

# Automação dos vários estágios de testes em aplicações Java

Matheus Fachine de Moura<sup>1</sup>

<sup>1</sup>Centro de Ciências Tecnológicas – Universidade de Fortaleza (Unifor)

CEP 60.811-905 – Fortaleza-CE – Brasil

matheusfmoura@gmail.com

**Resumo.** *Os sistemas atuais estão cada vez mais complexos, por isso executar testes automatizados está cada vez mais importante para o sucesso da construção de um programa. Automatizar testes é trabalhoso, mas tem seus benefícios desde que se tenha uma equipe experiente e que disponha de ferramentas corretas. Este artigo busca descrever o conceito de testes de software, tal qual unitário, de integração e de aceitação, mostrando na prática exemplos escritos na linguagem Java, concluindo com a importância da adoção deste tipo de metodologia em projetos.*

## 1. Introdução

Myers (2004) afirma que o teste de *software* é um processo, ou uma série de processos, projetado para se certificar a respeito do código de computador, além de fazer o que foi designado a cumprir, e sem realizar nada que não seja intencional. O *Software* deve ser previsível e consistente, não oferecendo surpresa para os usuários. Por ser muito trabalhoso de escrever e também por não saberem o real custo benefício que proporcionam em um projeto, muitas empresas no mercado não adotam esse tipo de teste. Fowler (2001) diz que nas empresas, as pessoas envolvidas em um projeto, perdem longas noites e até finais de semana tentando desfazer erros que estão fora do planejamento. O autor ainda afirma que, ironicamente, essas longas horas não levam a uma grande produtividade.

Imaginem um sistema de grande porte que possui várias funcionalidades. E uma delas é implementada, testada (por humano) e entregue ao cliente. Futuramente outra funcionalidade é instalada e afeta diretamente o fluxo da primeira. Um erro em outra parte do sistema poderá acontecer devido a essa mudança e passar despercebido pela equipe de desenvolvimento, sendo reportado quando o mesmo já estiver em produção. Segundo Leal (2009), o “Teste humano é menos eficiente, pois dificilmente uma pessoa conseguiria testar exaustivamente um sistema de forma que não restassem possibilidades possíveis para falha”. Daí o fato disso ser mais vantajoso quando automatizado.

Segundo *Selenium* (2011), automatizar teste significa usar uma ferramenta de software para executar tentativas repetitivas em uma aplicação a ser testada. É importante frisar que nem sempre é necessário automatizá-los. Completando o que *Selenium* (2011) diz, se uma aplicação tem um prazo muito apertado, e o projeto está atualmente sem cobertura, então o teste manual é a melhor solução. Neste artigo é descrita a definição de três estágios de testes (Unitário, Aceitação, Integração), mostrando em cada seção, na prática, alguns exemplos. Os exemplos deste artigo são mostrados através dos frameworks *JUnit* [4] e *Mockito* [2] para testes unitários, *DBUnit* [14] para testes de integração e *Selenium* [11] para testes de aceitação.

## **2. Teste Unitário**

### **2.1 Conceitos**

Segundo Meyers (2004), o teste de unidade (ou teste de módulo) é um processo que testa individualmente subprogramas, sub-rotinas ou procedimentos em um programa. Isto é, em vez de inicialmente testar o programa como um todo, o teste foca primeiramente na construção de blocos menores desse programa. Como o nome já diz, ele testa uma unidade de programa, seja ela uma função individual ou um procedimento. A ideia é de que se teste uma unidade independente das demais partes do sistema, o que possibilita ao programador fazer isso com cada módulo, isoladamente. No paradigma orientado a objetos, essa menor unidade pode ser entendida como um objeto ou mesmo um comportamento do mesmo, reagindo a um estímulo (métodos da classe).

Existem três motivações para que se realizem testes unitários. Em primeiro lugar, porque são formas de gerenciar elementos combinados de testes, já que são focados em menores partes de um programa. Em segundo, pelo fato de facilitarem a tarefa de depuração, uma vez que o erro, quando descoberto, encontra-se em um módulo particular. Finalmente, por introduzirem um paralelismo no processo de teste, tendo a oportunidade de testar vários módulos simultaneamente (Meyers, 2004).

## 2.2 Exemplo de Teste Unitário com *JUnit*

Para testes unitários na plataforma Java, um framework bastante conhecido e utilizado é o *JUnit*. *JUnit* é um simples framework de código aberto para escrever e executar testes repetitivos. Exemplificando o *JUnit*, pode-se citar uma conversão de moedas. Um usuário digita determinado valor que invocará um método de conversão. Esta é uma pequena unidade no sistema que pode ser testada isoladamente.

```
1 package br.com.matheusfechine.exemplo;
2
3 public class Conversor {
4
5     private Double taxaDeCambio;
6
7     public Conversor(Double taxaDeCambio) throws Exception{
8         if(taxaDeCambio <= 0){
9             throw new Exception("Erro de conversao");
10        }
11        this.taxaDeCambio = taxaDeCambio;
12    }
13
14    public Double converterParaReal(Double valorDollar) throws Exception {
15        return valorDollar * taxaDeCambio;
16    }
17 }
18
```

Figura 1. Classe Conversor

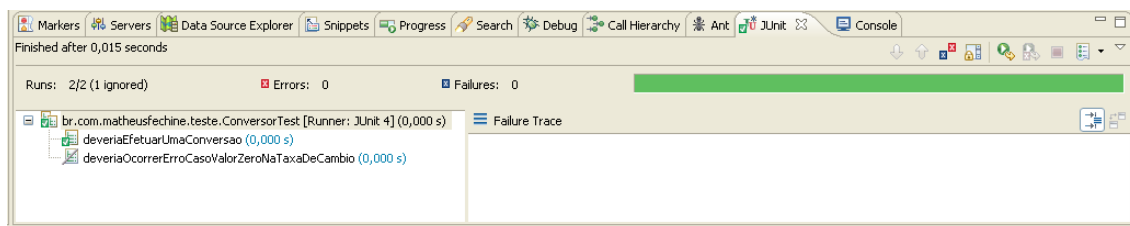
```

1 package br.com.matheusfechine.teste;
2 import org.junit.Test;
3 import static org.junit.Assert.*;
4 import br.com.matheusfechine.exemplo.Conversor;
5
6 public class ConversorTest {
7
8     private Conversor conversor;
9
10    @Test
11    public void deveriaEfetuarUmaConversao() throws Exception {
12        conversor = new Conversor(1.5D);
13        assertEquals(15, conversor.converterParaReal(10D), 0.0001);
14    }
15
16    @Test(expected= Exception.class)
17    public void deveriaOcorrerErroCasoValorZeroNaTaxaDeCambio() throws Exception{
18        conversor = new Conversor(0D);
19    }
20 }

```

**Figura 2. Classe de Teste Unitário**

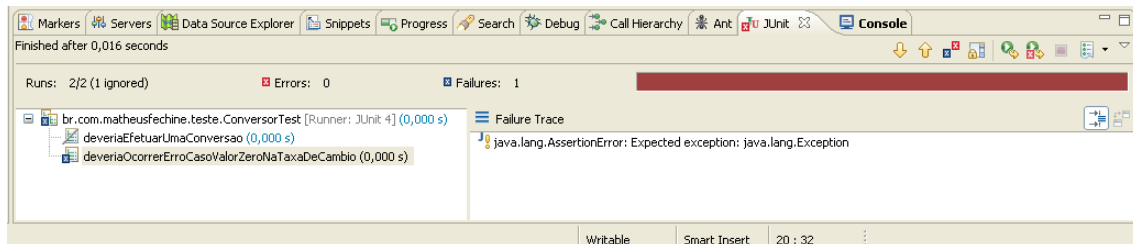
A figura 1 mostra a classe de negócio e a figura 2, o teste unitário. Podemos notar a existência, através da figura 2, de uma anotação *@Test*, do *JUnit*, demonstrando que esses dois métodos são de testes. O que o primeiro método espera é que, caso haja uma taxa de câmbio no valor de ‘1,5’, enviada no construtor, seja assegurado o retorno de ‘15’ quando o valor 10 for usado como parâmetro, diante da execução do método *converterParaReal* da classe *Conversor*. A classe *Assert* contém um método estático denominado *assertEquals* que afirma tal sentença. Se tudo ocorrer como esperado, então o teste passará a ser sinalizado pela cor verde.



**Figura 3. Teste realizado com sucesso**

O teste do método *deveriaOcorrerErroCasoValorZeroNaTaxaDeCambio* é mais simples que o anterior. Notemos que no construtor da classe *Conversor* existe uma verificação. E caso seja enviado qualquer valor menor ou igual a zero, o mesmo lança uma exceção. Supomos que uma regra de negócio diz que não é possível fazer qualquer conversão com taxa de câmbio menor ou igual a zero. A anotação *@Test* recebe como

parâmetro uma propriedade *expected*, sendo atribuído a ela um *Exception*. Isso significa que este método está esperando uma exceção ser lançada, e tal condição só irá ocorrer caso seja enviado por parâmetro no construtor um valor menor ou igual a zero. Para efeito de falha nesse tipo de teste, foi enviado o valor '1' no construtor. A figura 4 retrata-o falhando através do sinal vermelho.



**Figura 4. Teste falho.**

Os objetos *Mock* também são utilizados em testes unitários. Segundo Freeman (2000), o objeto *Mock* é uma implementação substituta para emular código. Ele deve ser mais simples do que o código real, e não duplicar a sua implementação, permitindo que seja configurado o estado privado para ajudar nos testes. A ênfase nessa implementação *Mock* é em sua simplicidade absoluta, ao invés de complexidade. O autor completa dizendo que escrever uma classe *Mock* fornece o mínimo de comportamento que esperamos de nosso banco de dados. É exibido, na sequência, um exemplo real de como se pode efetuar *Mock* em uma aplicação.

```
public class VerificaNFTeste{

    @Test
    public void mockando() {
        RepositorioDeNFs dao = mock(RepositorioDeNFs.class);
        List<NotaFiscal> nfs = new ArrayList<NotaFiscal>();
        nfs.add(new NotaFiscal("Cliente 1", 4000.00D));
        when(dao.pegarTodos()).thenReturn(nfs);
        Assert.assertEquals("Cliente 1", dao.pegarTodos().get(0).getNome());
    }
}
```

**Figura 5. Classe VerificaNFTeste.java.**

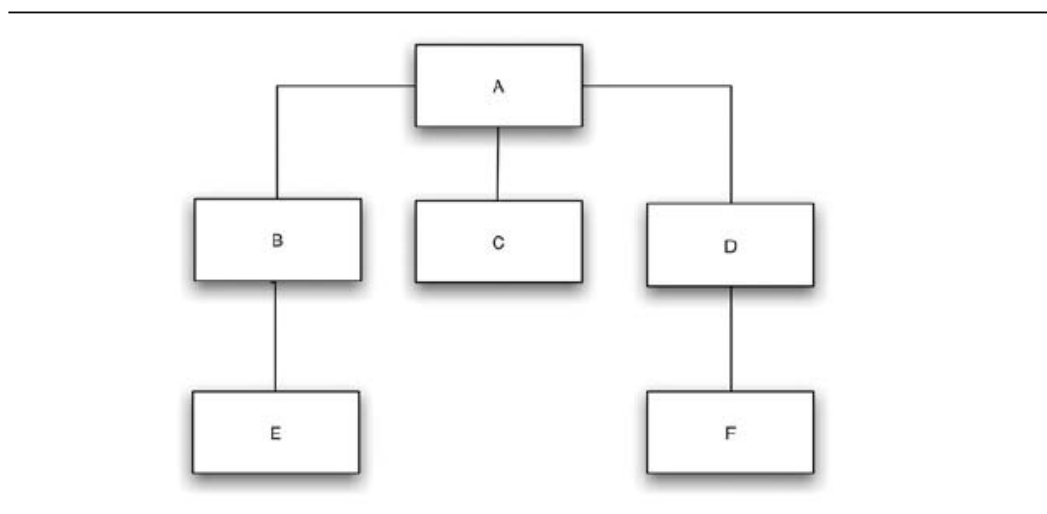
A figura 4 mostra uma classe teste com objeto *Mock*. Este objeto é criado para simular o comportamento da classe a ser testada. Por exemplo, existe um serviço DAO,

*RepositorioDeNFs*, que efetua uma pesquisa no banco de dados, e que retorna uma lista de *NotaFiscal*. Não é de interesse do teste unitário o acesso ao banco para buscar tal informação, portanto essa classe é “mockada” para simular o serviço. Para isso, é utilizada uma API denominada *Mockito*. *Mockito* é um *framework open source* para testes de *Mocks* na plataforma Java, que possui métodos estáticos de fácil entendimento e bem intuitivos. Analisando a linha que contém a sentença intitulada como *when(dao.pegarTodos()).thenReturn(nfs)*, do *Mockito*, o mesmo executa o comando que é capaz de dizer quando (*when*) o método *pegarTodos()* for invocado no serviço *Dao*. A partir daí, pode retornar a lista de *NotaFiscal* falsa criada (*thenReturn*). O método *assertEquals* apenas verifica se o atributo “nome” do primeiro elemento da lista é capaz de retornar o valor “Cliente 1”.

### **3. Teste Integração**

#### **3.1 Conceitos**

Teste de integração é o complemento do teste de unidade em que suas partes são colocadas para trabalhar juntas e verificar se ocorrem erros. Supomos então que seis unidades foram implementadas e testadas isoladamente, e as mesmas foram combinadas em um componente. A ideia desse tipo de teste é combinar tais unidades, testá-las entre si e identificar problemas que ocorram também entre elas, ou seja, testar fluxos em um programa.



**Figura 6. Módulos de um programa.**

A figura 5 mostra um exemplo de seis unidades isoladas que, em conjunto, se tornam um componente. No caso: um módulo A chama um módulo B, que por sua vez chama um C. É o que fizemos para exemplificar e explicar como efetuar esse tipo de teste, em que um controle de módulo A se comunica com outro de módulo B, e que este pode ser citado como um serviço qualquer: *DAO*, *JMS*, *WebService* etc. Diferente do *Mock*, em que os dados testados são falsos, esse teste é real, com dados verdadeiros obtidos nos serviços. É possível então ter acesso ao serviço do Banco de Dados, e o teste avalia se o que será inserido, deletado ou modificado está de acordo com o caso de teste.

### **3.3 Exemplo de Teste de Integração com *JUnit* e *DBUnit***

Mostrando na prática um teste de integração com acesso a determinado serviço em um Banco de Dados, utilizamos um arquivo XML para persistência dos dados através do *DBUnit*. Segundo DbUnit (2010), o *DbUnit* é uma extensão do *JUnit* orientada para banco de dados em que guia projetos e coloca este conjunto de registros em um estado conhecido entre as execuções de teste. É uma excelente maneira de evitar problemas que podem ocorrer quando um caso de teste corrompe o banco de dados e faz com que falhem.

Sempre que o teste for iniciado, esses dados serão gravados em tabelas reais. Como exemplo, temos uma tabela “Nota Fiscal” com código, nome do produto e valor, conforme a figura 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <notafiscal id="1" nome_produto="Borracha" valor="5.00" />
</dataset>
```

Figura 7. Arquivo XML que persiste no banco de dados.

```
public class NotaFiscalTeste extends DatabaseTestCase{
    private FlatXmlDataSetBuilder loadedDataSet;

    protected IDatabaseConnection getConnection() throws Exception{
        Class.forName("com.mysql.jdbc.Driver");
        Connection jdbcConnection = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/teste", "root", "root");
        return new DatabaseConnection(jdbcConnection);
    }

    protected IDataSet getDataSet() throws Exception{
        loadedDataSet = new FlatXmlDataSetBuilder();
        return loadedDataSet.build(new FileInputStream(
            "src/br/com/matheusfechine/xml/dataset.xml"));
    }

    public void testeNotaFiscal() throws Exception{
        DatabaseOperation.CLEAN_INSERT.execute(getConnection(), getDataSet());
        NotaFiscal nota = new NotaFiscal(1L, "Nota Teste", 1D);
        NotaFiscal notaFiscal = new NotaFiscalServiceImpl().getNotaFiscal(nota);
        DatabaseOperation.DELETE_ALL.execute(getConnection(), getDataSet());
        assertTrue(notaFiscal.getNome().equals("Borracha"));
    }
}
```

Figura 8. Classe de Teste com DBUnit.

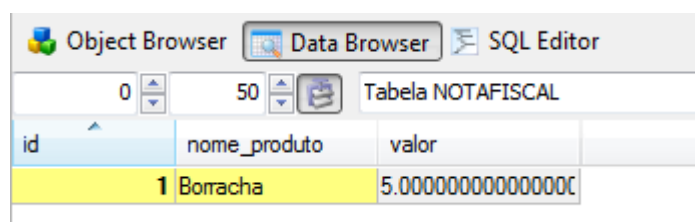
Analisando a figura 7, vemos que representa uma classe de teste simples que, em resumo, insere um registro no banco, verifica se ele foi persistido e depois o remove de modo a evitar que estes dados deixem “sujeira” na base.

A classe *NotaFiscalTeste*, que é uma extensão de *DatabaseTestCase*, uma classe do próprio *DBUnit*, que exige a implementação dos métodos *getConnection()*, retorna uma *IDatabaseConnection* (interface que representa uma conexão *DbUnit* para um



banco de dados), além de *getDataSet()*, que retorna uma *IDataSet* (interface que representa uma coleção de tabelas). Ambas citadas são também do *DBUnit*.

O método *testeNotaFiscal()* é quem realmente testará a determinada funcionalidade do sistema. A Classe *DatabaseOperation* é abstrata e representa uma operação realizada no banco de dados antes e após cada teste (DBUnit, 2010). A operação *CLEAN\_INSERT* realiza uma deleção de todos os dados contidos no arquivo XML, e em seguida por uma operação de inserção dos mesmos.



| id | nome_produto | valor              |
|----|--------------|--------------------|
| 1  | Borracha     | 5.0000000000000000 |

**Figura 9. Dados persistidos na base de dados com *DBUnit*.**

Após a execução desse comando, os dados contentes na figura 6 são persistidos na tabela *notafiscal* conforme visto na figura 8. O comando seguinte é a chamada ao método *getNotaFiscal()* no serviço, de modo a obter a *notafiscal* recentemente inserida. Por fim, é verificado se o atributo nome é obtido e está de acordo com o esperado.

#### **4. Teste de Aceitação**

Meyers (2004) define teste de aceitação como o processo de comparar o programa aos seus requisitos iniciais e as necessidades atuais de seus usuários finais. É um tipo incomum de teste em que geralmente é realizado pelo cliente do programa ou usuário final e, normalmente, não é considerada a responsabilidade da organização de desenvolvimento. Esse tipo de teste geralmente ocorre quando o sistema já está finalizado a ponto de ser implantado no cliente em que o mesmo também pode ter a participação direta, seja no planejamento ou na realização desta atividade. É possível

também automatizar, e existem ferramentas, como o *Selenium*, que gravam passos em tela de modo a simular ações que um usuário pode fazer.

Segundo definição de Selenium (2011), o *Selenium* é um conjunto de diferentes ferramentas de software, cada uma com uma abordagem diferente para apoiar a automação de teste para diferentes tipos de problema. Ele executa operações que são altamente flexíveis, permitindo muitas opções para a localização de elementos na interface e comparando os resultados dos testes esperados durante o comportamento de uma aplicação real. Uma das principais características do *Selenium* é o apoio para a execução de testes de um browser sobre múltiplas plataformas. Para mostrarmos na prática como funciona essa ferramenta, damos um exemplo simples em uma aplicação. Primeiramente são necessários dois componentes para geração dos testes:

1. *Selenium RC*.
2. *Selenium IDE*.

*Selenium RC (Remote Controller)* é um servidor, escrito em Java, que recebe chamadas “HTTP” vindas dos testes unitários (*JUnit*) e executa os testes. *Selenium IDE (Integrated Development Environment)* é uma ferramenta de prototipagem para a construção de scripts de teste. É um “plugin” do *Mozilla Firefox* que fornece uma ferramenta de interface fácil de usar para o desenvolvimento de testes automatizados. O *Selenium IDE* tem um recurso de gravação que registra as ações do usuário como elas são realizadas e, em seguida, é capaz de exportá-las como um script reutilizável em várias linguagens de programação, e que pode ser executado mais tarde (Selenium, 2011).

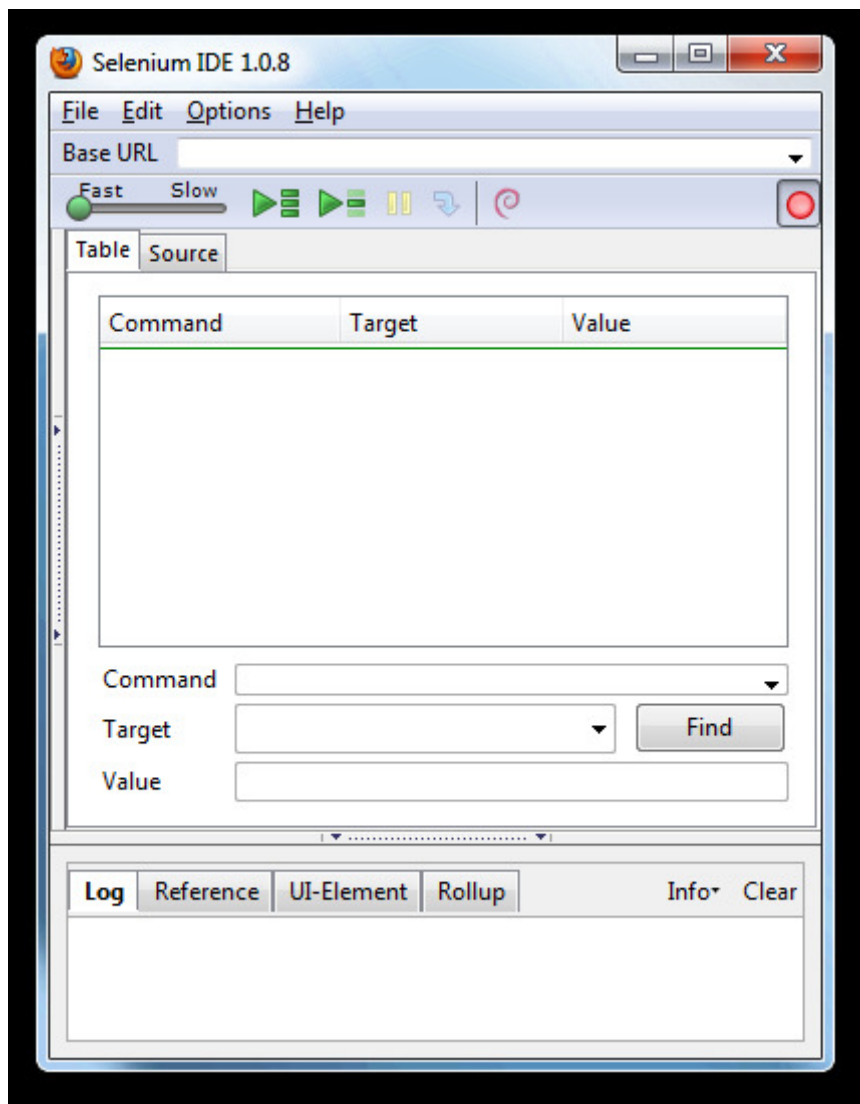


Figura 10. Selenium IDE para navegador Mozilla Firefox.

#### 4.1 Efetuando teste Selenium

Existem alguns tipos de testes possíveis de serem feitos pelo *Selenium*:

1. Teste de componentes estáticos.
2. Teste de links.
3. Teste de funções.
4. Teste de elementos dinâmicos.
5. Teste *Ajax*.

Como o próprio nome já diz, testes de componentes estáticos são elementos imutáveis de uma página. Como exemplo, nós os observamos no título de uma página *HTML* que é o esperado, ou se a página exibe o número do telefone da empresa etc. O teste de links pode ser efetuado para saber se o link de uma página está direcionando para o endereço esperado.

Já o teste de funções é um dos mais complexos, porém o mais importante. Nele, é testada uma função específica dentro de uma aplicação, requerendo algum tipo de entrada do usuário e retornando algum tipo de resultado. Alguns exemplos típicos são os de *login* de usuário, registro em um site, dentre outros. Os testes de função tipicamente espelham os cenários usados pelo usuário para especificar as características, design ou aplicação (Selenium, 2011).

O teste de elemento dinâmico efetua a validação do nome do identificador de um componente de uma página que é modificado dinamicamente pelo servidor, por exemplo:

```
<input type= "checkbox" value= "true" id= "addForm:_ID74:_ID75:0:_ID79:0:
checkbox" />
```

**Figura 11. Componente com nome de identificador modificado dinamicamente.**

O teste Ajax efetua uma verificação de ações isoladas de uma página solicitando requisição no servidor, que responde a ela sem a necessidade de carregá-la por completo.

## **4.2 Executando um teste Selenium**

Para exemplificar um teste de aceitação simples e automatizado, utilizamos o teste estático, na prática, em que é verificado se um elemento de *comboBox* está selecionado. É utilizado o *Selenium RC* para execução via código Java e *Selenium IDE* para gravar os passos da ação do usuário.

```

<html>
<head>
<title>Selenium e JUnit HowTo</title>
</head>
<body>
Size:
<select name="size">
  <option id="x-large">X-Large</option>
  <option id="large">Large</option>
  <option id="medium" selected>Medium</option>
  <option id="small">Small</option>
  <option id="tiny">Tiny</option>
</select>
</body>
</html>

```

Figura 12. Código de uma página HTML para uma simples comboBox.

```

private static DefaultSelenium selenium;

@BeforeClass
public static void setup() {
    String url = "http://localhost:8080";
    selenium = new DefaultSelenium("localhost", 4444, "firefox /usr/lib/firefox/firefox-bin", url);
    selenium.start();
}

@AfterClass
public static void tearDown() {
    selenium.stop();
}

@Test
public void testSelectedIdOfSizeComboBox() {
    selenium.open("/test/index.jsp");
    assertEquals("medium", selenium.getSelectedId("size"));
}

```

Figura 13. Demonstração de uma classe Java para execução de um teste com o *Selenium*.

A figura 12 mostra o teste de um método chamado *testSelectedIdOfSizeComboBox()*. Na montagem do HTML, a *comboBox* recebe o nome “size”. Primeiramente é executado um comando para o *Selenium RC*, para que este abra a página *teste.jsp* através do comando *selenium.open()*. A linha contendo “*assertEquals("medium", selenium.getSelectedId("size"))*,” está assegurando que o item com o nome “medium” esteja sendo selecionado no componente com o nome “size”. Com o *Selenium IDE*, é possível gerar automaticamente um código Java, por exemplo, ao realizar as operações que um usuário deseja testar. Após isso, basta adicionar as *assertions* desejadas em código Java via *Selenium RC*.

## 5. Conclusão

Neste artigo foram demonstradas as definições de teste unitário, de integração e de aceitação em projetos Java através de ferramentas *open-source*, efetuando na prática alguns exemplos de como executar esses testes em aplicações executadas em ambiente *web*.

Escrever testes é um trabalho árduo e necessita que a equipe de desenvolvimento tenha algum conhecimento nesta metodologia. E, além disso, é preciso ter o domínio de todas as ferramentas disponíveis, assim como saber o que realmente deverá ser testado. Há também a dificuldade em executar um teste bem feito, ou seja, aquele que prevê todas as possibilidades que podem causar erro em uma determinada funcionalidade como, por exemplo, saber se o tamanho de um texto é satisfatório de acordo com o limite do campo correspondente em que este será gravado ou todas as condições de uma exceção disparada.

Para um projeto em que o escopo constantemente é modificado, manter os testes coerentes com as mudanças exige certo esforço, pois estes provavelmente começarão a falhar e precisarão ser refeitos. Testes fazem com que a implementação de alterações no código sejam feitas com maior segurança.

Concluimos que o teste unitário é importante principalmente em um projeto em que a equipe de desenvolvimento é grande e a manutenção no código se torna constante. É muito comum o desenvolvedor que alterou uma parte do código nem se lembrar o porquê desta mudança. Testando cada unidade (método, por exemplo), serviço e interfaces gráficas, fica fácil de detectar o que e onde está falhando.

O teste de integração é importante para verificar se a conversa entre as camadas, isso inclui controle e *DAO*, estão trazendo os resultados esperados. Prever quando um resultado não é esperado também é importante.

Já um teste de aceitação, onde foi mostrado um exemplo simples utilizando o *Selenium* e focando mais a parte gráfica e usabilidade, é interessante para prever se a navegação de uma determinada funcionalidade está de acordo com o que o usuário (cliente) deseja e espera que aconteça. Teste é custoso, porém tem seus benefícios. Meyers (2004) define que, ao automatizar os testes, é possível ganhar confiança no

código que irá satisfazer as especificações que o cliente deseja. Além disso, implementar projetos fornece confiança em “refatorar” o código para melhorar o desempenho, sem se preocupar com a quebra da especificação.

## Referências

- [1] Testes de aceitação com o Selenium. Disponível em <<http://www.infoblogs.com.br/view.action?contentId=2030&Testes-de-aceitacao-com-o-Selenium.html>>. Acesso em 03/06/2011
- [2] WebSite do framework Mockito. Disponível em <<http://code.google.com/p/mockito/>>. Acesso em 05/08/2011
- [3] Aula 9 - Teste de Software - Parte 4. Disponível em <<http://pt.scribd.com/doc/51435639/3/Teste-de-Integracao>>
- [4] JUnit - Implementando testes unitários em Java – Parte I. Disponível em <<http://www.devmedia.com.br/articles/viewcomp.asp?comp=1432>>. Acesso em 10/06/2011
- [5] Conceitos: Teste de Aceitação. Disponível em <[http://www.wthree.com/rup/process/workflow/test/co\\_accte.htm](http://www.wthree.com/rup/process/workflow/test/co_accte.htm)>. Acesso em 15/06/2011
- [6] Desenvolvimento Orientado a Testes. Disponível em <<http://www.improveit.com.br/xp/praticas/tdd>>. Acesso em 18/06/2011
- [7] Leal, Igor (2009). Requisitos de Metodologias de Teste de software. Disponível em <<http://homepages.dcc.ufmg.br/~rodolfo/dcc823-1-09/Entrega2Pos/igor2.pdf>>. Acesso em 22/06/2011.
- [8] MYERS, Glenford. (2004). **The Art of Software Testing**. Hoboken: John Wiley & Sons Inc. 2004.
- [9] Integration Test Disponível em <<http://msdn.microsoft.com/en-us/library/aa292128%28v=vs.71%29.aspx>>. Acesso em 22/06/2011.
- [10] Freeman, Steve. Endo-Testing: Unit Testing with Mock Objects. Disponível em <<http://connextra.com/aboutUs/mockobjects.pdf>>. Acesso em 22/06/2011

[11] Selenium Documentation. Disponível em <[http://seleniumhq.org/docs/book/Selenium\\_Documentation.pdf](http://seleniumhq.org/docs/book/Selenium_Documentation.pdf)>. Acesso em 23/06/2011.

[12] Fowler, M. (2001). The Agile Manifesto. Disponível em <[http://andrey.hristov.com/fht-stuttgart/The\\_Agile\\_Manifesto\\_SDMagazine.pdf](http://andrey.hristov.com/fht-stuttgart/The_Agile_Manifesto_SDMagazine.pdf)> Acesso em 23/07/2011

[13] JUnit FAQ. Disponível em <<http://junit.sourceforge.net/doc/faq/faq.htm>>. Acesso em 24/07/2011

[14] DbUnit. Disponível em <<http://www.dbunit.org/>>. Acesso em 30/07/2011