UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

CAMPUS LUIZ MENEGHEL

MARCOS DI NALLO

TESTE DE UNIDADE UTILIZANDO JUNIT E OBJETOS MOCK – UM ESTUDO DE CASO

Bandeirantes

2010

MARCOS DI NALLO

TESTE DE UNIDADE UTILIZANDO JUNIT E OBJETOS MOCK – UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Norte do Paraná - campus Luiz Meneghel, como requisito para obtenção do grau de Bacharel em Sistemas de Informação, orientado pelo Prof. Carlos Eduardo Ribeiro.

Bandeirantes

2010

MARCOS DI NALLO

TESTE DE UNIDADE UTILIZANDO JUNIT E OBJETOS MOCK – UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Norte do Paraná - campus Luiz Meneghel, como requisito para obtenção do grau de Bacharel em Sistemas de Informação, orientado pelo Prof. Carlos Eduardo Ribeiro.

COMISSÃO EXAMINADORA

_____

Prof. Carlos Eduardo Ribeiro
Universidade Estadual do Norte do Paraná - Campus Luiz Meneghel

_____

Prof. Msc. André Luis Andrade Menolli
Universidade Estadual do Norte do Paraná - Campus Luiz Meneghel

_____

Prof. José Reinaldo Merlin
Universidade Estadual do Norte do Paraná - Campus Luiz Meneghel

Bandeirantes, 11 de Junho de 2010.

DEDICATÓRIA

Aos meus pais e irmão, principalmente pela paciência, apoio e motivação, não apenas nos momentos difíceis, mas também em meus melhores momentos, pois sem este apoio este trabalho como vários outros passos em minha vida seriam mais difíceis.

AGRADECIMENTOS

Em primeiro lugar a Deus, que atendeu as minhas orações nos momentos mais difíceis e que sempre me acompanha em todos os momentos.

A toda minha família, que sempre me ajudou.

A Glaucia, minha namorada que sempre me apoiou em minhas decisões e esteve sempre presente em minha caminhada.

Ao orientador Prof. Carlos Eduardo Ribeiro a quem sou muito grato, pela força, condução, apoio e incentivo nesse projeto.

Aos professores da comissão examinadora pelas dicas e sugestões, que auxiliaram na conclusão deste trabalho e a todos os professores e funcionários da UENP-Campus Luiz Meneghel.

Aos meus amigos com os quais passei momentos descontraídos e que sempre me incentivaram, principalmente ao Pietro e Du e aos meus primos Rômulo e Mario Henrique

Aos amigos que fiz em Bandeirantes, com os quais passei grandes momentos de diversão, principalmente aos meus amigos Zé e Emmanuel, que ainda são meus amigos, ao Jailton, Kenion e Rafael.

“Nunca, nunca desistas.” (Winston Churchill)

RESUMO

O uso de softwares tem aumentado tanto por parte de usuários domésticos como por empresas, fazendo também aumentar a procura por sistemas que ajam de maneira correta e sem apresentar falhas. Para tornar possível o desenvolvimento deste tipo de software é necessária a aplicação de testes, durante e depois do desenvolvimento de um aplicativo. Esse trabalho apresenta as principais técnicas de teste de software. Entretanto, a aplicação foi realizada com os testes de unidade, que visam testar o código do sistema, buscando falhas nos métodos e classes do mesmo. Para o desenvolvimento dos testes unitários usou-se o framework JUnit, cujo o objetivo é facilitar o desenvolvimento e automação destes testes. Foram usados também os Objetos Mock que têm a finalidade de simular objetos que foram implementados no sistema, a fim de diminuir a dependência entre objetos do sistema real. Quanto ao sistema no qual foram criados os objetos mock e aplicado os testes foi o Sysfarm que se trata de um sistema para o gerenciamento de propriedades rurais, testando assim o método salvar do aplicativo, simulando diversas entradas para o programa.

Palavras-chave: Teste de Software, Testes de Unidade, JUnit, Objetos Mock.

.

ABSTRACT

The use of software has increased both by home users as by companies, making also increase the demand for systems that have acted correctly and without failure. To make possible the development of this software is necessary to apply tests, during and after the development of an application. This paper presents the main techniques of Software Test. However, implementation was carried out with unit tests, which aim to test the system code, looking for flaws in the methods and classes of the same. For the development of unit tests used to the JUnit framework, whose goal is to facilitate the development and automation of these tests. Have also been used Mock Objects that are designed to simulate objects which were implemented in the system, in order to reduce the dependency between objects in the real system. The system in which the objects were created and implemented mock tests was Sysfarm that this is a system for managing farms, thereby testing the method to save the application, simulating different inputs to the program.

Key-Words: Software Testing, Unit Testing, JUnit, Mock Objects

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O uso de softwares nos dias de hoje tem aumentado significativamente, devido ao fato de as empresas e organizações buscarem um aumento de produtividade, segurança e exatidão em suas mais diversas áreas de atuação. Seja na área da medicina, engenharia, produção, educação, entre outras.

Devido a esta informatização global, os usuários de computadores domésticos e/ou de empresas tornam-se cada vez mais “reféns” da informática. Essa dependência cria a necessidade de se usar softwares cada vez mais confiáveis, que deem com exatidão a informação desejada, softwares que sejam estáveis e que apresentem pouca manutenção, sejam fáceis de operar e que atendam a todos os seus requisitos, e que, mesmo o software tendo todas estas qualidades, tenha preços acessíveis para o consumidor, seja ele uma empresa e/ou um usuário doméstico.

Com toda esta mudança de cenário (informatização), as empresas de desenvolvimento de sistemas também tendem a mudar para poder atender a todas essas exigências de mercado, pois elas terão que produzir softwares complexos, baratos e que satisfaçam seus clientes. Para tingir esse nível de desenvolvimento tornam-se fundamentais os testes de software.

Há muitas empresas de desenvolvimento e programadores que ainda não fazem a fase de testes de software por pensarem que tal procedimento encarece o produto final, porém esta é uma idéia errônea. Conforme será visto neste trabalho, os testes tornam os softwares mais baratos e confiáveis, pois as alterações que são feitas após o sistema ter sido implantado para o cliente, levam mais tempo para serem implementadas. Além disso, podem tornar o processo caro e prejudicial para o cliente, uma vez que o mesmo pode ficar com o sistema paralisado e erros ocorridos anteriormente podem contabilizar sérios prejuízos.

2 OBJETIVOS

2. 1 OBJETIVO GERAL

O objetivo geral deste projeto é trazer ao público o uso do “framework” JUnit e o uso dos objetos ‘mock” para o auxilio no desenvolvimento de testes de unidade, mostrando assim sua viabilidade e resultados obtidos nos testes.

2. 2 OBJETIVOS ESPECÍFICOS

- uma visão sobre testes de software,
- uma abordagem compreensiva sobre as fases e tipos de testes dando maior ênfase em Testes de Unidade,
- mostrar o uso do framework JUnit e o uso dos objetos Mock para a automação e auxílio em testes unitários.

3 JUSTICATIVA

A escolha do assunto deste projeto é fruto da observação sobre a importância de se fazer testes de software para se chegar a um software de alto nível. Hoje em dia é essencial a criação de software de alta qualidade, devido ao uso cada vez mais frequente de softwares e para as mais diversas áreas, como por exemplo, o uso de programas de computador na área da medicina.

Quanto à contribuição científica, este projeto terá como objetivo mostrar a aplicação das ferramentas JUnit e JMock para a realização de testes de unidade.

4 FUNDAMENTAÇÃO TEÓRICA

4. 1 Definição de Teste de Software

Teste de software é a atividade que tem como princípio encontrar erros em um sistema, sejam erros de requisitos, falhas de segurança ou exatidão, entre outros. Deve-se levar em consideração que um bom teste de software é aquele de detecta erros no sistema e principalmente erros que antes de se realizar o teste eram desconhecidos, pois não existem softwares livres de erros ou falhas, tendo em vista que quem desenvolve programas são humanos e todos estão sujeitos a falhas (Myers 1979 apud Franzem e Bellini 2005).

Delamaro, Maldonado e Jino (2007), reafirmam ainda que a construção de um software de qualidade é um processo muito complexo e que este processo é passível de falhas humanas e por isso deve-se fazer a atividade de teste antes da entrega do Sistema ao usuário final. Estes testes não se limitam a uma execução quando o sistema estiver finalizado, devem sim, ser efetuados antes, durante (Testes Unitários, serão o foco deste trabalho) e depois. A este conjunto de atividades denomina-se Validação, Verificação e Teste e, seguindo este Padrão de Desenvolvimento de Software evita-se a entrega de um produto com defeitos, sendo que estes podem ocasionar um erro durante a execução do programa levando assim a uma falha do sistema.

O teste de software pode ser comparado a um "*ckek-up*" feito em pessoas. O software pode estar cheio de erros, e estes podem não ser percebidos, porém com o tempo as falhas acontecerão e a reparação será muito mais complicada e custosa depois que o sistema estiver em pleno funcionamento. Consequências que podem ser evitadas se o problema for detectado através de testes e corrigido antes que acorra. Por isso que os testes de software são de grande importância para o desenvolvimento de um bom software, e sempre se deve realizar os testes durante o desenvolvimento e mesmo depois que o sistema estiver terminado e funcionando adequadamente, pois sempre poderá haver falhas (Molinari 2008). Segundo Delamaro, Maldonado e Jino (2007) os testes de Software

também devem seguir padrões, assim como o desenvolvimento do software, deve-se obedecer as fases de teste que são as fases de teste de unidade, integração, sistema e ainda o teste de regressão onde se aplicam as técnicas de teste estrutural ou funcional.

4. 2 Técnicas de Teste de Software

Atualmente existem muitas maneiras de se testar um software. Mesmo assim, existem as técnicas que sempre foram muito utilizadas em sistemas desenvolvidos sobre linguagens estruturadas que ainda hoje têm grande valia para os sistemas orientados a objeto. Apesar de os paradigmas de desenvolvimento serem completamente diferentes, o objetivo principal destas técnicas continua a ser o mesmo, encontrar falhas no software.

As duas principais técnicas existentes são o teste funcional e o teste estrutural, sendo que a diferença entre elas encontra-se na fonte utilizada para estabelecer os requisitos de teste (Molinari, 2008).

Como por exemplo, a utilização de um algoritmo incorreto utilizado para calcular mensalidades de um empréstimo, ou a não utilização de uma política de segurança em alguma funcionalidade do sistema, são dois tipos diferentes de erro e necessitam de técnicas de teste distintas. Para o caso do algoritmo aplica-se a técnica de teste estrutural, pois o problema encontra-se no código do programa, e para o problema de segurança aplica-se a técnica de teste funcional (Delamaro; Maldonado e Jino; 2007).

4. 3 Fases de Teste de Software

Dependendo da fase ou estágio em que se encontra um software, é possível saber qual tipo de teste deve ser utilizado em cada fase do seu desenvolvimento.

4.3.1 Teste de Integração

O teste de integração tem a finalidade de garantir que a combinação de duas ou mais partes ou módulos do sistema combinados funcionem corretamente (Molinari, 2008).

Esta fase de teste é executada após a conclusão dos testes de unidade, tem como foco a estrutura do sistema, por isso é preciso conhecer bem a estrutura do programa, tornando indispensável a equipe de desenvolvimento nesta fase para poder verificar se as diversas partes do software interagem de forma correta (Delamaro; Maldonado e Jino 2007).

Segundo Binder *apud* Lima e Travassos, [s.d], nesta fase de teste deve-se observar a dependência entre os componentes do sistema e fazer com que a partir desta dependência seja verificado se a interação entre eles está correta. Entretanto, é possível que um ou mais componentes desta interação não estejam prontos para esta fase de teste, devendo-se, então, criar os *stubs*, componentes criados para simular o componente real que não se encontra disponível ainda.

4.3.2 Teste de Sistema

O teste de sistema tem a finalidade de garantir que o aplicativo funcione corretamente como um todo, verifica-se o mesmo faz tudo aquilo que deveria fazer (Molinari, 2008).

Delamaro, Maldonado e Jino (2007), comentam ainda que o teste de sistema verifica se todas as funcionalidades descritas na documentação de requisitos estão corretamente implementadas e que aspectos de correção, completude, coerência devem ser verificados, assim como requisitos não funcionais como segurança, robustez e performance. Além disso, salientam que muitas organizações designam equipes independentes para realizar estes testes.

4.3.3 Teste de Regressão

É o teste que é feito em outra etapa do software. Este teste é realizado após o aplicativo ter sido entregue ao cliente, tem a finalidade de verificar se as novas implementações ou alterações que foram feitas no sistema estão corretas e se as funcionalidades antigas continuam válidas (Delamaro; Maldonado e Jino 2007).

Molinari (2008) destaca ainda que o teste de regressão é um dos mais importantes testes a ser realizado, pois sempre que se adiciona ou conserta algo novo pode-se estar danificando aquilo que estava correto anteriormente.

4.3.4 Teste de Unidade

Teste em nível de classe ou componente, em que o foco é dado no código da aplicação (Molinari, 2008).

Delamaro, Maldonado e Jino (2007), reforçam ainda que os testes de unidade ou testes unitários têm como foco as menores unidades do programa, como por exemplo, funções, procedimentos, métodos ou classe, visando assim, identificar erros no algoritmo do software, estrutura de dados incorreta ou ainda erros de programação. Pode-se ter como exemplo, um código que faça a multiplicação do número um por ele mesmo e o resultado esperado que era um não ocorrer, ou ter como resultado o número dois. Dessa forma, percebe-se que há um erro no código do programa que pode ser detectado pelo teste de unidade na fase de teste estrutural.

4. 4 Teste Funcional ou Teste Caixa Preta

O Teste funcional também é conhecido como teste caixa preta pelo fato do testador não verificar a parte interna do software e sim apenas sua parte externa buscando saber se a parte funcional do sistema está correta (Beizer 1990 *apud* Modesto 2006). Segundo Pressman *apud* Peter (2002), afirma ainda que o teste caixa preta verifica se os requisitos funcionais do sistema estão corretos, que a entrada de dados é aceita adequadamente, que a saída esperada proveniente desta entrada é produzida, que a integridade das informações são mantidas entre outras coisas, mostrando assim que não há a menor preocupação com o código do sistema.

Para a aplicação deste tipo de teste é necessário adotar alguns critérios, tendo em vista que não há como se testar todos os tipos de entradas possíveis, pois esta tarefa se tornaria muito exaustiva. Pode-se então, adotar os critérios de particionamento em classes de equivalência, que tem como objetivo separar as entradas em classes, diminuindo assim os casos de teste, pois se supõe que qualquer entrada pertencente à sua classe de equivalência terá o mesmo comportamento (Fabbri; Vincenzi e Maldonado 2007).

Outro critério de teste funcional segundo Molinari (2008), é a Análise do Valor Limite, o qual permite testar as fronteiras de entradas, tendo em vista que o maior número de erros ocorre nos limites dos valores de entrada, como por exemplo, se a entrada aceitar apenas números reais entre um e nove deve-se testar o número zero, um, nove e dez.

Destaca-se também o critério Grafo Causa-Efeito em que se cria um grafo de acordo com as entradas, analisando as saídas, comparando-as com as especificações do software, chegando assim, nos casos de teste (Myers 2004 apud Fabbri; Vincenzi e Maldonado 2007).

4.5 Teste Estrutural ou Teste Caixa Branca

O Teste estrutural pode ser denominado teste caixa branca, pois visa a implementação do aplicativo e requer a execução de partes ou componentes do programa (Myers 2004 e Pressman 2006 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007). Sendo assim, os caminhos lógicos do programa são testados, provando se as condições, laços e variáveis foram implementadas corretamente (Maldonado 1991 e Pressman 2006 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado (2007). Molinari (2008), afirma ainda que o teste caixa branca garante que todas as linhas de código e caminhos foram executas pelo menos uma vez e que todas estão corretas. Figura 1, exemplo de caminho de teste em que os losangos são os possíveis caminhos ou tomadas de decisões (*ifs*), e os retângulos representam as ações do sistema (Molinari 2008).

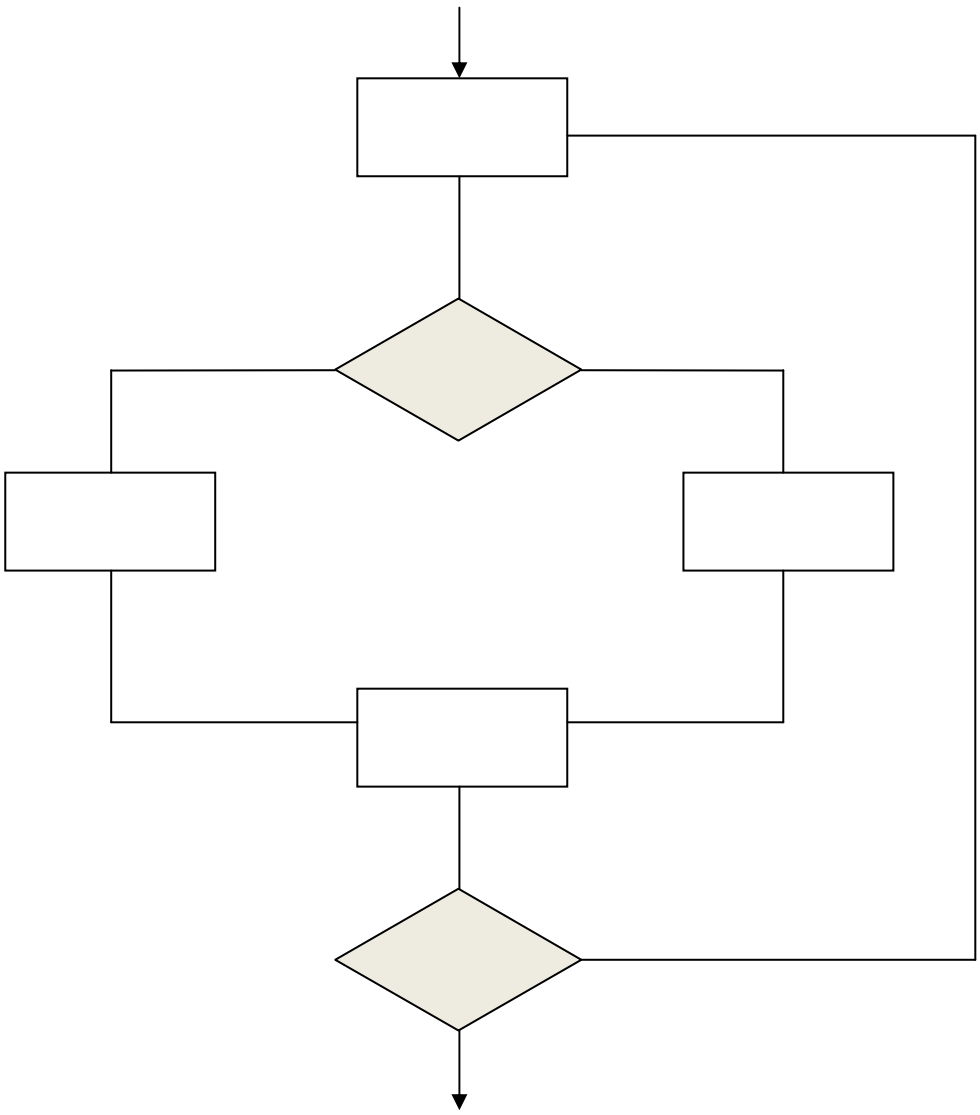


Figura 1 – Caminhos de Teste em uma Estrutura da Aplicação

Para o teste estrutural também se deve seguir alguns critérios de teste a fim de que esta tarefa não seja exaustiva e que sejam testados os caminhos e condições certas (Maldonado 1991 e Pressman 2006 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007).

4.5.1 Critérios Baseados na Complexidade

Este critério utiliza a complexidade ciclomática do grafo do programa que é o número de caminhos linearmente independentes, ou seja, um caminho que leve a uma nova condição (Pressman 2006 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007).

Para saber quantos caminhos devem ser percorridos podem ser usadas as seguintes fórmulas: (Pressman 2006 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007):

a) V(G) = E-N+2, tal que V é a complexidade, E é o número de arcos e N é o número de nós do grafo G

b) V(G) = P+1, tal que P é o número de nós contido no grafo do programa

Sendo que o valor da complexidade ciclomática é o número de casos de teste que deve ser aplicado.

4.5.2 Critérios Baseados em Fluxo de Controle

Os critérios baseados em fluxo de controle utilizam apenas características da execução do programa, como comandos e desvios para determinar em quais partes do sistema os testes devem ser aplicados. Os métodos mais utilizados nestes critérios são: todos-Nós que exigem que cada comando do programa deva ser testado pelo menos uma vez; todas-Arestas onde todos os desvios de fluxo sejam executados e por final todos-caminhos que requer que todos os possíveis caminhos do sistema sejam percorridos em busca de falhas. Entretanto, é preciso ter em vista que este método é impraticável, pois seria quase impossível testar todos os caminhos de um aplicativo (Myers 2004 e Pressman 2006 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007).

4.5.3 Critérios Baseados em Fluxo de Dados

Segundo Molinari (2008), os testes baseados em fluxo de dados se concentram nas definições e usos de variáveis no software, sendo importantes para selecionar caminhos que devam ser testados e que contenham condições aninhadas (IF aninhado).

De acordo com Ural (1988) *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado (2007), o critério baseado em fluxo de dados se mostra mais eficiente para classes com defeitos computacionais, nas quais as dependências de dados são identificadas.

4.5.4 Critérios de Rapps e Weyuker

Para se chegar aos casos de teste pelo critério Rapps e Weyuker deve-se fazer uma extensão do grafo do programa colocando em suas arestas as variáveis que tiverem algum valor atribuído, sejam elas globais ou locais e com base nesse conceito, propõem-se os principais métodos para seguir este critério que são, todas-definições que requer que toda variável seja testada não importa se esta seja global ou local; todos-usos, que determina que onde houver variáveis sejam elas globais ou locais devem ser testadas e que pelo menos um caminho onde não houver esta variável também seja testado e, por fim, todos-du-caminhos que necessita que todos os caminhos subsequentes às variáveis declaradas sejam testados (Rapps e Weyuker 1982 e 1985 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007).

4.5.5 Critérios Potenciais-Uso

Segundo Barbosa, Chaim, Vincenzi, Delamaro, Jino e Maldonado (2007), o critério potencial-uso visa testar os caminhos em que determinada variável irá passar. Na qual é possível aplicar casos de teste em todos-potenciais-usos, aplicando testes por todos os caminhos em que determinada variável passar; todos-potenciais-usos/du que requer que pelo menos um potencial caminho seja testado e todos-potenciais-du-caminhos, que requer que todas as variáveis e os caminhos atingidos por ela sejam testados (Maldonado 1991 *apud* Barbosa; Chaim; Vincenzi; Delamaro; Jino e Maldonado 2007).

5 O JUnit

Para facilitar os testes unitários, foi desenvolvido um framework denominado JUnit, que é uma poderosa ferramenta de automação de testes. Através dela é possível criar códigos que são capazes de testar unidades de um sistema (Massol e Husted 2005).

Segundo os autores o JUnit engloba três metas, ajudar a escrever testes, ajudar a criar testes que retenham seus valores com o passar do tempo e, principalmente ajudar a baixar o custo de escrever testes reutilizando o código.

Os benefícios que mais se destacam na implementação de testes com o JUnit de acordo com Myers (2004) *apud* Biasi (2006) são: criar testes unitários para métodos pertencentes a uma classe, permitir a execução de um conjunto de testes unitários, chamado de Suíte de Teste, permitir a execução de testes e relatar os problemas ocorridos e onde ocorreram.

Para um melhor entendimento do JUnit é possível observar na figura 2 a organização e o funcionamento das classes deste framework (Neto, [s.d]).

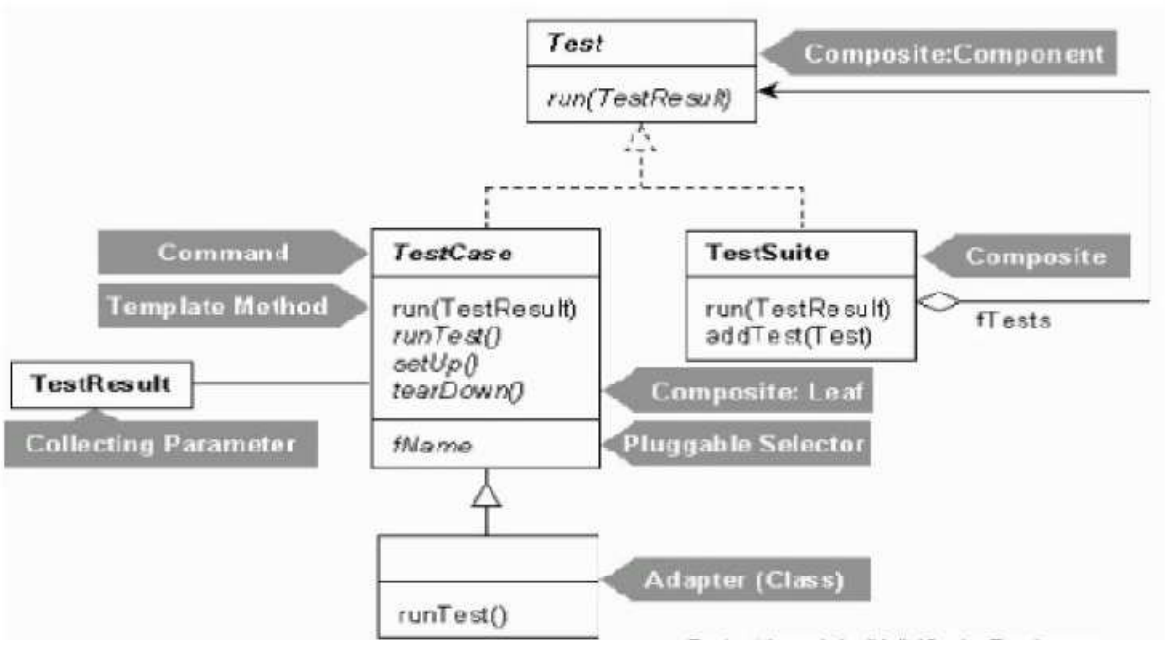


Figura 2 – Arquitetura do JUnit. Fonte Manual do Junit

Classe TestCase:

- *command* – Permite encapsular um pedido (de teste) como objeto e fornecendo o método *run*().
- *run*() – Cria um contexto (método *setUp*); em seguida executa o código e é feita a verificação do resultado (método *runTest*); terminado é feita a limpeza do contexto na memória (método *tearDown*).
- *setUp*() – Método chamado antes de cada método, pode ser utilizado para abrir uma conexão de banco de dados.
- *tearDown*() – Método chamado depois de cada método de teste, usado para desfazer o que o método *setUp*() fez, por exemplo fechar a conexão de banco de dados.
- *runTest*() – Método responsável por controlar a execução de um teste particular.

Classe TestSuite:

Com esta classe é possível executar um teste com vários métodos, registrando os resultados no *TestResult.*

- *composite* – O padrão (*pattern*) permite tratar objetos individuais e composições de objetos de forma uniforme.
- *addTest*() – Método responsável em adicionar um novo teste.

Dentro do framework JUnit encontram-se também os métodos para fazer a verificação de tomadas de decisão, que são: *asserTrue*, verifica se a expressão é verdadeira; *assertFalse*, verifica se a expressão é falsa; *assertEquals* que faz a comparação se o objeto “a” é igual ao objeto “b” e por fim o método *assertNull*, que verifica se o objeto é nulo (Myers 2004 *apud* Biasi 2006).

6 TESTES COM OBJETOS MOCK

É uma técnica em que se simula implementações do código do programa, permitindo criar testes de unidade de código em separado de outros objetos, através desta simulação pode-se testar mais rapidamente unidades de programa, pois às vezes os objetos reais podem ser difíceis de configurar, ou ele pode não existir (Massol e Husted 2005).

Esta técnica foi desenvolvida propondo a criação de um *Mock Object* (objeto falso), que emula um objeto real B podendo assim ser substituído, quando um objeto A a ser testado dependia da funcionalidade do objeto B, e devido ao fato de que um verdadeiro teste unitário deva testar apenas uma funcionalidade do sistema por vez, tendo em vista que em grande parte dos sistemas produzidos existem várias dependências entre seus objetos e métodos, melhorando assim tanto o código da aplicação como o código de testes, pois simplificam a estrutura do teste e evitam a poluição do código do sistema com a criação de elementos para teste (Mackinnon 2000 *apud* Prange 2007).

Mackinnon (2000) *apud* Prange (2007), descreve ainda que os *Mock Objects* não servem para testar os objetos que estão sendo simulados, servem apenas para testar códigos que ainda não foram escritos ou que já foram escritos e que dependam destes objetos e também que não devam reimplementar as funcionalidades que forem simuladas, devem apenas reproduzir as respostas necessárias para a realização de um teste particular.

Com o uso dos *Mock Objects* é possível não apenas substituir um objeto, mas também controlar todas as interações que são feitas com ele, como por exemplo: verificar chamadas de métodos, valores de parâmetros, definir valores de retorno e etc. A figura 3 mostra a diferença entre os objetos reais e os objetos simulados em um teste unitário de software.

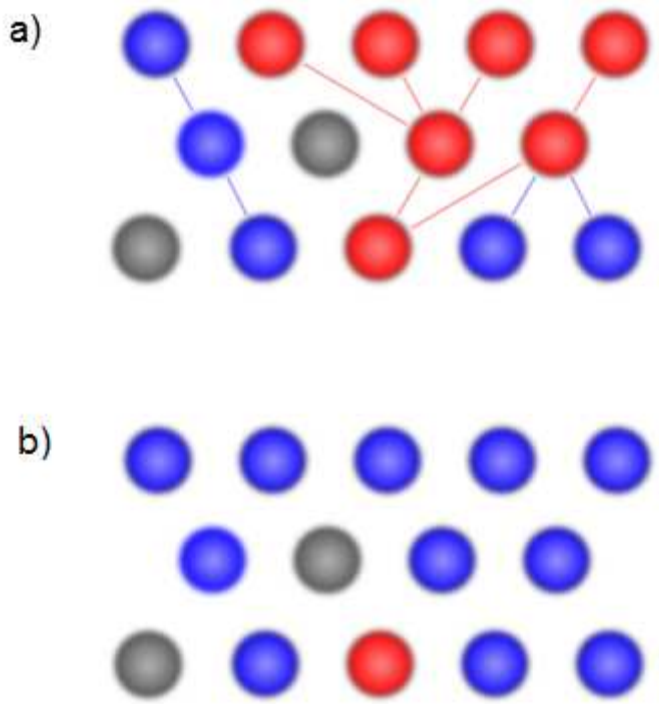


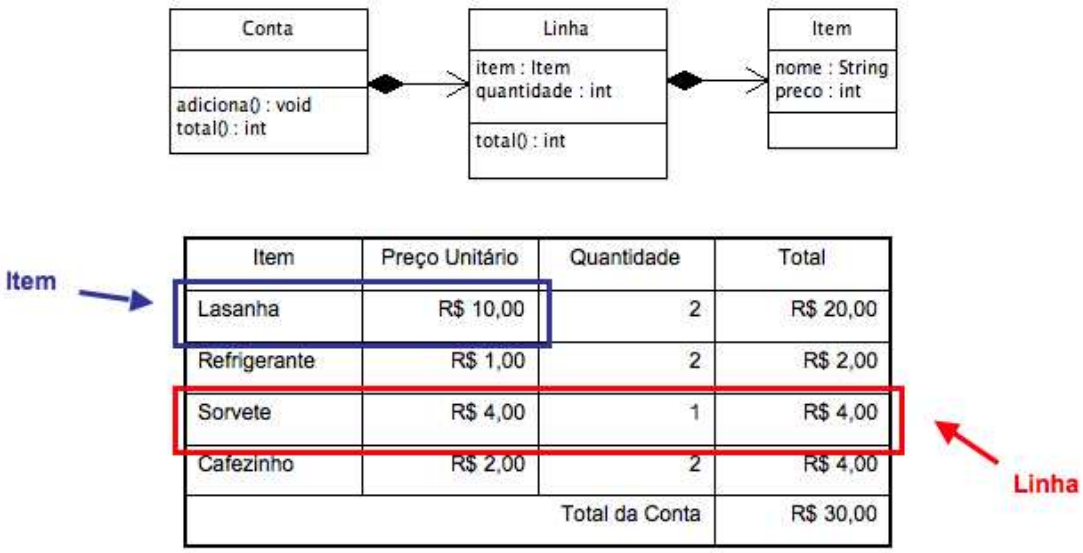
Figura 3: a) objetos reais com dependências, b) objetos simulados (livre de dependências)
Fonte: http://rafaelliu.net/?tag=mock-objects

Além das vantagens do desacoplamento dos objetos, segundo Liu (2009) existem outras vantagens na utilização dos *mock objects*, como por exemplo: a abstração das camadas mais baixas do sistema, pois elas podem não estar prontas e também devido ao fato de se trabalhar sem utilizá-las; centralizar a configuração de estado no próprio *mock* ao invés de espalhá-las pelo teste de unidade; a simulação de condições difíceis de serem reproduzidas, como por exemplo, um valor de retorno raro; e também é possível verificar mais rapidamente quando ocorre um erro.

Entretanto, o uso excessivo deste tipo de objeto pode comprometer os testes, pois poderá haver uma dependência muito grande entre vários *mock objects*, tornando os testes difíceis de serem realizados. Assim, os testes podem falhar, mesmo que estes obedeçam corretamente à lógica de programação, isto é, se o código faz realmente o que é esperado que ele faça, podendo resultar então em um grande crescimento de modificações a serem realizadas (Mackinnon 2000 *apud* Prange 2007).

Para um melhor entendimento do uso dos *mock objects* as figuras 5,

6 e 7 mostram as diferenças entre testes realizados sem e com objetos simulados da classe conta, de acordo com a figura 4.



Item	Preço Unitário	Quantidade	Total
Lasanha	R$ 10,00	2	R$ 20,00
Refrigerante	R$ 1,00	2	R$ 2,00
Sorvete	R$ 4,00	1	R$ 4,00
Cafezinho	R$ 2,00	2	R$ 4,00
		Total da Conta	R$ 30,00

Figura 4 – Classes usadas para modelar uma conta e exemplo de uma conta de restaurante - Fonte: www.improveit.com.br/XP/praticas/tdd/mock_objects

As figuras 5 e 6, mostram a criação da classe conta, o método total() que foi testado e também a criação das outras duas classes LinhaItem e Item, que eram precisas durante o teste tornando o método de teste maior e mais complexo, além disso, este método de se fazer um teste faz com que o teste não seja verdadeiramente um teste de unidade, pois a classe conta não está sendo testada isoladamente.

Classe TestCase

```
1.  import junit.framework.TestCase;
2.
3.  public class ContaTeste extends TestCase {
4.
5.      public void testTotalNota() {
6.          Conta conta = new Conta();
7.
8.          Item lasanha = new Item();
9.          lasanha.setNome("Lasanha a Bolognesa");
10.         lasanha.setPreco(10);
11.
12.         Item refrigerante = new Item();
13.         refrigerante.setNome("Guarana");
14.         refrigerante.setPreco(1);
15.
16.         Item sorvete = new Item();
17.         sorvete.setNome("Sorvete de Chocolate");
18.         sorvete.setPreco(4);
19.
20.         Item cafezinho = new Item();
21.         cafezinho.setNome("Cafe Expresso");
22.         cafezinho.setPreco(2);
23.
24.         LinhaItem linhaLasanha = new Linha(lasanha, 2);
25.         LinhaItem linhaRefrigerante = new Linha(refrigerante, 2);
26.         LinhaItem linhaSorvete = new Linha(sorvete, 1);
27.         LinhaItem linhaCafezinho = new Linha(cafezinho, 2);
28.
29.         conta.adiciona(linhaLasanha);
30.         conta.adiciona(linhaRefrigerante);
31.         conta.adiciona(linhaSorvete);
32.         conta.adiciona(linhaCafezinho);
33.
34.         assertEquals(30, conta.total());
35.     }
36. }
```

Figura 5 – Criação da classe *TestCase* (testes) - fonte:www.improveit.com.br/XP/praticas/tdd/mock_objects

Classe Conta

```
1.  public class Conta {
2.      private int total;
3.
4.      public void adiciona(LinhaItem linhaItem) {
5.          total += linhaItem.total();
6.      }
7.
8.      public int total() {
9.          return total;
10.     }
11. }
```

Classe LinhaItem

```
1.  public class LinhaItem {
2.      private Item item;
3.      private int quantidade;
4.
5.      public LinhaItem(Item item, int quantidade) {
6.          this.item = item;
7.          this.quantidade = quantidade;
8.      }
9.
10.     public int total() {
11.         return item.getPreco() * quantidade;
12.     }
13. }
```

Classe Item

```
1.  public class Item {
2.      private String nome;
3.      private int preco;
4.
5.      //... Métodos get/set
6.  }
```

Figura 6 – Criação das classes Conta, LinhaItem e item - fonte:www.improveit.com.br/XP/praticas/tdd/mock_objects

Na figura 7 foram introduzidos os *mock objects*, estes foram chamados de *LinhaMock*, que é um objeto que já recebe no construtor o valor total de uma linha. Usando a interface *Linha* pode-se evitar o uso da classe real *LinhaItem*.

```
1.  import junit.framework.TestCase;
2.
3.  public class ContaTeste extends TestCase {
4.      public void testTotalNota() {
5.          Conta conta = new Conta();
6.          conta.adiciona(new LinhaMock(20));
7.          conta.adiciona(new LinhaMock(2));
8.          conta.adiciona(new LinhaMock(4));
9.          conta.adiciona(new LinhaMock(4));
10.         assertEquals(30, conta.total());
11.     }
12. }
```

Nova interface Linha

```
1.  public interface Linha {
2.      int total();
3.  }
```

Classe LinhaMock

```
1.  public class LinhaMock implements Linha {
2.      private int total;
3.
4.      public LinhaMock(int total) {
5.          this.total = total;
6.      }
7.
8.      public int total() {
9.          return total;
10.     }
11. }
```

Figura 7 – Criação dos Mock Objects – fonte:www.improveit.com.br/XP/praticas/tdd/mock_objects

7 MATERIAL E MÉTODOS

Será usado o framework JUnit para a implementação do teste no método salvar, verificando se a saída de produtos do aplicativo Sysfarm ocorre de forma satisfatória, usando também os *mock objects* para o auxilio desta implementação visando isolar a classe no qual será testada.

8 CASO DE USO

A descrição do aplicativo Sysfarm segue da seguinte forma:

Funcionalidade: O software deve ser capaz de informar ao usuário todas as entradas e saidas de insumos agrícolas que foram aplicadas nas diversas propriedades rurais do cliente.

Apenas usuários cadastrados poderão operar o sistema.

O programa deverá cadastrar propriedades, fornecedores, produtos, entrada e saída de produtos, classes e tipos de produtos. Para o cadastro o software deverá fazer uma verificação de entradas, onde não poderão ser cadastrados caracteres inválidos e os principais campos (obrigatórios) não poderão ficar vazios, como por exemplo data e valor e quantidade.

Para o cadastro de entrada de produtos deverá ser feito um cadastro do produto no sistema caso este não contenha ainda o produto a ser dada a entrada, para isso o usuário devera obedecer o cadastro da classe e Tipo de produto a ser cadastrado, como por exemplo: produto Trifuralina, classe (cadastrado anteriormente) herbicida, tipo (cadastrado anteriormente) químico.

No momento em que for dada a saída do produto o sistema verificará no estoque da propriedade em que vai ocorrer a saída se há o produto e a quantidade a ser passada, havendo o produto estocado, o sistema dará baixa no estoque, armazenando, a data e o custo da aplicação, a quantidade de produto utilizado e a quantidade de terra aplicada, caso haja devolução de produto para o estoque deverá ser possível fazer o mesmo no sistema.

A qualquer momento o usuário do sistema poderá gerar relatórios que podem ser feitos por propriedades ou por produtos individuais, visando demonstrar todos os produtos aplicados em suas propriedades rurais, quantidades e datas, dando assim um controle de custo da safra, podendo também conhecer em quais propriedades foram aplicados determinado produto, isso deverá ocorrer de

acordo com a seleção do período do relatório, como por exemplo: período inicial 22/03/2010 até o Período de 22/08/2010. Gerando assim um relatório deste período.

Interfaces externas: O software deverá interagir com o usuário de forma simples, apenas dando o controle de aplicações e custo do mesmo, o usuário deverá apenas seguir uma hierarquia de cadastros como por exemplo, o sistema não deixará o usuário cadastrar um produto caso a classe do mesmo não estiver cadastrada, isso acontece também para a Saída de Produto não dará a saída do produto caso o mesmo não tenha ou não contenha a quantidade necessária para a aplicação.

Desempenho: Para o cadastro de Itens o sistema deverá executar a tarefa instantaneamente, para a saída haverá uma tolerância de até 3 segundos, já para pesquisa e relatório poderá ocorrer uma demora de 3 a 5 segundos caso aja uma sobrecarga do banco de dados no futuro

Requisitos lógicos de banco de dados: O banco de dados a ser utilizado: MySQL.

Para não haver uma sobrecarga do sistema o banco de dados terá quantidade de caracteres e cadastros básicos, com todas as suas entidades de dados com seus respectivos relacionamentos, como por exemplo produto relacionado com sua classe e tipo.

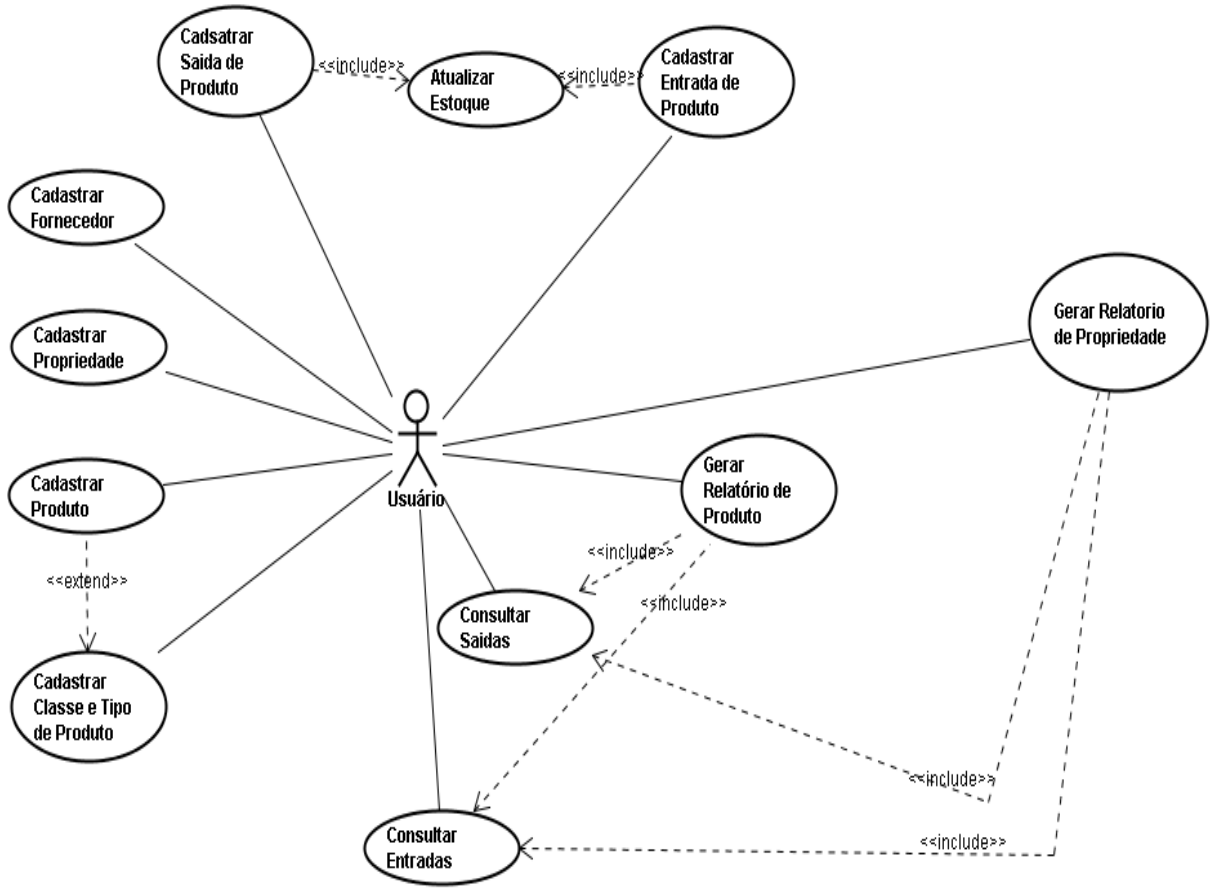
Atributos: O sistema não poderá falhar em situações de estocagem, e valores de produtos, ele será instalado em uma máquina com o sistema operacional Windows, podendo futuramente migrar para Linux, será acessado somente por pessoas cadastradas no sistema e principalmente ser de fácil uso, contendo apenas os requisitos impostos pelo usuário.

Restrições de design impostas na implementação: O aplicativo SYSFARM será desenvolvido na linguagem JAVA devido a sua portabilidade e ao fator de ser uma linguagem “*free*”, terá um padrão de janelas pois o mesmo já é conhecido pelos usuários, deverá haver uma fácil visualização dos recursos do

programa fazendo com que o usuário não tenha dúvidas sobre a sua funcionalidade.

Devido ao fato de o sistema ser pequeno, e haver apenas um computador para rodar o aplicativo, não haverá problemas com limites de recursos de Hardware.

- **Diagrama de Casos De Uso**



- **Interfaces do Sistema**

Figura 8 – Login, onde o usuário deverá digitar nome e senha para abrir o sistema.

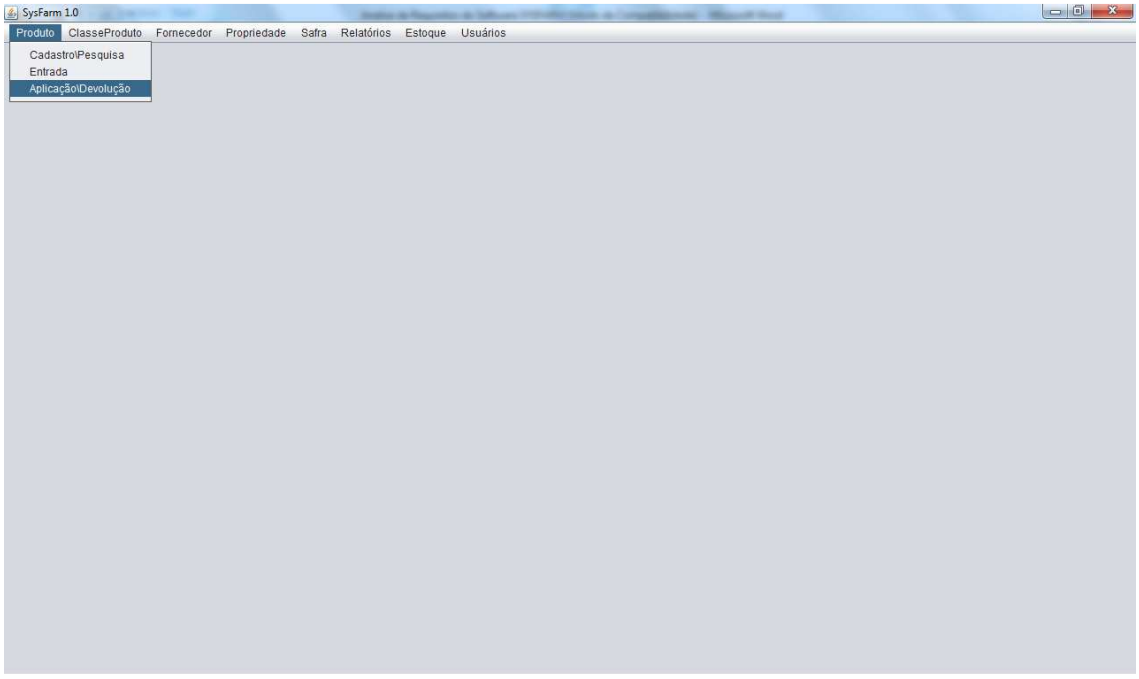


Figura 9 – Tela principal com menu para cadastros de Produto, Classe de Produto (fertilizantes, herbicidas, fungicidas e etc.), Fornecedores, Propriedades, Safra, Usuários e visualização de Estoque e Relatórios.

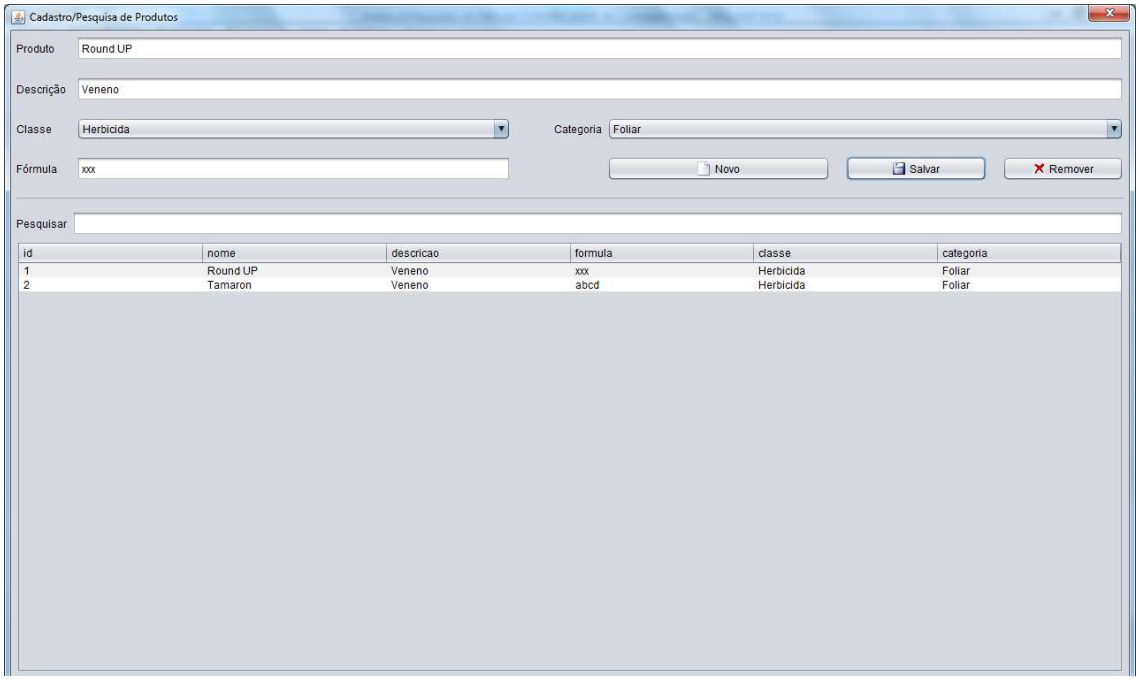


Figura 10 – Tela de cadastro de produtos, onde deverá ser informado seu nome, descrição, classe, categoria e formula, mostrando também todos os produtos cadastrados.

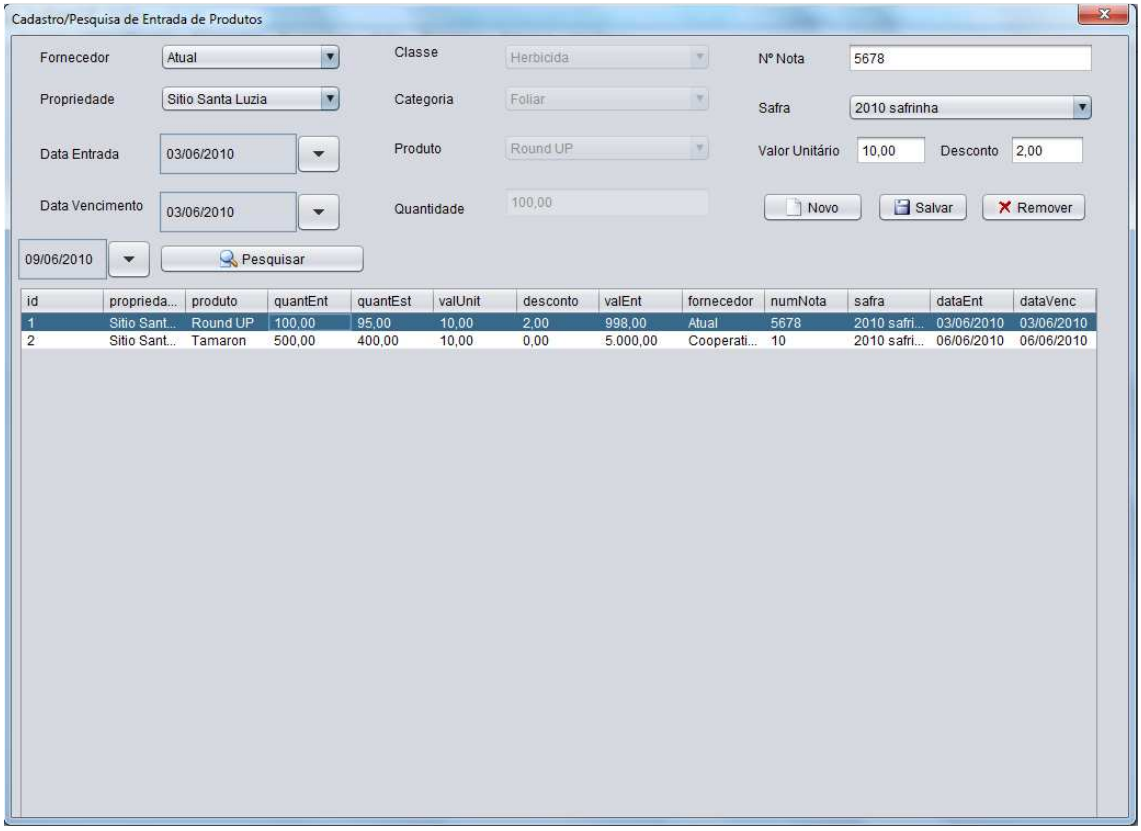


Figura 11 – Tela de Entrada de Produtos, após os cadastros de produtos, fornecedor e safra é possível cadastrar a entrada (compra) deste produto, abaixo é listada todas as entradas realizadas até o momento.

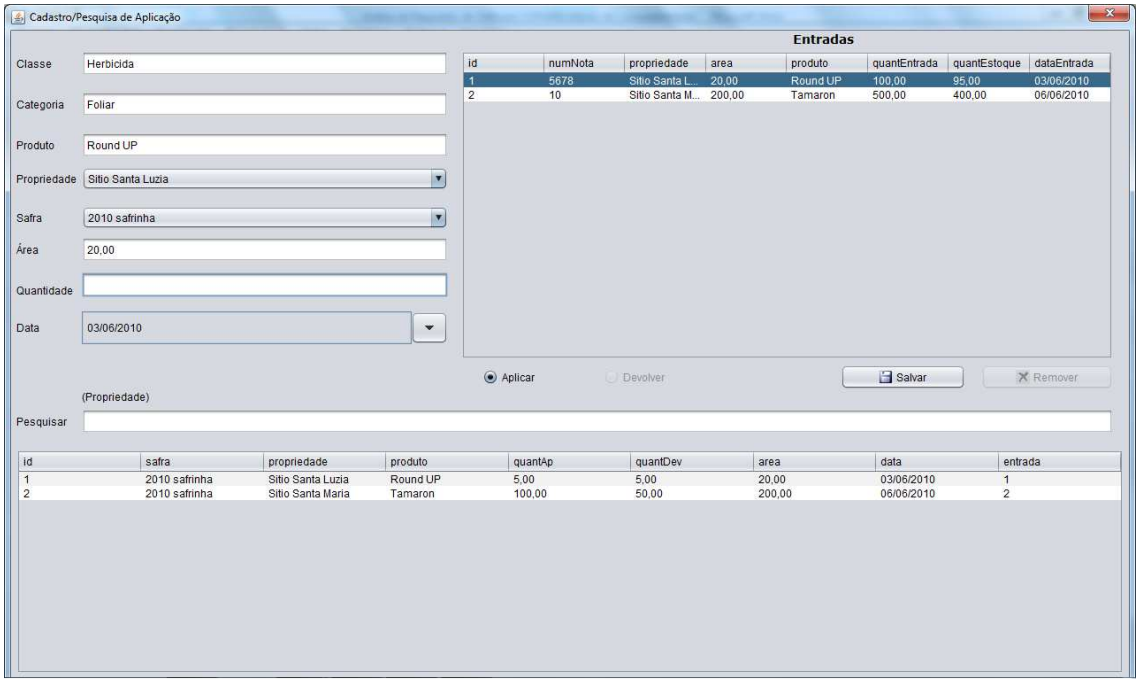


Figura 12 – Cadastro de Aplicação de Produto, ou Saída. Deve-se selecionar a entrada realizada, a propriedade onde o produto será utilizado, a safra corrente, o tamanho da área que será aplicado o produto e a data, fazendo a baixa do estoque, caso haja sobra de produto durante a aplicação deve-se selecionar a aplicação e fazer a devolução.

8.1 Descrição dos Testes Realizados

Foram criados objetos mock para testar o método salvar da classe SaídaProdutoBO, com o intuito de isolar essa classe do resto da aplicação para poder testá-la com mais precisão.

Código 1 – SaidaProdutoBO (código do aplicativo original).

Código da classe SaidaProdutoBO do aplicativo Sysfarm com o método Salvar que será testado, no qual recebe a proriedade que será aplicado um produto, o produto assim como sua quantidade e se caso haja devolução a quantidade de produto devolvido, a data da aplicação, a safra, e a quantidade de área que será aplicada.

Código 1.

```
package controller;

import daos.SaidaProdutoDAO;
import java.sql.Date;
import java.util.List;
import javax.swing.JComboBox;
import javax.swing.JTextField;
import model.EntradaProdutoVO;
import model.ProdutoVO;
import model.PropriedadeVO;
import model.SafraVO;
import model.SaidaProdutoVO;

public class SaidaProdutoBO {
   private SaidaProdutoDAO saidaProdutoDAO;
   private SaidaProdutoVO saidaProduto;
   private PropriedadeVO propriedadeVO;
   private ProdutoVO produtoVO;
   private EntradaProdutoVO entradaProdutoVO;
   private SafraVO safraVO;
   private PropriedadeBO propriedadeBO;
   private ProdutoBO produtoBO;
   private EntradaProdutoBO entradaProdutoBO;
   private SafraBO safraBO;

   public SaidaProdutoBO() {
      saidaProdutoDAO = new SaidaProdutoDAO();

      saidaProduto = new SaidaProdutoVO();
      produtoVO = new ProdutoVO();
      propriedadeVO = new PropriedadeVO();
      safraVO = new SafraVO();

      produtoBO = new ProdutoBO();
      propriedadeBO = new PropriedadeBO();
      entradaProdutoBO = new EntradaProdutoBO();
      safraBO = new SafraBO();
   }

   public int salvar(String prop, String prod, String safra, Double qtdApp, Double
qtdDev, Date data, Double area, Integer codEntrada) {
      if (!prop.equals("") && !prod.equals("") && !safra.equals("") && (qtdApp >= 0.0)
&& (qtdDev >= 0.0) && data != null && area != 0.0 && codEntrada != 0) {
         propriedadeVO = propriedadeBO.buscarPorNome(prop);
         produtoVO = produtoBO.buscarPorNome(prod);
```

```
            safraVO = safraBO.buscarPorSafra(safra);

            saidaProduto.setCodPropriedade(propriedadeVO.getId());
            saidaProduto.setCodProduto(produtoVO.getId());
            saidaProduto.setCodSafra(safraVO.getId());
            saidaProduto.setQuantAplicada(qtdApp);
            saidaProduto.setQuantDevolvida(qtdDev);
            saidaProduto.setData(data);
            saidaProduto.setArea(area);
            saidaProduto.setCodEntrada(codEntrada);
            if (saidaProdutoDAO.salvar(saidaProduto)) {
               return 1;
            } else {
               return 0;
            }
         } else {
            return -1;
         }
      }
```

A classe SaídaProdutoBO utilizava métodos das classes ProdutoBO (código 2), PropriedadeBO (código 3), SafraBO (código4) e SaidaProdutoDAO (código 5), assim foram criados objetos Mock para essas classes.

Código 2 – ProdutoBO (código do aplicativo original).

Classe que recebe o nome do produto, classe, categoria, fórmula e sua descrição.

Código 2.

```
package controller;

import daos.CategoriaDAO;
import daos.ProdutoDAO;
import java.util.List;
import java.util.Vector;
import model.CategoriaVO;
import model.ProdutoVO;

public class ProdutoBO {

   private ProdutoDAO produtoDAO;
   private CategoriaDAO categoriaDAO;
```

```
   private CategoriaVO categoriaProduto;
   private ProdutoVO produto;

   public ProdutoBO() {
      produtoDAO = new ProdutoDAO();
      produto = new ProdutoVO();
      categoriaDAO = new CategoriaDAO();
      categoriaProduto = new CategoriaVO();
   }

   public int salvar(String nome, String descricao, String classe, String categoria, String
formula) {
      if (!nome.equals("") && !descricao.equals("") && !classe.equals("") &&
!categoria.equals("") && !formula.equals("")) {
         categoriaProduto.setCategoria(categoria);
         categoriaProduto.setClasse(classe);

produto.setCategoria(categoriaDAO.buscarPorClasseCategoria(categoriaProduto).getId());
         produto.setDescricao(descricao);
         produto.setFormula(formula);
         produto.setNome(nome);
         if (produtoDAO.salvar(produto)) {
            return 1;
         } else {
            return 0;
         }
      } else {
         return -1;
      }
   }
```

Código 3 – PropriedadeBo (código do aplicativo original).

Código que cria a propriedade rural com todas suas descrições como por exemplo, nome, município, área e tipo (própria ou arrendada).

Código 3.

```
package controller;

import daos.PropriedadeDAO;
import java.util.List;
import model.PropriedadeVO;

public class PropriedadeBO {
```

```
   private PropriedadeDAO propriedadeDAO;
   private PropriedadeVO propriedade;

   public PropriedadeBO() {
      propriedadeDAO = new PropriedadeDAO();
      propriedade = new PropriedadeVO();
   }

   public int salvar(String nome, String municipio, Double area, String tipo) {
      if (!nome.equals("") && !municipio.equals("") && area != 0.0 && !tipo.equals("")) {
         propriedade.setNome(nome);
         propriedade.setMunicipio(municipio);
         propriedade.setArea(area);
         propriedade.setTipo(tipo);
         if (propriedadeDAO.salvar(propriedade)) {
            return 1;
         } else {
            return 0;
         }
      } else {
         return -1;
      }
   }
```

Código 4 – SafraBo (código do aplicativo original).

```
package controller;

import daos.SafraDAO;
import java.util.List;
import model.SafraVO;

public class SafraBO {

   private SafraDAO safraDAO;
   private SafraVO safraVO;

   public SafraBO() {
      safraDAO = new SafraDAO();
      safraVO = new SafraVO();
   }

   public int salvar(String safra) {
      if (!safra.equals("")) {
         safraVO.setSafra(safra);
```

```
            if (safraDAO.salvar(safraVO)) {
                return 1;
            } else {
                return 0;
            }
        } else {
            return -1;
        }
    }
```

Código 5 – SaidaProdutoDAO (código do aplicativo original).

Código que faz a inserção da aplicação de um produto em determinada propriedade.

```
package daos;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import banco.FabricaConexao;
import com.mysql.jdbc.Connection;
import javax.swing.JOptionPane;
import model.SaidaProdutoVO;

public class SaidaProdutoDAO {
    private Connection conexao;
    private PreparedStatement stmt;
    private ResultSet rs;

    public SaidaProdutoDAO() {
        conexao = FabricaConexao.getConexao();
    }

    public boolean salvar(SaidaProdutoVO saidaProduto) {
        String consulta = "INSERT INTO saidaProduto VALUES (?,?,?,?,?,?,?,?,?)";
        try {
            conexao = FabricaConexao.getConexao();
            stmt = conexao.prepareStatement(consulta);
            stmt.setInt(1, 0);
            stmt.setInt(2, saidaProduto.getCodProduto().intValue());
            stmt.setInt(3, saidaProduto.getCodPropriedade().intValue());
```

```
            stmt.setDouble(4, saidaProduto.getQuantAplicada().doubleValue());
            stmt.setDouble(5, saidaProduto.getQuantDevolvida().doubleValue());
            stmt.setDate(6, (java.sql.Date)saidaProduto.getData());
            stmt.setDouble(7, saidaProduto.getArea().doubleValue());
            stmt.setInt(8, saidaProduto.getCodEntrada().intValue());
            stmt.setInt(9, saidaProduto.getCodSafra());

            stmt.execute();
            conexao.commit();
        } catch (SQLException e) {
            try {
                conexao.rollback();
                System.out.println("Rollback feito!");
            } catch (SQLException ex) {
                ex.printStackTrace();
                JOptionPane.showMessageDialog(null, "Erro no rollback "+ex.getMessage());
            }
            e.printStackTrace();
            return false;
        } finally {
            try {
                stmt.close();
                conexao.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return true;
    }
```

Uma classe para qual vai ser criada um objeto mock deve ser uma inteface, assim as classes ProdutoBO, PropriedadeBO, SafraBO e SaidaProdutoDAO viraram intefaces, como é apresentado nos códigos 6, 7, 8 e 9, contendo apenas a assinatura dos métodos a as classes mock que implementaram essas classes devem também conter todos os métodos da interface, como demonstram os códigos 10, 11, 12 e 13.

Código 6 – ProdutoBo (interface da classe ProdutoBO).

```
package controller;

import model.ProdutoVO;
```

```
public interface ProdutoBO {


   public ProdutoVO buscarPorNome(String nome);

}
```

Código 7 – PropriedadeBO (interface da classe PropriedadeBo).

```
package controller;

import model.PropriedadeVO;

public interface PropriedadeBO {

     public PropriedadeVO buscarPorNome(String nome);
}
```

Código 8 – SafraBO (interface da classe SafraBo).

```
package controller;

import daos.SafraDAO;
import java.util.List;
import model.SafraVO;

public interface SafraBO {

   public SafraVO buscarPorSafra(String safra);

}
```

Código 9 – SaidaProdutoBO (interface da classe SaidaProdutoBo).

```
package daos;

import java.sql.SQLException;
import model.SaidaProdutoVO;

public interface SaidaProdutoDAO {
```

```
   public  boolean salvar(SaidaProdutoVO saidaProduto) throws SQLException;


   public boolean alterar(SaidaProdutoVO saidaProduto);

   public boolean remover(SaidaProdutoVO saidaProduto);

}
```

Código 10 – ProdutoBOMcok (classe mock).

Implementação do mock da classe ProdutoBO do aplicativo, implementada de forma simples, apenas recebendo seus valores.

```
package Mocks.controller;

import controller.ProdutoBO;
import model.ProdutoVO;


public class ProdutoBOMock implements ProdutoBO{
   public ProdutoVO buscarPorNome(String nome) {
      ProdutoVO produto = new ProdutoVO ();
      produto.setCategoria(2);
      produto.setDescricao ("Veneno");
      produto.setFormula("abcd");
      produto.setId(2);
      produto.setNome(nome);
      return produto;
   }
}
```

Código 11 – PropriedadeBOMock (classe mock).

Implementação do mock da classe PropriedadeBO do aplicativo, porem implementada de forma simples, apenas recebendo seus valores.

```
package Mocks.controller;


import controller.PropriedadeBO;
```

```
import model.PropriedadeVO;

public class PropriedadeBOMock implements PropriedadeBO {

   public PropriedadeVO buscarPorNome(String nome) {
      PropriedadeVO propriedade = new PropriedadeVO ();
      propriedade.setArea(200.0);
      propriedade.setId(2);
      propriedade.setMunicipio("Pedrinhas Paulista");
      propriedade.setNome(nome);
      propriedade.setTipo("Própria");
      return propriedade;
   }
}
```

Código 12 – SafraBOMock (classe mock).

Implementação do mock da classe SafraBO do aplicativo, porem implementada de forma simples, apenas recebendo seus valores.

```
package Mocks.controller;

import controller.SafraBO;
import model.SafraVO;

public class SafraBOMock implements SafraBO {

   public SafraVO buscarPorSafra(String nome) {
      SafraVO safra = new SafraVO ();
      safra.setId(1);
      safra.setSafra(nome);
      return safra;
   }
}
```

Código 13 – SaidaProdutoDAOMock (classe mock).

Implementação do mock da classe SaidaProdutoDAO do aplicativo, porem implementada de forma simples, apenas recebendo seus valores.

```
 package Mocks.daos;

import java.sql.PreparedStatement;
import java.sql.SQLException;

import banco.FabricaConexao;
import com.mysql.jdbc.Connection;
import daos.SaidaProdutoDAO;
import model.SaidaProdutoVO;


public class SaidaProdutoDAOMock implements SaidaProdutoDAO {

   private Connection conexao;
   private PreparedStatement stmt;

   public SaidaProdutoDAOMock(){
         conexao = FabricaConexao.getConexao();
   }

   public boolean salvar(SaidaProdutoVO saidaProduto) throws SQLException {
      String consulta = "INSERT INTO saidaProduto VALUES (?,?,?,?,?,?,?,?,?)";
       try {
         stmt = conexao.prepareStatement(consulta);
         stmt.setInt(1, 0);
         stmt.setInt(2, saidaProduto.getCodProduto().intValue());
         stmt.setInt(3, saidaProduto.getCodPropriedade().intValue());
         stmt.setDouble(4, saidaProduto.getQuantAplicada().doubleValue());
         stmt.setDouble(5, saidaProduto.getQuantDevolvida().doubleValue());
         stmt.setDate(6, (java.sql.Date)saidaProduto.getData());
         stmt.setDouble(7, saidaProduto.getArea().doubleValue());
         stmt.setInt(8, saidaProduto.getCodEntrada().intValue());
         stmt.setInt(9, saidaProduto.getCodSafra());
         conexao = FabricaConexao.getConexao();
         stmt.execute();
         conexao.commit();
      } catch (SQLException e) {
         return false;
      } finally {
         stmt.close();
         conexao.close();
      }
      return true;
   }
```

```
   public boolean alterar(SaidaProdutoVO saidaProduto)
   {
      return true;
   }

   public boolean remover(SaidaProdutoVO saidaProduto)
   {
      return true;
   }

}
```

Na classe SaídaProdutoBO (a que vai ser testada) foram criados objetos do tipo da interface, porem como interfaces não podem ser instanciada, esses objetos foram instanciados do tipo da classe Mock, assim a classe Saída Produto passa a usar os métodos reescritos nas classes Mock, como ilustra o código 14.

Código 14.

```
 package controller;

import Mocks.controller.ProdutoBOMock;
import Mocks.controller.PropriedadeBOMock;
import Mocks.controller.SafraBOMock;
import Mocks.daos.SaidaProdutoDAOMock;
import daos.SaidaProdutoDAO;
import java.sql.Date;
import java.sql.SQLException;
import java.util.List;
import javax.swing.JComboBox;
import javax.swing.JTextField;
import model.EntradaProdutoVO;
import model.ProdutoVO;
import model.PropriedadeVO;
import model.SafraVO;
import model.SaidaProdutoVO;


public class SaidaProdutoBO {
   //Criacao dos objetos
   private SaidaProdutoDAO saidaProdutoDAO;
   private SaidaProdutoVO saidaProduto;
   private PropriedadeVO propriedadeVO;
```

```
   private ProdutoVO produtoVO;
   private EntradaProdutoVO entradaProdutoVO;
   private SafraVO safraVO;
   private PropriedadeBO propriedadeBO;
   private ProdutoBO produtoBO;
   private EntradaProdutoBO entradaProdutoBO;
   private SafraBO safraBO;

   public SaidaProdutoBO() {
      //Instanciação dos objetos
      saidaProdutoDAO = new SaidaProdutoDAOMock();

      saidaProduto = new SaidaProdutoVO();
      produtoVO = new ProdutoVO();
      propriedadeVO = new PropriedadeVO();
      safraVO = new SafraVO();

      produtoBO = new ProdutoBOMock();
      propriedadeBO = new PropriedadeBOMock();
      entradaProdutoBO = new EntradaProdutoBO();
      safraBO = new SafraBOMock();
   }

   public int salvar(String prop, String prod, String safra, Double qtdApp, Double
qtdDev,/*Date data*/Double area, Integer codEntrada) throws SQLException {
      if (!prop.equals("") && !prod.equals("") && !safra.equals("") && (qtdApp >= 0.0) &&
(qtdDev >= 0.0) /*&& data != null*/ && area != 0.0 && codEntrada != 0) {
         propriedadeVO = propriedadeBO.buscarPorNome(prop);
         produtoVO = produtoBO.buscarPorNome(prod);
         safraVO = safraBO.buscarPorSafra(safra);

         saidaProduto.setCodPropriedade(propriedadeVO.getId());
         saidaProduto.setCodProduto(produtoVO.getId());
         saidaProduto.setCodSafra(safraVO.getId());
         saidaProduto.setQuantAplicada(qtdApp);
         saidaProduto.setQuantDevolvida(qtdDev);
        // saidaProduto.setData(data);
         saidaProduto.setArea(area);
         saidaProduto.setCodEntrada(codEntrada);
         if (saidaProdutoDAO.salvar(saidaProduto)) {
            return 1;
         } else {
            return 0;
         }
      } else {
         return -1;
      }
```

```
}
```

Para realização de testes com a biblioteca JUnit foi criada uma classe SaidaProdutoBOTest, é criado um objeto saídaProdutoBO, e com o uso da função assertEquals foi testado seu método salvar (código 15), onde foram passados parâmetros corretos e incorretos nos testes, verificando assim que o método esta funcionando adequadamente.

Código 15 – Classe SaidaProdutoBOTest.

```
package daos;

import controller.SafraBO;
import controller.SaidaProdutoBO;
import java.sql.SQLException;
import java.text.ParseException;
import java.util.Calendar;
import junit.framework.TestCase;
import org.junit.Test;

public class SaidaProdutoBOTest extends TestCase {
   //Criação do objeto saidaProdutoBO
   private SaidaProdutoBO saidaProdutoBO;

   @Override
   public void setUp() throws ParseException {

   }

   @Test
   public void testSaidaProduto() throws SQLException {
      //insciação do saidaProdutoBO.
      saidaProdutoBO = new SaidaProdutoBO();

     assertEquals(1,saidaProdutoBO.salvar("Sitio Santa Luzia", "Tamaron","12/12/2010",
10.1, 9.8,10.1, 1));
   }

   @Override
   public void tearDown() {
       saidaProdutoBO = null;
```

```
    }
}
```

O método assertEquals retorna valor um para testes satisfatórios e menos um para testes que falharam, para este teste como pode ser observado o método espera receber o valor 1, caso o método salvar de SaidaProdutoBO esteja inserindo os objetos corretamente.

9 RESULTADOS E CONCLUSÕES

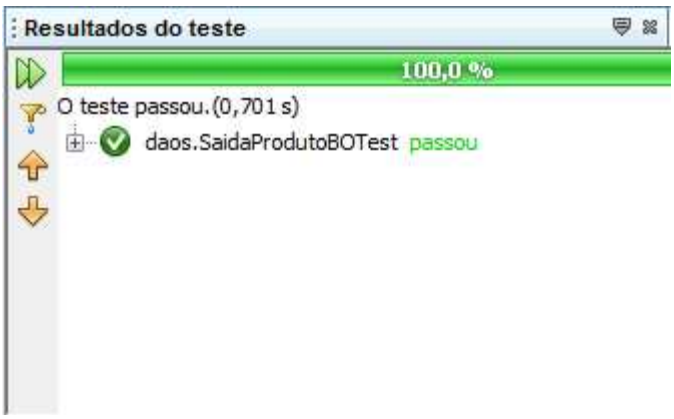


Figura 12 – Tela de resultado de teste.

Conforme visto na figura 12, o teste do método salvar da saída de produtos funciona corretamente, passado os dados para fazer uma inserção de aplicação de produto.

O uso dos testes de unidade mostra-se eficiente para a verificação de erros no código fonte.

O desenvolvimento dos objetos mock para confeccionar os testes simplificou o código, pois isolou as classes dependentes que foram usadas, facilitando observar o comportamento apenas do método que foi testado, e também devido a estes objetos pôde-se fazer alterações nas entradas de dados como por exemplo, testar diversas inserções de produtos e quantidades diferentes sem poluir o código original do sistema. O fato de haver de implementar interfaces para os objetos mock não dificultou o processo, pois o desenvolvimento desta é simples, como observado nos códigos.

O uso do framework JUnit é satisfatória pois o código fonte dos teste poderá ser reaproveitado em futuros testes para outros programas, reduzindo o tempo gasto para realizá-los, é de fácil implementação e também facilita a inserção de novos casos de teste.

REFERÊNCIAS BIBLIOGRÁFICAS

BARBOSA E. F.; CHAIM M. L.; VINCENZI A. M. R.; DELAMARO M. E.; JINO M.; MALDONADO J. C. Introdução ao Teste de Software. Elsevier, 2007, cap. 4, p. 47-74.

BIASI B. L. Geração Automatizada de Drivers e Stubs de Teste para JUnit a Partir de Especificações U2TP. Porto Alegre; 2006.

DELAMARO M.E.; MALDONADO J.C.; JINO M.. Introdução ao Teste de Sodtware. Elsevier, 2007 cap.1 p.1-7.

FABBRI S. C. P. F.; VINCENZI A. M. R. V.; MALDONADO J. C. Introdução ao Teste de Software. Elsevier, 2007, cap. 2, p. 9-24.

FRANZEN, M. B.; BELLINI, C.G.P. Arte ou Prática em Teste de Software. Read – Ed 45, V.1, n.3, maio-junho 2005.

LIMA G. M. P. S.; TRAVASSOS G. H. Teste de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes s.d.

LIU R. – Implementando Testes com Mock Objects; 2009.

MASSOL V.; HUSTED T. JUnit em Ação, Editora Ciência Moderna, cap. 1 p. 1-17.

MASSOL V.; HUSTED T. JUnit em Ação, Editora Ciência Moderna, cap. 7 p. 153-180.

MODESTO L. R. Teste Funcional Baseado em Diagramas da UML. 2006.

MOLINARI L. Teste de Software – Produzindo Sistemas Melhores e Mais Confiáveis. Érica ed.4 cap.6 p. 157-171 2008.

NETO P. D. V. A. Criando Testes com JUnit s.d

PETER R. Agregando Qualidade Com Testes Funcionais de Software.

PRANGE F. H. Uma Avaliação Empírica de um Ambiente Favorável para o Desenvolvimento Dirigido por Testes. Rio de Janeiro – 2007.

TELES M. V. Improve IT – 2006.