

# Testes Automatizados na Arquitetura J2EE

Fábio Henrique Barros<sup>1</sup>, Marum Simão Filho<sup>2</sup>

<sup>1</sup>Universidade de Fortaleza (Unifor) – <sup>2</sup>Faculdade 7 de Setembro (FA7)  
{fabio.hb}@gmail.com, {marumsimao}@gmail.com

**Resumo.** *O processo de desenvolvimento de software envolve diversas atividades passíveis de erro humano. Para minimizar a ocorrência desses erros, o desenvolvimento de software deve ser acompanhado de testes automatizados. Existem diversos tipos de testes automatizados, sendo os de unidade e funcionais [Cohen 2002] os principais e mais conhecidos. Para auxiliar na realização dos testes, utilizamos como base, o JUnit [Massol 2004] e suas extensões, tais como o DBUnit, EasyMock e o Shale Test [Rainsberger 2005], que foram utilizados para fornecer, respectivamente, suporte a Banco de Dados, simulação de objetos para Injeção de Dependências e Java Server Faces [Mann 2005] aos casos de teste. Para os testes funcionais, que simulam as ações do usuário sobre o sistema, utilizamos o Selenium [Koskela 2007] como facilitador desta atividade. Como construtor, integrador e executor de testes, indicamos o Maven [Sonatype 2008], por ter características como: execução automática dos testes de unidade e funcionais, por exemplo. Outro mecanismo importante no desenvolvimento voltado para testes é a Integração contínua. Essa prática consiste na integração periódica dos componentes do software durante seu desenvolvimento. O Continuum foi a ferramenta utilizada pra essa tarefa por ter integração com o Maven. A fim de comprovar os benefícios da utilização de Testes Automatizados, foram realizados diversos testes objetivando verificar suas vantagens e benefícios. Percebemos que este conjunto de práticas leva à redução de riscos do projeto, assim como permite obter software de alta coesão antecipadamente.*

## 1. Introdução

O processo de desenvolvimento de *software* envolve diversas atividades passíveis de erro humano. Existem diversas origens para esses erros, tais como requisitos mal elicitados, falhas de projeto do software, manutenções inadequadas e falhas humanas. Para minimizar a ocorrência desses erros, o desenvolvimento de *software* deve ser acompanhado de testes automatizados, ou, em um melhor caso, o

desenvolvimento deve ser dirigido por testes de *software* (*Test Driven Development - TDD*).

“A perspectiva dos testes exige que seja garantida que uma unidade de *software* não só executa de acordo com suas especificações, mas que realiza apenas a especificação.” [McGreg 2001]. Desta forma, os testes automatizados devem garantir não somente a sua correta execução, mas também que as funcionalidades solicitadas pelo cliente tenham sido atendidas.

Para o total aproveitamento de um teste, é necessário que sejam usadas ferramentas que possibilitem a automatização e aplicação destes. A principal importância de possuir testes automatizados é que este contribui para reduzir as falhas produzidas pela intervenção humana, aumentando a qualidade, produtividade e a confiabilidade do software.

Existem diversos tipos de testes automatizados, sendo os de unidade e funcionais os principais e mais conhecidos. Os Testes de Unidade são aqueles que visam garantir o correto funcionamento de uma unidade de *software*, testando, assim, uma parte isolada do código. Quando o desenvolvimento segue dirigido por testes, este é escrito antes do código e funciona como especificação, ajudando a pensar no *design* da aplicação. Os Testes Funcionais descrevem o funcionamento de um caso de uso do software, verificando, assim, se este atende às necessidades do cliente. Esse tipo simula a execução do produto sob o manuseio humano de forma automática, para garantir seu correto funcionamento. Objetivamos, portanto, neste artigo, explanar tais técnicas de forma que possamos verificar suas vantagens e benefícios.

Para alcançar esse objetivo, desenvolvemos uma aplicação simples com camadas bem definidas (Figura 1), de acordo com o padrão de projeto amplamente usado na WEB chamado MVC (*Model-View-Controller*) e utilizando os *frameworks* JSF, Spring [Walls 2005] e Hibernate [King 2005]. Cada camada terá seu teste, juntamente com suas características e particularidades.

Para explicar como este padrão funciona, as classes de controle recebem as requisições do utilizador e controla o fluxo de telas. O modelo de dados exerce as regras de negócio e manipula os dados armazenados no banco. Por fim, a camada de visão é responsável por formatar e apresentar as informações do sistema para o usuário.

A aplicação é composta por funcionalidades tais como: autenticação; cadastro de usuários; cadastro de estados e

ciudades; internacionalização do idioma; jogo utilizando a tecnologia Ajax.

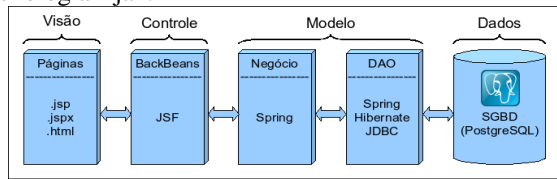


Figura 1: Modelo de camadas da aplicação

A organização deste artigo se faz em seis seções, contando com a introdução. A segunda seção explica a temática dos Testes de Unidade e suas particularidades para a arquitetura J2EE, seguido dos Testes Funcionais explanados na seção três. A seção quatro mostra a integração desses testes de forma contínua e seus resultados são apresentados na seção cinco. Por fim, tratamos dos resultados práticos deste estudo na seção seis.

## 2. Testes de Unidade

“Um teste de unidade é um trecho de código escrito por um programador que executa uma parte muito pequena ou área específica do código a ser testado. Geralmente um teste de unidade executa um método em contexto específico.” [Hunt 2003]. Em outras palavras, este tipo objetiva atestar que uma pequena parte do código está funcionando corretamente e conforme sua especificação.

Os programadores que não utilizam essa prática, podem sofrer as conseqüências de um código potencialmente não muito bem escrito. Quando um *software* é escrito sem testes, erros devido à falhas de programação podem ocorrer durante seu uso, assim, este acaba voltando para as mãos do programador para reparos. E quando será que podemos afirmar que um software está concluído? Provavelmente nunca, porém, sem testes, não poderíamos nem afirmar se está próximo ou não. Outro detalhe é que novos erros podem ser adicionados durante as manutenções corretivas e evolutivas do *software*. Via de regra, o que acaba acontecendo é que os programadores ficam “escravos” das correções. Certamente já ouvimos falar de sistemas que nunca estão bons o suficiente e que sempre há algo para “ajustar”, muitas vezes decretando prematuramente a morte do *software*. Esta situação se agrava quando manutenção ou alterações são feitas por diversos programadores durante o ciclo de vida.

Quando um programa é escrito sob essa ótica, a situação muda completamente, pois torna-se possível identificar erros prematuramente, assim como identificar que um determinado trecho do código está realmente concluído. Se o código escrito tiver sido verificado corretamente, podemos afirmar que este não sofrerá manutenções corretivas e sim evolutivas. Um fato importante é que os testes de unidade escritos para o sistema servirão de

“blindagem” contra erros de programação durante futuras manutenções. Outra vantagem é o fato de serem capazes de apontar o impacto das alterações no software e impedir que novos erros sejam criados. Observamos também que eles servem de documentação do código, pois demonstram como o software deve funcionar. Por fim, os testes podem ensinar a programadores inexperientes como desenvolver códigos sem erros.

Diversos *frameworks* de testes de unidades estão disponíveis para a linguagem Java, tais como: TestNG, GroboUtils, HASTE, Jetif, Mockrunner, Testare, Junit, etc. Para a realização deste artigo, escolhemos o JUnit, pois ele é atualmente o principal, mais utilizado e documentado *framework* de desenvolvimento de testes de unidade. Ele foi criado no intuito de facilitar o desenvolvimento de testes automáticos, que são chamados de “casos de teste”. O agrupamento de um ou mais casos é denominado “suite de teste”. Diversos *plugins* estão disponíveis para o JUnit, tais como o DBUnit, EasyMock e o Shale Test, que foram utilizados para fornecer, respectivamente, suporte a Banco de Dados, simulação de objetos para Injeção de Dependências e Java Server Faces aos casos de teste.

Para a realização dos testes das classes da aplicação analisada neste texto, dividimos em três categorias, Acesso a Dados, Regra de Negócio e Controle; explicados a seguir:

Na categoria “Acesso a Dados”, encontram-se as classes que fazem acesso direto ao Banco de Dados, simplesmente chamados de DAO (*Data Access Object*). Nesta categoria, utilizamos o DBUnit para facilitar os testes das operações básicas: criar, ler, alterar e excluir. Este *plugin* fornece facilidade de carregar, recuperar e comparar DataSet's que simulam registros no Banco de Dados.

Como podemos ver na Figura 2, o arquivo que define o um *DataSet* é um arquivo texto no formato XML, onde cada tabela é carregada com uma *tag* de mesmo nome.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <tabUsuario id="1" cpf="11111111111"
    nomeReduzido="Usuario" nomeCompleto="NomCompleto"
    email="usuario@gmail.com" senha="teste" />
  ...
</dataset>
```

Figura 2: Exemplo de um DataSet (usuarioDS.xml).

Para efetuar a carga do DataSet necessário aos testes, devemos iniciar a SessionFactory do Hibernate. Neste caso, optamos por criar uma classe abstrata (Figura 3) que conterá todos os métodos necessários para inicializar o ambiente de execução para os testes. No método “classSetup” iniciamos a SessionFactory a partir dos parâmetros contidos no arquivo de configuração do

Hibernate. Esse arquivo contém informações tais como: driver, endereço de conexão, dialeto de consulta, classes de persistência, usuário e senha do banco de dados. No método “recuperarDataSet” implementamos a carga dos dados na tabela através do DataSet em XML.

```

/** Javadoc */
public abstract class HibernateTestCase {
    ...
    /** Javadoc */
    @BeforeClass
    public static final void classSetup()
        throws Exception {
        Properties prop = new Properties();
        prop.load(HibernateTestCase.class
            .getResourceAsStream("dataSource.properties"));
        Configuration configuration =
            new AnnotationConfiguration();
        configuration.addProperties(prop)
            .configure(HibernateTestCase.class.getResource(
                "hibernateTestConfiguration.xml"));
        sessionFactory = configuration
            .buildSessionFactory();
        connectionProvider = ((SessionFactoryImpl)
            sessionFactory).getConnectionProvider();
    }
    /** Javadoc */
    protected IDataSet recuperarDataSet(
        final String arquivoDS) {
        try {
            return new FlatXmlDataSet(
                this.getClass().getResourceAsStream(
                    PASTA_DATA_SETS + "/" + arquivoDS));
        } catch (DataSetException e) {
            throw new AmbienteTesteException(e);
        } catch (IOException e) {
            throw new AmbienteTesteException(e);
        }
    }
}

```

Figura 3: Classe base dos testes de Banco de Dados.

Desta forma, criamos uma classe de teste, utilizando o JUnit, DBUnit e Hibernate (Figura 4). Esse caso refere-se ao método “selecionar”, assim, devemos criar um objeto da classe “Usuario” e carregá-lo com os dados que esperamos serem os mesmos do Banco de Dados. A verificação propriamente dito ocorre quando testamos se o objeto retornado pela consulta é igual ao esperado.

```

/** Javadoc */
public class TestUsuarioDaoImpl extends
    HibernateTestCase {
    /** Javadoc */
    @Test
    public void testSelecionarUsuario() {
        Usuario usuarioEsperado = new Usuario();
        usuarioEsperado.setId(1);
        usuarioEsperado.setCpf("11111111111");
        usuarioEsperado.setNomeReduzido("Usuario");
        usuarioEsperado.setEmail("usuario@gmail.com");
        Usuario usuario = new Usuario();
        usuario.setCpf("11111111111");
        assertEquals(usuarioEsperado,
            usuarioDao.selecionar(usuario));
    }
}

```

Figura 4: Caso de teste da classe UsuarioDaoImpl.

Para as classes da categoria de “Regra de Negócio”, utilizamos o EasyMock para simular a execução de suas dependências, no caso, os DAO's.

Este plugin auxilia a criação de objetos que simulam a execução (MockObject) previamente definida das dependências da classe a ser testada por meio de suas interfaces. Desta forma, o teste pode executar unitariamente, ou seja, o mesmo não depende da execução das dependências.

Neste tipo observado na figura abaixo (Figura 5), alguns passos são executados, tais como: criar objetos que representam o resultado esperado; criar os objetos que serão passados como parâmetros; criar os *MockObject's*; injetar os *MockObject's* no objeto da classe a ser validada; efetuar o teste propriamente dito; verificar se o resultado foi o esperado e se o objeto *Mock* foi chamado devidamente.

```

/** Javadoc */
public class TestUsuarioBusinessImpl {
    /** Javadoc */
    @Test
    public void testSelecionarUsuarios() {
        // Usuario de parametro
        Usuario usuarioParam = new Usuario();
        usuarioParam.setCpf("11111111111");
        // Usuario esperado
        Usuario usuarioEsperado = new Usuario();
        usuarioEsperado.setCpf("11111111111");
        usuarioEsperado.setNomeReduzido("Usuauro");
        // Criando o Mock
        UsuarioDao dao = EasyMock.createStrictMock(
            UsuarioDao.class);
        EasyMock.expect(dao.selecionar(usuarioParam))
            .andReturn(usuarioEsperado);
        EasyMock.replay(dao);
        // Injetando dependencia
        usuarioBusiness.setUsuarioDao(dao);
        // Testes
        Assert.assertEquals(usuarioEsperado,
            usuarioBusiness.selecionar(usuarioParam));
        EasyMock.verify(dao);
    }
}

```

Figura 5: Caso de teste da classe UsuarioBusinessImpl.

Finalmente, as classes de Controle também têm sua importância. Mesmo que seu papel seja apenas de coordenar as chamadas e fluxos de telas, um código mal escrito pode gerar complicações. Para esta camada, utilizamos o Shale Test da Apache Software Foundation, novamente um plugin para o JUnit construído no intuito de facilitar os testes de classe de controle do Java Server Faces. Este *framework* simula um ambiente de execução do JSF que somente estaria disponível se a aplicação estivesse em execução. Assim, todas as funcionalidades implementadas nas classes de controle podem ser testadas.

Analogamente à solução adotada com o DBUnit, também criamos uma classe base para todos os casos de teste das classes da camada em questão (Figura 6). Esta super-classe inicializa no método “setUp” todas as variáveis de ambiente que estariam disponíveis durante a execução do aplicativo, como, por exemplo, a sessão e localização padrão da língua através de um arquivo de propriedades.

```

/** Javadoc */
public abstract class JsftestCase extends
    AbstractViewControllerTestCase {
    ...
    /** Javadoc */
    protected void setUp() throws Exception {
        super.setUp();
        // Configurando arquivo de mensagens
        Locale locale = new Locale("pt", "BR");
        ResourceBundle bundle = ResourceBundle
            .getBundle("SystemMessages", locale);
        // Mocks
        MockApplication12 application =
            new MockApplication12();
        application.addResourceBundle("msg", bundle);
        this.application = application;
        facesContext.setApplication(application);
        facesContext.getViewRoot().setLocale(locale);
        session = new MockHttpSession(servletContext);
        request.setHttpSession(session);
    }
}

```

Figura 6: Classe base para os teste das classes de controle.

Dessa forma, as demais classes que implementam os casos de teste (Figura 7) devem herdar suas características e executar o teste necessário. Nesse exemplo, também utilizamos a técnica de MockObject's para simular o comportamento de suas dependências.

```

/** Javadoc */
public class TestUsuarioBean extends JsftestCase {
    /** Javadoc */
    @Test
    public final void testGetListaUsuarios() {
        String filtro = "Nome";
        Usuario usuario = new Usuario();
        usuario.setCpf("11111111111");
        usuario.setNomeReduzido(filtro);
        List<Usuario> usuariosEsperados =
            new ArrayList<Usuario>(1);
        usuariosEsperados.add(usuario);
        UsuarioBusiness usuarioBusiness =
            createStrictMock(UsuarioBusiness.class);
        expect(usuarioBusiness.selecionarUsuarios(
            filtro)).andReturn(usuariosEsperados);
        replay(usuarioBusiness);
        usuarioBean.setFiltro(filtro);
        usuarioBean.setUsuarioBusiness(usuarioBusiness);
        List<Usuario> resultado =
            usuarioBean.getListaUsuarios();
        assertEquals(usuariosEsperados.size(),
            resultado.size());
        for (int i=0; i < usuariosEsperados.size(); i++) {
            assertEquals(usuariosEsperados.get(i),
                resultado.get(i));
        }
        verify(usuarioBusiness);
    }
}

```

Figura 7: Caso de teste da classe UsuarioBean.

### 3. Testes Funcionais

Segundo Ricardo Ribeiro, “Testes funcionais são aqueles que encaram o sistema a ser testado como uma função que mapeia um conjunto de valores de entrada em um conjunto de valores de saída sem se preocupar com a forma como esse mapeamento foi implementado. Isto é, concentramos nossos esforços de testes nos requisitos funcionais do software” [Ribeiro 2008].

Este tipo de teste simula as ações do usuário sobre o sistema. Por isso, devemos verificar todas as possibilidades de uso do cliente, ou seja, não somente os fluxos principais, mas também os alternativos. Teste de Sobrecarga (*Stress Test*) é uma variante bastante conhecida desse tipo.

Existem diversos *frameworks* para testes funcionais e sobrecarga disponíveis para a linguagem Java. Segundo Carlos E. Perez, existem em torno de quarenta framework's com este intuito [Perez 2008]. Contudo, escolhemos o Selenium para a realização dos testes funcionais, pois este possui algumas vantagens sobre os outros, tais como: simplicidade de configuração; integração com ferramentas de *build*; possui uma extensão (Selenium IDE) para o *Firefox* que permite capturar a execução dos fluxos e gerar o código do caso de teste.

Novamente optamos por criar uma super-classe (Figura 8). Dessa forma as classes dos casos de teste têm a inicialização do Selenium em um único ponto. Nesse exemplo, o método “startSeleniumClient” configura o *framework* para utilizar o Mozilla Firefox.

```

/** Javadoc */
public abstract class SeleniumTestSupport {
    /** Usado para executar os comandos do selenium. */
    private Selenium selenium;
    protected final Selenium getSelenium() {
        return selenium;
    }
    /** Cria e inicia o cliente Selenium. */
    @Before
    public final void startSeleniumClient() {
        selenium = new DefaultSelenium("localhost", 4444,
            "firefox /usr/lib/firefox-2.0.0.7/firefox-bin",
            "http://localhost:8080");
        selenium.start();
    }
    /** Para o cliente selenium. */
    @After
    public final void stopSeleniumClient() {
        selenium.stop();
    }
}

```

Figura 8: Super-classe dos casos de teste do Selenium.

Com o Selenium IDE (Figura 9), gravamos o caso de teste e exportamos para uma classe de teste, assim, podemos executá-lo quando necessário. Essa ferramenta é bastante eficiente, pois à medida que utilizamos o sistema, ela armazena todas as ações do usuário como um gravador de macros. Permite ainda executar o teste, repetindo automaticamente todas as ações previamente gravadas.

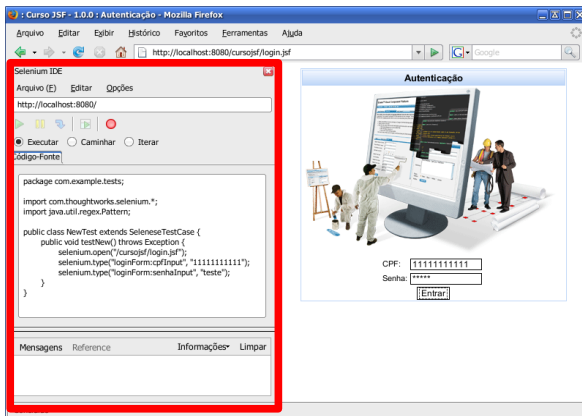


Figura 9: Tela do Selenium IDE

A própria ferramenta permite exportar o caso de teste para sua respectiva classe Java (Figura 10) como ambiente de execução baseado no JUnit. Esse teste refere-se ao caso de uso que dá acesso ao sistema, ou seja, o “Controle de Acesso”. Para ter acesso ao sistema, o utilizador deverá fornecer seu CPF, senha e pressionar o botão “Entrar”. O teste citado foi gravado pelo Selenium IDE e simula as ações do usuário, como podemos observar no método “testLogin”.

```

/** Caso de teste do caso de uso "Controle de Acesso".
 */
public class TestLogin extends SeleniumTestSupport {
    @Test
    public void testLogin() throws Exception {
        // Pagina de login
        get Selenium().open("/cursojsf/login.jsf");
        // Campos
        get Selenium().type(
            "loginForm:cpfInput", "1111111111");
        get Selenium().type(
            "loginForm:senhaInput", "teste");
        // Acao do botao de login
        get Selenium().click("loginForm:autenticaButton");
        get Selenium().waitForPageToLoad("30000");
        // Acao do link de logout
        get Selenium().click("logoutForm:logoutLink");
        get Selenium().waitForPageToLoad("30000");
    }
}

```

Figura 10: Classe de teste.

#### 4. Integração contínua dos testes

O renomado autor Martin Fowler escreveu em um artigo a seguinte definição para a Integração Contínua: “A Integração Contínua é uma prática de desenvolvimento de software onde os membros de uma equipe integram o seu trabalho frequentemente, normalmente cada pessoa integra seu código pelo menos diariamente, podendo levar a múltiplas integrações por dia. Cada integração é verificada por um construtor automatizado (inclusive os testes) para descobrir erros de integração o mais rápido possível. Muitas equipes percebem que esta prática leva a problemas de integração significativamente reduzidos e permite que uma equipe desenvolva o software coeso de forma mais rápida” [Fowler 2008]. Quando se trabalha

em equipe, a possibilidade de erros de integração é grande, por isso, integrar continuamente o *software* é tão importante.

Durante a integração, são executados automaticamente os teste de unidade e funcionais. Se estes executarem sem falhas, o integrador deve gerar uma versão compilada e testada do software, também conhecida como *build*. Caso ocorram erros durante os testes, o gerenciador de integração deverá alertar a equipe sobre os erros ocorridos. Dessa forma, sempre que uma falha de integração ocorrer, toda a equipe ficará sabendo e poderá tomar ações corretivas para o problema.

O gerenciador de integração não é responsável por executar a *build*, mas apenas por disparar o *software* construtor do projeto, ou *Project Builder*. O *Project Builder* será responsável por compilar, testar e empacotar em uma unidade instalável. Durante a execução deste projeto, identificamos necessidades de desenvolvimento tais como: execução automática dos testes de unidade e funcionais; gerência de dependências nos escopos de compilação, execução e testes; empacotamento e instalação no servidor de aplicações. Ao desenvolver os testes, verificamos que a melhor ferramenta de construção, de acordo com nossas necessidades, seria o Apache Maven.

Na arquitetura J2EE, existem diversos gerenciadores de integração contínua baseados nas licenças o software livre, tais como GPL, LGPL, BSD, Apache, Mozilla, etc. Dentre eles destacamos: Continuum (<http://continuum.apache.org/>), CruiseControl (<http://cruisecontrol.sourceforge.net/>) e Luntbuild (<http://luntbuild.javaforge.com/>). Estas três ferramentas são as mais utilizadas no mercado e por isso não cogitamos outras que não estivessem nesta lista. Também podemos afirmar que todas possuem quase as mesmas funcionalidades e que atendem às necessidades da aplicação desenvolvida para este artigo, por isso, a escolha da ferramenta ficou no âmbito da interface com o usuário.

O CruiseControl, a ferramenta mais antiga, ainda sofre com a configuração baseada em XML. Diferente da anterior, o Luntbuild tem sua configuração feita pelo próprio software, porém, sua interface é bastante confusa. Por fim, escolhemos o Continuum por ter maior comunidade ativa, por estar em amplo desenvolvimento e por ter a “melhor interface e facilidade de instalação” [Duvall 2008].

Como já citado, o gerenciador de integração contínua (Continuum) deve integrar-se com a ferramenta de construção da aplicação (Maven) para atingir o nosso maior objetivo: “software coeso de forma mais rápida” [Fowler 2008].

## 5. Resultados alcançados com a Integração Contínua dos Testes

Um dos maiores problemas em se usar a metodologia tradicional de desenvolvimento de software é nunca poder afirmar quando o produto poderá ser entregue, pois não se pode determinar quanto tempo levará para integrar todo o código. A Integração Contínua elimina todo esse problema, pois o código está constantemente sendo construído como se fosse ser entregue. O fato é que em qualquer momento do ciclo de vida do desenvolvimento, o *software* está pronto para ser implantado. Outro ponto que podemos citar é o fato de que quando um erro ocorre, toda a equipe é alertada, assim este erro pode ser corrigido tempestivamente.

“Erros de software são cumulativos. Quanto mais erros existem, mais difícil de retirá-los fica.” é o que afirma Fowler. A verdade é que o cenário pode ser pior, pois na tentativa de corrigir erros, podemos acabar criando novos, gerando assim, um ciclo de correções. Observamos que em projetos com Integração Contínua, onde o nível de cobertura das linhas do código por testes é alto, em torno de 90%, este fato não acontece.

## 6. Conclusão

Neste artigo, apresentamos práticas de testes automatizados em Java, como os Testes de Unidade com o JUnit e seus *plugins* DBUnit, EasyMock e Shale. Observamos ainda que testar somente a unidade do código não é suficiente e que devemos complementar com os Testes Funcionais, onde recomendamos o Selenium como ferramenta. Verificamos também que os testes e código devem ser integrados automaticamente e continuamente. Indicamos as ferramentas Maven e Continuum para realizar a tarefa chamada de Integração Contínua. Desta forma, problemas relacionados com baixa coesão do *software* são consideravelmente reduzidos, tornando-o pronto para implantação ao final de cada integração.

Assim, baseado nas experiências realizadas, percebamos que a maior vantagem da conjunção dos Testes de Unidade, Testes Funcionais, Ferramenta de Construção e Integração Contínua é a redução de riscos para o projeto.

## Referências

- [Cohen 2002] Frank Cohen. “Java Testing and Design – From Unit Testing to Automated Web Tests”. New Jersey: Prentice Hall. 2004.
- [Cohen 2004] Frank Cohen. “Java Testing, Design, and Automation”. Prentice Hall. 2004.
- [Duvall 2008] Paul Duvall. “Automation for the people: Choosing a Continuous Integration server”. <http://www-128.ibm.com/developerworks/java/library/j-ap09056/index.html>. Acessado em 10/05/2008.
- [Fowler 2008] Martin Fowler. “Continuous Integration”. <http://www.martinfowler.com/articles/continuousIntegration.html>. Acessado em 02/05/08.
- [Hunt 2003] Andy Hunt e Dave Thomas. “Pragmatic Unit Testing in Java with JUnit”. Dallas: The Pragmatic Bookshelf, 2003.
- [King 2005] Christian Bauer and Gavin King . “Hibernate in Action”. Greenwich: Manning Publications Co, 2005.
- [Koskela 2007] Lasse Koskela. “Test Driven: TDD and Acceptance TDD for Java Developers”. Greenwich: Manning Publications Co, 2007.
- [Mann 2005] Kito D. Mann. “Java Server Faces in Action ”. Greenwich: Manning Publications Co, 2005.
- [Massol 2004] Vincent Massol e Ted Husted. “JUnit in Action”. Greenwich: Manning Publications Co, 2004.
- [McGreg 2001] John D. McGregor e David A. Sykes. “A Practical Guide To Testing Object Oriented Software”. New York: Addison Wesley. 2001.
- [Perez 2008] Carlos E. Perez. “Open Source Automated Test Tools Written in Java”. <http://www.manageability.org/blog/stuff/open-source-automated-test-tools-written-in-java>. Acessado em 02/05/2008.
- [Rainsberger 2005] J. B. Rainsberger e Scott Stirling. “JUnit Recipes - Practical Methods for Programmer Testing”. Greenwich: Manning Publications Co. 2005.
- [Ribeiro 2008] Ricardo Lopes Ribeiro. “Testes de Software: Uma Visão para Aplicações Orientadas a Objeto”. <http://www.mundooo.com.br/php/modules.php?name=MOOArtigos&pa=showpage&pid=14>. Acessado em 02/05/2008.
- [Sonatype 2008] Sonatype Company. “Maven: The Definitive Guide”. O'Reilly Media, 2008
- [Walls 2005] Craig Walls e Ryan Breidenbach . “Spring in Action ”. Greenwich: Manning Publications Co, 2005.