02

# Testando e Debugando sua aplicação

Uma grande parte do desenvolvimento de software se resume a dar manutenção no código existente da aplicação, tanto para adicionar novas funcionalidades, quanto para corrigir algum problema - um bug. Uma das maneiras de facilitar esse trabalho é escrevendo testes automatizados (http://www.caelum.com.br/curso/fj-16-laboratorio-java-testes-xml-design-patterns/). Com eles podemos replicar o bug e garantir que ele nunca volte, verificar e documentar um comportamento (http://www.caelum.com.br/curso/pm-87-agile-praticas-ageis/) qualquer da aplicação ou até ajudar a melhorar o design da sua aplicação (http://blog.caelum.com.br/tdd-e-sua-influencia-no-acoplamento-e-coesao/).

Em Java, a ferramenta de testes automatizados mais utilizada é o JUnit (http://junit.org). Com ele conseguimos criar testes de maneira bem fácil, com classes java normais, rodar os testes e gerar relatórios sobre eles. O Eclipse já vem com suporte nativo ao JUnit, com uma view e alguns wizards dedicados a ele.

No nosso exemplo principal de projeto, uma classe um tanto complicada (e, portanto, mais sujeita a bugs) é a `ImportadorDeGastos`. Por conta disso, ela foi a eleita para começarmos nossos testes automatizados. Por convenção e também para que seja fácil separar classes de teste das principais, é prática comum criar uma *source folder* dedicada aos testes. Use o atalho de criação (**ctrl + N** *Source folder*) para criar o pacote `src/test/java`, seguindo a convenção de nomes do Maven (http://maven.apache.org/).

Para criar um novo teste, poderíamos simplesmente criar uma nova classe na *source folder* de testes, colocá-la no mesmo pacote da classe a ser testada e adicionar o sufixo `Test` a ela. Felizmente, o Eclipse possui um wizard que já cria a classe seguindo as convenções. Para acessá-lo, abra a classe `ImportadorDeGastos` e faça **ctrl + N** *JUnitTestCase*.

A janela de criação já virá preenchida com valores bons: o pacote certo, a classe que está sendo testada e o nome da classe de teste. Só precisamos mudar as duas primeiras informações da janela: troque a *source folder* para `src/test/java` e avise que é um *new JUnit 4 Test*. Ao confirmar a criação, o Eclipse já tomará a iniciativa de adicionar os JARs do JUnit 4 ao *Build path* do projeto: basta dar **enter**.

```
public class ImportadorDeGastosTest {

}
```

Agora que temos a classe, podemos começar a desenvolver o teste mais simples. Primeiramente precisamos dar um nome bom para o teste - ja que essa é a primeira coisa que vai aparecer se o teste falhar. Por exemplo, vamos criar o teste para o seguinte cenário: *"quando o arquivo passado for vazio, o importador deveria retornar uma lista vazia"*. A forma mais fácil de criar um método de teste para JUnit 4 é usando o template `Test` **ctrl+espaço+espaço** e apenas alterar o nome para algo mais descritivo.

```
@Test
public void deveRetornarUmaListaVaziaSeOArquivoPassadoForVazio() {
    |
}
```

Terminaremos de preencher o método passando uma entrada de dados vazia e chamando o importador. Finalmente, temos que verificar que a coleção devolvida não tem nenhum item. Note que, ao usar o template do método de teste, ele já faz o import estático dos métodos da classe `Assert` - faça **ctrl + espaço** dentro do método para ver todos os `assert*` disponíveis. Nosso código de teste vai ficar assim:

```
@Test
public void deveRetornarUmaListaVaziaSeOArquivoPassadoForVazio() throws ParseException {
    ByteArrayInputStream input = new ByteArrayInputStream(new byte[0]);

    ImportadorDeGastos importador = new ImportadorDeGastos();
    Collection<Gasto> lista = importador.importa(input);

    assertTrue("a lista não está vazia", lista.isEmpty());
}
```

Para rodar os testes, podemos usar o atalho **ctrl + F11**, que tenta descobrir o que você quer rodar baseado na classe que está aberta no editor... e acerta, na maior parte das vezes. O problema aparece quando há ambiguidade. Para evitá-la, muitos desenvolvedores preferem usar o atalho de execução **alt + shift + X** seguido da letra **T** de *testes*.

Para quem ainda não se desvinculou do mouse, botão direito > `Run As` > `JUnit Test` faz a mesma coisa.

A view do JUnit se abrirá e veremos a barra de testes verde, já que o teste está passando. Passemos, então, para o próximo teste, com um gasto para ser importado. Podemos nos basear no teste anterior para isso, selecionando todo o código do teste e usando o **ctrl+alt+seta pra baixo** para copiar um bloco. Devemos renomear o teste, por exemplo para `deveRetornarUmGastoDeUmArquivoNoFormatoCorreto` e passar como *input* uma `String` com os dados de um gasto:

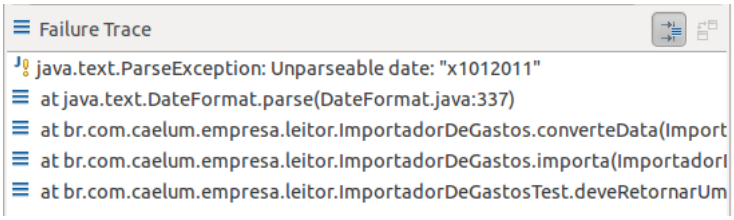
```
@Test
public void deveRetornarUmGastoDeUmArquivoNoFormatoCorreto() throws ParseException {
    String conteudo = "CARTAOx10120110000100000123jbjasbd jbjbbb               22071983\r\n";
    ByteArrayInputStream input = new ByteArrayInputStream(conteudo.getBytes());

    ImportadorDeGastos importador = new ImportadorDeGastos();
    Collection<Gasto> lista = importador.importa(input);

    assertTrue("a lista não está vazia", lista.isEmpty());
}
```

A barra, dessa vez, fica vermelha. Isso acontece quando qualquer um dos testes rodados falha. Um teste pode não passar por causa de uma *Failure* (ou seja, alguma asserção que fizemos falhou) ou por causa de um *Error*, que significa que alguma `Exception` foi lançada durante a execução do teste. Precisamos corrigir todos os testes quebrados e deixar a barra verde, assim sabemos que o nosso trabalho, pelo menos por enquanto, está terminado.

Para corrigir esse teste, perceba o que aparece na seção *Failure Trace* da aba JUnit:




Podemos usar essa stacktrace para ter uma idéia melhor do que está acontecendo. O principal é a mensagem de erro (ou a de falha do teste). Essa mensagem deveria facilitar a vida de quem está recebendo o erro, tanto para se recuperar desse erro, quanto para corrigir eventuais bugs. Se você viu uma mensagem que não te ajudou em nada, tente melhorá-la!

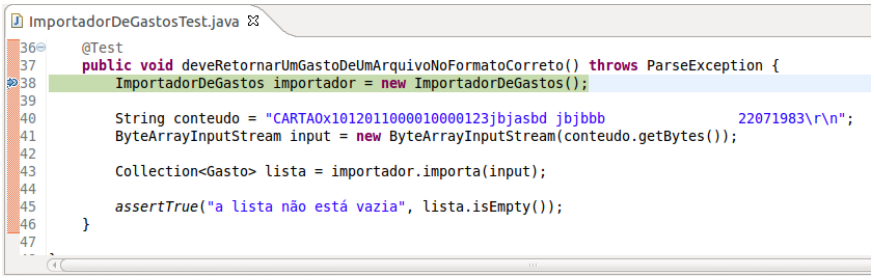
Cada item da stacktrace é clicável (com clique duplo), levando diretamente para o código. Com isso conseguimos ter uma idéia melhor do que causou o erro. Muitas vezes ter o código fonte das bibliotecas usadas no projeto ajuda a compreender melhor um erro estranho no projeto. Não deixe de baixá-las e configurá-las no eclipse, clicando na stacktrace em uma classe de uma biblioteca e em *Attach Source*. Esse trabalho pode ser facilitado pelo Maven (http://maven.apache.org/), que tem a opção de baixar os códigos fontes automaticamente.

Contudo, nem sempre a stacktrace é suficiente para entendermos a causa de um problema, por não sabermos os valores passados ou operações que aconteceram anteriormente. Isso é um forte indício de complexidade no código em que estamos trabalhando, então deveríamos refatorá-lo ou isolar os erros de forma melhor com mensagens de erro mais amigáveis. Mas, antes de corrigir a complexidade, precisamos corrigir o erro e, para isso, podemos usar o bom e velho *Debug*.

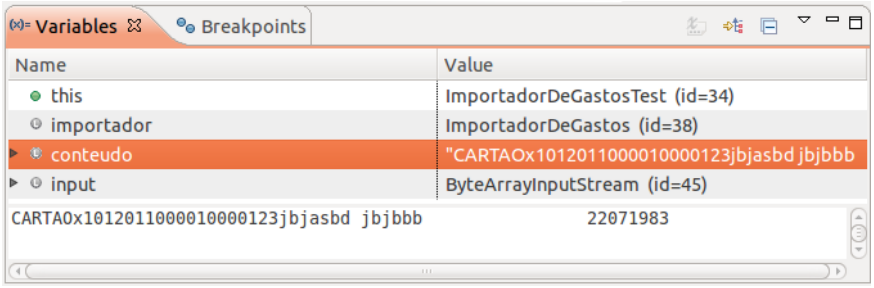
O Eclipse tem uma perspectiva inteiramente dedicada ao *debug*, com várias views que facilitam o processo e as inspeções comuns. Para debugar, precisamos configurar *breakpoints*, que param a execução nas linhas desejadas. Para isso podemos clicar duas vezes na faixa cinza à esquerda do editor na linha desejada, ou usar **ctrl+shift+B** com o cursor na linha. O próximo passo é rodar a aplicação em modo de *debug*, geralmente de uma forma bem similar ao modo normal. Ou seja, rodar uma *Java Application* (método main), um servidor ou um teste usando, por exemplo, o menu *Debug As*.

Para rodar o nosso teste com erro em modo de debug, vamos colocar um *breakpoint* na primeira linha (usando o **ctrl+shift+B**) e então **alt+shift+D T** - note que esse é o atalho de execução em debug de testes. O Eclipse vai perguntar se você quer abrir a perspectiva de *Debug*, que tem as views já configuradas. Vamos confirmar essa mudança (clique em *Remember my decision* se quiser que isso aconteça sempre).

Repare que o editor agora tem uma linha realçada, que indica onde a execução está parada:




Podemos usar o **F6**(*Step Over*) para executar a linha atual e ir para próxima linha do método. Com isso podemos ir até a linha que invoca o método importa e usar a view *Variables* para saber os valores das variáveis criadas.




Outra maneira de ver o valor de uma variável é simplesmente colocar o ponteiro do mouse em cima dela e esperar alguns segundos até um popup com o valor dela aparecer. Do lado da aba de *Variables* temos a de *Breakpoints*, onde podemos gerenciar todos os breakpoints existentes na aplicação. Podemos desabilitá-los no checkbox ao lado, removê-los com a tecla **Delete** ou os ícones  , que removem um e todos os breakpoints, respectivamente. Podemos ainda colocar um breakpoint em uma exceção, com o ícone  . Assim toda vez que essa exceção ocorrer, a execução para logo antes dela ser lançada.

Estamos na linha de execução do método importa mas não queremos ir para a próxima linha e, sim, entrar no método. Para isso usamos o **F5**(*Step Into*), que pula para a primeira linha do próximo método que o java executaria.

Dentro do método importa, sabemos que o erro está na conversão de datas, então podemos colocar o cursor na linha em que a conversão é feita e apertar **ctrl+R** (*Run to Line*), que continua a execução até ela passar pela linha do cursor.

```
double valor = Double.parseDouble(valorString);
int matricula = Integer.parseInt(matriculaString);
Calendar dataNascimento = converteData(df, dataNascTxt);
Calendar dataDespesa = converteData(df, data);
Funcionario funcionario = new Funcionario(nome, matricula,
        dataNascimento);
```

Antes de mandar o método executar de verdade, podemos selecionar partes do código que queremos ver o valor. Por exemplo, ao selecionar o `converteData(df, dataNascTxt)` e então **ctrl + shift + D** para mostrar o valor da expressão. Se quisermos uma versão mais detalhada do que o `toString` do valor, podemos usar o **ctrl + shift + I**. Se usarmos o **ctrl + shift + D** duas vezes em uma expressão, abrimos a view *Display*, onde podemos executar qualquer código. Podemos escrever, por exemplo, `gastos.size();` , selecionar esse código e apertar **ctrl + shift + D** ou **ctrl + shift + I** para ver o resultado.

Podemos, ainda, escrever uma expressão qualquer nessa aba **Displays** (ou mesmo no código real) e monitorar o resultado dela, à medida que o código é executado. Para isso usamos o **ctrl+shift+I** duas vezes. Isso abre a view *Expressions* onde vemos o valor de uma determinada expressão no contexto atual. Podemos adicionar uma expressão qualquer através do *Add new expression*.

A execução está parada antes da chamada do método `converteData` , então para ver o que está acontecendo vamos usar novamente o **F5**. Se olharmos os valores dos parâmetros passados para o método, vemos que a string passada é "22071983", que é uma data válida. Queremos então voltar para o método externo e continuar debugando, para isso usaremos o **F7** (*Step Return*). Então podemos usar o **F6** para ir para a próxima linha e o **F5** para entrar na próxima chamada do `converteData` .

Agora a string passada é a "x1012011", que não é uma data válida, que vai gerar o erro mostrado no teste. Para continuar a execução do método até o final (ou até o próximo breakpoint), podemos usar o **F8** (*Resume*), ou ainda podemos usar o **ctrl+F2** para parar a execução.

Voltemos, então, para o teste para corrigí-lo. Podemos fazer isso na perspectiva de debug, mas o melhor é voltar à perspectiva anterior com o **ctrl+F8**. Substituindo o x pelo zero, podemos rodar o teste novamente, que vai falhar. Dessa vez a falha é porque não mudamos a asserção do teste anterior. O esperado

não é uma lista vazia e, sim, uma lista com um elemento:

```
assertEquals(1, lista.size());
```

Com testes bons e um design simples, conseguimos testar a nossa aplicação sem ter que recorrer para sessões de debug mas, quando precisarmos, o Eclipse tem uma boa infra-estrutura para facilitar esse processo.

### Para saber mais: outra forma de adicionar bibliotecas ao projeto

Para começar a escrever os testes, precisamos que o *JUnit* esteja no classpath. Da forma como fizemos nessa aula, o próprio Eclipse adicionou a biblioteca ao *classpath*, mas essa não é a única forma de fazê-lo. Também é uma possibilidade baixar o JAR do JUnit ou usar o que já vem junto com o Eclipse, mas configurá-lo através do Build Path.

Essa configuração é feita nas preferências do projeto (botão direito no projeto > *Build path* > *Configure build path*). Vamos trabalhar com a aba *Libraries*. Nela, conseguimos adicionar bibliotecas ao classpath da aplicação, usando os botões do lado direito. Os mais importantes são *Add JAR* e *Add External JAR* para jars dentro e fora da workspace, e o *Add Library*, para adicionar bibliotecas já configuradas dentro do eclipse. Para adicionar o JUnit basta clicar em *Add Library* e selecionar *JUnit*. Na próxima tela, selecione a versão *JUnit 4*, que é baseada em anotações e bem mais fácil de usar.