03

Testes automatizados com JUnit

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/laboratorio-java/stages/03/capitulo3.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Nosso código está funcionando corretamente?

Escrevemos uma quantidade razoável de código nos capítulos anteriores, quase meia dúzia de classes. Elas funcionam corretamente? Tudo indica que sim, nós até criamos uma pequena classe com um **main** para verificar isso e fazer as perguntas corretas. Mas se você se lembrar bem, nosso método para testar não lá dos melhores.

O problema dos nossos testes atuais

Sabemos que testar nossas classes é importante. Com eles podemos pegar diversos casos de uso em que não sabemos se o comportamento da `CandlestickFactory` é o que esperamos.Mas os nossos testes, da maneira atual, são ruins de serem usados. Nós temos que rodar classe por classe, observar os resultados e verificar manualmente se era aquilo mesmo que esperávamos. Por isso, nós vamos estudar a biblioteca **JUnit**, que facilitará bastante todo esse processo de criar e rodar nossos testes.

JUnit e os testes de unidade (ou unitários!)

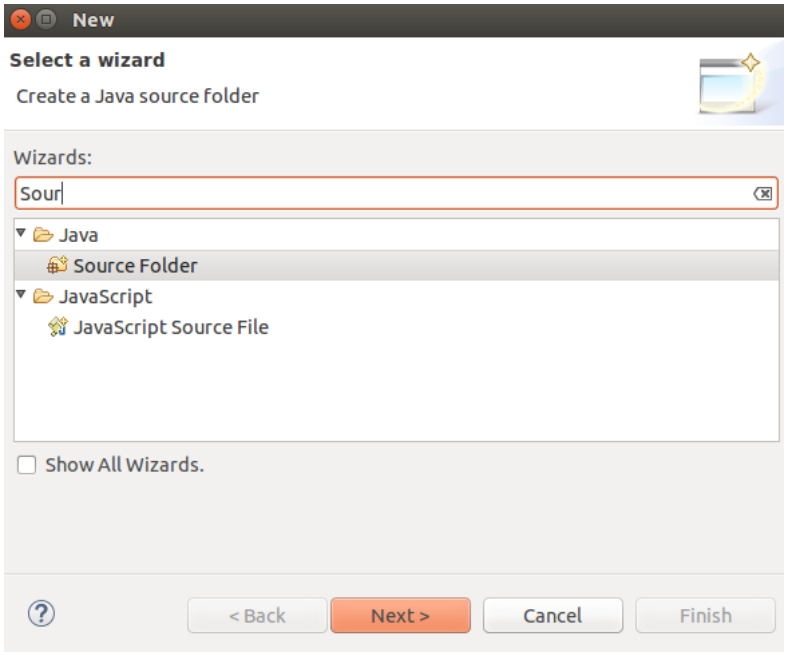
O **JUnit (http://www.junit.org")** é um *framework* muito simples, para facilitar a criação de testes e em especial sua execução. Ele possui alguns métodos que tornam o código de teste bem legível e fácil de fazer as **asserções**.

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.

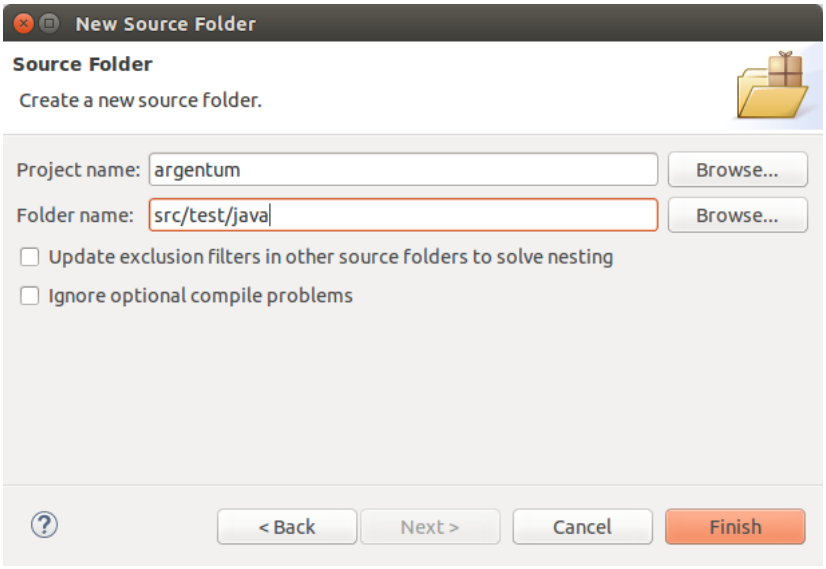
À medida que você mexe no seu código, você roda novamente toda aquela bateria de testes com um comando apenas. Com isso, você ganha a confiança de que novos bugs não estão sendo introduzidos (ou reintroduzidos) conforme você cria novas funcionalidades e conserta antigos bugs. Mais fácil do que ocorre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

Para cada classe, teremos uma classe correspondente (por convenção, com o sufixo `Test`) que conterá 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`).

Então de início, vamos criar esta pasta. Utilize o atalho **CTRL + N** do Eclipse e comece a digitar *Source Folder* até que o filtro encontre a opção correspondente:



Preencha com **src/test/java** e clique em *Finish*:



É nesta pasta que você colocará todos os seus testes!

Então, por exemplo, para a nossa classe `CandlestickFactory`, teremos a `CandlestickFactoryTest`, que criaremos dentro dessa pasta. 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 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 de `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

O tipo double é inerentemente inexato, já que não conseguimos estabelecer sua precisão no Java. Por exemplo, você sabe dizer quanto dá a conta :

```
double cem = 100.0;
double tres = 3.0;
double resultado = ???????? ;
```

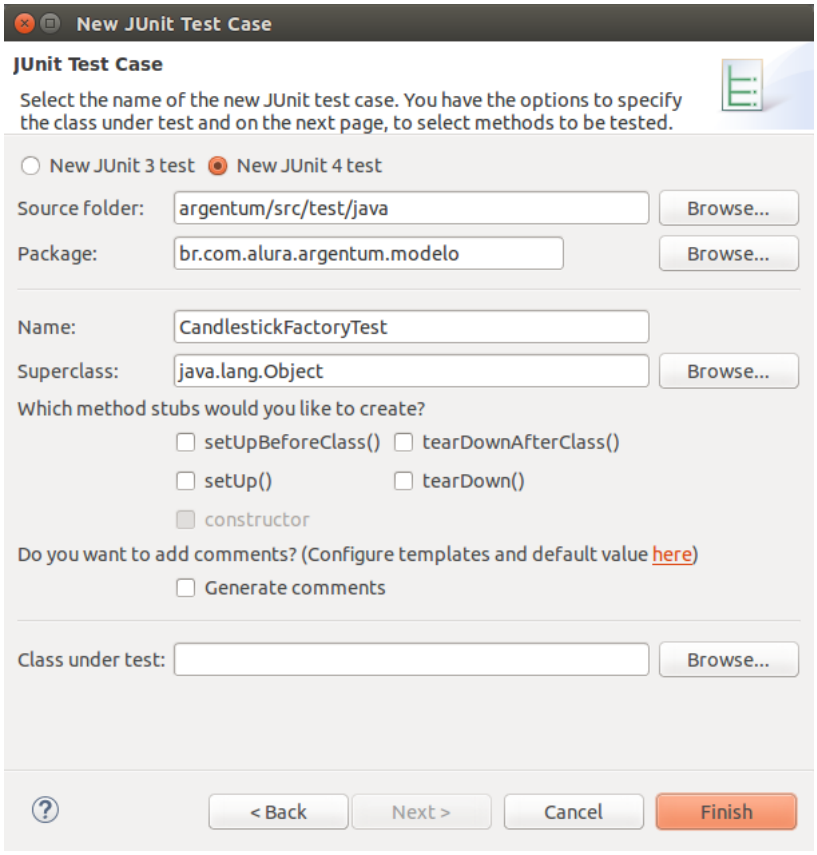
Será que é 33.33 ? Ou 33.3333 ? Ou 33.33333333 ? Não conseguimos controlar a precisão das casas decimais com o tipo double , por isso temos que ter cuidados com o arredondamento nestes casos.Mas diversas vezes, gostaríamos de comparar o `double` esperado e o valor real, sem nos preocupar com diferenças 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.

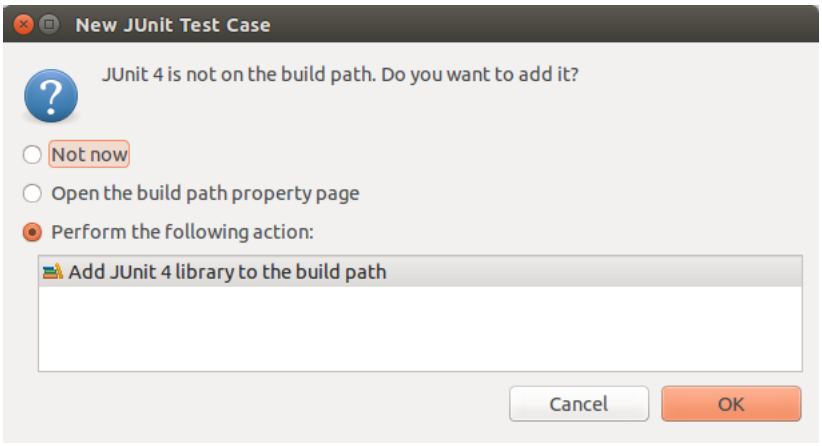
Por exemplo, quando lidamos com dinheiro, o que nos importa são as duas primeiras casas decimais e, portanto, não há problemas se o erro for na quinta casa decimal. Em softwares de engenharia, no entanto, um erro na quarta casa decimal pode ser um grande problema e, portanto, o *delta* deve ser ainda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

Sabendo disso, vamos montar o nosso primeiro teste, que vai fazer com que o nosso método crie uma série de negociações e as adicione numa lista, que utilizaremos para montar um *candlestick*. Parar criar um teste do JUnit, basta utilizar o atalho **CTRL + N** e buscar por *JUnit Test Case*.

Na Janela seguinte, selecione o *Source Folder* como **argentum/src/test/java**. Não esqueça, também, de **selecionar JUnit4**:



Ao clicar em *Finish*, o Eclipse te perguntará se pode adicionar os jars do Junit ao projeto, e você pode confirmar clicando em **OK**:



Agora, começando a escrever o nosso teste, vamos criar as negociações, montar a lista e pedir um novo *candlestick* para a nossa *factory*:

```
public class CandlestickFactoryTest {

    @Test
    public void sequenciaSimplesDeNegociacoes() {

        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);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
```

```
                negociacao3, negociacao4);

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

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 uma exceção é lançada.

Agora vamos pedir ao JUnit , através dos **Asserts** para que verifique se os valores que estão no *candlestick* são os valores que nós esperamos dele:

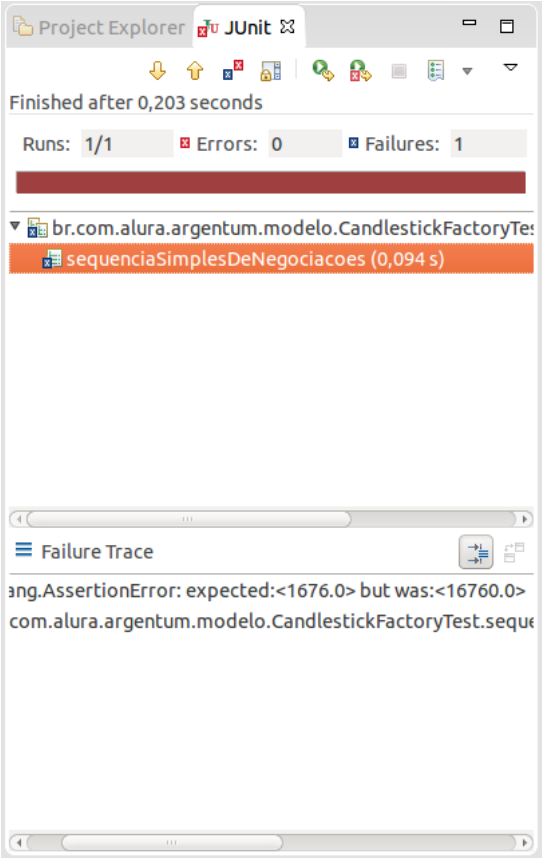
```
public class CandlestickFactoryTest {

    @Test
    public void sequenciaSimplesDeNegociacoes() {

        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);

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

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

        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);
    }
}
```

Com nosso teste criado, para executa-lo use qualquer um dos seguintes atalhos:

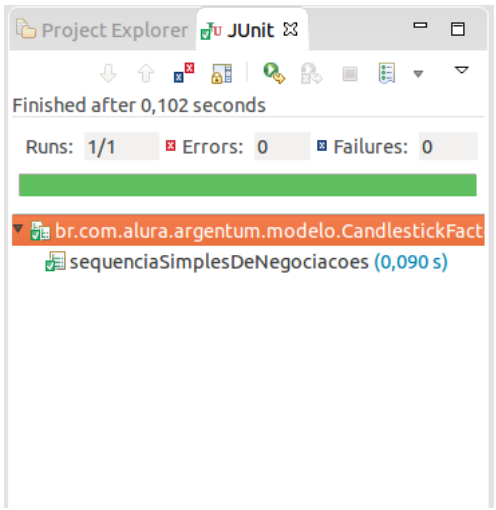
- **CTRL + F11**: Roda o que estiver aberto no editor.
- **ALT + SHIFT + X T**: Roda todos os testes do JUnit.

Ao rodar os teste, o Eclipse abre uma pequena aba na lateral com uma indicação clara de qual teste rodamos, e se ele falhou ou não.

Rodando esse teste , vemos que ele falha! Mas não se assuste. **Houve uma falha por que 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 ele falhou:



O número correto é **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!

O que aprendemos:

- A importância de testar o nosso código.
- O problema de realizar testes manuais.
- O *framework* de testes **JUnit**.
- Como criar corretamente um *Source Folder* para armazenar os testes.
- O que são asserções e como utilizar os métodos estáticos da classe `Assert` .
- A inexatidão do `double` e como o JUnit lida com isso.
- Como criar e executar um JUnit Test Case.

- A verificar a corretude dos nossos testes.