá um Candle inválido, com data nula, ou algum número zerado.

Faça um teste `geracaoDeCandleDeveTerTodosOsDadosNecessarios` que tente isso. O método `geraCandle` deveria lançar outra exception conhecida da biblioteca Java, a `IllegalStateException`, quando invocado antes dos seus outros seis métodos já terem sido.

O teste deve falhar. Corrija-o criando booleans que indicam se cada método *setter* foi invocado, ou utilizando alguma outra forma de verificação.

2. Se você fez os opcionais do primeiro exercício do capítulo anterior (criação do projeto e dos modelos) você tem os métodos `isAlta` e `isBaixa` na classe `Candlestick`. Contudo, temos um comportamento não especificado nesses métodos: e quando o preço de abertura for igual ao de fechamento?

Perguntando para nosso cliente, ele nos informou que, nesse caso, o candle deve ser considerado de alta.

Crie o teste `quandoAberturaIgualFechamentoEhAlta` dentro de `CandlestickTest`, verifique se isso está ocorrendo. Se o teste falhar, faça mudanças no seu código para que a barra volte a ficar verde!

3. O que mais pode ser testado? Testar é viciante, e aumentar o número de testes do nosso sistema começa a virar um hábito divertido e contagioso. Isso não ocorre de imediato, é necessário um tempo para se apaixonar por testes.

3.13 - Discussão em aula: testes são importantes?

CAPÍTULO ANTERIOR:

O modelo da bolsa de valores, datas e objetos imutáveis

PRÓXIMO CAPÍTULO:

Trabalhando com XML

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter