Aplicação da automação de testes de software em SGBD's: Um estudo de caso utilizando o banco de dados Oracle

Fernanda G. Rodrigues, Evaldo de O. da Silva

Centro de Ensino Superior de Juiz de Fora (CES/JF)
Juiz de Fora – MG – Brasil

fernandacesjf@yahoo.com.br, evaldo.oliveira@gmail.com

Resumo. *Perdas econômicas, ou até mesmo ameaças à vida humana podem ser evitadas caso sistemas computacionais sejam utilizados. Cabe a uma equipe de qualidade em desenvolvimento e teste de software garantir o nível de confiança e evidenciar o bom funcionamento do software. A confiança em um sistema de computador equivale à sua integridade o que significa o grau de confiança dos usuários de que a aplicação não "falhará" em seu uso normal. É possível perceber estas características também em projetos de banco de dados, quando há necessidade de construção de rotinas e funções. Desta forma, o objetivo do presente estudo é apresentar técnicas de testes unitário de software em Banco de Dados Oracle, utilizando o DBUnit em um projeto Java.*

Abstract. *Economic losses, or even threats to human life can be avoided where computer systems are used. It is a quality team in software development and testing to ensure the level of confidence and demonstrate the proper functioning of the software. Trust in a computer system equivalent to their integrity which simply means the level of confidence of users that the application does not "fail" in normal use. You can see these features also in database projects when there is need to build routines and functions. Thus, the aim of this study is to present software unit testing techniques in Oracle Database, using the DBUnit in a Java project.*

1. Introdução

Um sistema confiável é aquele no qual pode-se confiar a respeito dos serviços que ele oferece (SOMMERVILLE, 2006). Cada vez mais os *softwares* estão presentes no dia a dia das pessoas, podem ser encontrados nos aparelhos celulares, em procedimentos cirúrgicos, bancos, aviões, entre tantos outros setores. Logo, constata-se que esses *softwares* estão se tornando cada vez mais complexos devido ao surgimento de novas tecnologias.

Uma parte considerável das pessoas passam, ou já passaram, por alguma experiência decepcionante com alguma aplicação que não se comportou como o esperado. *Softwares* que não funcionam corretamente podem levar a muitos problemas e não inspiram confiança aos usuários.

Conforme Kaner (1999) e Vianna (2006), a realização de testes é vital para o desenvolvimento de *software* com qualidade, demandando grande esforço e tempo de projeto. Estudos demonstram que a atividade de testes representa 45% do custo de

desenvolvimento de um produto. Além disso, as consequências de não testar são maiores, já que os custos para reparação crescem em escala logarítmica no tempo (PATTON, 2006).

O conhecimento sobre o desenvolvimento de *software* não garante que este seja desenvolvido adequadamente, uma vez que, durante o projeto, um desenvolvedor pode cometer diversos equívocos. Para evitar este problema, os testes de *software* podem auxiliar a detecção de construções de projetos inadequados e possíveis erros.

Os bancos de dados são fundamentais para que uma aplicação se mantenha íntegra, principalmente, quando se trata de sistemas transacionais. Segundo Bassi (2010) testar o banco de dados é testar a interação entre uma aplicação e um banco de dados, trabalhando na qualidade do sistema. Os testes vão manipular o banco de dados, através das operações de inclusão, alteração, exclusão e consulta, ou seja, farão operações *CRUD* (*Create*: criação, *Retrieve*: consulta, *Update*: atualização e *Delete*: destruição), que são as operações básicas de um sistema.

A seção 2, abordará a definição de testes de *software*, a diferença entre defeito, erro e falha, assim como os principais níveis de testes. Além disso, relaciona técnicas de teste estrutural e funcional e apresenta as ferramentas de suporte a testes de *software*.

A seção 3, apresentará a definição de testes automatizados para banco de dados, os requisitos para execução dos testes e de um banco de dados definido e os processo para execução de testes unitários em banco de dados.

A seção 4, um estudo de caso é elaborado com o *SGBD Oracle* no intuito de apresentar a ferramenta *DBUnit*. Esta seção aborda também o conceito sobre a utilização deste banco, os arquivos necessários para implantação do *DBUnit* e a execução dos testes.

A seção 5 discorrerá sobre a conclusão do trabalho.

2. Referencial Teórico

A confiabilidade é um requisito importante de um *software*. A confiança em um sistema de computador é uma propriedade do sistema que equivale à sua integridade. A integridade significa, essencialmente, o grau de confiança dos usuários de que o sistema operará como eles esperam e não “falhará” em uso normal (SOMMERVILLE, 2006)**.**

No que diz respeito à segurança do sistema em relação a acidentes ou outros problemas que possam ser causados pelo *software*, a engenharia de segurança de *software* encontra-se num estágio de desenvolvimento inicial (BLOOMFIELD, 1994). A enorme flexibilidade do *software* opõe-se às técnicas que procuram tornar o *software* mais seguro e impede o desenvolvimento de *software* absolutamente seguro. Na maior parte das vezes, a segurança de *software* ainda é vista como apenas um dos fatores que compõem um modelo de qualidade.

O teste de *software* contribui para melhorar a confiabilidade, sendo uma área vinculada à Engenharia de *Software* é de extrema importância e relevância tanto para o meio acadêmico quanto para a sociedade como um todo. Isto porque são os testes que avaliam a falha do sistema, a tolerância a defeitos, a detecção e exclusão dos mesmos assim como a disponibilidade e confiança.

2.1. Testes de *Software*

Inthurn (2001), define testes como uma das áreas da Engenharia de *Software* que tem por objetivo aprimorar a produtividade e fornecer evidências sobre a confiabilidade e a qualidade do *software* em complemento a outras atividades de garantia de qualidade, ao longo do processo de desenvolvimento (INTHURN, 2001 e VIANNA, 2006).

Alguns conceitos relacionados a essa atividade é a diferença entre defeitos, erros e falhas. As definições que serão usadas a seguir, utilizam as terminologias para definição das área e conceitos da Engenharia de *Software* do *IEEE – Institute of Electrical and Electronics Engineers* – (IEEE 610, 1990):

> • **Defeito** é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto (NETO, 2015).
>
> • **Erro** é uma manifestação concreta de um defeito num artefato de *software*. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro (NETO, 2015).
>
> • **Falha** é o comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha (NETO, 2015).

A figura 1 (NETO, 2015), expressa a diferença entre esses conceitos. Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, através do mal uso de uma tecnologia. Defeitos podem ocasionar a manifestação de erros em um produto, ou seja, a construção de um *software* de forma diferente ao que foi especificado (universo de informação). Por fim, os erros geram falhas, que são comportamentos inesperados em um *software* que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar a utilização de um *software*.

Figura 1. Defeito x erro x falha. FONTE: (NETO, 2015)

Dessa forma, pode-se ressaltar que testes de *software* revelam simplesmente falhas em um produto. Após a execução dos testes é necessária a execução de um

processo de depuração para a identificação e correção dos defeitos que originaram essa falha.

Foram listados abaixo os principais níveis de teste de *software*, (ROCHA, 2001):

• **Teste de Unidade:** também conhecido como testes unitários. Tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código.

• **Teste de Integração:** visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do *software* que foi estabelecida na fase de projeto.

• **Teste de Sistema:** avalia o *software* em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia a dia de manipulação do *software*. Verifica se o produto satisfaz seus requisitos.

• **Teste de Aceitação:** são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.

• **Teste de Regressão:** teste de regressão não corresponde a um nível de teste, mas é uma estratégia importante para redução de "efeitos colaterais". Consiste em se aplicar, a cada nova versão do *software* ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema. Pode ser aplicado em qualquer nível de teste.

2.1.1. Técnicas para Teste Funcional

Segundo Howden (1991), o teste pode ser classificado de duas maneiras: teste baseado em especificação *(specification-based testing)* e teste baseado em programa *(program-based testing).* De acordo com tal classificação, tem-se que os critérios das técnicas funcional e baseada em estado são baseados em especificação, e os critérios das técnicas estrutural e baseada em erros são considerados critérios baseados em programa.

O teste funcional, também conhecido como teste caixa-preta, trata o *software* como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída (MYERS, 2004).

Para Beizer (1990) o testador utiliza, essencialmente, a especificação de requisitos do programa para derivar os requisitos de testes, ou mesmo os casos de teste que serão empregados, sem se importar com os detalhes de implementação Assim, uma especificação de qualidade, e de acordo com os requisitos do usuário é de fundamental importância para apoiar a aplicação dos critérios relacionados a essa técnica.

Alguns exemplos de teste funcional, de acordo com Almeida (2015) são:

• **Teste de Requisitos:** verifica se o sistema é executado conforme o que foi especificado. São realizados através da criação de condições de testes e *cheklists* de funcionalidades.

• **Teste de Tratamento de Erros:** determina a capacidade do *software* de tratar transações incorretas. Esse tipo de teste requer que o testador pense negativamente e conduza testes como: entrar com dados cadastrais impróprios, tais como preços, salários, para determinar o comportamento do *software* na gestão desses erros.

• **Teste de Suporte Manual:** verifica se os procedimentos de suporte manual estão documentados e completos, determina se as responsabilidades pelo suporte manual foram estabelecidas.

• **Teste de Interconexão:** garante que a interconexão entre os *softwares* de aplicação funcione corretamente. Pois, *softwares* de aplicação costumam estar conectados com outros *softwares* de mesmo tipo.

• **Teste de Controle:** assegura que o processamento seja realizado conforme sua intenção. Entre os controles estão a validação de dados, a integridade dos arquivos, as trilhas de auditoria, o *backup* e a recuperação, a documentação, entre outros.

• **Teste Paralelo:** comparar os resultados do sistema atual com a versão anterior determinando se os resultados do novo sistema são consistentes com o processamento do antigo sistema ou da antiga versão.

2.1.2. Técnicas para Teste Estrutural

Segundo Myers (2004) a técnica de teste de caixa branca é conhecida por vários nomes tais como teste estrutural e teste de caixa de vidro. O engenheiro de sistema realiza o teste direto no código fonte do *software*. São determinados os dados de entrada para analisar a lógica do *software*.

Lewis e Veerapillai (2005) afirmam que a desvantagem da técnica de caixa branca é que ela não analisa se a especificação está certa, concentra apenas no código fonte e não verifica a lógica da especificação.

Alguns exemplos de teste estrutural segundo Almeida (2015) são definidos a seguir:

• **Teste de Integração ou Iteração:** é feito ao término de cada iteração para validar a execução das funções referentes aos casos de uso, é feito normalmente pelo analista de sistemas.

• **Teste de Sistema:** executa o sistema como um todo para validar a execução das funções acompanhando cenários elaborados (casos de teste) por um analista de testes em um ambiente de testes.

• **Teste de Aceitação:** é feito antes da implantação do *software*, o cliente é quem executa este tipo de teste no ambiente de homologação, tem como objetivo verificar se o *software* está pronto para ser utilizado pelos usuários finais.

• **Teste de Unidade:** é aplicado aos menores componentes de código, é feito pelos programadores e testa as unidades individuais: funções, objetos e componentes.

2.1.3 Teste de Unidade

Segundo Massol (2003), o teste unitário é aquele implementado pelo desenvolvedor. Um teste unitário examina o comportamento de uma unidade distinta de trabalho.

De acordo com Rocha (2007) nesta definição, dois termos merecem ser destacados: comportamento e unidades de trabalho. Testes unitários, como o nome sugere, devem testar unidades de trabalho isoladas, e geralmente, em termos de códigos orientados a objeto (*OO*), estas unidades de trabalho são métodos de uma classe. Um teste unitário, tipicamente testa aquele e somente aquele método, evitando acesso à outros recursos como sistema de arquivos, banco de dados, rede, dentre outros.

Abaixo serão listados os principais fatores que motivam o uso sistemático da prática de testes unitários:

- Previne contra o aparecimento de *"Bug's"* oriundos de códigos mal escritos:
- Código testado é mais confiável;
- Permite alterações sem medo (coragem);
- Testa situações de sucesso e de falha;
- Serve como métrica do projeto;
- Gera e preserva um "conhecimento" sobre as regras de negócios do projeto.

O objetivo de testes unitários é ter a confirmação de que unidades de processamento como métodos e funções por exemplo, não façam apenas o que já é esperado delas, mas que mesmo com a evolução que possa haver do sistema, continuem mantendo o comportando inicial.

2.1.4 Ferramentas de Suporte a Teste de *Software*

O objetivo desta seção é apresentar algumas ferramentas de suporte ao teste de *software* que conferem organização e controle das atividades de teste. Classificadas através de diferentes tipos, essas aplicações podem auxiliar nas etapas do processo de teste. As ferramentas descritas neste item, foram selecionadas em virtude da praticidade e usabilidade implantadas em ambientes corporativos pela sua facilidade de aprendizado.

Conforme afirma Rodrigues (2012) as ferramentas são categorizadas de acordo com os tipos de teste que executam. A seguir algumas ferramentaas são citadas:

• ***JUnit*:** é a ferramenta mais conhecida, normalmente utilizada por desenvolvedores, para teste de unidades de código-fonte ao longo do desenvolvimento. Serve para garantir a corretude de determinados trechos de código;

• ***Selenium*:** ferramentas que tem como objetivo simular a utilização da aplicação ou sistema pelo usuário, para testes de aceitação e integração por exemplo. É uma ferramentas para aplicações *web* que simula a navegação pelas páginas a procura de falhas nas funções existentes;

• ***SiKuLi***: é uma ferramenta que faz uma análise das imagens contidas na tela para executar um *script* de teste previamente criado. *Sikuli script* automatiza qualquer coisa em tela, sem a *API* de suporte interno. Ou seja, é responsável pelos testes de imagem. É possível controlar programaticamente uma página *web*, uma aplicação de *desktop* que executam o *Windows / Linux / Mac OS X*, ou mesmo um aplicativo para *iPhone* rodando em um emulador (iTeste, 2015);

• ***JMeter***: o *JMeter* é uma ferramenta *desktop* para testes de *performance*, desenvolvida utilizando a linguagem *Java* e licenciada sob os termos da "*Apache License, Version 2.0*". Esta ferramenta foi primeiramente utilizada para realizar testes em aplicações *web*, mas tem expandido suas funcionalidades, podendo realizar testes funcionais, testes em bancos de dados entre outros (FAPEG, 2013);

• ***Mantis***: mais conhecido, por sua praticidade, simplicidade e robustez. É uma ferramenta capaz de manter e administrar os registros e solicitações de colaboradores e usuários de sistemas e recursos de informática, facilitando o rastreamento de falhas e problemas em potencial, gerando estatísticas e documentando todo o processo de resolução das ocorrências ou problemas (Roessier, 2010). Se trata de uma ferramenta de gerenciamento de testes;

• ***DBUnit*:** é uma extensão do *JUnit* que oferece funcionalidades específicas para testes envolvendo banco de dados (Adams, 2011). Utilizado em testes unitários, o *framework* possui métodos específicos para comparação de registros, e facilita a carga de registros e a "limpeza" do banco (Rocha, 2007).

2.2. O que é Banco de Dados

Segundo Date (2004) "Um banco de dados é uma coleção de dados persistentes, usada pelos sistemas de aplicação de uma determinada empresa". Em outras palavras, banco de dados é formado por um conjunto organizado de informações relacionadas, criadas com objetivo específico de atender usuários, onde ao agrupá-los para uma mesma finalidade, é possível determiná-lo como banco de dados.

Os objetivos de um sistema de banco de dados são o de isolar o usuário dos detalhes internos do banco de dados (abstração de dados) e promover a independência dos dados em relação às aplicações, ou seja, tornar independente da aplicação, a estratégia de acesso e a forma de armazenamento (Rezende, 2014).

2.2.1. Objetos de Banco de Dados

Segundo Spinola (2012) objetos de banco de dados são as unidades lógicas que compõem os blocos de construção dos bancos de dados.

Um objeto de banco de dados se refere a qualquer objeto que é utilizado para armazenar ou referenciar dados. São exemplos desses objetos: tabelas que é a unidade básica de armazenamento de dados sendo composta por linhas e colunas (UNESP, 2000), *views* que é uma maneira alternativa de observação de dados de uma ou mais tabelas em uma base de dados (Bianchi, 2015), *functions* ou funções em português que são blocos *PL/SQL* semelhante à *Stored Procedures* com retorno de valores (Prado, 2012), *clusters* segundo Golin (2015) garante a integridade dos dados em todos os nós ao mesmo tempo certificando que inseriu o dado em cada nó (máquinas que compõe o cluster) antes de disponibilizar esse dado para outro usuário (replicação síncrona), *sequences* são objetos em que se gera números sequenciais dentro de um banco de dados (Monteiro, 2015), índices que são utilizados para performance do banco de dados otimiza as buscas de informações (Bianchi, 2015), dentre outros.

2.2.2. Testes em Banco de Dados

Segundo Rocha (2007), os testes representam uma etapa fundamental no desenvolvimento de qualquer *software*, e nessa etapa as atividades de teste de unidade são sem dúvida as mais conhecidas. Porém, testes que envolvem bancos de dados são geralmente trabalhosos, pois é necessário configurações diversas do banco, além de conhecimentos de *SQL* mais profundos.

Pede ser executado testes de unidade de banco de dados para verificar se altera para um ou mais objetos de banco de dados em um esquema desfeito funcionalidade existente em um aplicativo de banco de dados. Esses testes complementam os testes de unidade a criar seus desenvolvedores de *software* (MSDN, 2015).

As ferramentas de teste de banco de dados, surgem como opção para facilitar os testes além de dinamizar tais procedimentos, tornando-os mais simples ao desenvolvedor tal verificação. O teste é uma ferramenta essencial no desenvolvimento de *software* e assim também deve ser focado nos testes em banco de dados.

3. Definição de testes automatizados para Banco de Dados

Testes unitários executados de forma automatizada têm o objetivo de minimizar que qualquer tipo de alteração efetuada no sistema possam afetar a sua funcionalidade. Sabe-se que o teste unitário também conhecido como teste de caixa branca, usa da aplicação à ser testada o uso das suas classes e métodos, buscando resultados de processamento.

Sem uma base de dados consistente é inútil a utilização de testes automatizados em banco de dados. Como alternativa, a ferramenta *DBUnit*, que é um extensão da ferramenta *JUnit*, surge como solução para tais testes onde utilizando definições, busca-se resultados esperados.

Bancos de Dados são utilizados essencialmente para armazenar informações de aplicações. Porém, erros nessa camada podem facilmente afetar o resultado dos testes, tornando a experiência mais complexa e de difícil execução principalmente quando a

aplicação estiver em fases pós-desenvolvimento. O *DBUnit* auxilia os testes nesta camada e ajuda a prever prováveis problemas com testes de bancos de dados em partes específicas do código, caracterizando os testes unitários.

Com isso, neste artigo, o *DBUnit* será instalado no ambiente de desenvolvimento *Eclipse* juntamente com o banco de dados *Oracle* por ser uma plataforma robusta utilizada pela maioria das grandes corporações, além de ser gratuita, de código livre, o *DBUnit* possui uma comunidade ativa em toda *internet* (em artigos, fóruns e afins) assim como o *JUnit* e é também utilizado no ambiente acadêmico pela sua simplicidade de aprendizado e interface amigável.

A aplicação de testes automatizados é útil, mas ineficaz se a base de dados não estiver consistente. Neste contexto, a ferramenta *DBUnit*, uma extensão do framework de testes *JUnit* para execução de testes unitários, é capaz de efetuar testes sobre resultados esperados, como, comparar o nome do atributo referente à chave primária de uma tabela com o resultado que é esperado (SILVA, 2008).

Como exemplos de outras ferramentas para testes em banco de dados, segundo (Pierazo, 2015) a *JMeter* para testes de performance é citada, *DTM DB Stress* para testes de *stress* em banco de dados, *DBTester* para realização de testes de integridade, conectividade, monitoração dentre outras (ferramenta paga), *DBMonster* para testes aleatórios ou testes que atenda as regras de negócio dentro de um banco *SQL*, *GS Data Generator* para geração de dados aleatórios na base, dentre outras.

3.1. Requisitos Básicos

Para os testes de *software* no *DBUnit*, é necessária a ferramenta de desenvolvimento *Eclipse kepler* (4.3) instalada com o plugin do *JUnit* para execução dos testes e de um banco de dados definido, neste caso, o *Oracle 11g*.

Além do *DBUnit*, serão necessários outros "*jar*" responsáveis pelo processo de testes da base de dados. São eles:

- ***DBUnit:*** Com o *Eclipse* e o banco de dados já instalado, faz-se o *download* da ferramenta *DBUnit* (*dbunit-2.4.9.jar*);
- ***JUnit:*** Utilizar o pacote *JUnit* "*jar*" mais recente (*junit-4.10.jar*);
- ***Oracle:*** É o "*.jar"* de banco de dados *Oracle (JDBC)*. O arquivo "*ojdbc6.jar*", classe desenvolvida pela própria *Oracle* para acesso do *Java* ao seu banco de dados;
- ***SLF4J:*** ferramenta para geração dos *logs* dos testes, os arquivos do *SLF4J – Simple Logging Facade for Java* (*slf4j-jdk14-1.7.12* e *slf4j-api-1.7.12*).

3.2. Processo para execução de Testes Unitários em Banco de Dados

Usualmente, testes em bancos de dados em aplicações maiores são realizados com servidores dedicados. Porém, no exemplo apresentado neste trabalho, será evidenciado a parte de *login* de usuários e seus perfis em um banco de dados localmente, com quantidades de informações reduzidas para exemplificar o processo.

Em um projeto *Java*, utilizado como exemplo, são adicionadas as bibliotecas *".jar"* segundo o item 3.1, que deverão ser integradas ao projeto conforme é possível verificar na figura 2.

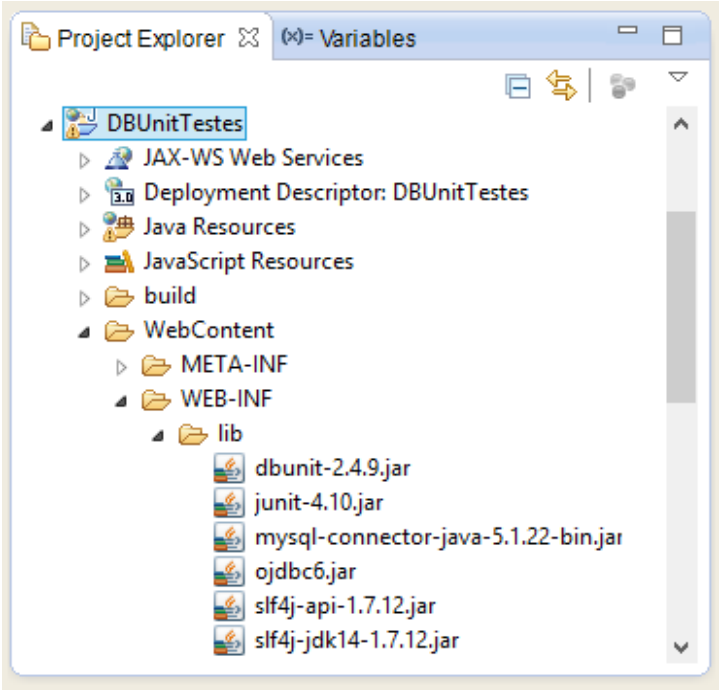

Figura 2. Eclipse – ProjectExplorer: libs. FONTE: (O Autor, 2015)

Inserir o *DBUnit* (*dbunit-2.4.9.jar*) e o *JUnit* (*junit-4.10.jar*) no projeto, o *JDBC* de conexão do banco de dados do *Oracle* (*ojdbc6.jar*), e os arquivos do *SLF4J - Simple Logging Facade for Java* (*slf4j-jdk14-1.7.12* e *slf4j-api-1.7.12*) responsável pela criação de logs do *DBUnit*.

4. Teste Unitário em Banco de Dados: um estudo de caso com o *SGBD Oracle*

O estudo de caso abordado neste artigo, com o intuito de apresentar a ferramenta *DBUnit*, considera um fragmento de um site que evidencia a parte de usuários e perfis. Na figura 3, observa-se o modelo entidade relacionamento de forma a gerar o seu banco de dados no *Oracle*.

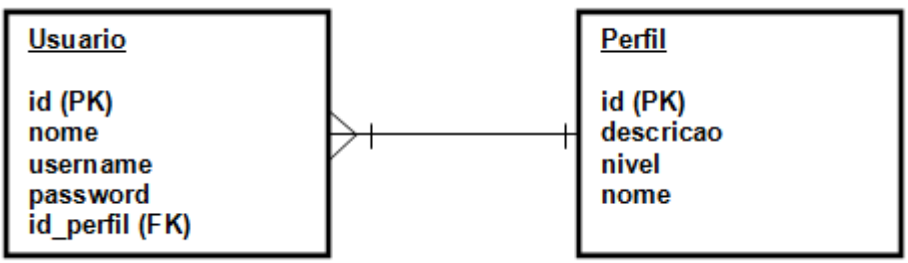

Figura 3. Estudo de Caso. FONTE: (O Autor, 2015)

A partir desse modelo, serão criados os arquivos *".xml"*. Arquivos estes que serão comparados para realização dos testes. Esse será o início para utilização da ferramenta *DBUnit*.

Mesmo com o crescimento do número de usuários que se conectam a um sistema, bem como os seus perfis, o banco de dados deverá tratar as informações, a segurança da aplicação e evitar que o usuário tenha acesso a áreas nao permitidas na aplicação, carregando corretamente as suas informações para a liberação do acesso. Neste contexto serão criados casos de testes baseados nessa premissa, onde as buscas no banco de dados localizam registros sem quedas no serviço e o sistema garantirá que suportará tal demanda mesmo existindo uma grande quantidade de registros.

4.1. *Oracle* - Banco de Dados

Um dos fatores para utilização do *Oracle* neste artigo, é que atualmente o *Oracle* é o banco de dados mais requisitado de acordo o ranking existente no site *DB-Engines* (2015) conforme demonstrado na figura 4.

277 systems in ranking, June 2015

Rank			DBMS	Database Model	Score		
Jun 2015	May 2015	Jun 2014			Jun 2015	May 2015	Jun 2014
1.	1.	1.	Oracle	Relational DBMS	1466.36	+24.26	-34.56
2.	2.	2.	MySQL	Relational DBMS	1278.36	-15.91	-31.20
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1118.05	-12.98	-105.74
4.	↑ 5.	4.	PostgreSQL	Relational DBMS	280.90	+7.39	+40.92
5.	↓ 4.	5.	MongoDB	Document store	279.05	+1.73	+47.61
6.	6.	6.	DB2	Relational DBMS	198.70	-2.35	+0.67
7.	7.	7.	Microsoft Access	Relational DBMS	146.49	+0.91	+4.13
8.	8.	↑ 9.	Cassandra	Wide column store	108.91	+2.36	+27.06
9.	9.	↓ 8.	SQLite	Relational DBMS	107.97	+2.81	+18.79
10.	10.	↑ 12.	Redis	Key-value store	95.49	+0.76	+30.36

Figura 4. DB-Engines – Ranking Banco de Dados. FONTE: (DB-Engines, 2015)

O cálculo desse ranking considera: número de menções do sistema em *WebSites*, buscas no *Google* e *Bing*, discussões técnicas sobre o *SGBD* em alguns sites de fóruns, número de vagas de trabalho oferecidas para o *SGBD*, número de profissionais que mencionam o uso do SGBD em seus *profiles*.

Um segundo fator, Prado (2012) nos informa que o *Oracle* tem um custo mais alto que outras plataformas no mercado e sua administração é uma das mais complexas, porém é um produto superior no quesito de segurança e performance, que podem ser muito importantes e cruciais para empresas que possuem aplicações críticas e que possuem muitos dados e muitos usuários concorrentes, em geral.

Em vista de tamanha complexidade, o *Oracle* é mais indicado para grandes empresas ou grandes aplicações, que concentram requisitos de negócios mais complexos e críticos, e que possuem capital para investir nos recursos de segurança e performance adicionais que ele oferece.

O mercado de trabalho está cada vez mais selecionando pessoas que tem como diferencial o conhecimento desse *SGBD* sendo este a motivação para elaboração do conteúdo desse trabalho.

4.2. Implantação do *DBUnit* - arquivos necessários

Após implantado no projeto as bibliotecas *"jar"*, conforme visto no item 3.2, o ambiente está pronto para desenvolver os testes propostos, com isso, será criado um pacote no projeto específico para os testes, além de criar a classe teste para o projeto que fará o acesso ao banco de dados *Oracle*. Assim, será criada a classe "*TesteOracle.java*", e no seguinte contexto, conforme figura 5.

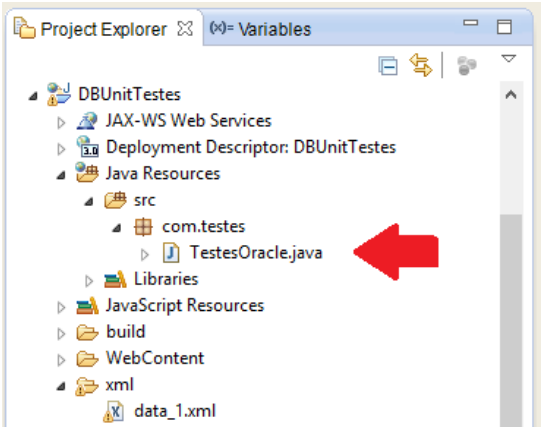

Figura 5. Eclipse: TestesOracle.java. FONTE: (O Autor, 2015)

Inicia-se a criação da classe, extendendo à classe *"DBTestCase"*, utilizada para os testes do *DBUnit* criando as variáveis e parametrizações.

```
public class TestesOracle extends DBTestCase {


	private FlatXmlDataSet bdcarregado;
	private String serverName = "127.0.0.1";
	private String portNumber = "1521";
	private String sid = "xe";
	// Super Classe: Conexão à base de dados definida e schema
	public TestesOracle(String name){
	super(name);
	System.setProperty(PropertiesBasedJdbcDatabaseTester.DBUNIT_DRIVER_CLASS,
	"oracle.jdbc.driver.OracleDriver");
	System.setProperty(PropertiesBasedJdbcDatabaseTester.DBUNIT_CONNECTION_URL,
	"jdbc:oracle:thin:@" + serverName + ":" + portNumber + ":" + sid);
	System.setProperty(PropertiesBasedJdbcDatabaseTester.DBUNIT_USERNAME,
	"vida");
	System.setProperty(PropertiesBasedJdbcDatabaseTester.DBUNIT_PASSWORD,
	"vida");
	System.setProperty(PropertiesBasedJdbcDatabaseTester.DBUNIT_SCHEMA,
	"vida"); // Oracle: definição do esquema (mesmo user do bd)
	}
```

Figura 6. Parametrizações: TestesOracle.java. FONTE: (O Autor, 2015)

Configura-se na classe o arquivo "*.xml*", que possui o conteúdo dos testes.

```
        // conteudo dos testes (arquivo xml)
        protected IDataSet getDataSet() throws Exception {
        bdcarregado = new FlatXmlDataSetBuilder().build(new
FileInputStream("xml/data_1.xml"));
        return bdcarregado;
        }
```

Figura 7. Carregamento xml: TestesOracle.java. FONTE: (O Autor, 2015)

Nos processos do *DBUnit*, determinam-se as operações de pré-execução e pós execução dos testes, como descrito na figura 8.

```
        // Operações antes dos testes
        protected DatabaseOperation getSetUpOperation()

        throws Exception {
        return DatabaseOperation.REFRESH; // Refresh antes de começar testes
        }

        // Operações pós execução
        protected DatabaseOperation getTearDownOperation()

        throws Exception {
         return DatabaseOperation.NONE; // None não executa nada
        }
```

Figura 8. Pré e pós execuções: TestesOracle.java. FONTE: (O Autor, 2015)

Para o banco de dados *Oracle*, sobrescreve-se a configuração padrão do DBUnit, para o seu funcionamento.

```
        // É necessário sobrescrever esta configuração (override)
        // do dbunit DatabaseConfig, para o funcionamento correto do Oracle.
        @Override
        protected void setUpDatabaseConfig(DatabaseConfig config){
        config.setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY, new
Oracle10DataTypeFactory());
        }
```

Figura 9. *DatabaseConfig*: TestesOracle.java. FONTE: (O Autor, 2015)

Após as configurações, determinam-se os testes à serem feitos. No primeiro caso de uso utilizando o *DBUnit*, testa-se a existência de determinado usuário na base de dados.

```
        /////// INICIO TESTES
        // Teste 1: Verifica-se se o usuário id = 2 existe no banco de dados
        public void testarUsuarioId() throws Exception, SQLException, Throwable {
        ITable registrosBd = getConnection().createQueryTable("u",
        "SELECT username " +
        "FROM usuario " +
        "WHERE id = 2");
        assertEquals("rsilveira", registrosBd.getValue(0, "username"));
        }
```

Figura 10. Teste 1: TestesOracle.java. FONTE: (O Autor, 2015)

No segundo caso, um exemplo para localização correta do perfil de um usuário específico.

```
        // Teste 2: Verifica-se o perfil do usuário: rsilveira
        public void testarPerfilUsuario() throws Exception, SQLException, Throwable {
        ITable registrosBd = getConnection().createQueryTable("p",
        "SELECT p.nome " +
        "FROM usuario u, perfil p " +
        "WHERE u.id_perfil = p.id " +
        "AND u.username = 'rsilveira' ");
        assertEquals("Vendedor", registrosBd.getValue(0, "nome"));
        }
```

Figura 11. Teste 2: TestesOracle.java. FONTE: (O Autor, 2015)

No terceiro e último caso de uso, é varrida a tabela em busca da quantidade de registros esperados na consulta.

```
        // Teste 3: Conta-se a quantidade de registros na tabela "usuario"
        public void testarQtdeRegistrosTabela() throws Exception {
        IDataSet registrosBd = getConnection().createDataSet();
        int qtdeRegistrosBd = registrosBd.getTable("usuario").getRowCount();
        assertEquals(3, qtdeRegistrosBd); // setam-se 3 registros manualmente,
comparado à busca na base
        }
}
```

Figura 12. Teste 3: TestesOracle.java. FONTE: (O Autor, 2015)

Observam-se os seguintes pontos na classe "*TestesOracle.java*".

• **Super Classe:** define-se a conexão ao banco de dados, com o *driver* de acesso à base, local do banco de dados, *login*, senha e o *schema* para o banco *Orcale*.

• ***getDataSet*( ):** define-se o arquivo *.xml* e o seu local que fará os carregamentos das informações para testes.

• ***getSetUpOperation*( ):** nesse método informam-se os processos que ocorrerão antes do início dos testes. Neste caso, faz-se um "*refresh*" (atualização) das informações.

• ***getTearDownOperation*( ):** nesse método informam-se os processos que ocorrerão após o término dos testes. Neste caso, não executa-se nada "*none*". Um outro exemplo, é que pode ser apresentado é fazer a limpeza das informações presentes no banco de dados através da chamada "*delete_all*".

• ***setUpDatabaseConfig*(*DatabaseConfig config*):** esse método sobrescreve o mesmo existente nas parametrizações do *DBUnit* e é necessário para funcionamento da base de dados *Oracle*.

• ***testarUsuarioId*( ):** o primeiro método de testes da aplicação. Nele, a busca do usuário é definida de "id = 2". Espera-se o resultado "rsilveira".

• ***testarPerfilUsuario*( ):** o segundo método de testes da aplicação. Nele, a busca do perfil do usuário com o "*username = rsilveira*" deve ser "Vendedor".

• ***testarQtdeRegistrosTabela*( ):** o terceiro método de testes da aplicação. Nele, a quantidade de registros no banco de dados é pesquisada. Atribui-se que deverá conter 3 registros.

Para auxiliar, será criado um arquivo "*.xml*" contendo as mesmas informações da tabela do banco de bados (para ser usada como teste), respeitando a parametrização conforme descrito na figura 13.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
	<usuario id="2" nome="Roberto Silveira" username="rsilveira"
password="123" id_perfil="3" />
	<usuario id="3" nome="Maria Roberta" username="mroberta"
password="123" id_perfil="2" />
	<usuario id="4" nome="João José" username="jjose" password="123"
id_perfil="1" />
</dataset>
```

Figura 13. data_1.xml. FONTE: (O Autor, 2015)

Obs: O arquivo é salvo na pasta raiz do projeto, de nome *"xml"*, para que a execução dos testes o localize.

4.3. Implantação do *DBUnit* – execução dos testes

A partir da implantação do código fonte, a execução dos testes diretamente na aplicação estará pronta e o *plugin* do *JUnit* deve estar instalado no *Eclipse* para a execução.

Para executar os testes, clica-se com o botão direito na classe *"TestesOracle.java"*, e na sequencia em *Run As >> JUnit Test*, conforme figura 14.

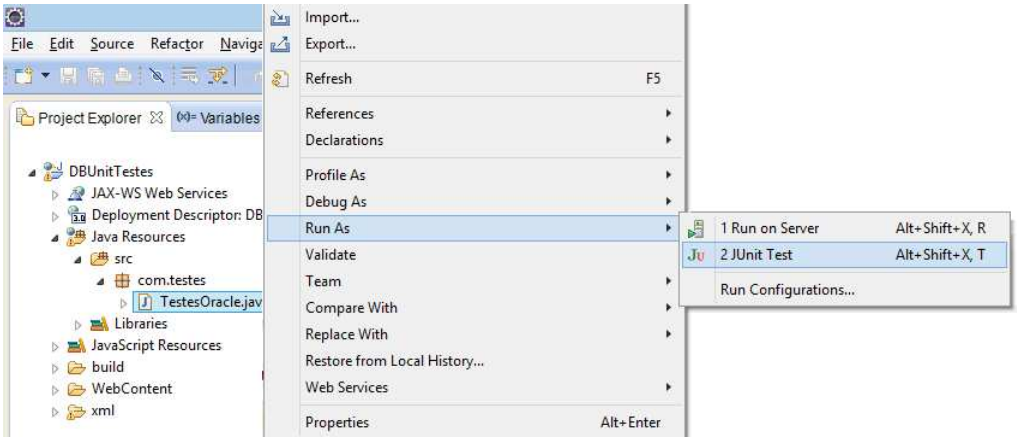

Figura 14. Eclipse: Run as – JUnit Test. FONTE: (O Autor, 2015)

Aguarda-se a execução e ao final, são apresentados os resultados exibidos na figura 15.

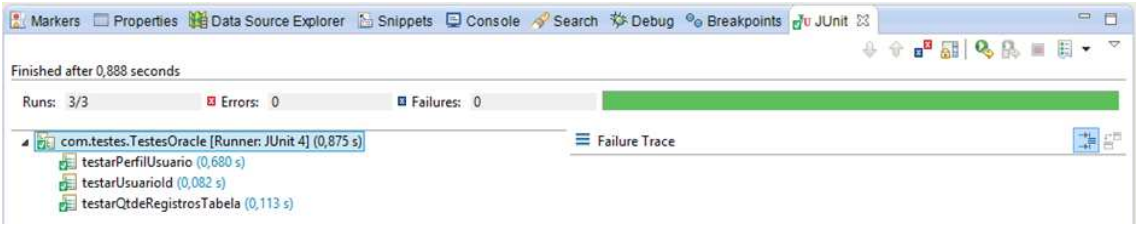

Figura 15. Eclipse: Run as – JUnit Test: sucesso. FONTE: (O Autor, 2015)

A figura 15 mostra que quando a execução é feita com sucesso, a cor apresentada é "verde". Um outro detalhe é ao lado do nome dos métodos testados, o *status* de cada processamento (à esquerda) e a informação do tempo de processamento (à direita).

Como outro exemplo, ajusta-se o fonte dos testes, para forçar um erro. Assim, será apresenta-se o método que ocorreu o erro, conforme figura 16.

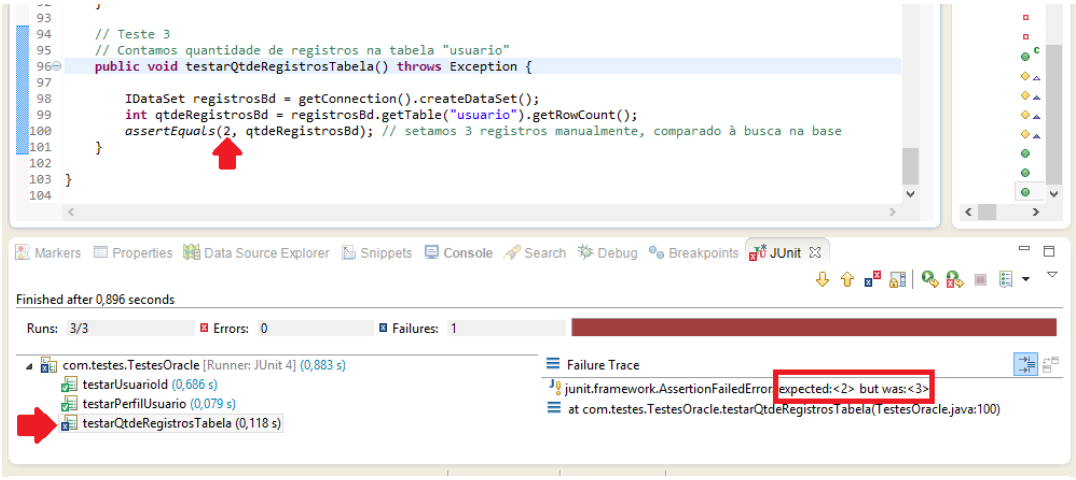

Figura 16. Eclipse: Run as – JUnit Test: erro. FONTE: (O Autor, 2015)

Como pode-se observar na figura 16, o resultado de busca de registros de usuários de 3 para 2 foi alterado, apesar disso, o banco de dados permaneceu com os 3 registros na tabela, ocasionando o erro dos testes.

O objetivo do *DBUnit* foi cumprido e abre-se um leque de opções ao desenvolvedor a evitar futuros erros em implementações, dando ao cliente maiores garantias da funcionalidade, criando *scripts* e testes unitários efetivos nos projetos.

5. Conclusão

Testes unitários em Banco de Dados muitas vezes são menosprezados e implementados com muitas regras de negócio dentro do próprio banco, através de *Stored Procedures*, *Triggers* e Funções.

A ferramenta adotada para este trabalho, *DBUnit*, assim como outras de testes de *software* possuem instalações complexas, necessitando um conhecimento abrangente na plataforma de desenvolvimento *Eclipse*, como instalação de *jar's* e *plugins* específicos para o seu correto funcionamento. Dos conhecimentos adquiridos na execução deste trabalho destacam-se: a implantação de testes unitários em Bancos de Dados *Oracle* utilizando o *DBUnit*, elaboração de *xml´s* com massa de dados para efeito comparativo e tipos de testes de *software* presentes no mercado. Com base no ambiente e no projeto *Java* utilizados para execução dos testes, foi possível obter diversas informações sobre a implementação dos métodos e funções necessários para parametrização do teste unitário utilizando o *DBUnit*, onde foram consultados dados e comparados aos eventos propostos.

Com essas informações e experiências é possível a validação ou não de um teste unitário, melhorando o processo de testes. Estudos futuros poderão ser realizados, implementando mais testes de banco de dados com outras ferramentas e exemplos com massa de dados reais, com uma maior proximidade a um ambiente empresarial. Uma opção é a utilização do *DBUnit* em paralelo ao *JMeter* para testes de performance em ambientes equivalentes testando assim o potencial de um Banco de Dados.

Outra contribuição importante a este trabalho é a adoção desses estudos de testes a um projeto que esteja em produção.

Referências

ADAMS, Y. **Testes de Integração: Validando ainda mais sua aplicação**. 2011. Disponível em: <https://yuriadamsmaia.wordpress.com/tag/dbunit/>. Acesso em: 10 jun. 2015.

AGILE, M. **Artigo Perdendo ou ganhando tempo com testes de unidade**. 2010. Disponível em: <http://blog.caelum.com.br/perdendo-ou-ganhando-tempo-com-testes-de-unidade>. Acesso em: 5 jun. 2015.

ALMEIDA, C. **Artigo Introdução ao teste de Software**. Disponível em: <http://www.linhadecodigo.com.br/artigo/2775/introducao-ao-teste-de-software.aspx>. Acesso em: 24 mai. 2015.

BASSI, G. **Testando o Banco de Dados: com Infra Microsoft**. 2010. Disponível em: <http://blog.lambda3.com.br/2010/07/testando-o-banco-de-dados-com-infra-microsoft/>. Acesso em: 10 jun. 2015.

BEIZER, B. **Software Testing Techniques**, Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

BIANCHI, W. **Entendendo e Usando Índices – Parte 1**. Disponível em: <http://www.devmedia.com.br/entendendo-e-usando-indices-parte-1/6567>. Acesso em: 3 jun. 2015.

BIANCHI, W. **Introdução a Views**. Disponível em: <http://www.devmedia.com.br/introducao-a-views/1614#ixzz3coyGQpXn>. Acesso em: 3 jun. 2015.

DATE, Christopher J. **Introdução a Sistemas de Bancos de Dados**. Rio de Janeiro: Elsevier, 8. edição, 2003.

DB-ENGIMES. **DB-Engines Ranking.** 2015. Disponível em: <http://db-engines.com/en/ranking>. Acesso em: 2 jun. 2015.

FAPEG. **Manual de Utilização da Ferramenta JMeter**. 2013. Disponível em: <http://www.freetest.net.br/downloads/Ferramentas/JMeter/Manual_JMeter.pdf>, Acesso em: 2 jun. 2015.

GOLIN, R. **MySQL Cluster Disponibilidade Total**. Disponível em: <http://www.devmedia.com.br/mysql-cluster-disponibilidade-total/6921>. Acesso em: 11 jun. 2015.

IEEE Standard 610-1990. **IEEE Standard Glossary of Software Engineering Terminology**, IEEE Press, 1990.

ITESTE. **Uso do SIKULI para automação**. Disponível em: <http://iteste.com.br/LinkClick.aspx?fileticket=U5Ihjq-6Rxk%3D&tabid=320&mid=1205>. Acesso em: 11 jun. 2015.

MALDONADO, José Carlos. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, 1991.

MASSOL,Vincent; Husted, Ted. **JUnit in Action**. Manning Publications, 2003.

MONTEIRO, D. **Usando o Objeto Sequence do Oracle**. Disponível em: <http://www.devmedia.com.br/usando-o-objeto-sequence-do-oracle/9303>. Acesso em: 20 jun. 2015.

MSDN, Microsoft. **Criar e definir os testes de unidade de banco de dados**. Disponível em: <https://msdn.microsoft.com/pt-br/library/vstudio/dd193286%28v=vs.100%29.aspx>. Acesso em: 26 mai. 2015.

MYERS, J. , SANDLER, C., BADGETT, T., THOMAS, T. **The Art of Software Testing.** John Wiley & Sons, 2nd. Edition, 2004.

NETO, A. **Artigo Engenharia de Software – Introdução a Teste de Software**. Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acesso em: 16 mai. 2015.

NOGUEIRA, A, Guimarães, L. **Testes: Uma Abordagem para Melhoria da Qualidade do Software**. 2008. Disponível em: <http://www.academia.edu/4313796/Monografia_de_Testes_Qualidade_do_Software>. Acesso em: 20 mai. 2015.

PIERAZO, C. **Análise Comparativa de Ferramentas de Teste para Aplicações em Banco de Dados**. Disponível em: <http://www.computacao.unitri.edu.br/erac/index.php/e-rac/article/viewFile/121/154>. Acesso em: 12 jun. 2015.

PRADO, F. **Qual é o melhor Banco de Dados: ORACLE ou SQL SERVER?** 2012. Disponível em: <http://www.fabioprado.net/2012/01/qual-e-o-melhor-banco-de-dados-oracle.html>. Acesso em: 13 jun. 2015.

PRADO, F. **Stored Procedures, Functions e Packages em banco de dados Oracle**. Disponível em: <http://www.devmedia.com.br/stored-procedures-functions-e-packages-em-bancos-de-dados-oracle/25390#ixzz3cp5nPL8G>. Acesso em: 21 jun. 2015.

PRESSMAN, R. **Software Engineering - A Practitioner's Approach**. McGraw-Hill, 5 edition, 2001.

REZENDE, Ricardo. **Conceitos Fundamentais de Banco de Dados**. 2014. Disponível em: <http://www.devmedia.com.br/conceitos-fundamentais-de-banco-de-dados/1649>. Acesso em: 14 jun. 2015.

ROCHA, A. **Artigo Java Magazine 48 - Iniciando com o DbUnit**. 2007. Disponível em: <http://www.devmedia.com.br/artigo-java-magazine-48-iniciando-com-o-dbunit/8535>. Acesso em: 26 mai. 2015.

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. et al. **Qualidade de software – Teoria e prática**. Prentice Hall, São Paulo, 2001.

RODRIGUES, P. **Artigo Sobre Ferramentas de Testes.** 2012. Disponível em: <https://bugless.wordpress.com/2012/01/11/sobre-ferramentas-de-testes/>. Acesso em: 25 mai. 2015.

ROESSIER, R. **Como instalar o Mantis**. 2010. Disponível em: <http://testersoftware.blogspot.com.br/2010/11/como-instalar-o-mantis.html>. Acesso em: 21 jun. 2015.

SILVA, B. **Artigo da Revista SQL Magazine - Edição 44 - Implementando testes ubnitários em base de dados com DBUnit**. 2008. Disponível em: <http://www.devmedia.com.br/artigo-sql-magazine-44-implementando-testes-unitarios-em-bases-de-dados-com-dbunit/7094>. Acesso em: 20 jun. 2015.

SOMMERVILLE, Ian. **Engenharia de Software**. Pearson Addison Wesley, 6ª ed. São Paulo, 2003.

SPINOLA, R. **Gerenciando Objetos de Banco de Dados – Revista SQL Magazine 90**. 2012. Disponível em: <http://www.devmedia.com.br/gerenciando-objetos-de-banco-de-dados-revista-sql-magazine-90/22004#ixzz3cov8QOBX>. Acesso em: 20 jun. 2015.

UNESP . **Norma Técnica para Definição de Objetos de Banco de Dados e de Estruturas de Armazenamento que Constituem o Banco de Dados Corporativo**. 2000. Disponível em: <http://www.unesp.br/ai/pdf/nt-ai.04.04.01.pdf>. Acesso em: 13 jun. 2015.