UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

CURSO SUPERIOR DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

JONATHAN HEVERSON RIBAS

DESENVOLVIMENTO DE CLASSES DE TESTE PARA A CAMADA DE PERSISTÊNCIA DO FRAMEWORK DE PREÇO DE VENDA (FRAMEMK) USANDO JUNIT

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2014

JONATHAN HEVERSON RIBAS

DESENVOLVIMENTO DE CLASSES DE TESTE PARA A CAMADA DE PERSISTÊNCIA DO FRAMEWORK DE PREÇO DE VENDA (FRAMEMK) USANDO JUNIT

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, da Coordenação do Curso de Análise e Desenvolvimento de Sistemas (COADS), da Universidade Tecnológica Federal do Paraná.

Orientadora: Profª. Drª. Simone Nasser Matos.

PONTA GROSSA

2014

Ministério da Educação

Universidade Tecnológica Federal do Paraná

Câmpus Ponta Grossa

Diretoria de Graduação e Educação Profissional

TERMO DE APROVAÇÃO

Desenvolvimento de Classes de Teste para a Camada de Persistência do Framework de Preço de Venda (FRAMEMK) usando JUnit

por

JONATHAN HEVERSON RIBAS

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 11 de novembro de 2014 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Profª. Drª Simone Nasser Matos
Orientadora

_____
Profª. Drª Simone de Almeida
Membro titular

_____
Profª. Drª Tânia Lúcia Monteiro
Responsável pelos Trabalhos
de Conclusão de Curso

_____
Prof Dr. Tarcizio Alexandre Bini
Membro titular

_____
Profª Drª Simone de Almeida
Coordenadora do curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

AGRADECIMENTOS

Agradeço a Deus e peço que abençoe as mulheres que fizeram comigo este TCC. Primeiramente a minha mulher, Layse, pela sua teimosia e fé, pelo que acreditou em mim até mais do que eu mesmo. Sem seu companheirismo, força e amor eu nunca teria chegado aonde cheguei. Eu amo você por tudo isso, pela pessoa que me completa e me ajuda nesta caminhada. Muito obrigado meu amor.

Pela minha Orientadora, Profª Drª Simone Nasser Matos que não mediu esforços em me ajudar, fazendo um trabalho muito além do excepcional em me orientar e me injetar altas doses de força de vontade para que eu nunca desistisse. Sem a sua ajuda nem o primeiro parágrafo deste trabalho teria sido feito, meu mais que sincero muito obrigado!

Pela minha mãe, que sempre acreditou em mim, independente de quantas vezes eu errasse. Nos momentos mais difíceis eu sempre ouço sua voz na minha mente dizendo para continuar. Suas mensagens de ânimo e força foram o fator decisivo para a conclusão deste trabalho, muito obrigado mãe.

Pela nossa coordenadora de curso, Profª Drª Simone de Almeida, que fez o possível e impossível para que eu pudesse me formar, além das suas aulas e orientações em outros projetos que fizemos, onde não importava a situação você nunca se deixou vacilar. Muito obrigado.

Por fim quero agradecer também os professores que tiveram paciência comigo por tantos anos, acreditando que eu iria conseguir me formar algum dia. Muito obrigado!

RESUMO

RIBAS, Jonathan Heverson. **Desenvolvimento de classes de teste para a camada de persistência do framework de preço de venda (FrameMK) usando JUnit.** 2014. 75 f.Trabalho de Conclusão de Curso Superior de Análise e Desenvolvimento de Sistemas- Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2014.

O teste de unidade permite avaliar a menor unidade de um sistema, que na orientação a objetos é a classe. Seu objetivo é permitir a identificação de erros no funcionamento da lógica interna desta unidade. Este trabalho realizou o teste de unidade em um Framework de Formação de Preço de Venda (FrameMK) que está em desenvolvimento pelo Grupo de Pesquisa em Sistemas de Informação. A realização do teste se deu por meio de uma adaptação de uma metodologia destinada a aplicação de testes, a qual permitiu a criação de casos de teste. A partir dos casos, foram criadas as classes de teste na ferramenta *JUnit* para a camada de persistência do framework. Com as classes de teste foi possível avaliar os vários cenários de um método, o que ajudou a aumentar a confiabilidade no sistema e as possíveis inserções de funcionalidades.

Palavras-chave: *JUnit. Classes de teste. Camada de persistência.*

ABSTRACT

RIBAS, Jonathan Heverson. **Test class development for the persistence layer of the selling price framework (FRAMEMK) using JUnit.** 2014. 75f.Trabalho de Conclusão de Curso Superior de Análise E Desenvolvimento de Sistemas- Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2014.

The unit test allows evaluating the smallest system unit, which is the class within object orientation. Its objective is allowing the error identification in the business logic inside this unity. This paper performed unit tests in a Framework of sell price formation (FrameMK) which is developed by the Research Group in Information Systems. The test creation is made by an adaptation of one methodology intended to test application, which provided the creation of test cases. From this test cases, was been created the test classes in the JUnit tool, to the framework persistence layer. Finally, with the test classes was possible to evaluate the method scenarios, which help to increase the system confiability and the possible functionality insertions.

Keywords: *JUnit. Test class. Persistence layer*.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Durante o desenvolvimento de sistemas de software existe a probabilidade de ocorrência de erros humanos, como falhas na comunicação ou requisitos especificados de forma incorreta. Para garantir a qualidade do software é fundamental a criação de atividades de teste de software (VAZ, 2003).

O teste de software minimiza o custo da correção de um sistema, encontrando o mais cedo possível a maior quantidade de erros no ciclo de desenvolvimento do software. Devido a sua complexidade, a automação de parte do teste de software é visto como uma forma favorável de aumentar sua eficiência e por consequência a qualidade do produto final para o cliente (FANTINATO et al., 2005).

As atividades de teste de software devem ser feitas ao longo do próprio processo de desenvolvimento, em geral se dividindo em três fases de teste: unidade, integração e sistema (BARBOSA et al., 2000).

O teste de unidade verifica a menor unidade de um módulo ou componente de software, procurando possíveis erros dentro dos limites da unidade testada. Este teste possui complexibilidade proporcional ao tamanho do escopo da unidade, pois tem por objetivo a lógica interna do processamento e a estruturas de dados do componente (PRESSMAN, 2011). Os testes de unidade podem ser realizados de forma manual ou automatizados.

Criado em 1997 durante uma viagem de avião por Kent Beck e Erick Gamma o framework *JUnit* é uma ferramenta focada na criação de testes automatizados de unidade considerando a linguagem de programação em Java. Esta ferramenta verifica se cada classe funciona como o esperado, gerando diagnósticos dos erros encontrados e pode ser utilizada até mesmo para baterias de testes (DALCIN; FERREIRA; D'ORNELLAS, 2007; DIAS; DALCIN; D'ORNELLAS, 2006). Os testes podem ser usados para validar as camadas de visão, regra de negócio ou persistência.

Testes na camada de persistência, são cruciais para garantir a qualidade do software, pois erros não detectados podem resultar em corrupção de dados irrecuperáveis (CHAN; CHEUNG, 1999).

O Grupo de Pesquisa em Sistemas de Informação (GPSI), Linha de Pesquisa de Engenharia de Software da Universidade Tecnológica Federal do

Paraná, Câmpus Ponta Grossa está desenvolvendo na linguagem Java um framework que implementa diversos métodos de precificação, para definição de preço de venda de um produto ou serviço (MAZER, 2013). Este framework foi criado por vários acadêmicos, que testam os requisitos somente em tempo de execução. O problema desta abordagem está na inserção de novas funcionalidades ao framework, pois o mesmo não contém um conjunto de classes de testes para validação do requisito. Além disto, as funcionalidades que existem ainda não foram validadas.

Este trabalho criou um conjunto de classes de teste para a camada de persistência do Framework para Formação de Preço de Venda (FrameMK), adaptando a metodologia de Crespo et al. (2004) para a criação dos casos de testes que facilitou a criação destas classes. As classes de teste foram desenvolvidas usando o framework *JUnit* no ambiente Eclipse.

1.1 OBJETIVO

A seguir serão descritos os objetivos gerais e específicos deste trabalho.

1.1.1 Objetivo Geral

Criar classes de testes de unidade utilizando o framework *JUnit* na camada de persistência do Framework para Formação de Preço de Venda (FrameMK).

1.1.2 Objetivos Específicos

Os objetivos específicos identificados para o desenvolvimento desta pesquisa são:

- Adaptar uma metodologia para criação das classes de testes.
- Analisar o funcionamento dos métodos de cada classe de persistência do FrameMK.
- Identificar os métodos no *JUnit* para criação das classes de teste na camada de persistência.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho é separado em cinco capítulos. O Capítulo 1 exibe uma introdução sobre o tema do trabalho, seus objetivos gerais e específicos.

O Capítulo 2 narra com mais detalhes o processo de teste, fornecendo informações sobre ferramentas de testes automatizados e a importância do processo de teste no desenvolvimento de sistemas.

O Capítulo 3 apresenta uma definição sobre frameworks e fala sobre o framework utilizado neste trabalho.

O Capítulo 4 descreve a metodologia adaptada para criação dos casos e classes de teste, bem como a metodologia base utilizada para sua concepção.

O último capítulo exibe os resultados da metodologia utilizada, as classes de testes geradas pelos casos de teste e as informações do levantamento da quantidade e resultados dos casos de testes executados.

2 TESTES DE SOFTWARE

Este Capítulo apresenta conceitos sobre teste de software. A Seção 2.1 relata a importância do uso dos testes de software para o desenvolvimento de sistema. A Seção 2.2 apresenta os estágios do processo e os seus respectivos tipos de teste. A Seção 2.3 descreve os testes de unidade em orientação a objeto. A Seção 2.4 discorre algumas informações sobre as ferramentas automatizadas para testes de unidade.

2.1 IMPORTÂNCIA DE TESTES DE SOFTWARE

O teste de software é um ou uma série de processos que tem por objetivo a comprovação de que o software faz somente o que foi designado para fazer. O software deve ser previsível e consistente, sem oferecer surpresas aos usuários. O teste em si é o processo de execução de um programa com o intuito de encontrar erros (MYERS; SANDLER; BADGETT, 2004).

Sommerville (2007) contextualiza estas definições em duas metas principais do processo de software:

- Provar ao desenvolvedor e cliente que o software provê todos os requisitos estipulados: Se o cliente solicita o requisito diretamente, é necessário ter ao menos um teste para cada requisito solicitado. Se os produtos de software são genéricos, é preciso validar todas as funcionalidades geradas no *release* do sistema. Em alguns sistemas é permitido que o próprio cliente teste com seus dados (teste de aceitação) para verificar a conformidade da especificação em comparação ao do sistema.
- Encontrar defeitos ou falhas gerados pelo mau comportamento do software: Por meio dos testes de defeitos é possível remover os tipos de comportamento incorretos do sistema, tais como: falhas na lógica de programação, no processamento de dados, queda de desempenho, corrupção de dados e integrações incorretas com outros sistemas.

Ao gerar o código fonte é necessário testar o software com o intuito de detectar e corrigir a maior quantidade de erros possível, antes de entregar o sistema ao cliente (PRESSMAN, 2011).

A etapa de teste de software é vista como uma das mais importantes dentro do ciclo de vida de geração de software, pois garante a qualidade do software tanto em aplicações críticas como sistemas de controle de foguetes quanto em aplicações menos complexas como sistemas para celulares (OLEGÁRIO, 2005).

Para Myers; Sandler e Badgett (2004) a atividade de teste ocorre basicamente quando se executa o software com o intuito de se encontrar erros. Um bom caso de teste é aquele que tem grande possibilidade de revelar os erros e um caso de teste bem sucedido é aquele que encontra erros ainda não descobertos.

Conceitualmente, o teste é simples, mas na prática dado o enorme (praticamente infinito) espaço para entradas de teste, é necessário resolver certo número de problemas desafiadores, incluindo avaliar e reusar os testes eficientemente e efetivamente durante o desenvolvimento do software (ZHANG, 2014).

Para melhorar a qualidade de um *software*, o desenvolvedor pode executar os testes de unidade, verificando toda a estrutura interna do projeto para encontrar possíveis condições de erro que podem ser tratados por meio de caminhos alternativos e previstos com maior facilidade (OLEGÁRIO, 2005).

O teste no banco de dados em aplicações é importante em ambas às fases: desenvolvimento e produção. Ressalta-se que falhas não rastreadas podem resultar em modificações incorretas ou remoção acidental de dados fundamentais (CHAN; CHEUNG, 1999).

Representando pelo menos 50% do orçamento dos projetos de software, o teste de software em geral tem como foco demonstrar a confiabilidade do software. Entretanto, em softwares de tamanhos maiores, os testes correspondem a um julgamento aproximado da segurança do software, pois o número de possíveis entradas é praticamente infinito, impedindo que todos os testes possíveis sejam feitos (PAŁKA, 2014).

A atividade de teste se enquadra como uma etapa crítica do desenvolvimento de um software, pois durante todo o processo de desenvolvimento apesar de serem utilizadas técnicas e métodos para se evitar os erros, eles acontecem e na atividade de teste estes são detectados e posteriormente eliminados (DIAS; DALCIN; D’ORNELLAS, 2006).

2.2 ESTÁGIOS DE TESTES DE SOFTWARE

Sommerville (2007) classifica os estágios do processo de teste em três itens, sendo eles: unidade, sistema e de integração. Estes estágios estão detalhados nas subseções a seguir.

2.2.1 Testes de unidade

Para garantia da operabilidade de um componente este é testado separadamente, em outras palavras, sem a intervenção de outros componentes do sistema. Estes componentes podem ser tanto: entidades simples, classes de objetos e funções provenientes destas entidades simples (SOMMERVILLE, 2007).

O teste de unidade em programação estruturada é um processo de validação de subprogramas, subrotinas ou procedimentos individuais em um programa, ou seja, antes de iniciar o teste no programa como um todo, este foca nos pequenos blocos de código do programa (MYERS; SANDLER; BADGETT, 2004).

Existem três motivações para se fazer este tipo de teste. Em primeiro lugar, o teste de unidade é uma forma de controlar os elementos combinados para teste, onde a atenção é focada inicialmente nas menores unidades do programa. Em segundo lugar o teste de unidade facilita a tarefa de depuração, pois quando um erro é encontrado, a sua existência fica conhecida em um módulo particular. Em último lugar, o teste de unidade apresenta um paralelismo no processo de teste por apresentar como a funcionalidade deve testar vários módulos simultaneamente (MYERS; SANDLER; BADGETT, 2004).

2.2.2 Teste de sistema

Um sistema é composto por integrações de componentes. Estas integrações estão sujeitas a erros inesperados causados por interações nos componentes que não foram previamente analisadas. O teste de sistema visa encontrar erros em: interações de componentes e nos problemas gerados na interface (SOMMERVILLE, 2007).

O software é uma parte do sistema de computador, pois ao terminá-lo ele será integrado com as outras partes (hardware, *peopleware* e informações, por exemplo). A série de testes executados para exercitar o sistema como um todo é chamado de teste de sistema. A lista a seguir descreve alguns dos tipos de teste de sistema para o software (PRESSMAN, 2011):

- Teste de recuperação: É um teste que tem o objetivo de verificar como o sistema se recupera após sofrer algum tipo de falha. Se a operação de recuperação for automática é avaliado sua reinicialização, mecanismos de verificação de erros e recuperação das informações perdidas. Se a recuperação for manual, é avaliado se o tempo médio para reparação do sistema está entre os limites aceitáveis.
- Teste de segurança: Checa se o sistema está protegido contra ataques ou invasões. O testador procura formas de extrair informações do sistema com qualquer abordagem, realizando por exemplo uma sobrecarga no sistema, utilizando um software de rastreamento de pacotes, forjando falhas para obter acesso durante a recuperação ou procurando credenciais nos arquivos desprotegidos do sistema.
- Teste por esforço**:** Verifica o comportamento do sistema diante uma demanda de recursos fora do normal, causando estresse no sistema até que ele falhe. Isto é útil para conhecer os limites do sistema, verificando até onde o sistema consegue ficar estável em relação aos recursos solicitados.
- Teste de desempenho**:** É utilizado para testar a rapidez e o desempenho de uma funcionalidade do sistema. Em certos tipos de sistemas, como os de tempo real e embutidos é inaceitável uma função que não esteja conforme os seus requisitos de desempenho. Este tipo de teste pode ser feito até em nível de unidade, testando individualmente uma função complexa que pode gerar um impacto negativo no sistema. Porém, para se testar o desempenho real é necessário que estejam integrados todos os elementos do sistema.
- Teste de disponibilização**:** Também pode ser chamado de teste de configuração. Procura erros nos diversos ambientes em que o cliente pode instalar o sistema, em outras palavras, verifica os “instaladores”

do sistema utilizados pelos clientes nas diversas plataformas disponibilizadas. Quando o sistema é acessível pela Internet este teste verifica o comportamento do sistema em cada navegador ou as combinações de sistemas operacionais com vários navegadores que possam ser utilizados pelo cliente.

Portanto, para integrar e validar um sistema são executados testes por outras pessoas além dos engenheiros de software. Todavia os cuidados realizados durante o desenvolvimento do projeto de software vão impactar diretamente o sucesso da integração do software com um sistema maior (PRESSMAN, 2011).

2.2.3 Teste de aceitação

Para finalizar a validação do software, os testes de aceitação utilizam dados provenientes do cliente, promovendo uma variação do comportamento do sistema em relação aos dados simulados para teste. Nesta variação de comportamento é possível encontrar erros, falta de especificações dos requisitos do sistema, problemas de desempenho e outras falhas que poderiam tornar o sistema fora do padrão para uso operacional (SOMMERVILLE, 2007).

O teste de aceitação é o processo de comparar o programa com seus requisitos iniciais e a necessidade atual de seus usuários. É uma forma de teste incomum, pois é realizada geralmente pelo próprio cliente ou pelo usuário final do software e normalmente não se considera responsabilidade da organização que desenvolveu o software (MYERS; SANDLER; BADGETT, 2004).

Pressman (2011) divide os testes de aceitação em três tipos:

- Teste alfa: Em um ambiente controlado, um grupo que representa os usuários finais são conduzidos pelo desenvolvedor que registra erros e problemas de uso.
- Teste beta: É realizado na instalação do sistema em uma ou mais estações de um cliente. Ao contrário do teste alfa o desenvolvedor não está acompanhando os testes e também não tem domínio sobre o ambiente para onde o sistema está sendo instalado. O usuário anota os problemas encontrados por ele e reporta em intervalos controlados para o desenvolvedor. Estes problemas podem ser erros do sistema ou imaginados pelo usuário que encontrou alguma dificuldade na

utilização do software. Com os resultados, os engenheiros de software realizam as modificações necessárias e preparam a versão estável do software para todos os clientes.

- Teste de aceitação do cliente: Uma variante do teste beta que pode ser executado quando o software possui uma personalização para um cliente específico por meio de um contrato. Este cliente testa o sistema no intuito de encontrar erros antes de confirmar que o software está de acordo com sua solicitação. Quando o software é utilizado por um grupo corporativo grande ou governamental, o teste de aceitação do cliente pode levar semanas para ser concluído, pois o processo de aceitação pode ser formal.

O intuito do teste de aceitação é verificar se os requisitos personalizados atendem as solicitações do cliente. Uma bateria de testes de aceitação pode variar de um simples acompanhamento até um esquema de testes planejados e sistematicamente realizados, podendo ser executado por longos períodos, encontrando desta maneira erros que poderiam corromper o sistema depois de uma grande parcela de tempo (PRESSMAN, 2011).

2.3 TESTES DE UNIDADE EM ORIENTAÇÃO A OBJETO

O conceito de teste de unidade modifica-se em um sistema orientado a objeto, em que uma classe encapsulada se torna o foco do teste. Os métodos no interior desta classe são as unidades testáveis de menor tamanho. Uma classe pode ter um método que envolva outras classes, sendo necessário modificar a tática do teste de unidade. Diferente do teste de unidade convencional, o teste de unidade em orientação a objeto é influenciado pelos métodos encapsulados na classe e pelo seu estado de comportamento (PRESSMAN, 2011).

Em outras palavras, o teste de unidade é aplicado para verificar a consistência entre a implementação de uma classe e sua descrição. Comparando ao teste de unidade tradicional que testa somente uma operação, uma classe possui múltiplas operações a serem validadas pelo teste (ZHANG et al., 2013).

No software orientado a objeto as descrições das classes são testadas pelos casos de teste baseados em uma sequência de operação apropriada. Inicialmente,

os casos de testes são determinados a partir das descrições da classe. Após esta etapa, os casos de teste adicionais são estendidos, baseados no valor de fronteira apresentado pelas implementações da classe. Por comparação, no teste de software tradicional, os casos de testes são desenhados dos detalhes do algoritmo de cada módulo (ZHANG et al., 2013).

Uma forma de lidar com as dependências de um método quando se irá testar em unidade é simular as classes dependentes. Esta é uma extensão de uma técnica de teste conhecida como *drivers* ou *stubs.* Tradicionalmente, *drivers* e *stubs* são escritos como métodos "burros" para os métodos dependentes. Em teste de unidade em orientação a objeto este conceito é estendido para todas as classes (GORDON; ROGGIO, 2014).

O *driver* é escrito quando a classe depende de outra para processar os dados, sendo normalmente usado com uma camada de baixo nível em um modelo de desenvolvimento hierárquico. O *stub* é um método escrito que controla dados para processar enquanto o módulo que processa o dado ainda não foi escrito, ou quando o módulo escrito ainda não foi testado. *Stubs* são escritos em geral quando é testado as classes de alto nível em um *design* hierárquico (GORDON; ROGGIO, 2014).

2.4 FERRAMENTAS AUTOMATIZADAS PARA TESTES DE UNIDADE

Frameworks de teste foram criados a fim de auxiliar e padronizar a escrita de testes automatizados, desse modo o isolamento do código de teste do código da aplicação é facilitado. Alguns exemplos são: o arcabouço pioneiro *SUnit 3* para *SmallTalk* criado por Kent Beck, que foi seguido por implementações para outras linguagens, tais como: *JUnit 4* e *TestNG 5* para Java, *JSUnit 6* para *Javascript*, *CppTest 7* para C++, *csUnit* 8 para .NET (BERNARDO; KON, 2008).

Conhecidas como *XUnit*, as ferramentas para teste de unidade verificam resultados de testes e diagnosticam o erro automaticamente, além de outras vantagens como por exemplo guardar o código que coordena o teste em um repositório. Isso favorece a sua manutenção e o maior comprometimento do desenvolvedor, que ao executar o código automaticamente terá o resultado já testado pela ferramenta (BIASI; BECKER, 2006; SILVA; PRICE, 1998).

Entre as ferramentas do *XUnit* para linguagens conhecidas estão a *cppUnit* para C++, *dUnit* para *Delphi*, *VBUnit* para Visual Basic, *NUnit* para .NET e *JUnit* para Java (BIASI; BECKER, 2006; SILVA; PRICE, 1998).

O Quadro 1 ilustra algumas ferramentas de teste automatizado para as linguagens mais conhecidas no mercado.

Linguagem	Framework	Descrição	Licença
Java	JUnit	O framework XUnit mais famoso para Java	Código aberto
	JTest	Uma ferramenta comercial que inclui geração e execução de testes automatizados	Comercial
	JMock	Uma extensão do framework JUnit para criar objetos mock	Gratuito
JavaScript	DOH	Executa testes dentro do browser ou independente	Código aberto
	Qunit	Testa qualquer JavaScript genérico, sendo muito útil para teste de regressão	Gratuito
	JSTest.Net	Habilita os testes de unidade em JavaScript a serem executados diretamente em outros frameworks XUnit	Gratuito
C/C++	C++ test	Um framework comercial que gerações de teste de unidade e relatórios de cobertura de código	Comercial
	Cantata++	Um framework comercial designado para teste de sistemas embarcados	Comercial
	Opmock	Um framework com *stubs* e *mock* para C e C++ baseado em geração de cabeçalhos de código.	Gratuito
.NET	NUnit	Um framework integrado com o Visual Studio para criar e executar testes de unidade	Gratuito
	DbUnit.NET	Um framework XUnit para testar bancos de dados.	Código Aberto
	MbUnit	Um framework XUnit baseado em modelo	Gratuito
	QuickUnit.NET	Desenha testes sem código e é muito útil em desenvolvimento orientado a testes	Comercial
PHP	PHPUnit	Um framework XUnit que exporta resultados em XML e HTML, incluindo informações de cobertura	Código aberto
	Apache-Test	Uma implementação PHP de um módulo de teste de unidade em Perl chamado Test::More	Código aberto
	Enhance PHP	Um framework XUnit que inclui recursos como *mock* e *stub*	Comercial
Internet	HtmlUnit	Uma extensão do JUnit que permite testar código HTML	Código aberto
	Selenium	Um framework gravação-reprodução que trabalha com a maioria dos navegadores da *Web*	Código aberto

Quadro 1 - Algumas ferramentas de teste automatizado.
Fonte: Adaptado de Polo et al. (2013, p. 86).

Os testes automatizados reduzem o custo e aumentam a qualidade das tarefas de teste. Na prática, os frameworks *XUnit* são as tecnologias mais utilizadas para automatizar testes. Nestes frameworks os casos de teste são escritos em uma

linguagem executável e podem ser iniciados automaticamente. Eles também permitem especificar operações para implementar os artefatos de casos de teste (POLO et al., 2013).

Para executar testes em bancos de dados a extensão DBUnit do framework JUnit cria uma especificação completa do estado de um banco de dados no início de um teste unitário e o estado que o banco deve possuir quando o teste unitário for executado com sucesso (WILLMOR; EMBURY, 2009).

A seguir será descrito o framework *JUnit* que será usado para o desenvolvimento dos casos de testes propostos neste trabalho.

2.4.1 JUnit

Desenvolvido por Kent Beck e Erick Gamma, o *JUnit* é um considerado um framework horizontal de código aberto, que suporta criação de testes automatizados em Java. Este framework facilita a geração de código para a automação de testes apresentando bons resultados. Por meio dele é possível verificar se cada método de uma classe funciona como se espera, exibindo possíveis erros ou falhas, podendo ser utilizado tanto para a execução de baterias de testes ou para extensão (DIAS; DALCIN; D'ORNELLAS, 2006).

Esta ferramenta possibilita a verificação dos testes de forma clara, visualmente e em formato de texto, evitando assim que o desenvolvimento dos casos de testes repita testes já realizados e o resultado seja positivo (BIASI, 2006; CIRILO, 2008).

O *JUnit* possibilita a criação de suítes de testes, ou seja, uma grande coleção de classes de testes, onde cada conjunto de classes é responsável por testar uma classe ou uma pequena funcionalidade do código que será colocado em produção (JUNIT, 2014; LANGR, 2001).

Além de possuir suporte para testes automatizados, o *JUnit* é uma ferramenta gratuita em que é possível obter rapidez na execução de testes das classes, evitando duplicação e permitindo o reuso dos testes (BIASI, 2006). A Figura 1 apresenta o diagrama de classes do Framework *JUnit*.

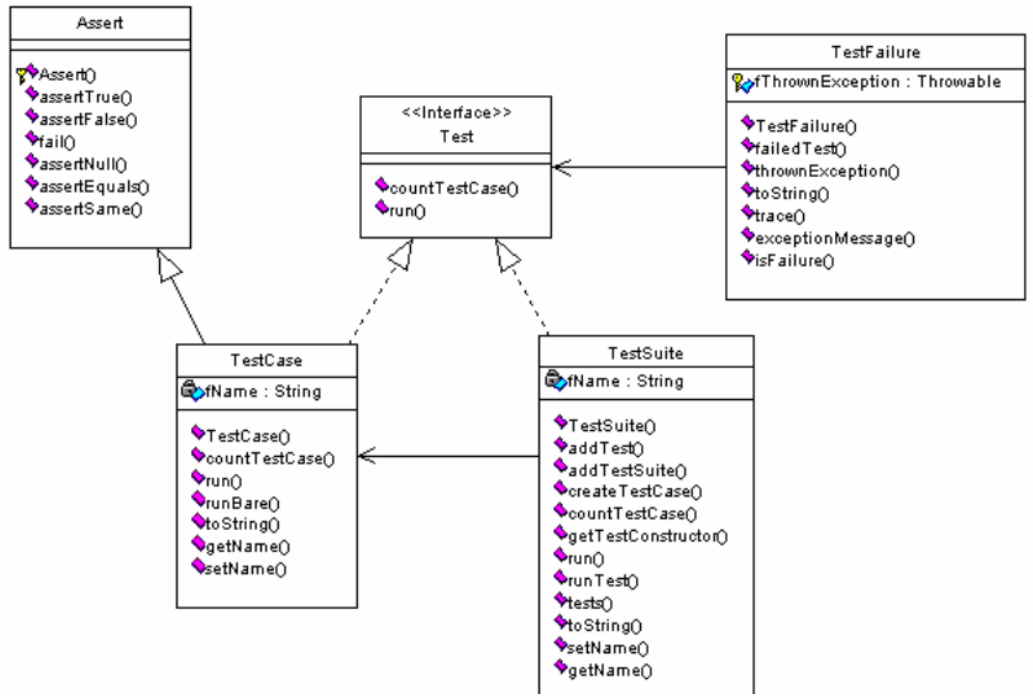

Figura 1 - Diagrama de classes do Framework JUnit
Fonte: Biasi (2006, p. 28).

Neste framework as classes de teste podem ser organizadas hierarquicamente, possibilitando que o sistema possa ser testado em partes separadas, algumas integradas ou até mesmo todas juntas de uma só vez (BIASI, 2006; CIRILO, 2008).

Quando se utiliza o *JUnit* o programador tem uma ferramenta útil que permite detectar *bugs* do código. Como se pressupõe que os programadores gostam de programar, criou-se uma forma interessante de realizar os testes facilitando desta forma o trabalho do programador (DIAS; DALCIN; D'ORNELLAS, 2006).

Para se utilizar o *JUnit*, faz-se necessário criar uma classe que estenda *jnuit.framework.TestCase.* Então, para cada método que será testado, é necessário definir um método público e sem retorno de argumentos, como *testeTestando()* na classe de teste. Esses métodos de testes devem ser *public void* e não podem receber nenhum parâmetro. Eles criam um objeto que definem o ambiente de teste, executando assim os testes com a utilização desse ambiente que foi criado e verificam os resultados que significam sucesso, falha ou exceção (DALCIN; FERREIRA; D'ORNELLAS, 2007).

Para a realização de teste de comparação entres os resultados obtidos e esperados, utilizam-se os métodos: *assertEquals, assertTrue, assertNull*. O método *setUp()* é usado para inicialização da suíte de testes antes da execução dos mesmos, enquanto o método *tearDown()* é usado após finalizar a suíte de testes (OLAN, 2003).

Algumas vantagens de se utilizar o *JUnit* são (CLARK, 2005):

- Não se faz necessário criar uma ferramenta de testes desde o começo.
- É gratuito.
- Existe muita documentação criada por desenvolvedores que utilizam software de código aberto.
- Quando os testes estão escritos podem ser executados de forma rápida, sem interromper o processo de desenvolvimento.
- Possibilita a criação de forma rápida do código de teste enquanto melhora a qualidade do sistema sendo desenvolvido e testado.

O *JUnit* possui uma integração com outras ferramentas de desenvolvimento, como: *Jbuilder*, *Kawa* e *Jdeveloper*. Também foram projetados outras extensões para se utilizar o *JUnit* voltados para diversos segmentos como banco de dados, XML, J2EE e WEB (DIAS; DALCIN; D'ORNELLAS, 2006).

3 FRAMEWORK DE DOMÍNIO

Este Capítulo relata algumas informações sobre desenvolvimento de frameworks. A Seção 2.1 descreve o que são frameworks e como podem ser classificados. A Seção 2.2 narra sobre o framework que será usado para a criação das classes de testes de unidade. A Seção 2.3 apresenta a arquitetura do framework usado no estudo de caso desta pesquisa.

3.1 FRAMEWORK

Um framework é um software que provê funcionalidades genéricas que podem ser modificadas seletivamente adicionando código escrito pelo usuário, gerando um software específico para uma aplicação. Um framework é uma plataforma de software universal, reutilizável e empregado para desenvolver aplicações, produtos e soluções. Frameworks de software incluem programas de apoio, compiladores, bibliotecas de código, ferramentas e interface de programação de aplicativos (API) que trazem juntos todos os diferentes componentes para habilitar o desenvolvimento de um projeto ou solução (FAYAD; SCHMIDT; JOHNSON, 1999).

Em outras palavras, um framework basicamente é o conjunto de classes que são cooperativas e formam um projeto que pode ser reutilizado para categorias específicas de software. Ele fornece direcionamento arquitetural pelo particionamento do projeto em classes abstratas, definindo suas responsabilidades e colaborações. Dessa forma, o desenvolvedor, modifica um framework para uma determinada aplicação (GAMMA et al., 1994).

A construção de um framework é formada por pontos comuns e específicos no domínio. A base corresponde aos aspectos que são comuns, a saber, é a parte que inclui os pontos de estabilidade; enquanto os pontos de flexibilidade ou específicos podem ser dos dois tipos seguintes (BOUASSIDA; BEN-ABDALLAH; BEN-HAMADOU, 2005): caixa-branca e caixa-preta.

Em um ponto de flexibilidade caixa-branca, o desenvolvedor pode adicionar classes de herança, redefinir código de método, entre outras possibilidades. Em contrapartida, em um ponto de flexibilidade caixa-preta, o desenvolvedor não precisa

“olhar” para o código interno para personalizar seu sistema, ou seja, os pontos de flexibilidade são preenchidos com a composição de objetos que os implementam (BOUASSIDA; BEN-ABDALLAH; BEN-HAMADOU, 2005).

Taligent (1994) assegura que *frameworks* têm a capacidade de resolver tanto os problemas que justificam sua criação quanto os que vão além da ideia inicial de seu desenvolvimento, sendo essa uma das razões pelas quais eles se diferem de aplicações independentes.

Dentre os *frameworks* existentes, tem-se sua classificação referente a seu reuso, sendo (FAYAD; SCHMIDT; JOHNSON, 1999):

- Caixa-branca (*White-box*): são estendidos usando conceitos de herança a partir de classes base do *framework.*
- Caixa-preta (*Black-box*): provêm interfaces que possibilitam a composição de objetos para obtenção de um resultado final.
- Caixa-cinza (*Gray-box*): podem ser chamados de híbridos e permitem sua extensão a partir de herança e utilização de interfaces por composição.

Os frameworks podem ser caracterizados por diferentes dimensões: aplicação, domínio e suporte. Os frameworks de aplicação preocupam-se basicamente com problemas internos de desenvolvimento de software; são independentes do domínio de aplicação-exemplo. Os frameworks de domínio, por sua vez, têm como objetivo apoiar o desenvolvimento de aplicações dirigidas aos usuários e produtos em domínios específicos. Os frameworks de suporte oferecem serviços de sistema de baixo nível, tais como dispositivos de interface para periféricos (*drivers*) (TALIGENT, 1994).

Dentre as dimensões, este trabalho tem como foco o estudo de um framework de domínio para criação das classes de teste, por isto esse será detalhado na próxima seção.

3.2 UM FRAMEWORK DE DOMÍNIO PARA FORMAÇÃO DE PREÇO DE VENDA: FRAMEMK

O FrameMK é um framework de domínio que está sendo desenvolvido pelo Grupo de Pesquisa em Sistema de Informação (GPSI) da UTFPR Câmpus Ponta Grossa, na linha de pesquisa de Engenharia de Software.

Este framework contempla vários métodos de preço de venda para produtos, a saber, Sebrae, Custo Pleno e ABC e tem por finalidade oferecer ao usuário um ambiente no qual se pode gerar o preço de venda para um produto ou serviço. A Figura 2 ilustra um exemplo do ambiente do FrameMK para gerar o preço de um produto usando o método de precificação Custo Pleno.

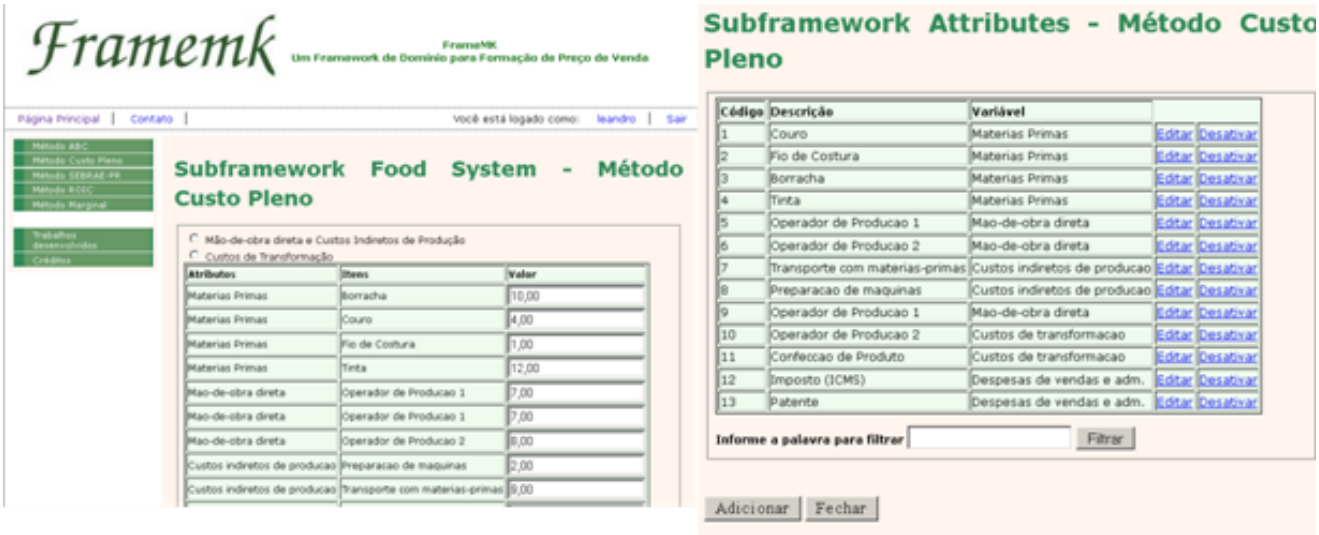

Figura 2 - FrameMK: Exemplo de geração de preço usando o método Custo Pleno

Fonte: Autoria própria.

A linha cronológica para o desenvolvimento do FrameMK foi marcada por um conjunto de trabalhos desenvolvidos pelos acadêmicos. A Figura 3 ilustra os autores dos trabalhos que já contemplados no FrameMK a partir de 2008 até 2014.

Figura 3 - Linha Cronológica do Desenvolvimento do FrameMK

Fonte: Autoria Própria.

Anderson Crazuski, Leandro Botelho Feitosa e Thiago Luiz Cordeiro iniciaram o estudo do domínio em 2008 (CRAZUSKI; FEITOSA; CORDEIRO, 2008). O trabalho se baseou em levantar os pontos de estabilidade e de flexibilidade usando o processo dirigido por responsabilidades para criar a modelagem de três métodos, Método de Custeio em Atividades (ABC), Método Sebrae e Método Custo Pleno.

Rafael Rudnik de Oliveira e Rudy José Crissi Crema, no ano de 2009, prosseguiram com o estudo sobre o domínio. Elaboraram um trabalho que tinha como finalidade apresentar os conceitos sobre a formação de preço de venda e do lucro, modelando dois métodos de formação de preço de venda, sendo eles o método do Custeamento Marginal e o do Retorno Sobre o Capital (ROIC) (RUDNIK; CREMA, 2009).

No ano de 2010, os alunos Paulo Eduardo Boeira Capeller e Vinícius Camargo Andrade começaram a desenvolver a arquitetura inicial do FrameMK (CAPELLER; ANDRADE, 2010). A arquitetura proposta utiliza três métodos de domínio para formação de preço de venda: Custo Pleno, Custeio Baseado em Atividade e SEBRAE, os quais já tinham sidos modelados individualmente pelo trabalho dos alunos Crazuski; Feitosa e Cordeiro (2008).

No mesmo ano (2010), Claudinei Rodrigues Junior (JUNIOR, 2010) desenvolveu, voltado para a WEB, um serviço de código aberto para a busca de preços de venda de produtos em sítios de *e-commerce* que é importante para a implementação do método baseado nas decisões das empresas concorrentes.

Renato Ramos desenvolveu em 2011, tendo como base o trabalho de Capeller; Andrade (2010), a criação da primeira versão do framework para a web, refatorando a camada de apresentação do framework utilizando uma ferramenta para interfaces web para java (*Struts)* e seu sistema de templates (*Tiles*) (RAMOS, 2011).

Em 2012, Leandro Siqueira da Silva desenvolveu um módulo de *login*, utilizado no FrameMK, baseado em aspectos para controle de acesso aos usuários no aplicativo (SILVA, 2012). O método identifica aspectos na fase de análise usando uma matriz adjacência. A maior utilidade desse método é que os analistas não necessitam ter conhecimentos sobre aspectos, tendo que estes serão identificados a partir das similaridades com requisitos não funcionais.

No mesmo ano (2012), Victor Schnepper Lacerda desenvolveu o trabalho de refatoração do sítio ArcaboMK, que teve como objetivo armazenar as informações dos trabalhos realizados pelo grupo de pesquisa e de acadêmicos envolvidos nos projetos. A refatoração foi realizada para framework Struts e sua validação foi obtida por meio do uso de métricas de qualidade e desempenho (LACERDA, 2012).

Em 2013, o mestrando Ademir Mazer Junior desenvolveu um *web service* para o FrameMK (MAZER JUNIOR, 2013).

E em 2014, os alunos Alexandre Prado Barbosa e Luan Bukowitz Beluzzo adaptaram um processo para extensão do FrameMK no qual realizaram a inserção de novas funcionalidades do método ABC e criaram a modelagem para os métodos Marginal e ROIC (BARBOSA; BELUZZO, 2014). Além deste trabalho, o aluno Everaldo Veres Zahaikevitch criou um sistema especialista que é capaz de determinar o melhor de método para ser usado na formação de preço de venda a partir das características das empresas (ZAHAIKEVITCH, 2014).

3.3 ARQUITETURA

O FrameMK foi construído em Java usando o *framework* de aplicação *Struts.* Sua arquitetura é composta pelas principais divisões: a) *app* são referentes a interfaces gráficas do sistema, *BusinessRule* possui as regras de negócio da aplicação e *Persistence* realizam a conexão e comunicação com o banco de dados, b) *web services* são referentes ao serviço para o FrameMK, c) *test* representam as classes de testes do sistema *web service* e d) *sistemasPOA* refere-se o sistema de *login* usando orientação a aspectos. A Figura 4 ilustra essas divisões.

Este trabalho irá apresentar as classes de teste para o pacote de persistência porque se deseja verificar o registro dos dados no banco. Esses dados são usados durante o cálculo do preço de venda. Por isso, as classes deste pacote serão explicadas neste trabalho.

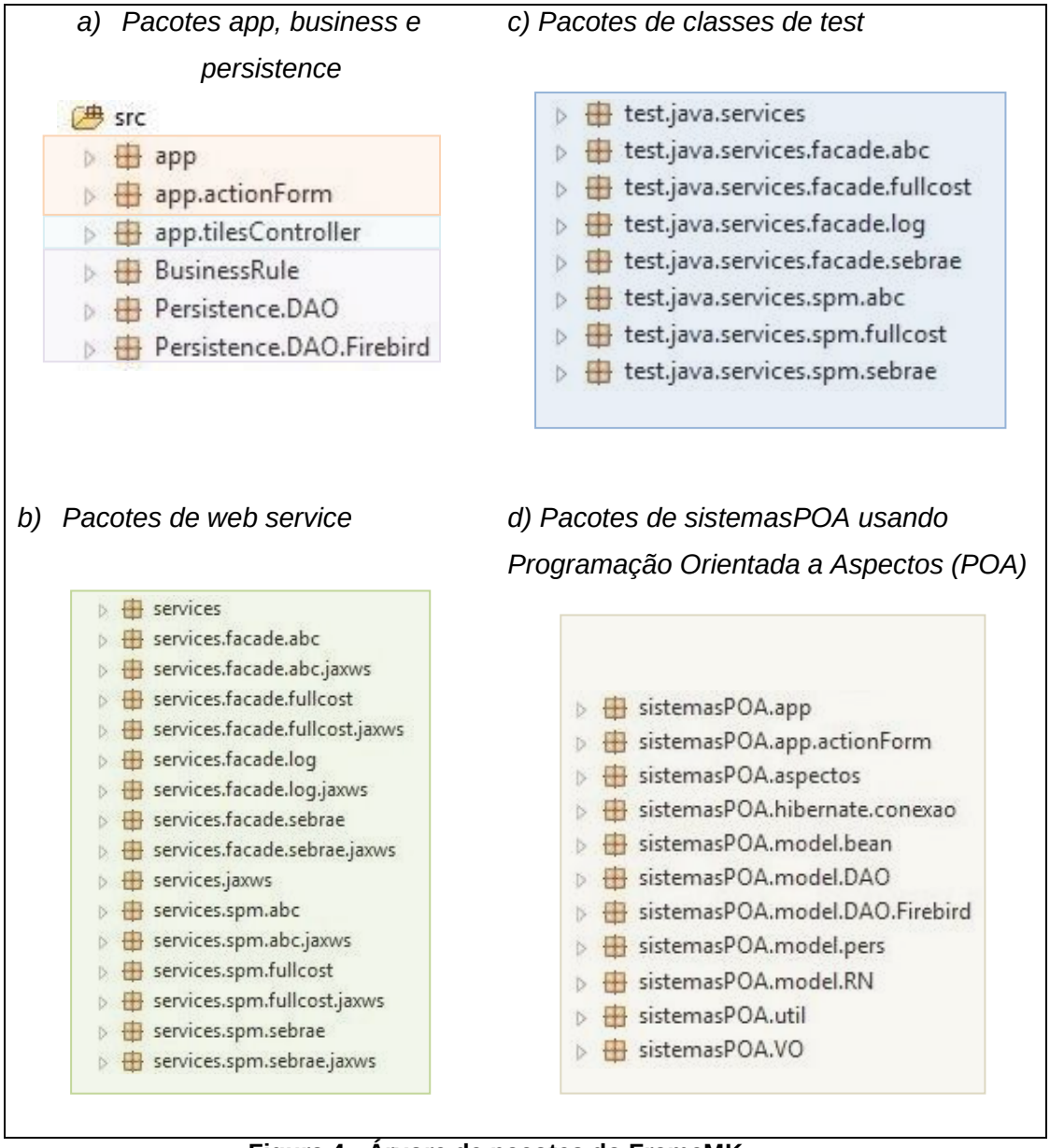

Figura 4 - Árvore de pacotes do FrameMK.
Fonte: Autoria própria.

Conforme ilustra a Figura 4a) dentro do pacote *Persistence.DAO* se tem a interface *DAOFactory* que contém os métodos abstratos *getABCAttributeDAO()*, *getSebraeAttributeAttributeDAO()*, *getFullCostAttributeDAO(),* os quais retornam os atributos referentes aos métodos de formação de preço de venda ABC, Sebrae e Custo Pleno, respectivamente. Exemplos de atributos foram ilustrados na Figura 2 para o método Custo Pleno, tal como: mão de obra direta, matérias prima, entre outras.

Outro pacote de persistência é o *Persistence.DAO.Firebird*, Figura 4a), que tem classes *AbcActivityDAOFirebird*, *AbcAttributeDAOFirebird*, *AbcLogValueDAOFirebird*, *AbcProductDAOFirebird*, AbcProductionLineDAOFirebird, *FBDDAOFactory*, *FullCostAttributeDAOFirebird*, *FullCostItemDAOFirebird*, *FullCostLogValueDAOFirebird*, *LogValuesDAOFirebird*, *SebraeAttributeDAOFirebird*, *SebreaAttributeDAOFirebird*, *SebraeLogValueDAOFirebird*, que são específicas para cadastrar, editar, excluir, consultar dados referentes aos métodos de formação de preço de venda.

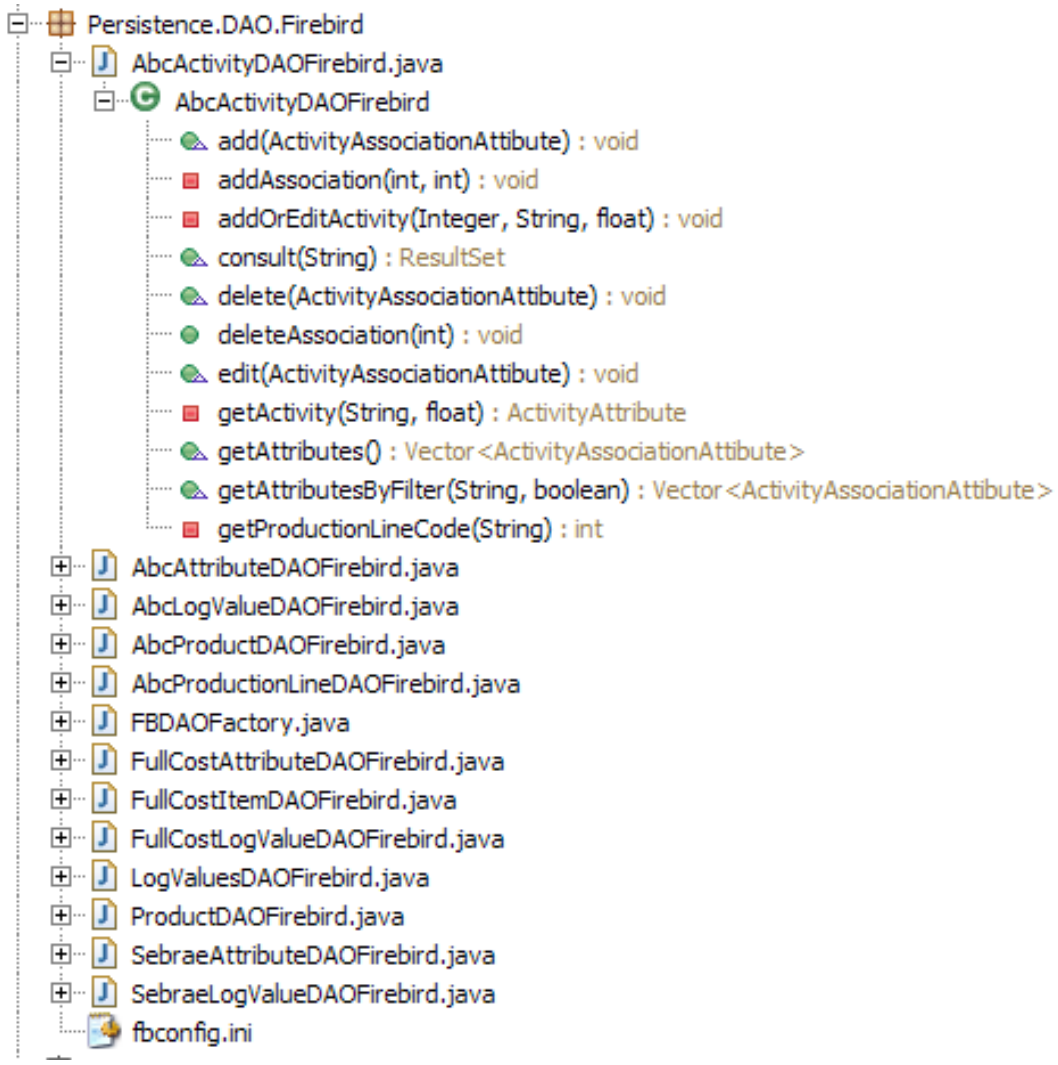

Figura 5 – Classes do Pacote *Persistence.DAO.Firebird*

Fonte: Autoria própria.

A Figura 5 ilustra também um exemplo dos métodos e atributos da classe *AbcActivityDAOFirebird* que implementa a adição de uma associação entre atividade e atributo (*add(ActivityAssociationAttribute); addAssociation(int,int)*), adicionar ou editar uma atividade (*addOrEditActivity(Integer, String, float),*

edit(ActivityAssociationAttibute)); consultar uma atividade (*consult(String)*), excluir uma atividade (*delete(ActivityAssociationAttribute), deleteAssociation(int)*), retornar um conjunto de atributo ou um atributo específico (*getActivity(String, float)*, *getAtributes()*), retornar os atributos que estão associados a uma atividade (*getAttributesByFilter(String, boolean)*) e obter o código do método ABC (*getProductionLineCode(String)*).

As outras classes são implementadas da mesma forma, porém considerando as particularidades de cada método. O Apêndice A apresenta os métodos e atributos das outras classes, exceto a *AbcActivityDAOFirebird*.

4 METODOLOGIA PARA A CRIAÇÃO DAS CLASSES DE TESTES

Este Capítulo descreve a metodologia base para a criação do processo usando para gerar as classes de testes para o FrameMK. A Seção 4.1 descreve a metodologia para o projeto de teste desenvolvida por Crespo et al. (2004). A Seção 4.2 relata sobre a metodologia adaptada. A Seção 4.3 apresenta a aplicação da metodologia no framework FrameMK.

4.1 METODOLOGIA BASE

Utilizou-se como base do processo da criação das classes de testes a metodologia desenvolvida por Crespo et al. (2004), conforme ilustra a Figura 6.

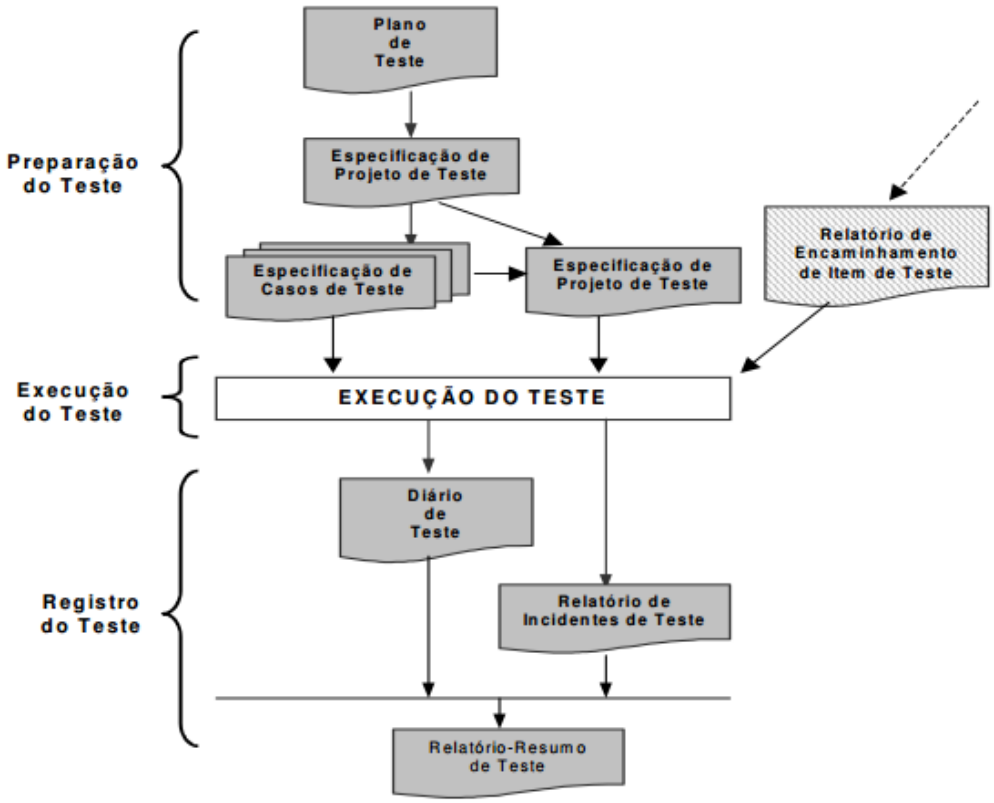

Figura 6 - Fluxograma do processo de criação das classes de teste
Fonte: Crespo et al. (2004, p. 8)

Nesta metodologia o plano de teste deve começar com a definição da estratégia de teste, que compreende os seguintes itens (CRESPO et al., 2004):

- Nível do teste: Definição da etapa do desenvolvimento do software em que os testes serão aplicados. Pode ser aplicado para verificar: a codificação nos módulos do sistema (teste de unidade), a forma de como estes módulos interagem entre si (teste de integração), se o sistema está funcionando de acordo com os requisitos solicitados (teste de sistema), se o cliente aprova a maneira de como os requisitos foram resolvidos (teste de aceitação) ou para verificar a integridade na manutenção do sistema (teste de integração).
- Técnica de teste: Especifica a finalidade para qual serão feitos os testes. Assim, como o nível do teste, este item depende da etapa em que os testes serão executados. Existem dois tipos de técnicas de teste: o teste estrutural que verifica a estrutura do software e o teste funcional que utiliza casos de teste para verificar se os requisitos solicitados foram solucionados corretamente.
- Critérios do teste: Define o que será testado no sistema, orientando a geração dos casos de teste. O critério de teste possui elementos e/ou características do software que devem ser testadas. Os casos de teste devem verificar os elementos e/ou características definidos nos critérios de teste.
- Tipos de teste: Quais aspectos do software serão testados. Os principais tipos de teste verificam a funcionalidade, interface, desempenho, carga, usabilidade, volume e segurança do sistema.

Definido o plano do teste é necessária a capacitação das pessoas que irão elaborar e realizar os testes criados. Este é um processo genérico e deve se adequar ao tipo de equipe e ao plano de teste (CRESPO et al., 2004).

Depois da capacitação, são criadas as especificações dos casos de teste e o cronograma que irá ser seguido para a execução dos casos de teste (projeto de teste) (CRESPO et al., 2004).

Após a execução dos testes os seus resultados são documentados para análise e correção, seguindo a norma IEEE 829-1998 (descrita na Seção 4.1.1) que descreve a geração de oito documentos no final da fase de registro de teste. Com estes documentos prontos, o gerente de teste analisa e encaminha novamente os itens que deverão ser re-testados até que o líder do projeto considere o produto adequado para instalação no cliente (BLANCO, 2012; CRESPO et al., 2004).

4.1.1 DOCUMENTOS DA NORMA IEEE 829-1998

A metodologia de Crespo et al. (2004) segue como referência o guia para elaboração de documentos de teste, que estão presentes desde o início (Plano de teste) até o final (Resumo de teste) do projeto de teste. O guia contém os seguintes elementos:

- Plano de teste – Gerado no início da execução do projeto de teste, define recursos, abrangência, abordagem, tempo, requisitos e documentos que serão criados durante todo o projeto de teste.
- Especificação do projeto de teste – Processa as informações do plano de teste em associações com os requisitos a serem testados e os testes a serem utilizados. Neste documento também é especificado os casos, procedimentos e critérios para os testes que serão executados.
- Especificação de casos de teste – Utiliza as informações processadas pela especificação do projeto de teste para criação dos casos de teste, validação dos dados de entrada e saída, pré-requisitos e resultados esperados.
- Especificação de procedimento do teste – Detalha os passos a serem seguidos para executar um ou mais casos de teste.
- Diário de teste – Registra cronologicamente eventos importantes que aconteceram durante a execução dos testes.
- Relatório de incidente de teste – Grava os eventos críticos durante a execução dos testes, que necessitam de uma análise posterior.
- Relatório resumo de teste – Apresenta de forma sucinta os resultados da execução dos testes em relação ao planejamento dos mesmos, com uma análise baseada nos resultados obtidos.
- Relatório de encaminhamento de Item de Teste – Este documento deve ser utilizado quando existe mais de uma equipe de teste, para identificar a origem e a situação dos itens encaminhados para teste.

Este trabalho adaptou a metodologia de Crespo et al. (2004) para ser usada no estudo de caso, lembrando que o foco desta pesquisa é a criação dos casos e classes de teste e a metodologia de Crespo et al. (2004) é bastante abrangente. A metodologia adaptada será descrita na próxima seção.

4.2 METODOLOGIA PROPOSTA PARA CRIAÇÃO DAS CLASSES DE TESTE

Para criação das classes de teste, a metodologia criada por Crespo et al. (2004) foi modificada e alguns documentos foram unificados e atualizados conforme a nova regra de criação de documentos de teste, a IEEE 898 (BLANCO, 2012). A metodologia de teste utilizada neste trabalho está ilustrada na Figura 7.

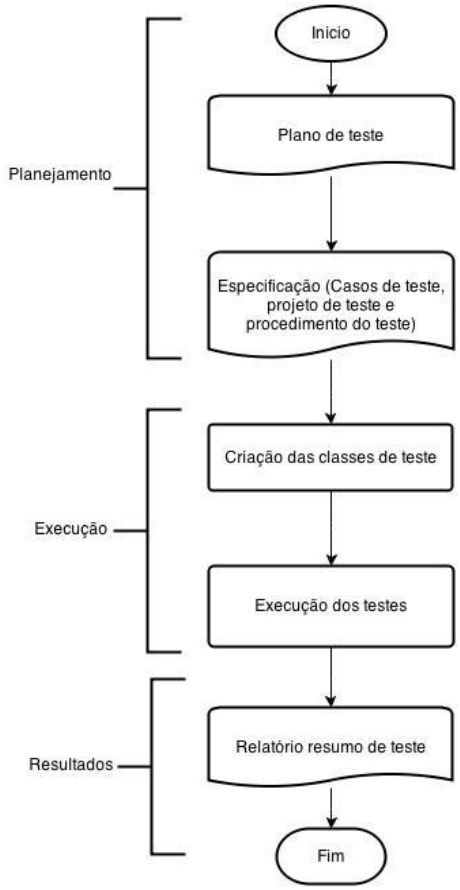

Figura 7 - Metodologia adaptada para geração das classes de teste
Fonte: Autoria própria.

O plano de teste foi direcionado para a camada de persistência do FrameMK, porém sua estrutura pode ser reutilizada em outras camadas do framework. Aplicou-

se os quatro itens principais da metodologia descrita por Crespo et al. (2004) para definir o plano de teste do projeto proposto neste trabalho, a saber:

1. Nível de teste: A camada de persistência do FrameMK consiste nos pacotes *DAO* e *Firebird*, os quais serão testados ao nível de código (teste de unidade).
2. Técnica de teste: Será testada a parte funcional do código, em outras palavras, será verificado se o código cumpre com os requisitos necessários para o funcionamento do framework.
3. Critérios de teste: Serão testadas as funções de recebimento e envio dos dados pelo framework, bem como seus objetos criados.
4. Tipos de teste: Serão realizados testes de funcionalidade nas classes de persistência do FrameMK.

Os documentos de especificação de teste, especificação de documentos de teste e especificação dos procedimentos de teste (casos de teste) foram unidos para facilitar a criação e manutenção dos casos e classes de teste, resultando no documento do plano de teste, que está ilustrado no Apêndice B deste trabalho.

Os elementos que compõe os casos de teste tais como: nome, status, pacotes, entre outros, foram tirados da norma 898-2008 (BLANCO, 2012). Alguns itens tais como: classes pacotes de teste foram incorporados para atender as necessidades deste trabalho. Os itens utilizados na criação dos elementos do caso de teste que estão na documentação do IEEE estão listados abaixo (IEEE Computer Society, 2008).

- Design de teste: Identificam os refinamentos da abordagem de testes, os recursos que serão testados e os testes associados a esta abordagem.
- Caso de teste: define as informações necessárias das entradas e saídas do software que será testado.
- Procedimento de teste: Especificar os passos para execução de um conjunto de casos de teste ou mais generalizadamente, os passos usados para exercitar o produto de software para avaliar um conjunto de recursos.
- Registro de teste: Provê um registro cronológico dos detalhes relevantes da execução dos testes. Uma ferramenta automatizada pode capturar todo ou parte destas informações.

Alguns itens do caso de teste foram inseridos ou alterados para se adaptarem a necessidade da criação das classes de teste no FrameMK, por exemplo, os cenários, métodos de teste, resultados esperados e resultados obtidos podem ter mais de um valor, de acordo com a quantidade de cenários testada em um caso de teste.

Número	Número integral, sequencial e único que registra o caso de teste (por exemplo: 0001)
Nome	Nome do caso de teste (por exemplo: *Teste de Inserção de Cliente*)
Status	Verificar se o caso de teste já foi realizado, neste caso, recebe a palavra reservada *Testado,* do contrário *Pendente*
Pacote	Nome do pacote que será realizado o teste (por exemplo: *Venda*)
Classe	Nome da classe que será realizado o teste (por exemplo: *Cliente*)
Pré-condição	Identificar dependência dos objetos dentro do método (por exemplo, se no *método inserirCliente()* ele instância um objeto do tipo Conexão, este deveria ser listado neste item)
Método	Nome do método a ser testado (por exemplo, *inserirCliente()*)
Pacote de teste	Nome do pacote em que a classe de teste será codificada (por exemplo, segue-se o padrão do *JUnit* → *test.java.Venda*)
Classe de teste	Nome para a classe de teste (por exemplo, *ClienteTest*)
Método(s) de teste	Nome do método de teste (por exemplo, *inserirClienteTest*). Caso o método a ser testado tenha *N* cenários a serem validados, o nome do método de teste deve ser o nome do método seguido pelo número do cenário (por exemplo, *inserirClienteTest1,…, inserirClienteTestN*)
Cenário	Descrição do que será testado dentro do método (por exemplo, dados em branco). Lembrando que podem existir *N* cenários para um método.
Resultado esperado	Descrição do que se espera com o caso de teste (por exemplo, dados em branco não podem ser armazenados)
Resultado obtido	Descrição do resultado atingido com o método de teste (por exemplo, o sistema informa que os dados não podem ser armazenados, pois são nulos)
Data do último teste	01/10/2014

Quadro 2 – Modelo de Caso de teste

Fonte: Autoria Própria.

Após a criação dos casos de teste é necessária a instalação do ambiente para o correto funcionamento do framework. Os requisitos deste ambiente estão dentro do plano de teste que está ilustrado no Apêndice B.

Instalado o ambiente são criadas as classes de teste, que se baseiam nas informações inseridas nos casos de teste, conforme modelo do Quadro 2. Estas classes por sua vez são executadas e seus retornos são gravados no documento de resumo do teste, que serão analisados para confirmar os resultados deste trabalho.

4.3 APLICAÇÃO DA METODOLOGIA PROPOSTA

Seguindo o plano de teste do Apêndice B, os casos de teste devem focar as classes de persistências do FrameMK que estão dentro do pacote *Persistence.DAO.Firebird.* O pacote *Persistence.DAO* contém apenas classes do tipo *Interface*, em outras palavras, não há código para se testar nelas.

Deve ser criado um caso de teste para os métodos de cada classe do pacote a ser testado, como ilustram os Quadros 3, 4 e 5 para a classe *AbcActivityDAOFirebird*.

Número	0001
Nome	Testar associação entre a atividade e a linha de produto.
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory; Connection; CallableStatement
Método	addAssociation(int activityCode, int productionLineCode)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	addAssociationTest(int activityCode, int productionLineCode) addAssociationTest2(int activityCode, int productionLineCode) addAssociationTest3(int activityCode, int productionLineCode)
Cenário	Cenário1: Número da atividade e da linha de produto com valores nulos. Cenário2: Número da atividade da linha do produto com valores que não estejam cadastrados. Cenário3: Número de atividade e linha de produto estão cadastrados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve executar corretamente a associação.
Resultado obtido	
Data do último teste	

Quadro 3 - Caso de teste 0001

Fonte: Autoria própria.

Número	0002
Nome	Deletar associação entre atividade e linha do produto
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory; Connection; StringBuilder
Método	deleteAssociation(int codigo)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	deleteAssociationTest(int codigo) deleteAssociationTest2(int codigo) deleteAssociationTest3(int codigo)
Cenário	Cenário1: Número da associação nulo. Cenário2: Número da associação com valores não cadastrados. Cenário3: Número da associação com valores cadastrados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve deletar a associação corretamente.
Resultado obtido	
Data do último teste	

Quadro 4 - Caso de teste 0002
Fonte: Autoria própria.

Número	0003
Nome	Atualizar valores do produto
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcProductDAOFirebird
Pré-condição	FBDAOFactory; Connection; StringBuilder; AbcProduct
Método	saveValuesTest(AbcProduct abcProduct) saveValuesTest2(AbcProduct abcProduct) saveValuesTest3(AbcProduct abcProduct)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcProductDAOFirebirdTest
Método(s) de teste	saveValues(AbcProduct abcProduct)
Cenário	Cenário1: Objeto nulo. Cenário2: Objeto com valores inexistentes no banco de dados. Cenário3: Objeto com valores existentes no banco de dados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção a mensagem de erro “Problema em atualizar valor” Cenário3: Deve atualizar a associação corretamente.
Resultado obtido	
Data do último teste	

Quadro 5 - Caso de teste 0003
Fonte: Autoria própria.

Foram realizados casos de teste para os principais métodos do pacote, somando 16 casos de testes ao todo. Os restantes dos casos de teste estão descritos no documento de plano de teste, que pode ser encontrado no Apêndice B deste trabalho.

Algumas classes possuíam métodos semelhantes com os casos de teste já criados e visando poupar a quantidade de testes realizados nesta camada, optou-se por não duplicar casos de teste para métodos com a mesma implementação. Desta forma, criou-se uma tabela de relacionamento com o intuito de exibir os métodos das classes com seus respectivos casos de teste, conforme ilustra o Quadro 6.

Classe	Método	Número do caso de teste
AbcActivityDAOFirebird	addAssociation(int activityCode, int productionLineCode)	0001
	deleteAssociation(int codigo)	0002
	consult(String sql)	0004
	add(ActivityAssociationAttibute attribute)	0005
	addOrEditActivity(Integer codigo, String descricao, float custo)	0006
	getProductionLineCode(String description)	0007
	getActivity(String descricao, float custo)	0008
	edit(ActivityAssociationAttibute attribute)	0009
	delete(ActivityAssociationAttibute attribute)	0010
	deleteAssociation(int codigo)	0011
	getAttributes()	0012
	getAttributesByFilter(String filter, boolean filterIsTheCode)	0013
AbcLogValueDAOFirebird	setSalePrice(AbcLogValue abcLogValue, double value)	0014
AbcProductDAOFirebird	listartable(int code)	0015
	saveValues(AbcProduct abcProduct)	0003
	saveValues(int methodCode, int iteration, int productCode, int attributeCode, double value, String date)	0016

Quadro 6 - Relacionamento entre método e classe de teste

Fonte: Autoria Própria.

As classes de teste criadas a partir dos casos de testes estão descritas no próximo capítulo.

5 RESULTADOS

Este Capítulo mostra os resultados obtidos por este trabalho. A Seção 5.1 apresenta algumas classes de testes que foram criadas a partir dos casos elaborados na seção 4.3. A Seção 5.2 apresenta uma estatística sobre as classes de testes que obtiveram sucesso ou falha.

5.1 CLASSES DE TESTES PARA A CAMADA DE PERSISTÊNCIA

Conforme anteriormente mencionado, foram elaborados 16 (dezesseis) casos de testes e como cada caso de teste tinha em média 3 (três) cenários por método, então foram elaborados 49 (quarenta e nove) métodos de testes.

As classes de teste foram implementadas utilizando *JUnit*, porém os métodos de teste foram customizados para se adaptarem aos casos de teste que contém um conjunto de cenários, visto que a ferramenta gera um método de teste para cada método da classe de aplicação.

Alguns métodos de teste implementados necessitam de ajustes para serem testados, como por exemplo o caso de teste 0001 necessita que o registro utilizado para adicionar a associação seja apagado do banco de dados, para ser incluído novamente pelo teste. Isto significa que os testes devem ser gerados utilizando uma base de dados de testes, para evitar perda e corrupção de dados na base de dados de produção. No JUnit os ajustes feitos para execução dos testes são definidos no método *setUp()* conforme ilustra a Figura 8.

```
@Before
public void setUp() throws Exception {
    abcActivityDAOFirebird = new AbcActivityDAOFirebird();

    Connection con;

    con = FBDAOFactory.connect();
    con.prepareStatement("delete from atividade_linha").executeUpdate();
    con.prepareStatement("delete from atividade where codigo > 3").executeUpdate();
    FBDAOFactory.disconnect();
}
```

Figura 8 - Método *setUp()* do caso de teste 0001
Fonte: Autoria própria.

Considerando o caso de teste 0001 elaborado para a classe *AbcActivityDAOFirebird* criou-se a classe de teste ilustrada na Figura 9.

```
// Caso de teste 0001
public boolean addAssociationTest(int activityCode, int productionLineCode) {
    parameterTypes = new Class[2];
    parameterTypes[0] = int.class;
    parameterTypes[1] = int.class;
    try {
        method = abcActivityDAOFirebird.getClass().getDeclaredMethod(
                "addAssociation", parameterTypes);
    } catch (Exception e) {
        fail("Falha na localização do método" + e.toString());
    }
    method.setAccessible(true);
    parameters = new Object[2];
    parameters[0] = activityCode;
    parameters[1] = productionLineCode;
    try {
        method.invoke(abcActivityDAOFirebird, parameters);
        return true;
    } catch (Exception e) {
        return false;
    }
}

// Cenário 1: Número da atividade e da linha de produto com valores nulos.
@Test
public void addAssociationTest() {
    assertFalse(addAssociationTest(-1, -1));
}

// Cenário 2: Número da atividade da linha do produto com valores que não
// estejam cadastrados.
@Test
public void addAssociationTest2() {
    assertFalse(addAssociationTest(9999, 9999));
}

// Cenário 3: Número de atividade e linha de produto estão cadastrados.
@Test
public void addAssociationTest3() {
    if (!addAssociationTest(3, 1)) {
        fail("Falha na execução do método");
    } else {
        ResultSet rs = null;
        try {
            Connection con = FBDAOFactory.connect();
            Statement stm = con.createStatement();
            rs = stm.executeQuery("select 1 from atividade_linha "
                    + " where codigo_atividade = 3 "
                    + " and codigo_linha = 1 ");
            FBDAOFactory.disconnect();
            if (rs == null) {
                fail("Método não inseriu a associação");
            }
        } catch (Exception e) {
            fail("Falha na execução do método sql");
        }
    }
}
```

Figura 9 - Métodos de teste do cenário 0001
Fonte: Autoria própria.

Para acessar o método privado *addAssociation(int activityCode, int productionLine)* da classe *AbcActivityDAOFirebird* o método *addAssociationTest(int activityCode, int productionLine)* utiliza um conceito chamado reflexão computacional, que executa uma interceptação do objeto em tempo de execução, permitindo assim a utilização dos seus métodos privados. Este método também cria um ambiente propício ao reuso, utilizado pelos três cenários de teste do *caso de teste 0001*.

Os métodos de teste utilizam o método *addAssociationTest(int activityCode, int productionLine)* e outros métodos de *Assert*. O método *assertFalse* verifica a saída do método testado e sinaliza como teste realizado com sucesso se esta saída vier com o valor booleano *False*, assim como o método *assertTrue*, que sinaliza com sucesso se a saída for *True*.

O método *addAssociationTest3()* além de fazer o teste de inserção também verifica se no banco de dados os valores solicitados foram realmente inseridos, utilizando o método *fail()* nativo do JUnit para sinalizar um teste com erro. Note que não se utilizou o Mock para objetos de classe de persistência, devido a baixa complexidade do sistema e por alguns métodos possuírem uma grande parte da lógica dentro de *stored procedures* do banco de dados.

As classes de teste *AbcActivityDAOFirebirdTest*, *AbcLogValueDAOFirebirdTest* e *AbcProductDAOFirebirdTest* estão presentes na íntegra nos Apêndices C, D e E, respectivamente.

A classe de teste criada é executada pelo framework *JUnit*, que retorna uma janela com os resultados do teste, conforme ilustra a Figura 10.

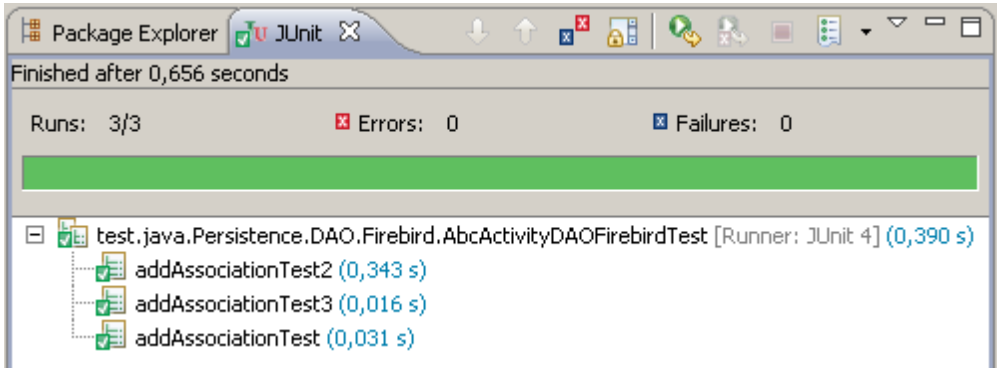

Figura 10 - Resultados da execução do caso de teste 0001
Fonte: Autoria própria.

No caso de teste 0001 todos os testes executados foram realizados com sucesso. Nos cenários de teste 1 e 2 para ser sinalizado com sucesso o método

deve falhar criando exceções, enquanto no cenário 3 o método deve ser executado com sucesso e o registro no banco de dados deve ser criado.

O caso de teste 0002 não utiliza o conceito de reflexão, devido sua visibilidade ser pública, porém este teste depende do método *addAssociationTest(int activityCode, int productionLine)* para seu funcionamento, já que para deletar um registro no banco de dados é necessário obter um registro válido. A Figura 11 ilustra os métodos de teste criados para execução do caso de teste 0002.

```
// Caso de teste 0002
// Cenário 1: Número da associação nulo.
@Test(expected = Exception.class)
public void deleteAssociationTest() throws Exception {
    abcActivityDAOFirebird.deleteAssociation(-1);
}

// Cenário 2: Número da associação com valores não cadastrados.
@Test(expected = Exception.class)
public void deleteAssociationTest2() throws Exception {
    abcActivityDAOFirebird.deleteAssociation(9999);
}

// Cenário 3: Número da associação com valores cadastrados.
@Test
public void deleteAssociationTest3(){
    int codigo;
    if (!addAssociationTest(3, 1)) {
        fail("Falha na adição de uma nova associação");
    } else {
        ResultSet rs = null;
        try {
            Connection con = FBDAOFactory.connect();
            Statement stm = con.createStatement();
            rs = stm.executeQuery("select codigo from atividade_linha order by codigo desc");
            FBDAOFactory.disconnect();
            if (rs == null) {
                fail("Método não inseriu a associação");
            } else {
                rs.next();
                codigo = rs.getInt("codigo");
                abcActivityDAOFirebird.deleteAssociation(codigo);
                con = FBDAOFactory.connect();
                stm = con.createStatement();
                rs = stm.executeQuery("select 1 from atividade_linha where codigo =" +
                        " " + Integer.toString(codigo));
                FBDAOFactory.disconnect();

                if (rs == null) {
                    fail("Método não deletou a associação");
                }
            }
        } catch (Exception e) {
            fail("Falha na execução do método sql");
        }
    }
}
```

Figura 11 - Métodos de teste do cenário 0002
Fonte: Autoria própria.

Conforme ilustra a Figura 11 a classe de teste utilizou o parâmetro *@Test*. Este parâmetro possui uma expectativa de retorno, que caso não atendida o teste é sinalizado com erro. O resultado deste teste não obteve sucesso em todos os cenários, conforme ilustra a Figura 12. A causa deste resultado é a implementação do método, que não possui um tratamento adequado de valores incorretos, diferente do método *addAssociation(int activityCode, int productionLine)* que possui em seu registro do banco de dados uma chave estrangeira que bloqueia a inserção de dados inválidos.

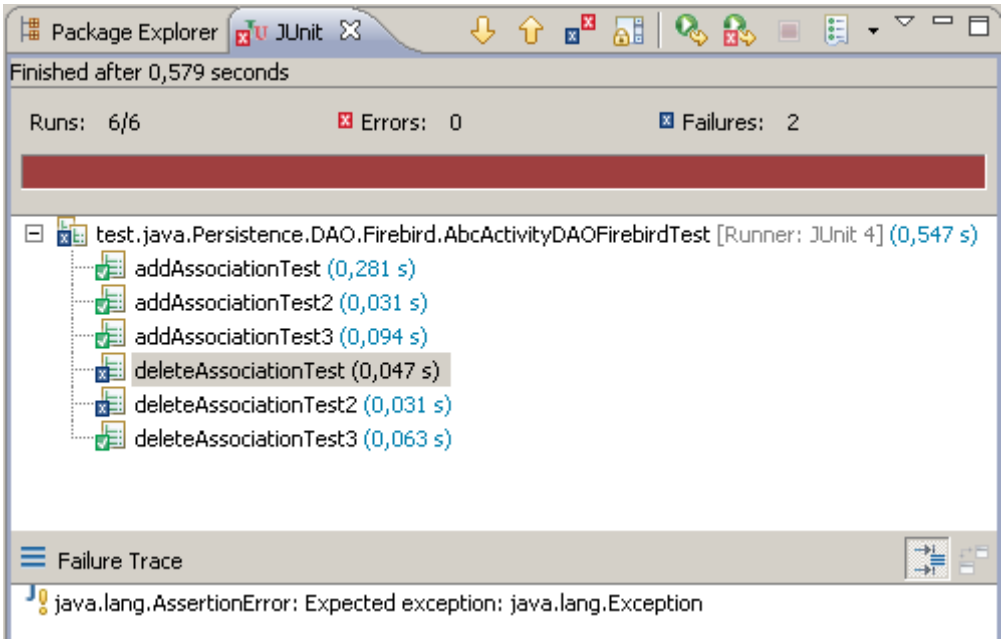

Figura 12 - Resultados da execução do caso de teste 0002
Fonte: Autoria própria.

Nos resultados do *caso de teste 0002* ilustrado na Figura 12 é possível observar o caminho de falha do teste, onde é descrito que o teste estava esperando por uma exceção, o que não ocorreu, sinalizando o teste como insucesso.

O *caso de teste 0003* utiliza um objeto como parâmetro para teste. Neste caso de teste é testado tanto a lógica da aplicação (valor nulo) quanto à lógica da *stored procedure* do banco de dados (valor inválido), conforme ilustra a Figura 13.

```
private AbcProduct abcProduct;
private AbcProductDAOFirebird abcProductDAOFirebird;
@Before
public void setUp() throws Exception {
      abcProductDAOFirebird = new AbcProductDAOFirebird();
      abcProduct = new AbcProduct();
}
//Caso de teste 0003
//Cenário1: Objeto nulo.
@Test(expected = Exception.class)
public void saveValuesTest() throws Exception {
     abcProduct = null;
     abcProductDAOFirebird.saveValues(abcProduct);
}

//Cenário2: Objeto com valores inexistentes no banco de dados.
@Test(expected = Exception.class)
public void saveValuesTest2() throws Exception {
     abcProduct = new AbcProduct();
     abcProduct.setCode(999999999);
     abcProduct.setValue(999999999);
     abcProductDAOFirebird.saveValues(abcProduct);
}

//Cenário3: Objeto com valores existentes no banco de dados.
@Test
public void saveValuesTest3() throws Exception {
     abcProduct = new AbcProduct();
     abcProduct.setCode(1);
     abcProduct.setValue(11);
     abcProductDAOFirebird.saveValues(abcProduct);
```

Figura 13 - Métodos de teste do cenário 0003
Fonte: Autoria própria.

Todos os métodos de teste do cenário 0003 foram executados com sucesso, pois a implementação deste método prevê entradas inválidas ou nulas, criando exceções e impedindo que registros incorretos sejam gravados no sistema. A Figura 14 ilustra os resultados deste caso de teste.

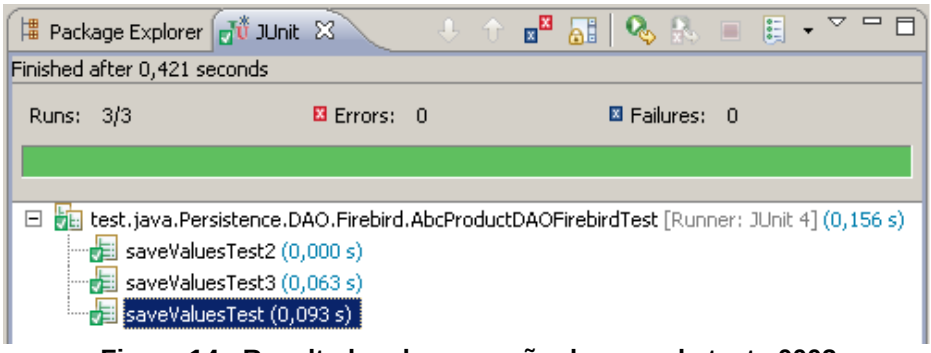

Figura 14 - Resultados da execução do caso de teste 0003
Fonte: Autoria própria.

Os métodos de teste criados nestas classes de testes podem ser reutilizados em outras classes, pois na camada de persistência a complexidade do código é menor, resultando em uma oportunidade de reuso em outras classes desta camada.

5.2 IMPORTÂNCIA DA APLICAÇÃO DOS MÉTODOS DE TESTE

Como se pode observar na seção anterior, os resultados dos testes unitários estão intimamente relacionados a implementação escolhida e na forma de capturar entradas e saídas incorretas nos métodos.

Um conjunto de cenários maior poderia revelar até mais erros na implementação, uma vez que estes somente testam a funcionalidade dos métodos, não a sua estrutura. Com um conjunto de métodos eficazes o desenvolvedor pode organizar seu código mais rapidamente, pois ao realizar a manutenção nos métodos as classes para teste e funcionamento deles já estão prontas.

Este conceito de criação de testes antes mesmo da lógica é chamado de *Test Driven Development* ou TDD, sendo utilizado para uma criação de um sistema de qualidade superior, pois ao testar todas as entradas e saídas possíveis de um sistema. Desta forma a segurança dos dados que entrarão e sairão da aplicação serão mais consistentes, garantindo uma vida útil maior do software, diminuindo manutenções e aumentando a confiança do cliente, que fica satisfeito ao ver seus requisitos não apresentarem falhas na etapa de aceitação.

Optou-se a aplicação dos cenários e classes de teste na camada de persistência do FrameMK pela baixa complexidade da lógica das classes, que em sua maioria possuíam métodos para a utilização das operações de criação, leitura, atualização e exclusão de dados. Os cenários e classes de testes criados podem ser utilizados como referência para criação de outros cenários e classes de teste, porém como foram especificados para a camada de persistência do FrameMK não podem ser diretamente reutilizados em outras camadas.

A adaptação da metodologia proposta por Crespo et al. (2004) foi imprescindível para a criação dos casos e classes de teste. A norma 898-2008 para criação dos documentos de teste também foram essenciais para o aumento da produtividade na produção do plano de teste e dos cenários de teste desenvolvidos para a camada de persistência do FrameMK.

6 CONCLUSÃO

Este trabalho adaptou uma metodologia para aplicação de testes usando como base a que foi proposta por Crespo et al. (2004). Esta adaptação foi necessária para direcionar o plano de teste para a camada de persistência do FrameMK. O plano foi composto por: nível, técnica, critérios e tipos de testes. Os documentos de especificação e procedimentos de testes foram integrados para facilitar a criação dos casos de testes. Os itens que faziam parte do modelo de casos de teste foram retirados da norma 898-2008 (por exemplo: número, *status*) e outros foram criados para atender aos requisitos deste trabalho, tais como: classes, pré-condição e cenários.

O desenvolvimento das classes de teste, baseadas nos casos de testes, foi possível após uma análise da arquitetura e instalação do FrameMK. Esta análise foi realizada procurando entender o funcionamento interno das classes e seus respectivos métodos. No processo de análise procurou-se estudar os trabalhos que já haviam sido desenvolvidos pelos acadêmicos que estavam ou estão envolvidos no desenvolvimento do framework.

Os casos de testes criados receberam um número sequencial para facilitar o relacionamento de métodos com comportamento similares. Ou seja, verificou-se que duas classes distintas possuíam métodos com o mesmo comportamento, mudando somente o tipo de objeto. Portanto, para métodos com comportamento semelhante criou-se apenas um método de teste que atendesse este funcionamento.

As classes de testes foram codificadas no framework *JUnit* usando como base os casos de testes gerados no Plano de Teste. Na implementação das classes de testes padronizou-se a inserção de um comentário acima do método de teste a fim de indicar qual cenário do caso de teste ele está atendendo.

A criação das classes de testes permitiu avaliar a camada de persistência do FrameMK, deixando-a mais consistente em termos do que já estava implementado e facilitando o processo de inserção de novas funcionalidades nas classes já geradas.

6.1 TRABALHOS FUTUROS

Como trabalho futuro a este, podem ser desenvolvidas pesquisas em:

- Aprimoramento dos casos de teste: Alterar os tipos de casos de testes para que possam suportar outras técnicas de teste. Como teste estrutural, por exemplo.
- Inclusão do *Mock* de objetos: Para lógicas mais complexas o *Mock* pode ajudar no reuso do código e facilitar a criação dos testes com banco de dados.
- Aplicação do teste em outras camadas do framework: O FrameMK possui várias camadas sem nenhum tipo de teste de unidade, o que abre uma grande oportunidade para verificação e ajuste no sistema através de casos de teste consistentes, que podem até servir de base de uma documentação da funcionalidade dos métodos das classes do framework.
- Adicionar à metodologia a utilização de novos conceitos de desenvolvimento orientado a testes que possuem compatibilidade com os documentos criados na metodologia deste trabalho, seguindo as mesmas diretrizes de documentação utilizadas (IEEE 828-2008).

REFERÊNCIAS

BARBOSA, A. P.; BELUZZO, L. B. **Um Processo de Extensão de Framework de Domínio: Um Estudo de Caso no FrameMK (Framework para Formação de Preço de Venda).** 2014. Trabalho de Conclusão de Curso. (Graduação em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2014.

BARBOSA, Ellen Francine; MALDONADO, José Carlos; VINCENZI, Auri Marcelo Rizzo; DELAMARO, Márcio Eduardo; SOUZA, Simone do Rocio Senger de; JINO, Mario. Introdução ao teste de software**.** In: Simpósio Brasileiro de Engenharia de Software, 14, 2000. **Anais...** Disponível em: <www.inf.ufpr.br/silvia/topicos/apostilaUSP.pdf.gz>. Acesso em: 20 ago. 2014.

BERNARDO, Paulo Cheque; KON, Fabio. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54-57, 2008.

BIASI, Luciano Bathaglini. **Geração Automatizada de Drivers e Stubs de Teste para JUnit a partir de Especificações U2TP.** 2006. 153 f. Dissertação (Mestrado) - Curso de Ciência da Computação, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2006.

BIASI, Luciano Bathaglini; BECKER, Karin. Geração automatizada de drivers e stubs de teste para JUnit a partir de especificações U2TP. In: Simpósio Brasileiro de Engenharia de Software, 20, 2006, Florianópolis. **Anais eletrônicos...** Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbes/2006/003.pdf>. Acesso em: 24 ago. 2014.

BLANCO, M. Z. Documentação de teste baseado na Norma IEEE 829 – estudo de caso: “sistema de apoio a tomada de decisão”. **Revista T.I.S.** São Carlos, v. 1, n. 1, p. 91-97, 2012.

BOUASSIDA, N.; BEN-ABDALLAH, H.; BEN-HAMADOU, A. An Evaluation of the FBDM Framework Design Method. **Special Issue of the 8th MCSEAI**, v. 4, p. 1-16.2005.

BRYCE, R; KUHN, R. Software Testing [Guest editors' introduction]. **Journal Computer,** v. 47, n. 2, p. 21-22, 2014.

CAPELLER, P. E. B.; ANDRADE, V. C. **Uso do Processo Dirigido a Responsabilidades no Desenvolvimento da Arquitetura e Modelagem do Framework de Preço de Venda**. 2010, 164f. Trabalho de Conclusão de Curso - Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2010.

CHAN, Man-Yee; CHEUNG, Shing-Chi. **Testing Database Applications with SQL Semantics.** In: International Symposium on Cooperative Database Systems for Advanced Applications, 2, 1999, Wollongong (Australia). Anais Eletrônicos... Wollongong: International Symposium on Cooperative Database Systems for

Advanced Applications, 1999, p. 363-374. Disponível em: <http://www.cs.ust.hk/~scc/publ/CODAS99.pdf>. Acesso em: 22 ago. 2014.

CIRILO, Elder Jose Reioli. **GENARCH: UMA FERRAMENTA BASEADA EM MODELOS PARA DERIVAÇÃO DE PRODUTOS DE SOFTWARE.** 2008. 100 f. Dissertação (Mestrado) - Curso de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2008.

CLARK, Mike. **JUnit Primer.** 2005. Disponível em: < http://pesona.mmu.edu.my/~wruslan/SE1/Readings/detail/Reading-69.pdf>. Acesso em: 27 set. 2014

CRAZUSKI, A.; FEITOSA L. B.; CORDEIRO, T. L. **Identificação dos pontos de estabilidade e de flexibilidade dos métodos para o estabelecimento de preço de venda.** 2008. 157f. Trabalho de diplomação (Tecnologia em Análise e Desenvolvimento de Sistemas) – UTFPR, Ponta Grossa, 2008.

CRESPO, A. N.; SILVA, O. J.; BORGES, C. A.; SALVIANO, C. F.ARGOLLO JUNIOR, M. T.; JINO, M. Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo. **Simpósio Brasileiro de Qualidade de Software**, p. 271-285, 2004.

DALCIN, Sabrina Borba; FERREIRA, Adriane Pedroso Dias; D'ORNELLAS, Marcos Cordeiro. Utilizando Testes de Unidade no Ciclo de Desenvolvimento de Software para Processamento e Análise de Imagens. **RESI – Revista Eletrônica de Sistemas de Informação,** ed. 10, n. 1, p. 1-7, 2007.

DIAS, Adriane Pedroso; DALCIN, Sabrina Borba; D'ORNELLAS, Marcos Cordeiro. Aplicando Testes em XP com o Framework JUnit. **Journal of Computer Science,** v. 5, n. 2, 2006.

FANTINATO, Marcelo; CUNHA, Adriano Camargo Rodrigues da; DIAS, Sindo Vasquez; CARDOSO, Sueli Akiko Mizuno; CUNHA, Cleida Aparecida Queiroz. AutoTest – Um framework reutilizável para a automação de teste funcional de software. **Cad. CPqD Tecnologia,** Campinas, v. 1, n. 1, p. 119-131, jan./dez. 2005.

FAYAD, Mohamed E.; SCHMIDT, Douglas C.; JOHNSON, Ralph E.**Implementing application frameworks: object-oriented frameworks at work**. John Wiley & Sons, Inc., 1999.

GAMMA, E.R.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **DESIGN PATTERNS: ELEMENTS of REUSABLE OBJECT-ORIENTED SOFTWARE.** 1 ed, Estados Unidos: Addison-Wesley,1994.

GORDON, Jamie S.; ROGGIO, Robert F. A Comparison of Software Testing Using the Object-Oriented Paradigm and Traditional Testing. **Journal of Information Systems Applied Research**, 2014.

IEEE Computer Society. **Standard for Software and System Test Documentation 829-2008**. New York, 2008.

JUNIOR, C. R. **Um Web Service para busca de Preço de Venda.** 2010, 72f Trabalho de diplomação (Tecnologia em Análise e Desenvolvimento de Sistemas) – UTFPR, Ponta Grossa, 2010.

JUNIT. **JUNIT**. Disponível em: <http://www.junit.org>. Acesso em: 26 set. 2014.

LACERDA, V. S. **Refatoração do Aplicativo Gerenciador de Menus Dinâmicos do Sítio ArcaboMK.** 2012. Trabalho de Conclusão de Curso. (Graduação em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2012.

LANGR, Jeff. **Evolution of Test and Code Via Test-First Design.** 2001. Disponível em: < http://www.objectmentor.com/resources/articles/tfd.pdf>. Acesso em: 26 set. 2014.

MAZER JUNIOR, A. **Métodos de formação de preço de venda em sistemas ERP por intermédio de arquitetura orientada à serviços do framework frameMK.** 2013. 115 f. Dissertação (Mestrado em Engenharia de Produção: Gestão do Conh) - Programa Pós-Graduação em Engenharia de Produção, Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2013.

MYERS, G. J.; SANDLER, C.; BADGETT, T. The art of software testing. **Wiley,** v. 3, p. 2, 2004.

OLAN, M. Unit testing: test early, test often. **Journal of Computing Sciences in Colleges**, v. 19, p. 319-328, Dec., 2003.

OLEGÁRIO, P. L. **Suporte a Teste de Unidade de Aplicativos J2ME no Ambiente Eclipse.** 2005. 72 f. Trabalho de diplomação (Curso de Engenharia da Computação), Universidade de Pernambuco, Recife, 2005.

PAŁKA, M. H. **Random Structured Test Data Generation for Black-Box Testing.** 2014. 76 f. Tese (Doutorado) - Curso de Philosophy, Chalmers University Of Technology And Göteborg University, Sweden, 2014.

POLO, M; REALES, P.; PIATTINI, M.; EBERT, C. Test automation. **IEEE Software**, v. 30, p. 84-89. 2013.

PRESSMAN, Roger S. **Engenharia de software:** uma abordagem profissional. 7. ed. Porto Alegre: Bookman, 2011.

RAMOS, R. **Refatoração da camada de apresentação do framework de preço de venda (Framemk).** 2011, 64f. Trabalho de diplomação (Tecnologia em Análise e Desenvolvimento de Sistemas) – UTFPR, Ponta Grossa, 2011.

RUDNIK, R.; CREMA, R. J. C. **Definição dos Pontos de Estabilidade e Flexibilidade, em nível de Requisitos, no Domínio de Preço de Venda.** 2009. Trabalho de Conclusão de Curso. (Graduação em Tecnologia em Análise e Desenvolvimento de Sistema) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2009.

SILVA, L. S. **Um Método para Identificação de Aspectos em Nível de Análise Baseado em Atributos de Requisitos Não-Funcionais**. 2012. Trabalho de Conclusão de Curso. (Graduação em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2012.

SILVA, R. P.; PRICE, R. T. **A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos.** In: Workshop Iberoamericano de Requisitos e Ambientes de Software, 2, 1998. Anais... Torres/RS: Instituto de Informática (UFRGS) 1998. p. 298-309.

SOMMERVILLE, I. **Engenharia de software.** 8. ed. São Paulo: Pearson Education Companion, 2007.

TALIGENT. **Building object-oriented frameworks.** A Taligent White Paper. 1994.

VAZ, Rodrigo Cardoso. **JUnit - Framework para Testes em Java.** 2003. Disponível em: <http://arquivo.ulbra-to.br/ensino/43020/artigos/relatorios2003-2/TCC/JUnit.pdf >. Acesso em: 20 ago. 2014.

WILLMOR, David; EMBURY, Suzanne M. **An Intensional Approach to the Specification of Test Cases for Database Applications.** In: International Conference on Software Engineering, 28, 2009, Shanghai (China). Anais eletrônicos... Disponível em: <http://www.cs.man.ac.uk/~willmord/files/WillmorEmbury-ICSE06.pdf >. Acesso em: 22 ago. 2014.

ZAHAIKEVITCH, E. V. **Sistema Especialista para Identificação do Método de Custeio para Formação de Preço de Venda.** 2014. Dissertação (Mestrado em Mestrado em Engenharia de Produção: Gestão do Conh) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2014.

ZHANG, L.; MU, X.; ZHANG, H.; SONG, W.Comparison between Object-Oriented Software Testing and Traditional Software Testing**. Applied Mechanics and Materials**, v. 411, p. 497-500, 2013.

ZHANG, Lingming. **Unifying regression testing with mutation testing.** 2014. 230 f. Tese (Doutorado) - Curso de Philosophy, The University Of Texas At Austin, Austin, 2014.

APÊNDICE A – MÉTODOS E ATRIBUTOS DAS CLASSES DO PACOTE *PERSISTENCE.DAO.FIREBIRD*

Apresentam-se nas Figuras 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26 as especificações das classes do pacote *Persistence.DAO.Firebird*.

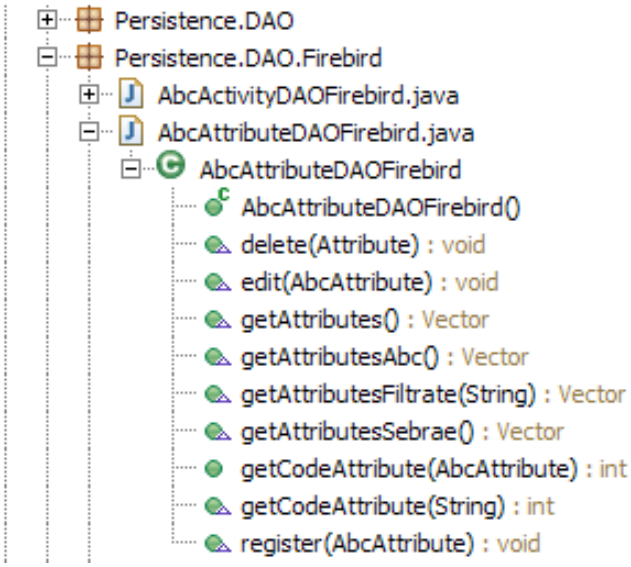

Figura 15 - Especificação da classe AbcAttributeDAOFirebird

Fonte: Autoria própria.

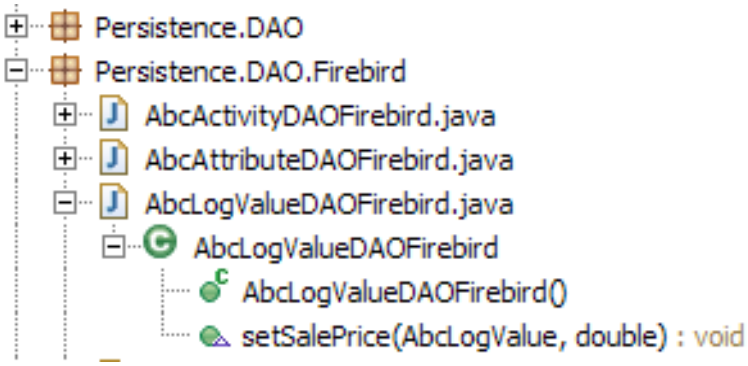

Figura 16 - Especificação da classe AbcLogValueDAOFirebird

Fonte: Autoria própria.

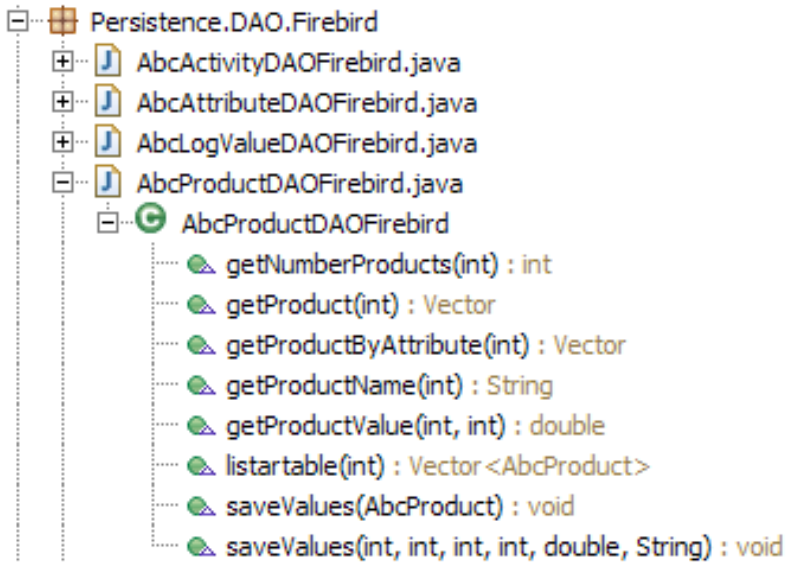

Figura 17 - Especificação da classe AbcProductDAOFirebird

Fonte: Autoria própria.

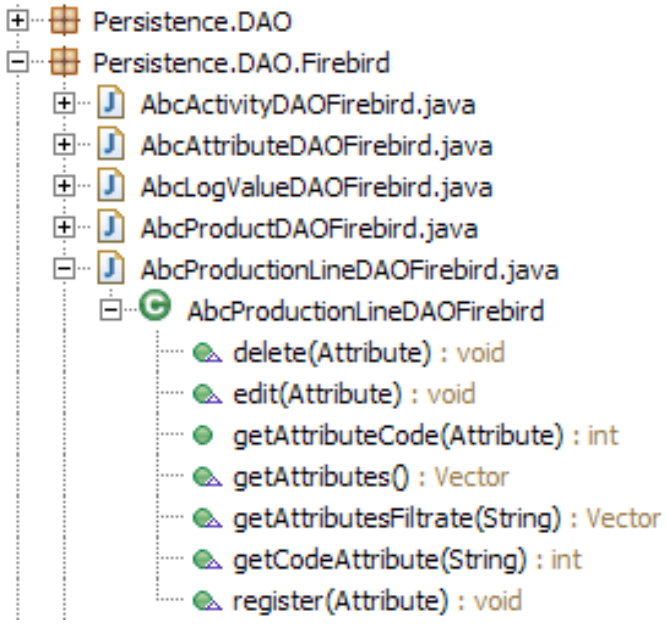

Figura 18 - Especificação da classe AbcProductionLineDAOFirebird

Fonte: Autoria própria.

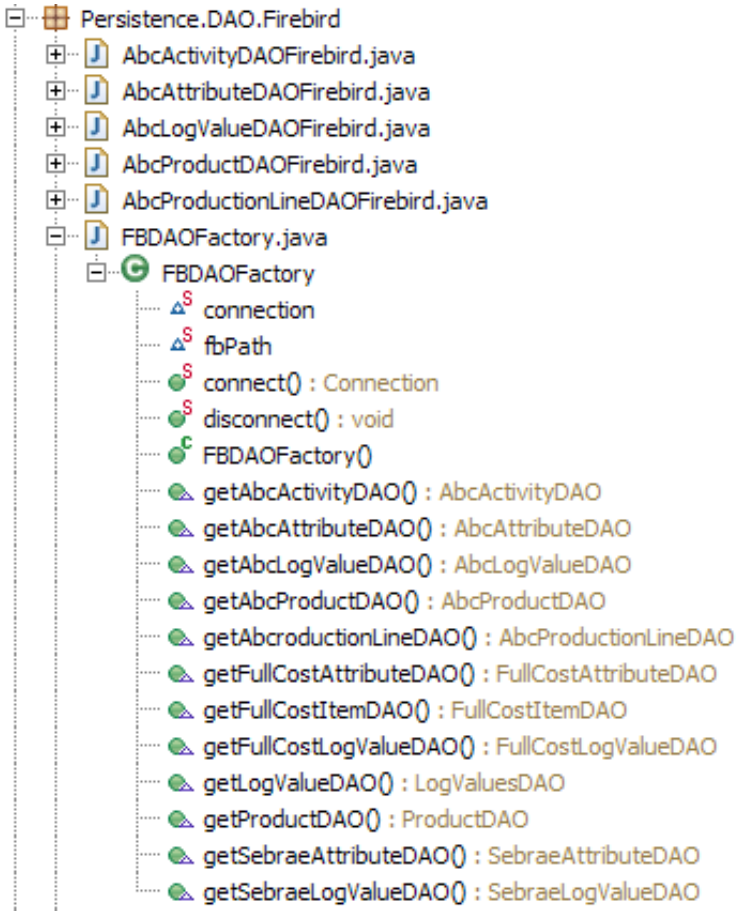

Figura 19 - Especificação da classe FBDAOFactory

Fonte: Autoria própria.

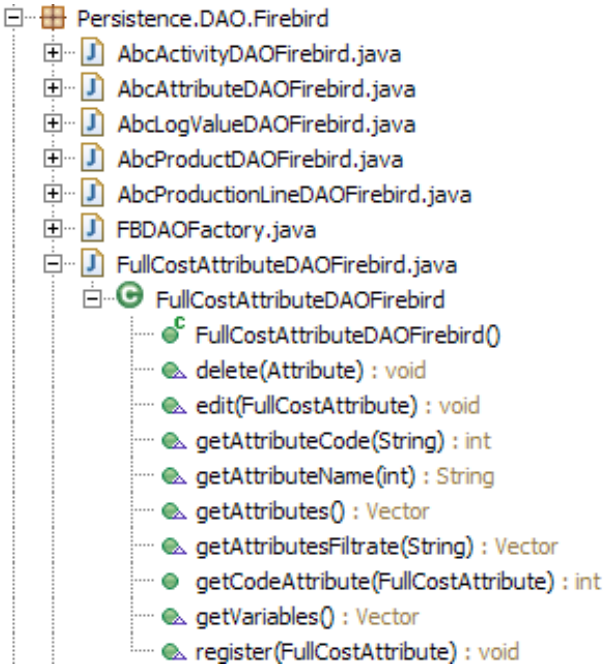

Figura 20 - Especificação da classe FullCostAttributeDAOFirebird

Fonte: Autoria própria.

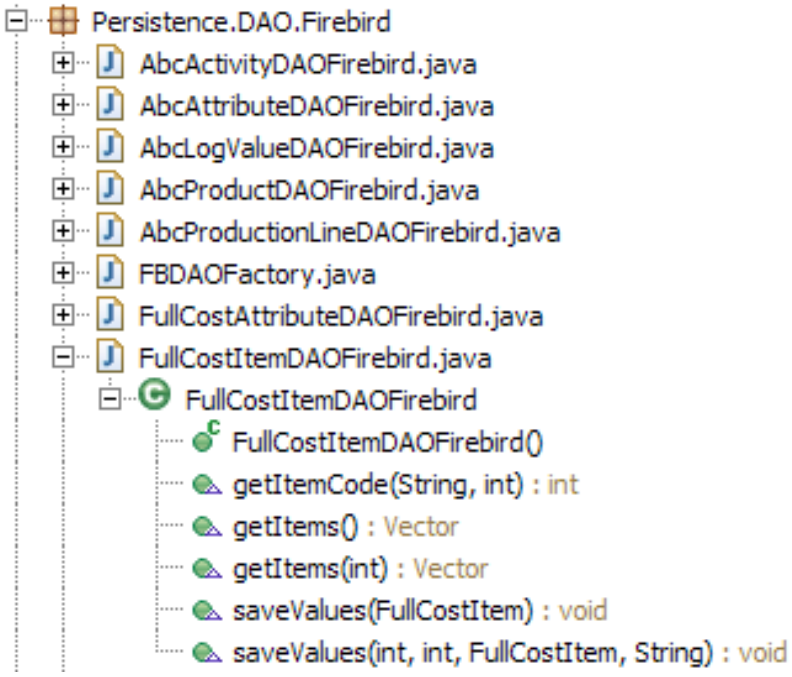

Figura 21 - Especificação da classe FullCostItemDAOFirebird

Fonte: Autoria própria.

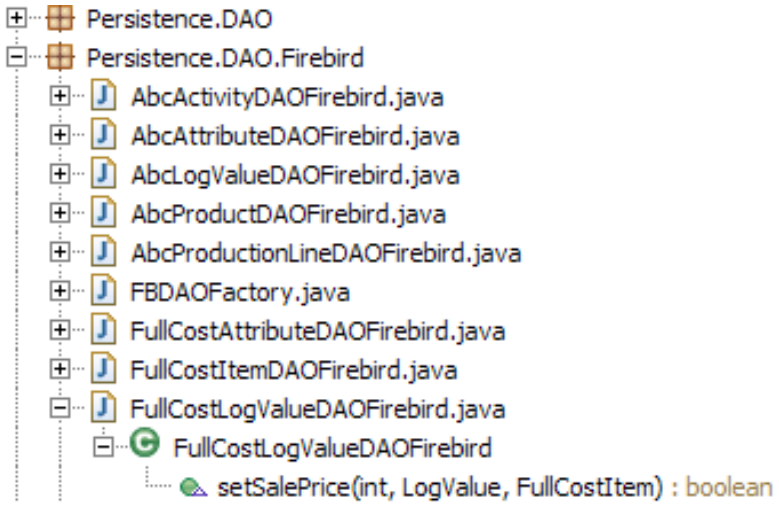

Figura 22 - Especificação da classe FullCostLogValueDAOFirebird

Fonte: Autoria própria.

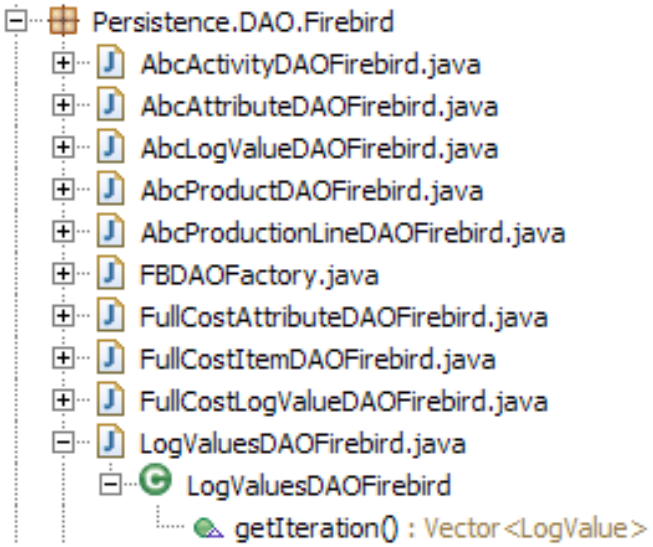

Figura 23 - Especificação da classe LogValuesDAOFirebird

Fonte: Autoria própria.

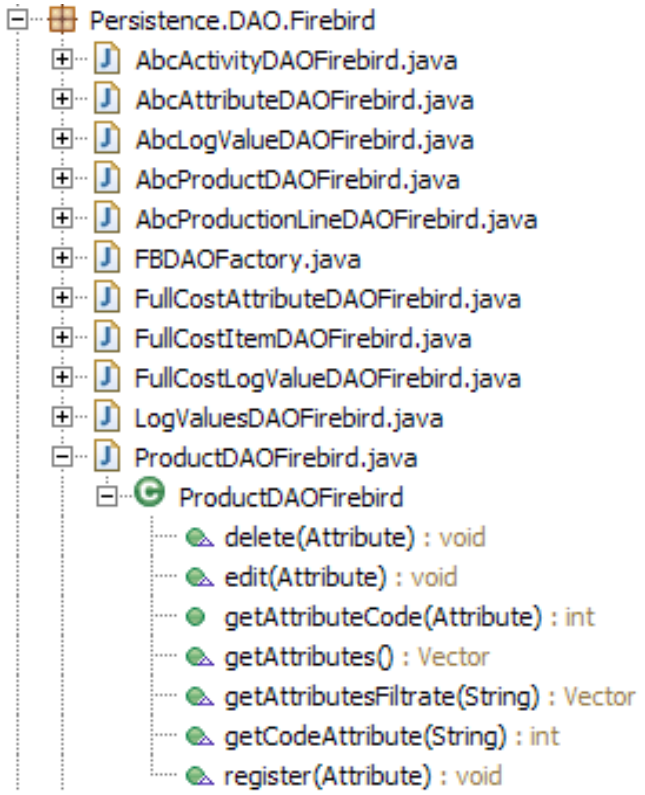

Figura 24 - Especificação da classe ProductDAOFirebird

Fonte: Autoria própria.

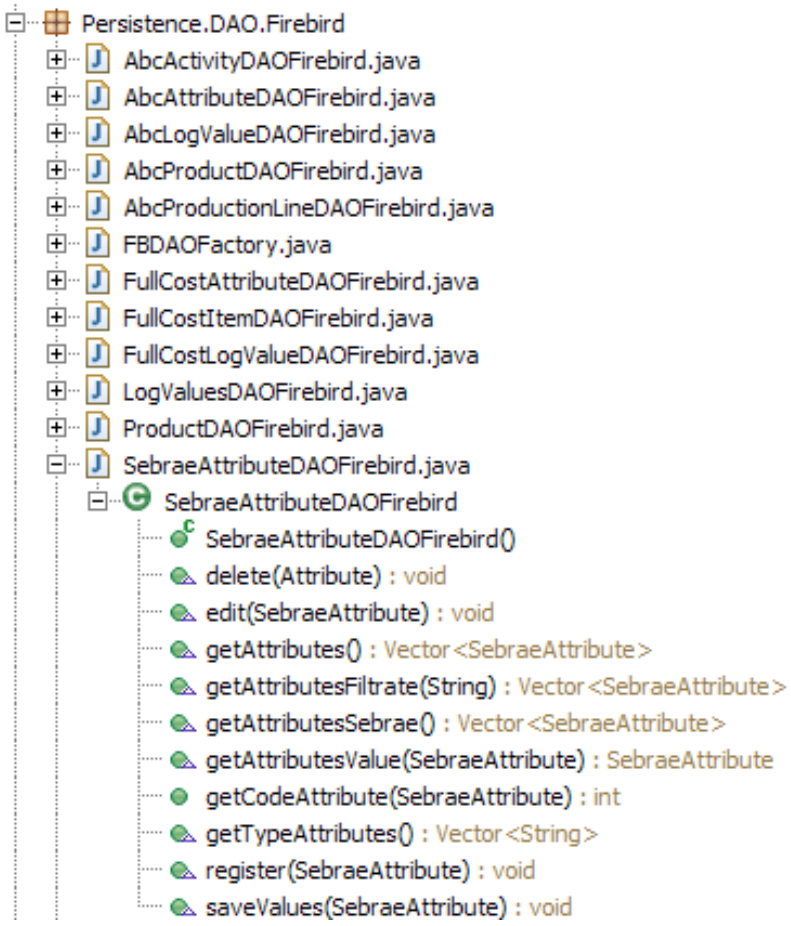

Figura 25 - Especificação da classe SebraeAttributeDAOFirebird

Fonte: Autoria própria.

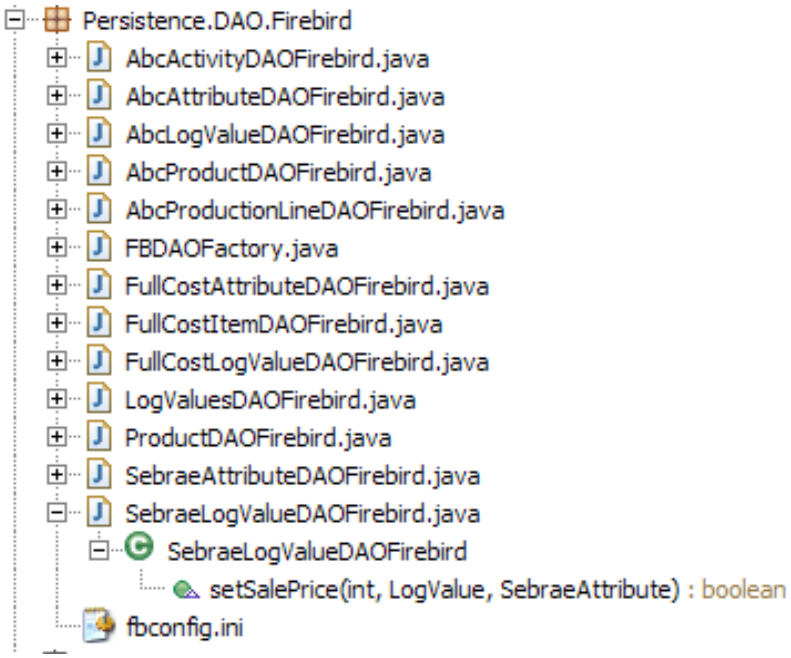

Figura 26 - Especificação da classe SebraeLogValueDAOFirebird

Fonte: Autoria própria.

APÊNDICE B – PLANO DE TESTE

PLANO DE TESTE

Este documento visa estabelecer e definir os itens essenciais para o planejamento, criação, execução e documentação do projeto de teste da camada de persistência do framework de preço de venda, FrameMK.

Os itens deste plano de teste foram baseados na norma IEEE 829-2008 para criação de documentos para teste.

1) Nome do projeto

O nome do projeto de teste será definido com base no nível e no critério de teste que serão utilizados, neste caso será chamado “Testes de unidade na camada de persistência do FrameMK”.

2) Pessoas/Responsabilidades

Este item agrupa os recursos humanos disponíveis para este projeto de teste:

- Professora Simone Nasser Mattos: Criadora e revisora dos casos de teste;
- Jonathan Heverson Ribas: Criador de casos de teste e desenvolvedor de testes de unidade.

3) Módulos

Classes de persistência pertencentes ao pacote *Persistence.DAO.Firebird*.

4) Hardware/Software mínimo necessário

Pentium 4, 512 mb de RAM com Windows XP, Java 1.5 eclipse e Firebird Super Server.

5) Cronograma

Data de início dos testes: 25/10/2014.
Data de término dos testes 31/10/2014.

6) Local dos testes

Não há local específico para realização dos testes. Com a máquina descrita no item 6 e com os fontes do FrameMK já é possível a execução dos testes.

7) Nível escolhido

As classes de persistência serão testadas a níveis de unidade.

8) Técnicas aplicadas

Será testada a funcionalidade do código, ou seja, se os módulos se comportam de acordo com que foram desenhados.

9) Critérios elencados

Verificar o retorno dos métodos repassando valores diferenciados de parâmetros.

10) Tipos de teste

Serão utilizados testes de funcionalidade nos módulos de persistência.

11) Tabela de relacionamento método x caso de teste

Classe	Método	Número do caso de teste
AbcActivityDAOFirebird	addAssociation(int activityCode, int productionLineCode)	0001
	deleteAssociation(int codigo)	0002
	saveValues(AbcProduct abcProduct)	0003
	consult(String sql)	0004
	add(ActivityAssociationAttibute attribute)	0005
	addOrEditActivity(Integer codigo, String descricao, float custo)	0006
	getProductionLineCode(String description)	0007
	getActivity(String descricao, float custo)	0008
	edit(ActivityAssociationAttibute attribute)	0009
	delete(ActivityAssociationAttibute attribute)	0010
	getAttributes()	0011
	getAttributesByFilter(String filter, boolean filterIsTheCode)	0012
AbcLogValueDAOFirebird	setSalePrice(AbcLogValue abcLogValue, double value)	0013
AbcProductDAOFirebird	listartable(int code)	0014
	saveValues(int methodCode, int iteration, int productCode, int attributeCode, double value, String date)	0015

12) Caso de teste 0001 – Testar associação entre a atividade e a linha de produto

Número	0001
Nome	Testar associação entre a atividade e a linha de produto.
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory Connection CallableStatement
Método	addAssociation(int activityCode, int productionLineCode)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	addAssociationTest(int activityCode, int productionLineCode) addAssociationTest2(int activityCode, int productionLineCode) addAssociationTest3(int activityCode, int productionLineCode)
Cenário	Cenário1: Número da atividade e da linha de produto com valores nulos. Cenário2: Número da atividade da linha do produto com valores que não estejam cadastrados. Cenário3: Número de atividade e linha de produto estão cadastrados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve executar corretamente a associação.
Resultado obtido	
Data do último teste	

13) Caso de teste 0002 –Deletar associação entre atividade e linha do produto

Número	0002
Nome	Deletar associação entre atividade e linha do produto
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory Connection StringBuilder
Método	deleteAssociation(int codigo)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	deleteAssociationTest(int codigo) deleteAssociationTest2(int codigo) deleteAssociationTest3(int codigo)
Cenário	Cenário1: Código da associação nulo. Cenário2: Código da associação com valores não cadastrados. Cenário3: Código da associação com valores cadastrados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve deletar a associação corretamente.
Resultado obtido	
Data do último teste	

14) Caso de teste 0003 – Atualizar valores do produto

Número	0003
Nome	Atualizar valores do produto
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcProductDAOFirebird
Pré-condição	FBDAOFactory Connection PreparedStatement
Método	saveValues(AbcProduct abcProduct)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcProductDAOFirebirdTest
Método(s) de teste	saveValuesTest(AbcProduct abcProduct) saveValuesTest2(AbcProduct abcProduct) saveValuesTest3(AbcProduct abcProduct))
Cenário	Cenário1: Objeto nulo. Cenário2: Objeto com valores inexistentes no banco de dados. Cenário3: Objeto com valores existentes no banco de dados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção a mensagem de erro “Problema em atualizar valor” Cenário3: Deve atualizar a associação corretamente.
Resultado obtido	
Data do último teste	

15) Caso de teste 0004 –Método de consulta

Número	0004
Nome	Método de consulta
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory Connection Statement ResultSet
Método	consult(String sql)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	consultTest(String sql) consultTest2(String sql)
Cenário	Cenário1: Consulta com sintaxe SQL mal formada. Cenário2: Consulta com sintaxe SQL correta.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar um objeto do tipo ResultSet com os resultados da SQL solicitada.
Resultado obtido	
Data do último teste	

16) Caso de teste 0005 – Adicionar atividade e associação no atributo

Número	0005
Nome	Adicionar atividade e associação no atributo
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	ActivityAssociationAttibute
Método	add(ActivityAssociationAttibute attribute)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	addTest1(ActivityAssociationAttibute attribute) addTest2(ActivityAssociationAttibute attribute) addTest3(ActivityAssociationAttibute attribute)
Cenário	Cenário1: Adição de um novo atributo nulo. Cenário2: Adição de um novo atributo com valores inválidos. Cenário3: Adição de um novo atributo com valores válidos.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve adicionar a atividade e a associação com sucesso.
Resultado obtido	
Data do último teste	

17) Caso de teste 0006 – Adicionar ou editar uma atividade

Número	0006
Nome	Adicionar ou editar uma atividade
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory Connection CallableStatement
Método	addOrEditActivity(Integer codigo, String descricao, float custo)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	addOrEditActivityTest(Integer codigo, String descricao, float custo) addOrEditActivityTest2(Integer codigo, String descricao, float custo) addOrEditActivityTest3(Integer codigo, String descricao, float custo)
Cenário	Cenário1: Adição de uma atividade com valores inválidos. Cenário2: Adição de uma atividade com valores válidos. Cenário3: Adição de uma atividade com valores válidos já existentes no sistema.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma nova atividade com os valores cadastrados. Cenário3: Deve alterar os valores já cadastrados no sistema.
Resultado obtido	
Data do último teste	

18) Caso de teste 0007 – Recuperar código de linha do produto

Número	0007
Nome	Adicionar ou editar uma atividade
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory
Método	getProductionLineCode(String description)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	getProductionLineCodeTest(String description) getProductionLineCodeTest2(String description)
Cenário	Cenário1: Recuperar o código com uma descrição inválida. Cenário2: Recuperar o código com uma descrição válida.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve recuperar um inteiro com o código da linha do produto.
Resultado obtido	
Data do último teste	

19) Caso de teste 0008 – Recuperar uma atividade

Número	0008
Nome	Recuperar uma atividade
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	FBDAOFactory StringBuilder ActivityAttribute
Método	getActivity(String descricao, float custo)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	getActivityTest(String descricao, float custo) getActivityTest2(String descricao, float custo) getActivityTest3(String descricao, float custo)
Cenário	Cenário1: Descrição e valor de custo com valores nulos. Cenário2: Descrição e valor de custo com valores que não estejam cadastrados. Cenário3: Descrição e valor de custo com valores que estão cadastrados.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção avisando que não há registro com os valores solicitados. Cenário3: Deve retornar um objeto ActivityAttribute com os valores solicitados.
Resultado obtido	
Data do último teste	

20) Caso de teste 0009 – Editar uma associação

Número	0009
Nome	Editar uma associação
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	ActivityAssociationAttribute
Método	edit(ActivityAssociationAttibute attribute)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	editTest(ActivityAssociationAttibute attribute) editTest2(ActivityAssociationAttibute attribute) editTest3(ActivityAssociationAttibute attribute)
Cenário	Cenário1: Editar com um objeto nulo. Cenário2: Editar com um objeto de valores inválidos. Cenário3: Editar com um objeto de valores válidos.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção avisando que não há registro com os valores solicitados. Cenário3: Deve retornar um objeto ActivityAttribute com os valores solicitados.
Resultado obtido	
Data do último teste	

21) Caso de teste 0010 – Deletar uma atividade

Número	0010
Nome	Deletar uma atividade
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	StringBuilder Connection FBDAOFactory
Método	delete(ActivityAssociationAttibute attribute)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	deleteTest(ActivityAssociationAttibute attribute) deleteTest2(ActivityAssociationAttibute attribute) deleteTest3(ActivityAssociationAttibute attribute)
Cenário	Cenário1: Deletar uma atividade com um objeto nulo. Cenário2: Deletar uma atividade com um objeto válido não cadastrado. Cenário3: Deletar uma atividade com um objeto válido cadastrado no sistema.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção avisando que não há registro com o objeto solicitado. Cenário3: Deve deletar a atividade solicitada no atributo.
Resultado obtido	
Data do último teste	

22) Caso de teste 0011 – Listar Atributos

Número	0012
Nome	Listar Atributos
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	
Método	getAttributes()
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	getAttributesTest()
Cenário	Cenário1: Retornar um vetor de atributos de atividades e linhas de atividade.
Resultado esperado	Cenário1: Deve retornar um vetor de atributos de atividades e linhas de atividade.
Resultado obtido	
Data do último teste	

23) Caso de teste 0012 – Listar atributos por filtros

Número	0013
Nome	Listar atributos por filtros
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcActivityDAOFirebird
Pré-condição	StringBuilder ActivityAssociationAttibute
Método	getAttributesByFilter(String filter, boolean filterIsTheCode)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcActivityDAOFirebirdTest
Método(s) de teste	getAttributesByFilterTest(String filter, boolean filterIsTheCode) getAttributesByFilterTest1(String filter, boolean filterIsTheCode) getAttributesByFilterTest3(String filter, boolean filterIsTheCode)
Cenário	Cenário1: Filtro e booleano inválidos. Cenário2: Filtro com valores inválidos e booleano como true. Cenário3: Filtro e booleano com valores válidos.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve retornar uma lista de atributos válidos
Resultado obtido	
Data do último teste	

24) Caso de teste 0013 – Setar preço de venda

Número	0014
Nome	Setar preço de venda
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcLogValueDAOFirebird
Pré-condição	Connection CallableStatement FBDAOFactory
Método	setSalePrice(AbcLogValue abcLogValue, double value)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcLogValueDAOFirebirdTest
Método(s) de teste	setSalePriceTest(AbcLogValue abcLogValue, double value) setSalePriceTest2(AbcLogValue abcLogValue, double value) setSalePriceTest3(AbcLogValue abcLogValue, double value)
Cenário	Cenário1: Setar objeto e valor de preço de venda nulo. Cenário2: Setar objeto e valor de preço de venda inválido. Cenário3: Setar objeto e valor de preço de venda válido.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve adicionar ou atualizar o valor de preço de venda corretamente.
Resultado obtido	
Data do último teste	

25) Caso de teste 0014 – Listar tabela de produtos

Número	0015
Nome	Listar tabela de produtos
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcProductDAOFirebird
Pré-condição	AbcProduct Connection Statement ResultSet FBDAOFactory
Método	listartable(int code)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcProductDAOFirebirdTest
Método(s) de teste	listartableTest(int code) listartableTest2(int code) listartableTest3(int code)
Cenário	Cenário1: Recuperar produtos com um código nulo. Cenário2: Recuperar produtos com um código inválido. Cenário3: Recuperar produtos com um código válido.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve retornar um vetor de produtos válidos.
Resultado obtido	
Data do último teste	

26) Caso de teste 0015 – Salvar produto ABC2

Número	0017
Nome	Salvar produto ABC2
Status	Pendente
Pacote	Persistence.DAO.Firebird
Classe	AbcProductDAOFirebird
Pré-condição	AbcProduct Connection CallableStatement FBDAOFactory
Método	saveValues(int methodCode, int iteration, int productCode, int attributeCode, double value, String date)
Pacote de teste	test.java.Persistence.DAO.Firebird
Classe de teste	AbcProductDAOFirebirdTest
Método(s) de teste	saveValuesTest15(int methodCode, int iteration, int productCode, int attributeCode, double value, String date) saveValuesTest152(int methodCode, int iteration, int productCode, int attributeCode, double value, String date) saveValuesTest153(int methodCode, int iteration, int productCode, int attributeCode, double value, String date)
Cenário	Cenário1: Salvar valores com valores nulos. Cenário2: Salvar valores com valores inválidos. Cenário3: Salvar valores com valores válidos.
Resultado esperado	Cenário1: Deve retornar uma exceção com uma mensagem de erro. Cenário2: Deve retornar uma exceção com uma mensagem de erro. Cenário3: Deve alterar os valores do objeto solicitado. Também deve registrar os valores no log de produtos ABC.
Resultado obtido	
Data do último teste	

APÊNDICE C – CLASSE DE TESTE *AbcActivityDaoFirebirdTest*

```
package test.java.Persistence.DAO.Firebird;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import sun.misc.JavaLangAccess;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Vector;

import Persistence.DAO.AbcActivityDAO;
import VO.AbcProduct;
import VO.ActivityAssociationAttibute;
import VO.ActivityAttribute;
import VO.ProductionLineAttribute;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

import Persistence.DAO.Firebird.AbcActivityDAOFirebird;
import Persistence.DAO.Firebird.FBDAOFactory;

public class AbcActivityDAOFirebirdTest {
    private AbcActivityDAOFirebird abcActivityDAOFirebird;
    private Method method;
    private Class[] parameterTypes;
    private Object[] parameters;

    @Before
    public void setUp() throws Exception {
        abcActivityDAOFirebird = new AbcActivityDAOFirebird();

        Connection con;

        con = FBDAOFactory.connect();
        con.prepareStatement("delete from
atividade_linha").executeUpdate();
        con.prepareStatement("delete from atividade where codigo > 3")
                .executeUpdate();
        FBDAOFactory.disconnect();
    }

    // Caso de teste 0001
    public boolean addAssociationTest(int activityCode, int
productionLineCode) {
        parameterTypes = new Class[2];
        parameterTypes[0] = int.class;
        parameterTypes[1] = int.class;
        try {
            method = abcActivityDAOFirebird.getClass().getDeclaredMethod(
                    "addAssociation", parameterTypes);
```

```
        } catch (Exception e) {
            fail("Falha na localização do método" + e.toString());
        }
        method.setAccessible(true);
        parameters = new Object[2];
        parameters[0] = activityCode;
        parameters[1] = productionLineCode;
        try {
            method.invoke(abcActivityDAOFirebird, parameters);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    // Cenário 1: Número da atividade e da linha de produto com valores
nulos.
    @Test
    public void addAssociationTest() {
        assertFalse(addAssociationTest(-1, -1));
    }

    // Cenário 2: Número da atividade da linha do produto com valores que
não
    // estejam cadastrados.
    @Test
    public void addAssociationTest2() {
        assertFalse(addAssociationTest(9999, 9999));
    }

    // Cenário 3: Número de atividade e linha de produto estão cadastrados.
    @Test
    public void addAssociationTest3() {
        if (!addAssociationTest(3, 1)) {
            fail("Falha na execução do método");
        } else {
            ResultSet rs = null;
            try {
                Connection con = FBDAOFactory.connect();
                Statement stm = con.createStatement();
                rs = stm.executeQuery("select 1 from atividade_linha "
                        + " where codigo_atividade = 3 "
                        + " and codigo_linha = 1 ");
                FBDAOFactory.disconnect();
                if (rs == null) {
                    fail("Método não inseriu a associação");
                }
            } catch (Exception e) {
                fail("Falha na execução do método sql");
            }
        }
    }

    // Caso de teste 0002
    // Cenário 1: Número da associação nulo.
    @Test(expected = Exception.class)
    public void deleteAssociationTest() throws Exception {
        abcActivityDAOFirebird.deleteAssociation(-1);
    }

    // Cenário 2: Número da associação com valores não cadastrados.
```

```
    @Test(expected = Exception.class)
    public void deleteAssociationTest2() throws Exception {
        abcActivityDAOFirebird.deleteAssociation(9999);
    }

    // Cenário 3: Número da associação com valores cadastrados.
    @Test
    public void deleteAssociationTest3() {
        int codigo;
        if (!addAssociationTest(3, 1)) {
            fail("Falha na adição de uma nova associação");
        } else {
            ResultSet rs = null;
            try {
                Connection con = FBDAOFactory.connect();
                Statement stm = con.createStatement();
                rs = stm.executeQuery("select codigo from atividade_linha
order by codigo desc");
                FBDAOFactory.disconnect();
                if (rs == null) {
                    fail("Método não inseriu a associação");
                } else {
                    rs.next();
                    codigo = rs.getInt("codigo");
                    abcActivityDAOFirebird.deleteAssociation(codigo);
                    con = FBDAOFactory.connect();
                    stm = con.createStatement();
                    rs = stm.executeQuery("select 1 from atividade_linha
where codigo ="
                            + " " + Integer.toString(codigo));
                    FBDAOFactory.disconnect();

                    if (rs.next()) {
                        fail("Método não deletou a associação");
                    }
                }
            } catch (Exception e) {
                fail("Falha na execução do método sql");
            }
        }
    }

    // Caso de teste 0004
    // Cenário 1: Consulta com sintaxe SQL mal formada.
    @Test(expected = Exception.class)
    public void consultTest() throws Exception {
        abcActivityDAOFirebird.consult("select x from atividade_linha");
    }

    // Cenário 2: Consulta com sintaxe SQL correta.
    @Test
    public void consultTest2() throws Exception {
        if (!(abcActivityDAOFirebird.consult("select 1 from
atividade_linha") instanceof ResultSet)) {
            fail("O resutado deve ser do tipo ResultSet");
        }

    }

    // Caso de teste 0005
    // Cenário 1: Adição de um novo atributo nulo.
```

```
    @Test(expected = Exception.class)
    public void addTest1() throws Exception {
        ActivityAssociationAttibute activityAssociationAttibute;
        activityAssociationAttibute = null;
        abcActivityDAOFirebird.add(activityAssociationAttibute);
    }

    // Cenário 2: Adição de um novo atributo com valores inválidos.
    @Test(expected = Exception.class)
    public void addTest2() throws Exception {
        ActivityAssociationAttibute activityAssociationAttibute;
        ActivityAttribute activityAttribute;
        ProductionLineAttribute productionLineAttribute;

        activityAssociationAttibute = new ActivityAssociationAttibute();
        activityAttribute = new ActivityAttribute();
        productionLineAttribute = new ProductionLineAttribute();

        activityAttribute.setCode(-1);
        activityAttribute.setCost(-1);
        activityAttribute.setName("");

        productionLineAttribute.setCode(-1);
        productionLineAttribute.setName("");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);
        activityAssociationAttibute.setCode(-1);
        activityAssociationAttibute.setName("");

        abcActivityDAOFirebird.add(activityAssociationAttibute);
    }

    // Cenário 3: Adição de um novo atributo com valores válidos.
    @Test
    public void addTest3() throws Exception {
        ActivityAssociationAttibute activityAssociationAttibute;
        ActivityAttribute activityAttribute;
        ProductionLineAttribute productionLineAttribute;

        activityAssociationAttibute = new ActivityAssociationAttibute();
        activityAttribute = new ActivityAttribute();
        productionLineAttribute = new ProductionLineAttribute();

        activityAttribute.setCost(0);
        activityAttribute.setName("TesteActivity");

        productionLineAttribute.setName("PIZZA");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);
        activityAssociationAttibute.setCode(1);
        activityAssociationAttibute.setName("teste");

        abcActivityDAOFirebird.add(activityAssociationAttibute);

        ResultSet rs = null;

        Connection con = FBDAOFactory.connect();
```

```
        Statement stm = con.createStatement();
        rs = stm.executeQuery("select codigo_linha from atividade_linha
order by codigo desc");
        FBDAOFactory.disconnect();

        rs.next();
        if (rs.getInt(1) != 1) {
            fail("Não gravou a atividade corretaente");
        }
    }

    // Caso de teste 0006
    public boolean addOrEditActivity(Integer codigo, String descricao,
            float custo) {
        parameterTypes = new Class[3];
        parameterTypes[0] = Integer.class;
        parameterTypes[1] = String.class;
        parameterTypes[2] = float.class;
        try {
            method = abcActivityDAOFirebird.getClass().getDeclaredMethod(
                    "addOrEditActivity", parameterTypes);
        } catch (Exception e) {
            fail("Falha na localização do método" + e.toString());
        }
        method.setAccessible(true);
        parameters = new Object[3];
        parameters[0] = codigo;
        parameters[1] = descricao;
        parameters[2] = custo;

        try {
            method.invoke(abcActivityDAOFirebird, parameters);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    // Cenário 1: Adição de uma atividade com valores inválidos.
    @Test
    public void addOrEditActivityTest() {
        assertFalse(addOrEditActivity(-1, "", -1));
    }

    // Cenário 2: Adição de uma atividade com valores válidos.
    @Test
    public void addOrEditActivityTest2() {
        assertTrue(addOrEditActivity(null, "Testando", 10));
    }

    // Cenário 3: Adição de uma atividade com valores válidos já existentes
no
    // sistema.
    @Test
    public void addOrEditActivityTest3() throws Exception {
        assertTrue(addOrEditActivity(null, "Testando", 10));
        int codigo;
        ResultSet rs = null;

        Connection con = FBDAOFactory.connect();
        Statement stm = con.createStatement();
```

```
        rs = stm.executeQuery("select codigo from atividade order by codigo
desc");
        FBDAOFactory.disconnect();
        rs.next();
        codigo = rs.getInt(1);

        assertTrue(addOrEditActivity(codigo, "addOrEditActivityTest3",
8002));
    }

    // Caso de teste 0007
    public int getProductionLineCode(String description)
            throws IllegalAccessException, IllegalArgumentException,
            InvocationTargetException {
        parameterTypes = new Class[1];
        parameterTypes[0] = String.class;

        try {
            method = abcActivityDAOFirebird.getClass().getDeclaredMethod(
                    "getProductionLineCode", parameterTypes);
        } catch (Exception e) {
            fail("Falha na localização do método" + e.toString());
        }
        method.setAccessible(true);
        parameters = new Object[1];
        parameters[0] = description;

        return (Integer) method.invoke(abcActivityDAOFirebird, parameters);

    }

    // Cenário 1: Recuperar o código com uma descrição inválida.
    @Test
    public void getProductionLineCodeTest() throws IllegalAccessException,
            IllegalArgumentException, InvocationTargetException {
        assertTrue(getProductionLineCode("XTESTEX") == 0);
    }

    // Cenário 2: Recuperar o código com uma descrição válida.
    @Test
    public void getProductionLineCodeTest2() throws IllegalAccessException,
            IllegalArgumentException, InvocationTargetException {
        assertTrue(getProductionLineCode("PIZZA") == 1);
    }

    // Caso de teste 0008
    public ActivityAttribute getActivity(String descricao, float custo)
            throws Exception {
        parameterTypes = new Class[2];
        parameterTypes[0] = String.class;
        parameterTypes[1] = float.class;

        try {
            method = abcActivityDAOFirebird.getClass().getDeclaredMethod(
                    "getActivity", parameterTypes);
        } catch (Exception e) {
            fail("Falha na localização do método" + e.toString());
        }
        method.setAccessible(true);
        parameters = new Object[2];
        parameters[0] = descricao;
```

```
            parameters[1] = custo;

            return (ActivityAttribute) method.invoke(abcActivityDAOFirebird,
                    parameters);
        }

        // Caso de teste 0009
        // Cenário1: Descrição e valor de custo com valores nulos.
        @Test(expected = Exception.class)
        public void editTest() throws Exception {
            // Cria uma linha de atividade
            ActivityAssociationAttibute activityAssociationAttibute;
            ActivityAttribute activityAttribute;
            ProductionLineAttribute productionLineAttribute;

            activityAssociationAttibute = null;
            activityAttribute = null;
            productionLineAttribute = null;

            activityAssociationAttibute.setActivity(activityAttribute);

    activityAssociationAttibute.setProductionLine(productionLineAttribute);

            activityAssociationAttibute.setName(null);

            abcActivityDAOFirebird.edit(activityAssociationAttibute);
        }

        // Cenário2: Descrição e valor de custo com valores que não estejam
        // cadastrados.
        @Test(expected = Exception.class)
        public void editTest1() throws Exception {
            // Cria uma linha de atividade
            ActivityAssociationAttibute activityAssociationAttibute;
            ActivityAttribute activityAttribute;
            ProductionLineAttribute productionLineAttribute;

            activityAssociationAttibute = new ActivityAssociationAttibute();
            activityAttribute = new ActivityAttribute();
            productionLineAttribute = new ProductionLineAttribute();

            activityAttribute.setCost(2200);
            activityAttribute.setName("editTest2NÃO EXISTE");

            productionLineAttribute.setName("PIZZA NÃO EXISTE");

            activityAssociationAttibute.setActivity(activityAttribute);

    activityAssociationAttibute.setProductionLine(productionLineAttribute);

            activityAssociationAttibute.setName("Edit teste NÃO EXISTE");

            abcActivityDAOFirebird.edit(activityAssociationAttibute);
        }

        // Cenário3: Descrição e valor de custo com valores que estão
    cadastrados.
        @Test
        public void editTest2() throws Exception {
            // Cria uma linha de atividade
            ActivityAssociationAttibute activityAssociationAttibute;
```

```
        ActivityAttribute activityAttribute;
        ProductionLineAttribute productionLineAttribute;

        activityAssociationAttibute = new ActivityAssociationAttibute();
        activityAttribute = new ActivityAttribute();
        productionLineAttribute = new ProductionLineAttribute();

        addOrEditActivity(null, "editTest2", 2200);
        activityAttribute.setCost(2200);
        activityAttribute.setName("editTest2");
        activityAttribute = getActivity(activityAttribute.getName(),
                activityAttribute.getCost());

        productionLineAttribute.setName("PIZZA");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);

        activityAssociationAttibute.setName("Edit teste");

        abcActivityDAOFirebird.add(activityAssociationAttibute);

        // Verifica se a linha foi criada - Linha 1 - PIZZA
        StringBuilder sql = new StringBuilder();
        ResultSet rs = null;
        Connection con = FBDAOFactory.connect();
        Statement stm = con.createStatement();

        sql.append("select codigo from atividade_linha ");
        sql.append(" where (codigo_atividade = "
                + String.valueOf(activityAttribute.getCode()) + ") ");
        sql.append(" and (codigo_linha = 1) ");

        rs = stm.executeQuery(sql.toString());
        FBDAOFactory.disconnect();

        // Se não há a linha criada então o teste não criou a atividade
        assertFalse(!rs.next());

        // Verifica se a linha foi editada - Linha 3 - Ar Condicionado
        activityAssociationAttibute.setCode(rs.getInt(1));
        productionLineAttribute.setName("Ar Condicionado");

        abcActivityDAOFirebird.edit(activityAssociationAttibute);

        con = FBDAOFactory.connect();
        stm = con.createStatement();

        sql = new StringBuilder();
        sql.append("select codigo from atividade_linha ");
        sql.append(" where (codigo_atividade = "
                + String.valueOf(activityAttribute.getCode()) + ") ");
        sql.append(" and (codigo_linha = 3) ");

        rs = stm.executeQuery(sql.toString());
        FBDAOFactory.disconnect();

        assertTrue(rs.next());
    }
```

```
    // Caso de teste 0010
    // Cenário1: Deletar uma atividade com um objeto nulo.
    @Test(expected = Exception.class)
    public void deleteTest() throws Exception {
        // Cria uma linha de atividade
        ActivityAssociationAttibute activityAssociationAttibute;
        ActivityAttribute activityAttribute;
        ProductionLineAttribute productionLineAttribute;

        activityAssociationAttibute = null;
        activityAttribute = null;
        productionLineAttribute = null;

        productionLineAttribute.setName("PIZZA");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);

        activityAssociationAttibute.setName("Delete teste");

        abcActivityDAOFirebird.delete(activityAssociationAttibute);
    }

    // Cenário2: Deletar uma atividade com um objeto válido não cadastrado.
    @Test(expected = Exception.class)
    public void deleteTest2() throws Exception {
        // Cria uma linha de atividade
        ActivityAssociationAttibute activityAssociationAttibute;
        ActivityAttribute activityAttribute;
        ProductionLineAttribute productionLineAttribute;

        activityAssociationAttibute = new ActivityAssociationAttibute();
        activityAttribute = new ActivityAttribute();
        productionLineAttribute = new ProductionLineAttribute();

        activityAttribute.setCost(2200);
        activityAttribute.setName("deleteTest2");
        activityAttribute = getActivity(activityAttribute.getName(),
                activityAttribute.getCost());

        productionLineAttribute.setName("PIZZA");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);

        activityAssociationAttibute.setName("Delete teste2");

        abcActivityDAOFirebird.delete(activityAssociationAttibute);
    }

    // Cenário3: Deletar uma atividade com um objeto válido cadastrado no
    // sistema.
    @Test
    public void deleteTest3() throws Exception {
        // Cria uma linha de atividade
        ActivityAssociationAttibute activityAssociationAttibute;
        ActivityAttribute activityAttribute;
        ProductionLineAttribute productionLineAttribute;
```

```
        activityAssociationAttibute = new ActivityAssociationAttibute();
        activityAttribute = new ActivityAttribute();
        productionLineAttribute = new ProductionLineAttribute();

        addOrEditActivity(null, "deleteTest3", 2200);
        activityAttribute.setCost(2200);
        activityAttribute.setName("deleteTest3");
        activityAttribute = getActivity(activityAttribute.getName(),
                activityAttribute.getCost());

        productionLineAttribute.setName("PIZZA");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);

        activityAssociationAttibute.setName("Delete teste 3");

        abcActivityDAOFirebird.add(activityAssociationAttibute);

        // Verifica se a linha foi criada - Linha 1 - PIZZA
        StringBuilder sql = new StringBuilder();
        ResultSet rs = null;
        Connection con = FBDAOFactory.connect();
        Statement stm = con.createStatement();

        sql.append("select codigo from atividade_linha ");
        sql.append(" where (codigo_atividade = "
                + String.valueOf(activityAttribute.getCode()) + ") ");
        sql.append(" and (codigo_linha = 1) ");

        rs = stm.executeQuery(sql.toString());
        FBDAOFactory.disconnect();

        // Se não há a linha criada então o teste não criou a atividade
        assertFalse(!rs.next());
        activityAssociationAttibute.setCode(rs.getInt(1));

        abcActivityDAOFirebird.delete(activityAssociationAttibute);

        // Verifica se a linha foi deletada
        con = FBDAOFactory.connect();
        stm = con.createStatement();

        sql = new StringBuilder();
        sql.append("select codigo from atividade_linha ");
        sql.append(" where (codigo_atividade = "
                + String.valueOf(activityAttribute.getCode()) + ") ");
        sql.append(" and (codigo_linha = 3) ");

        rs = stm.executeQuery(sql.toString());
        FBDAOFactory.disconnect();

        assertTrue(!rs.next());
    }

    // Caso de teste 0011
    // Cria uma linha de atividade
    public void createActivityAssociationAttibute() throws Exception {
        ActivityAssociationAttibute activityAssociationAttibute;
        ActivityAttribute activityAttribute;
```

```
        ProductionLineAttribute productionLineAttribute;

        activityAssociationAttibute = new ActivityAssociationAttibute();
        activityAttribute = new ActivityAttribute();
        productionLineAttribute = new ProductionLineAttribute();

        activityAttribute.setCost(7000);
        activityAttribute.setName("Preparacao de Maquinas");
        activityAttribute = getActivity(activityAttribute.getName(),
                activityAttribute.getCost());

        productionLineAttribute.setName("PIZZA");

        activityAssociationAttibute.setActivity(activityAttribute);

activityAssociationAttibute.setProductionLine(productionLineAttribute);

        activityAssociationAttibute.setName("teste");

        abcActivityDAOFirebird.add(activityAssociationAttibute);
    }

    // Cenário1: Retornar um vetor de atributos de atividades e linhas de
    // atividade.
    @Test
    public void getAttributesTest() throws Exception {
        createActivityAssociationAttibute();

        // Verifica se a atividade é listada
        Vector<ActivityAssociationAttibute>
activityAssociationAttibuteVector;

        activityAssociationAttibuteVector = abcActivityDAOFirebird
                .getAttributes();


assertTrue(activityAssociationAttibuteVector.elementAt(0).getActivity()
                .getCode() == 1);
    }

    // Caso de teste 0012
    // Cenário1: Filtro e booleano inválidos.
    @Test(expected = Exception.class)
    public void getAttributesByFilterTest() throws Exception {
        createActivityAssociationAttibute();

        // Verifica se a atividade é listada
        Vector<ActivityAssociationAttibute>
activityAssociationAttibuteVector;

        activityAssociationAttibuteVector = abcActivityDAOFirebird
                .getAttributesByFilter(null, false);

        assertFalse(activityAssociationAttibuteVector.elementAt(0)
                .getActivity().getCode() == 1);

    }

    // Cenário2: Filtro com valores inválidos e booleano como true.
    @Test(expected = Exception.class)
    public void getAttributesByFilterTest2() throws Exception {
```

```
        createActivityAssociationAttibute();

        // Verifica se a atividade é listada
        Vector<ActivityAssociationAttibute>
activityAssociationAttibuteVector;

        activityAssociationAttibuteVector = abcActivityDAOFirebird
                .getAttributesByFilter(null, true);

        assertFalse(activityAssociationAttibuteVector.elementAt(0)
                .getActivity().getCode() == 1);
    }

    // Cenário3: Filtro e booleano com valores válidos.
    @Test
    public void getAttributesByFilterTest3() throws Exception {
        createActivityAssociationAttibute();

        // Pega o código da linha atividade
        StringBuilder sql = new StringBuilder();
        ResultSet rs = null;
        Connection con = FBDAOFactory.connect();
        Statement stm = con.createStatement();

        sql.append("select codigo from atividade_linha ");
        sql.append(" where (codigo_atividade = 1)");
        sql.append(" and (codigo_linha = 1) ");

        rs = stm.executeQuery(sql.toString());
        FBDAOFactory.disconnect();

        // Se não há a linha criada então o teste não criou a atividade
        assertFalse(!rs.next());

        // Verifica se a atividade é listada
        Vector<ActivityAssociationAttibute>
activityAssociationAttibuteVector;

        activityAssociationAttibuteVector = abcActivityDAOFirebird
                .getAttributesByFilter(String.valueOf(rs.getInt(1)), true);


assertTrue(activityAssociationAttibuteVector.elementAt(0).getActivity()
                .getCode() == 1);
        ;
    }
}
```

APÊNDICE D – CLASSE DE TESTE *AbcProductDAOFirebirdTest*

```
package test.java.Persistence.DAO.Firebird;


import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import Persistence.DAO.AbcProductDAO;
import Persistence.DAO.Firebird.AbcProductDAOFirebird;
import VO.AbcProduct;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Vector;

public class AbcProductDAOFirebirdTest {
    private AbcProduct abcProduct;
    private AbcProductDAOFirebird abcProductDAOFirebird;

    @Before
    public void setUp() throws Exception {
        abcProductDAOFirebird = new AbcProductDAOFirebird();
        abcProduct = new AbcProduct();
    }

    // Caso de teste 0003
    // Cenário1: Objeto nulo.
    @Test(expected = Exception.class)
    public void saveValuesTest() throws Exception {
        abcProduct = null;
        abcProductDAOFirebird.saveValues(abcProduct);
    }

    // Cenário2: Objeto com valores inexistentes no banco de dados.
    @Test(expected = Exception.class)
    public void saveValuesTest2() throws Exception {
        abcProduct = new AbcProduct();
        abcProduct.setCode(999999999);
        abcProduct.setValue(999999999);
        abcProductDAOFirebird.saveValues(abcProduct);
    }

    // Cenário3: Objeto com valores existentes no banco de dados.
    @Test
    public void saveValuesTest3() throws Exception {
        abcProduct = new AbcProduct();
        abcProduct.setCode(1);
        abcProduct.setValue(11);
        abcProductDAOFirebird.saveValues(abcProduct);
    }

    // Caso de teste 0014
    // Cenário1: Recuperar produtos com um código nulo.
    @Test(expected = Exception.class)
    public void listartableTest() throws Exception {
        abcProductDAOFirebird.listartable(-1);
    }

    // Cenário2: Recuperar produtos com um código inválido.
    @Test(expected = Exception.class)
```

```
    public void listartableTest2() throws Exception {
        abcProductDAOFirebird.listartable(9999);
    }

    // Cenário3: Recuperar produtos com um código válido.
    @Test
    public void listartableTest3() throws Exception {
        Vector<AbcProduct> abcProductVector;
        abcProductVector = abcProductDAOFirebird.listartable(1);
        // Produto 1 - Calabresa
        assertTrue(abcProductVector.get(1).getCode() == 2);
    }

    // Caso de teste 0015
    // Cenário1: Salvar valores com valores nulos.
    @Test(expected = Exception.class)
    public void saveValuesTest15() throws Exception {
        abcProductDAOFirebird.saveValues(-1, -1, -1, -1, -1, "");
    }

    // Cenário2: Salvar valores com valores inválidos.
    @Test(expected = Exception.class)
    public void saveValuesTest152() throws Exception {
        abcProductDAOFirebird.saveValues(9999, 9999, 99999, 99999, 99999,
"");
    }

    // Cenário3: Salvar valores com valores válidos.
    @Test
    public void saveValuesTest153() throws Exception {
        abcProductDAOFirebird.saveValues(1, 1, 1, 1, 3000, "09/12/2014");
    }

}
```

APÊNDICE E – CLASSE DE TESTE *AbcLogValueDAOFirebirdTest*

```
package test.java.Persistence.DAO.Firebird;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import sun.misc.JavaLangAccess;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Vector;

import Persistence.DAO.AbcActivityDAO;
import VO.AbcLogValue;
import VO.AbcProduct;
import VO.ActivityAssociationAttibute;
import VO.ActivityAttribute;
import VO.ProductionLineAttribute;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

import Persistence.DAO.Firebird.AbcActivityDAOFirebird;
import Persistence.DAO.Firebird.AbcLogValueDAOFirebird;
import Persistence.DAO.Firebird.FBDAOFactory;

public class AbcLogValueDAOFirebirdTest {
    private AbcLogValueDAOFirebird abcLogValueDAOFirebird;
    private AbcLogValue abcLogValue;

    @Before
    public void setUp() throws Exception {
        abcLogValueDAOFirebird = new AbcLogValueDAOFirebird();
        abcLogValue = new AbcLogValue();
    }

    // Caso de teste 0013
    // Cenário1: Setar objeto e valor de preço de venda nulo.
    @Test(expected = Exception.class)
    public void setSalePriceTest() throws Exception {
        abcLogValue = null;
        abcLogValueDAOFirebird.setSalePrice(abcLogValue, 0);
    }

    // Cenário2: Setar objeto e valor de preço de venda inválido.
    @Test(expected = Exception.class)
    public void setSalePriceTest2() throws Exception {
        abcLogValue = new AbcLogValue();

        abcLogValue.setCodeAttribute(99999);
        abcLogValue.setCodeProduct(99999);
        abcLogValue.setDate("");
```

```
        abcLogValue.setIterationCode(99999);
        abcLogValue.setLogCod(99999);
        abcLogValue.setValue(99999);

        abcLogValueDAOFirebird.setSalePrice(abcLogValue, 0);
    }

    // Cenário3: Setar objeto e valor de preço de venda válido.
    @Test
    public void setSalePriceTest3() throws Exception {

        abcLogValue = new AbcLogValue();

        abcLogValue.setCodeAttribute(1);
        abcLogValue.setCodeProduct(1);
        abcLogValue.setDate("2014-01-01");
        abcLogValue.setIterationCode(1);
        abcLogValue.setLogCod(1);
        abcLogValue.setValue(1);

        abcLogValueDAOFirebird.setSalePrice(abcLogValue, 0);
    }

}
```