UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

***CAMPUS* LUIZ MENEGHEL**

ALLAN COUTINHO ABUCARUB

UMA ABORDAGEM PARA TESTES DE REGRESSÃO UTILIZANDO JUNIT

Bandeirantes

2012

ALLAN COUTINHO ABUCARUB

UMA ABORDAGEM PARA TESTES DE REGRESSÃO UTILIZANDO JUNIT

Trabalho de Conclusão de Curso apresentado à Universidade Estadual do Norte do Paraná – *campus* Luiz Meneghel – como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. José Reinaldo Merlin

Bandeirantes

2012

ALLAN COUTINHO ABUCARUB

UMA ABORDAGEM PARA TESTES DE REGRESSÃO UTILIZANDO JUNIT

Trabalho de Conclusão de Curso apresentado à Universidade Estadual do Norte do Paraná – *campus* Luiz Meneghel – como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

_____
Prof. José Reinaldo Merlin
UENP – *Campus* Luiz Meneghel

_____
Prof. Daniela de Freitas G. Trindade
UENP – *Campus* Luiz Meneghel

_____
Prof. Carlos Eduardo Ribeiro
UENP – *Campus* Luiz Meneghel

Bandeirantes, 26 de novembro de 2012

“Tenha coragem vá em frente. Determinação, coragem e autoconfiança são fatores decisivos para o sucesso. Não importa quais sejam os obstáculos e as dificuldades. Se estamos possuídos de uma inabalável determinação,conseguiremos superá-los independentemente das circunstancias, devemos ser sempre humildes, recatados e despidos de orgulho.”

Dalai Lama

RESUMO

Os testes são fundamentais para a qualidade do software. Assim como o software evolui, os testes também devem acompanhar essa evolução. A cara manutenção do software os testes devem ser refeitos a fim de assegurar que a alteração não introduza defeitos, o que é chamado de teste de regressão. Este trabalho apresenta uma abordagem para testes de regressão com a utilização da biblioteca JUnit, para automatizar os testes.

Os resultados dessa utilização são economia de tempo e garantia de cobertura durante os testes de regressão.

Palavras chave: Teste de software, Teste de regressão, JUnit, Abordagem.

ABSTRACT

The tests are crucial to software quality. As software evolves, the tests should also follow this trend. Each maintenance performed the tests should be redone to ensure that the change does not introduce defects, which is called regression testing. This paper presents an approach to regression testing using JUnit library to automate testing.

The results of such use are time saving and warranty coverage during the regression testing.

Keywords: Software testing, Regression testing, JUnit, Approach.

AGRADECIMENTOS

Agradeço a Deus por ter me acompanhado e dado forças para não desanimar e sempre acreditar em um futuro bom e promissor, não só durante esses 4 anos e meio, mas também durante toda minha vida.

A toda minha família por sempre estar preocupada em como estava indo na faculdade, sempre estarem dispostos a ajudarem no que forem preciso e simplesmente por serem como são, o que já é suficiente pra me inspirar a lutar e seguir em frente.

A minha namorada por sempre me incentivar e ajudar no que fosse preciso e possível, e ser compreensiva nos momentos que tive que dar atenção para os trabalhos ao invés dela.

A todos os amigos que fiz durante esse percurso, aos quais pude ajudar e ser ajudado todos com o objetivo da tão sonhada formação.

Agradeço também a todos os professores que se dedicaram a passar o seu conhecimento e que estiveram sempre disponíveis para sanar as dúvidas que surgiam, e ao meu orientador Merlin que prontamente respondia a meus e-mails e sempre me auxiliou no que foi preciso.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Os testes são uma fase muito importante no que diz respeito ao desenvolvimento de software, pois contribuem para a criação de um software de qualidade, que passe segurança aos clientes que o adquirem. Quanto menor a quantidade de erros ocorridos na utilização do software, maior a confiança do cliente nele, e é exatamente isso que os testes se propõem a fazer, diminuir ao máximo qualquer possibilidade de ocorrência de um erro no software em questão. Além desse aumento de qualidade do software e confiabilidade dos clientes, se aplicados devidamente, os testes também otimizam o trabalho e podem reduzir os custos da empresa desenvolvedora.

Uma importante e eficiente fase de teste de software é o teste de regressão. Esta fase tem por finalidade após cada alteração realizada no código do programa, fazer o re-teste desse programa utilizando os casos de teste previamente criados, com o objetivo de captar qualquer possível defeito originado pela alteração feita. Uma grande vantagem de fazer este reuso de casos de teste é justamente a de poupar o retrabalho economizando tempo e custo, além de identificar mais facilmente as possíveis anomalias geradas por essa mudança no código fonte.

1.1 OBJETIVOS

O objetivo deste trabalho é elaborar uma abordagem para reutilização de casos de teste durante os testes de regressão.

1.2 JUSTIFICATIVA

A proposta da elaboração desta abordagem para testes de regressão, parte da ideia de que com a aplicação da mesma o processo de teste de software ganhará um tempo significativo, pois sua essência é trabalhar com a técnica de reutilização de casos de teste. Esse tempo que pode ser poupado é de grande valia, pois pode refletir

diretamente no resultado do software como um todo diminuindo, por exemplo, os riscos que se têm de o projeto ultrapassar o tempo proposto e também reduzindo significativamente o re-trabalho no processo de criação de casos de teste.

1.3 MÉTODO

Este trabalho foi desenvolvido, por meio de pesquisa bibliográfica seguido de estudo empírico. Foram buscados na literatura os conceitos sobre teste de software e reuso de casos de teste. Em seguida, foi definido um conjunto de passos que o testador deve seguir para um reuso eficiente de casos de teste. Esses quatro passos poderão ser executados com o acompanhamento da explanação dos mesmos na seção da abordagem proposta, que também apresenta uma simulação do teste de regressão aplicado em uma aplicação de exemplo, para avaliar a abordagem proposta.

1.4 ORGANIZAÇÃO DO TRABALHO

O trabalho está divido em quatro seções: Introdução, Fundamentação Teórica, Abordagem Proposta e Conclusão. Na seção 2 são mostrados conceitos sobre teste de software, técnicas de teste de software, fases de teste, e o framework JUnit que será utilizado no trabalho. Na seção 3 a abordagem é apresentada, validada e seus resultados são apresentados.

2 FUNDAMENTAÇÃO TEÓRICA

Na presente seção são abordados os temas: teste de software, técnicas de teste de software, principais técnicas e seus conceitos, fases de teste, principais fases e seus conceitos, apresentação e conceitos JUnit.

2.1 TESTES DE SOFTWARE

O teste é uma das etapas do desenvolvimento de um software, que tem como objetivo minimizar a ocorrência de erros e os riscos associados, o que auxilia na garantia da qualidade do software desenvolvido (MALDONADO et al*.,* 2004).

Na execução de um teste o objetivo é descobrir o máximo de defeitos possíveis, pois o que define se um teste foi bem planejado são os seus resultados, sendo assim, quanto mais defeitos virem a tona, melhor o teste.

Segundo Myers, o Teste de Software é uma das etapas de verificação e validação (V & V), a qual tem por objetivo revelar a presença de defeitos no produto por meio de uma análise dinâmica do software e, indiretamente, aumentar a confiança na qualidade desse produto. Um teste bem sucedido é aquele que revela a presença de um ou mais defeitos até então não encontrados (MYERS, 2004).

Alguns princípios importantes a serem considerados sobre testes de software (MIRANDA JUNIOR, s.d).

a. **Testar totalmente não é possível.** É impossível detectar todos os erros presentes em um software. Logicamente, com exceção dos softwares extremamente simples.
b. **Testar é difícil e exige muita criatividade.** É necessário que o encarregado pelos testes possua experiência e seja devidamente treinado, para que se tenha um bom teste. Testar acaba se tornando difícil, pois é necessário que se conheça profundamente o sistema para que o teste seja efetivo e, normalmente, os sistemas não são tão simples.

c. **Testes devem ser planejados e projetados.** Deve se criar um bom planejamento dos casos de teste que serão utilizados, principalmente dependendo da complexidade do software. Um bom teste é aquele que tem a maior probabilidade de encontrar erros, gerando assim alta confiança.

d. **Testar requer independência.** O ideal é um testador sem vícios, ocasionados por profundo conhecimento do programa. Esses vícios podem levar o testador a ignorar certas partes do programa que julga estar correto, podendo assim, deixar passar alguns erros.

Para guiar esses testes são desenvolvidos os casos de teste, que nada mais são do que um conjunto de condições que podem verificar tanto se as especificações do software foram atendidas, quanto a estrutura interna do mesmo. buka

Segundo Myers(2004), existe um conjunto de dez importantes diretrizes que devem ser consideradas no processo de teste de um software, cujas são apresentadas no Quadro 1.

Quadro 1 - Diretrizes para o processo de teste visto de um perfil psicológico

	Diretrizes
1	Uma parte necessária de um caso de teste é a definição de um resultado esperado.
2	Um programador deve evitar a tentativa de testar o seu próprio programa.
3	A organização de programação não deve testar os seus próprios programas.
4	Inspecione os resultados de cada teste.
5	Casos de teste devem ser escritos para as condições de entrada que são inválidas e inesperadas, bem como para aqueles que são válidas e esperadas.
6	Examinando um programa para ver se ele não faz aquilo que é suposto a fazer é apenas metade da batalha, a outra metade é ver se o programa faz o que não é suposto fazer.
7	Evite casos de teste descartáveis a menos que o programa seja realmente um programa descartável.
8	Não planejar um esforço de teste sob a suposição tácita de que erros não serão encontrados.
9	A probabilidade da existência de mais erros em uma seção de um programa é proporcional ao número de erros já encontrados nesta seção.
10	O teste é uma tarefa extremamente criativa e intelectualmente desafiadora.

Fonte: Myers, 2004

A seguir serão detalhadas as diretrizes propostas por Myers (2004):

Diretriz 1: Um dos erros cometidos com mais frequência em testes de software, pois se um resultado esperado de caso de teste não foi predefinido há chances de um resultado errado passar como correto, e isso pode ser combatido por meio de uma descrição dos dados de entrada para o programa e uma descrição precisa da saída correta, para tais dados de entrada.

Diretriz 2: É extremamente difícil um programador olhar para um software, que ele mesmo desenvolveu, com outra perspectiva, ficando assim mais improvável de se encontrar erros. Outro problema significativo é que se o programador entender errado a declaração das especificações, provavelmente também se equivocará ao fazer os testes. Vale ressaltar que esses argumentos não se aplicam a depuração, pois esta é mais eficaz se executada pelo programador.

Diretriz 3: Este argumento é semelhante ao anterior, pois a organização do projeto tem problemas semelhantes aos dos programadores individuais, e além disso ela é altamente cobrada para entregar o software em uma determinada data e com certo custo. Então é importante deixar a parte de teste para uma equipe independente, para maior eficácia, consequentemente economizando com os custos ocasionados com a ocorrência de falhas.

Diretriz 4: Pode parecer o princípio mais óbvio, mas muitas vezes é esquecido. Inúmeras vezes erros não são detectados, ou são detectados novamente, por não ter sido feita uma inspeção em resultados de testes anteriores.

Diretriz 5: Geralmente quando se pensa em testar um software, se pensa nas entradas válidas e esperadas para a execução do teste, negligenciando assim as condições inválidas e inesperadas, quando na verdade se deve dar atenção para os dois tipos de entradas e não somente as válidas e esperadas, até porque, casos de teste que representam condições de entrada inválidas e inesperadas tem um maior rendimento de detecção de erros.

Diretriz 6: Este é um complemento do princípio anterior. Os programas devem ser examinados para efeitos colaterais indesejados. Por exemplo, se o software faz algo a mais do que é proposto a fazer, também deve ser tratado como erro.

Diretriz 7: Evitar criar casos de teste, utilizá-los e descartá-los, pois em uma nova situação que venha a necessitar desse teste já criado, terá que se reinventar o mesmo teste, por não ter sido armazenado anteriormente, gerando assim um trabalho muito custoso e perda de tempo.

Então é altamente recomendado que os testes de caso criados sejam armazenados, para que posteriormente possam ser reutilizados.

Guardar casos de teste e utilizá-los novamente após uma alteração é conhecido como teste de regressão.

Diretriz 8: Este é um sinal de a utilização da definição incorreta de teste, erro que gestores muitas vezes cometem, pois supõem-se de que o teste é o processo de mostrar que as funções do programa estão funcionando corretamente, o que é totalmente diferente da real definição de teste, que é o processo de execução de um programa com a intenção de detecção de erros.

Diretriz 9: Exemplificando, se um software possui os módulos A e B, e são encontrados 5 erros no módulo A e apenas 1 no módulo B, pressupõem-se que a probabilidade de ocorrência de erros no modulo A é maior que no módulo B.

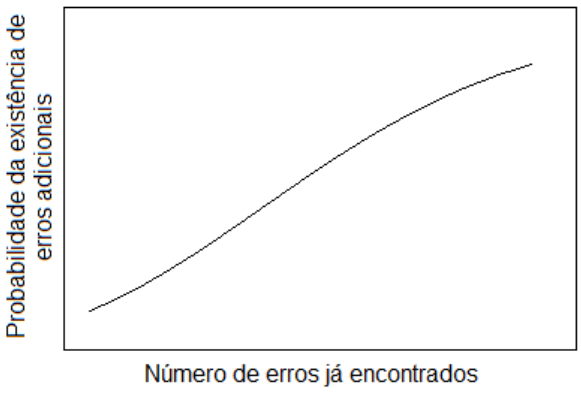

Figura 1 - Relação dos erros encontrados com os provavelmente existentes

O fenômeno ilustrado na Figura 1 é útil na medida em que nos dá uma visão ou *feedback* no processo de teste. Se uma seção específica de um software parece ser muito mais propensa a erros do que outras seções, então este fenômeno nos diz que,

para termos de rendimento do nosso investimento, os esforços de testes adicionais devem ser mais focados nesta seção mais propensa a erros.

Diretriz 10: Geralmente a criatividade necessária para testar um programa de grande porte excede a criatividade necessária na elaboração desse programa, mesmo utilizando de metodologias para o desenvolvimento de um conjunto razoável de casos de teste, é necessário uma quantidade significativa de criatividade.

2.2 TÉCNICAS DE TESTE DE SOFTWARE

2.2.1 Teste Estrutural

Também conhecida como teste “caixa-branca” essa técnica se baseia no código fonte do software para gerar os casos de teste (MYERS, 2004).

Essa técnica avalia aspectos de teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos e geralmente é utilizada na fase de Unidade, mas também pode ser aplicada as fases de Integração, Regressão e Sistema.

Uma desvantagem da técnica de caixa de caixa branca é que não analisa se a especificação está certa, concentra-se apenas no código fonte e não verifica a lógica da especificação (CARDILLI; apud LEWIS e VEERAPILLAI, 2005) e uma vantagem é que, como essa técnica trabalha direto com o código fonte do programa é mais fácil encontrar os valores de entrada mais úteis para os testes.

2.2.2 Teste Funcional

Também chamado de teste “caixa-preta”, não procura se preocupar com o comportamento interno do programa.

> Seu objetivo é ser completamente despreocupado com o comportamento interno e estrutura do programa. Em vez disso, concentrar em encontrar circunstâncias em que o programa não se comporta de acordo com as suas especificações.
> Nesta abordagem, os dados de teste são derivados somente a partir das especificações (isto é, sem tomar vantagem do conhecimento da estrutura interna do programa).

> Se você quiser usar essa abordagem para encontrar todos os erros no programa, o critério é exaustiva teste de entrada, fazendo uso de todas as condições de entrada possível, como um caso de teste *(*MYERS; 2004, p. 13).

Um exemplo de aplicação do teste é fazer entradas erradas e monitorar o comportamento do programa, verificando assim se tudo ocorreu conforme o esperado e se o resultado foi o esperado.

Nos testes funcionais a técnica de particionamento de equivalência é uma das mais simples, porém importante, técnicas utilizadas, que de acordo com as funcionalidades do software efetua uma divisão das entradas de valores em subconjuntos, por exemplo, a inserção de uma massa de dados para validar o processo de inserção. Seu princípio é a escolha da melhor abordagem a ser utilizada e a melhor maneira de se obter a validação dos erros e aumento da confiabilidade (BRUNELI, 2006).

Há também, além da técnica de particionamento, outras técnicas que podem ser adotadas, como a de Grafo Causa-Efeito e Análise do valor limite.

2.2.2.1 Particionamento por Equivalência

Particionamento por equivalência diz respeito a um critério pertencente a técnica de teste funcional, da mesma forma que outros critérios conhecidos como análise do valor limite e grafo causa-efeito, mas nesse trabalho especificamente será utilizado perticionamento por equivalência.

Segundo (PRESSMAN, 2006) este critério tem por finalidade classificar as entradas do sistema como pertencentes a classes de equivalência válidas ou inválidas e uma vez classificadas tais entradas supõe-se que quando um elemento de determinada classe provocar a falha de um teste, todos os demais membros desta mesma classe também provocarão falhas. O mesmo vale para o contrário, se um elemento obtiver sucesso como resultado de um teste, então todos os membros da classe que este elemento pertence obterão o mesmo resultado. Com a utilização deste critério consequentemente serão necessários um número muito menor de casos de teste para cobrir as possibilidades de ocorrência do teste.

Por exemplo, uma função que calcula a quantidade de dias do mês que recebe como parâmetro um número inteiro correspondente ao mês, com isso já se pode definir que:

- Classe válida = { 13 > entrada > 0 }
- Classe inválida 1 = { 12 < entrada }
- Classe inválida 2 = { 1 > entrada }

Portanto qualquer entrada que corresponda aos números de 1 á 12 pertencerá a classe válida, qualquer entrada que corresponda a números maiores que 12 pertencerá a classe inválida 1 e qualquer entrada que corresponda a números menores que 1 pertencerá a classe inválida 2. Assim definimos ao menos 1 classe válida e duas inválidas, que é o aconselhável para um teste coeso.

2.3 FASES DE TESTE

O processo de teste possui suas estratégias, que se diferem por suas funções, momento de aplicação e modo de como são aplicados, proporcionando assim uma maior abrangência referente aos diversos tipos de software.

Esse processo, de acordo com Bruno, Silva e Alves (2012), pode ser dividido basicamente nas seguintes etapas:

Teste de Unidade: Testa a menor parte do aplicativo, que é a unidade. Na programação orientada a objeto a unidade geralmente é a classe ou um método, já na procedimental a unidade pode ser uma função individual ou um módulo. Esses testes são tipicamente escritos e dirigidos pelos desenvolvedores e procura por falhas ocasionadas por defeitos de implementação e de lógica em cada módulo, separadamente. Ao aplicar os testes de unidade em cada unidade identificada do programa, o desenvolvedor tem a garantia de que, pelo menos individualmente o programa está funcionando corretamente. Portanto se algum erro for ocasionado devido a alguma alteração feita no código fonte, os testes de unidade irão identificar esse erro.

Teste de Integração: é a fase que testa a integração dos componentes do programa, visando sua funcionalidade em conjunto, verificando assim erros associados a interface.

Teste de Validação: objetivamente o teste de validação tem a função de verificar se o sistema esta de acordo com as regras de negócio estabelecidas em sua elaboração, e se ele esta funcionando devidamente de acordo com todas suas especificações de requisitos.

Teste de Sistema: é executado após o término do software e avalia o software em busca de falhas por meio da utilização do mesmo, incluindo todos os itens de software e de hardware. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do software verificando se o produto satisfaz seus requisitos. Alguns dos testes de sistemas são: Teste de Desempenho, Teste de Segurança, Teste de Recuperação e Teste de Inicialização.

Teste de Regressão: Segundo Pfleeger(2004), o teste de regressão tem a função de identificar novos defeitos que podem ter sido introduzidos no ato da correção de outros defeitos. Assim sendo o teste de regressão é um teste aplicado a uma nova versão para verificar se a versão atual está realizando as mesmas funções da mesma maneira que na versão anterior.

De acordo com Oliveira (2007 apud BLACK, 2007), existem três tipos de regressão:

- **Regressão Local:** Um novo erro é gerado em decorrência da correção ou mudança de um erro que já existe;
- **Regressão Exposta:** Um erro existente é revelado por uma alteração ou correção de outro erro;
- **Regressão Remota:** Certa área do sistema é afetada devido a uma alteração ou correção de um erro em outra área. Normalmente este tipo é o mais difícil de ser detectado.

Geralmente, um software é constantemente modificado devido aos pedidos de alterações dos usuários, sendo assim controlado através de versões. Por isso, após a criação de uma nova versão se faz necessário certificar-se que nenhuma outra funcionalidade do software foi prejudicada por essa nova alteração. Segundo Pontes(s.d.), é muito comum que outras partes do sistema sejam danificadas com essas constantes modificações no programa, devido a inúmeros pedidos de alteração que o usuário solicita. Para evitar que ocorra essa geração indesejada de erros, o teste de regressão é aplicado. Esse teste nada mais é que o re-teste desses casos de teste

elaborados desde a primeira versão até a última versão, com a finalidade de captar todos os efeitos colaterais ou erros, caso existam, causados pela última modificação no programa. Estudos mostram que quando se prioriza esses casos de teste, há um aumento na taxa de detecção de defeitos.

Nem sempre se torna possível a inclusão de todos esses casos de teste no ciclo de regressão mesmo sendo o ideal, pois se existir uma quantidade massiva de casos, esse re-teste exigirá um certo tempo disponível o que as vezes não se tem.

Uma solução eficaz e que traria inclusive bons resultados, se tratando de agilidade no processo, seria a de automatizar o processo de re-execução de todos esses casos de teste utilizando a ferramenta JUnit, a qual permite montar classes em Java referentes a esses testes e instanciá-las dentro de uma suíte de testes que executaria todos esses testes de uma só vez.

2.4 JUNIT

JUnit é um framework de código aberto, desenvolvido por Kent Beck e Erick Gamma, que possibilita a criação de testes unitários em Java. É uma ferramenta de fácil acesso, visto que, a maioria das IDEs atuais incorporam o JUnit em seu ambiente de desenvolvimento.

O JUnit[1] é de grande valia para criação e execução de código para a automação de testes, com ele é possível se verificar cada método de uma classe, sendo assim as unidades, para se certificar que estão funcionando de forma esperada. Caso contrário são exibidos possíveis erros ou falhas que venham a ocorrer. No momento em que todas as condições do teste estiverem sido testadas em determinado método e o resultado obtido for o esperado, incrementa-se a qualidade da unidade testada pois será considerado que tal unidade está atendendo as especificações necessárias (DIAS; DALCIN; D'ORNELLAS, 2006).

[1] http://www.junit.org/.

2.4.1 Vantagens

Segundo Aristides(s.d.), existem algumas vantagens na utilização do framework JUnit, as quais são:

a. Permite a criação rápida de código de teste possibilitando um aumento na qualidade do desenvolvimento e teste;
b. Amplamente utilizado pelos desenvolvedores da comunidade código-aberto, possuindo um grande número de exemplos;
c. Uma vez escritos, os testes são executados rapidamente sem que, para isso, seja interrompido o processo de desenvolvimento;
d. JUnit checa os resultados dos testes e fornece uma resposta imediata;
e. JUnit é livre e orientado a objetos.

3 UTILIZANDO JUNIT EM TESTES DE REGRESSÃO

Nesta seção será apresentada a abordagem proposta, com os seus passos devidamente descritos e explicados. Será também utilizado um sistema para que se possa validar a abordagem. Nessa validação o sistema utilizado será descrito e serão apresentadas simulações e testes que foram aplicados ao mesmo.

3.1 ABORDAGEM PROPOSTA

A abordagem foi proposta com intuito de gerar alguns benefícios importantes ao sistema que venha a utilizá-la, dentre esses benefícios estão a maior qualidade o armazenamento e a organização dos testes criados, e o poder de manter os testes criados sempre atualizados em correspondência com as alterações efetuadas.

Basicamente a abordagem se dá em quatro passos, que devem ser executados sequencial e continuamente para que a abordagem funcione adequadamente.

O primeiro passo consiste em criar as unidades do sistema, os métodos. O segundo passo consiste em criar um teste para cada unidade existente, armazenando esses testes em um diretório. O terceiro passo consiste em monitorar esses testes para

que eles se mantenham atualizados, e por fim o quarto passo consiste em executar os testes de regressão sempre que uma alteração for efetuada.

Agora será apresentada a descrição em detalhes de cada um dos passos citados anteriormente.

3.1.1 Criar unidades necessárias

Neste primeiro passo devem ser criadas as unidades do sistema.

Essas unidades são criadas de acordo com a necessidade da empresa, em decorrência das alterações solicitadas por seus clientes, ou então, em decorrência do planejamento de construção de algum módulo do sistema ou do sistema em si.

Esta tarefa é designada a equipe de desenvolvimento da empresa, os quais recebem os pedidos de alteração solicitados pelos clientes ou recebem diretamente as orientações dos encarregados pela ideia de construção de um novo módulo, nova funcionalidade no sistema ou do sistema em si. Orientações essas que podem ser apresentadas a equipe de desenvolvimento em forma de documento, escopo ou oralmente, dependendo da necessidade em questão, do porte e da forma de trabalhar da empresa.

Um exemplo de uma unidade criada pode ser visualizado na figura 2.

```
    public boolean verificaIndicado(double estMax, double estAtu, double estMin) {

        if (estAtu > estMax && estMax > estMin) {
            return true;
        } else {
            return false;
        }
    }
}
```

Figura 2 - Método pertencente ao sistema

3.1.2 Criar teste para cada unidade existente

No segundo passo da abordagem devem ser criados testes para cada unidade existente, o que é de responsabilidade da equipe de teste, pois como a segunda diretriz

identificada por Myers diz, é contra-indicado que quem desenvolveu o programa também faça os testes.

Visto que a unidade especificada para essa abordagem foi o método, para cada método criado deve ser criado seu respectivo teste, claro que obedecendo as exigências do critério de particionamento por equivalência, que foi o critério definido para essa abordagem.

Os testes que vão sendo criados devem ser sempre armazenados no diretório definido pela equipe de testes, até mesmo, se embasando na sétima diretriz identificada por Myers, que releva a importância do armazenamento dos testes criados.

Portanto, é indicado que quando a equipe de desenvolvimento terminar a criação de uma classe e suas unidades, já repassem essa classe para a equipe responsável de criar os testes, para que eles já possam trabalhar em paralelo criando os testes correspondentes a cada unidade ao invés de repassar todas as classes ao final do desenvolvimento, até mesmo para colaborar com a agilidade do processo.

Observe na figura 3 a ilustração do processo.

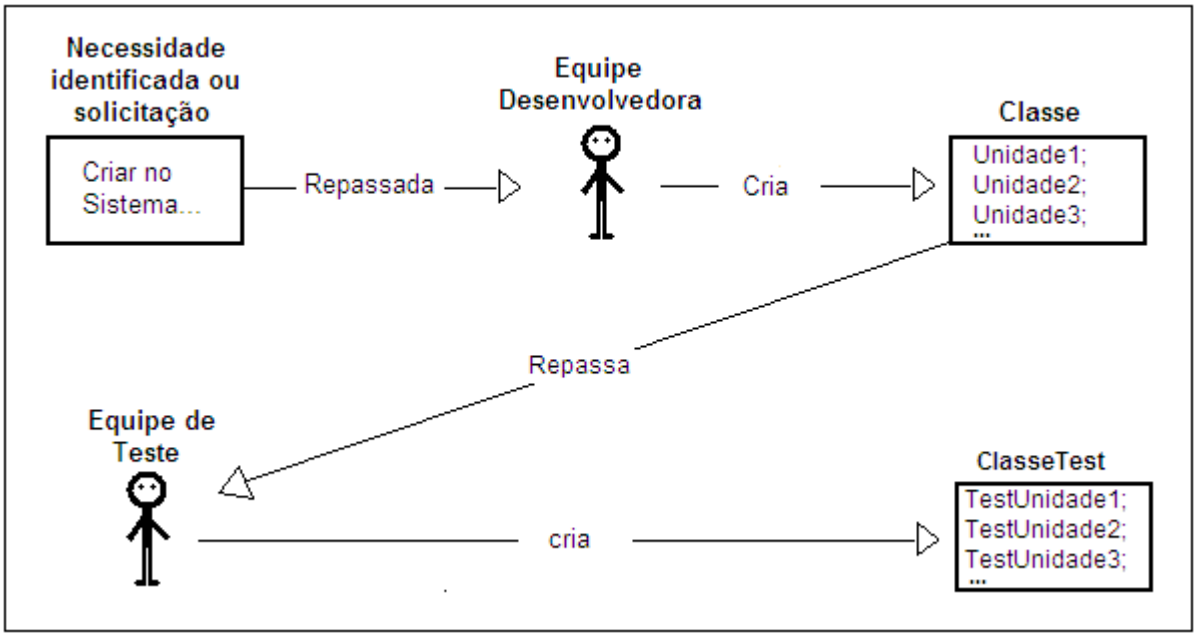

Figura 3 – Funcionamento do primeiro passo em conjunto com o segundo

A figura 3 mostra desde a solicitação que surge mediante as necessidades dos clientes ou as necessidades de criação de funcionalidades identificadas pela própria

empresa, até o processo de criação dos testes, que são criados pela equipe de teste conforme as unidades criadas são repassadas a mesma, pela equipe de desenvolvimento.

Observe que a classe de teste e as unidades de teste manterão os mesmos nomes da classe e unidades que representam, apenas adicionando o sufixo *Test* nas classes e o prefixo *Test* nas unidades, sendo isso procedimento padrão do framework JUnit, o que facilitará no controle de suas versões pelo programa proposto no terceiro passo.

Na figura 4 mostra-se um exemplo de teste criado, teste este, que equivale a unidade exemplificada na figura 2.

```
@Test
public void testVerificaIndicado() {
    System.out.println("Testando método verificaIndicado() da classe TelaProdutosPromocao");
    ResultSetData rs = new ResultSetData();
    double estMax = 0, estAtu = 0, estMin = 0;

    ResultSet retorno = rs.getByQuery("SELECT \"id\",\"descricao\",\"estoquemax\",\"estoqueatual\",\"estoquemin\" "+
                                      "FROM \"produtos\" where \"estoqueatual\" > \"estoquemax\" ");

    TelaProdutosPromocao instance = new TelaProdutosPromocao();
    boolean expResult = true; // Resultado esperado

    try {
        while (retorno.next()) {
            estMax = retorno.getDouble("estoquemax");
            estAtu = retorno.getDouble("estoqueatual");
            estMin = retorno.getDouble("estoquemin");
            String nome = retorno.getString("descricao");
            int cod = retorno.getInt("id");

            if ((estMin <= 0) || (estAtu <= estMin || estMax <= estMin)) {// Se a entrada se enquadrar em uma classe inválida..

                assertEquals("Entrada referente ao produto \"" + cod + " - " + nome // ..o teste é barrado
                        + "\" faz parte de uma classe inválida!", true, false);
            }
        }

        boolean result = instance.verificaIndicado(estMax, estAtu, estMin);
        assertEquals("Teste não passou!", expResult, result);

    } catch (SQLException e) {
        System.out.println("" + e.getMessage());
    }
}
```

Figura 4 - Teste criado

O teste criado na figura 4 utilizando da ferramenta JUnit e sua criação foi baseada no critério de particionamento por equivalência, que utiliza classes válidas e inválidas para a validação do teste.

Os testes que vão sendo criados não devem ser descartados, mas sim armazenados em um diretório específico, para que sempre estejam a disposição da equipe de teste para que possam aplicar os testes de regressão.

3.1.3 Monitorar versão dos testes

O terceiro passo da abordagem consiste em apresentar uma forma para que se tenha um monitoramento constante das versões dos testes, e para que isto ocorra as unidades também tem que ser monitoradas, pois a partir delas que será identificado se os testes tem que ter sido atualizados ou não.

Tendo-se esse controle evita-se que os testes fiquem desatualizados e acabem perdendo a eficácia por não estarem de acordo com a unidade que correspondem.

Com isso a proposta é simplesmente o desenvolvimento de um programa que faça esse monitoramento e que seja sempre executado antes de se aplicar os testes de regressão, para que os testes não estejam desatualizados no momento de aplicá-los diminuindo assim sua eficácia.

O programa proposto deve funcionar da seguinte maneira:

- É necessário que o programa receba como entrada o caminho do projeto do sistema que deve ter seus testes monitorados e a partir desse caminho leia todos os arquivos com extensão .java, classificando-os em classes de unidade e classes de teste, faça uma cópia desses arquivos e armazene-as classes de unidade em um determinado diretório e as classes de teste em outro, tendo-se assim uma forma de ter sempre armazenado as classes da ultima versão.
- Com base nesses arquivos copiados e devidamente classificados, o programa deve comparar as classes de unidade copiadas com as classes de unidade originais, para verificar se alguma unidade da classe original teve alguma alteração realizada.
- Se for identificada alguma diferença em qualquer unidade original em relação a unidade correspondente armazenada, o teste original também deve ser comparado com o teste copiado correspondente a ele, para verificar se o teste original também difere do copiado, pois se a unidade original foi alterada o teste também teria que ter sido, do contrário ele não foi atualizado.

- O programa identificando que há testes desatualizados deve informar ao operador do sistema tal fato, que preferencialmente será o testador, para que ele atualize o teste informado pelo sistema.
- O programa também oferecerá a opção de atualizar as classes desatualizadas, para que depois dos testes serem atualizados o operador do sistema possa atualizar os diretórios de cópia com as classes originais atuais.

Com a aplicação correta de um programa com essas características se terá uma garantia que os testes estarão sempre atualizados por consequência terão a integridade preservada.

3.1.4 Executar o teste de regressão a cada alteração no sistema

Este quarto passo da abordagem tem por função aplicar os testes de regressão, sendo esses testes aplicados toda vez que o sistema sofre uma alteração. Pois a alteração em uma determinada unidade pode causar efeitos colaterais em outras funcionalidades do sistema, sendo eles na mesma área da alteração efetuada ou em outra área. Com isso, acabam comprometendo o funcionamento do mesmo, podendo ocasionar erros e/ou apresentar inconsistências.

Portanto após ter efetuado todos os passos anteriores adequadamente, onde foram criadas as unidades, criados seus respectivos testes, os quais estarão sob constante monitoramento garantindo que estejam sempre atualizados, os testes de regressão podem e devem ser aplicados, para que antes da versão ser liberada ela passe por essa fase que á deixará muito menos suscetível a erros.

Para essa aplicação dos testes de regressão basta utilizar a ferramenta JUnit, onde o testador responsável abrirá os testes armazenados e os executará.

Com essa ação todos os testes serão simultaneamente executados, devido a função proporcionada pelo JUnit, a qual permite a inserção de todas as chamadas para as classes de teste em uma suíte e executando essa suíte os testes são executados instantaneamente.

Cabe ressaltar que a eficácia desse quarto passo está diretamente ligada com a correta aplicação do terceiro passo, uma vez que o terceiro passo não for devidamente

executado os testes ficarão desatualizados e consequentemente os testes de regressão, que utilizam desses testes, terão sua eficácia diminuída.

3.2 VALIDAÇÃO DA ABORDAGEM

Nesta seção apresenta-se o que foi feito para validação da aplicação da abordagem. Será apresentado o sistema utilizado para validação da abordagem, também será visto os testes efetuados para a validação e esclarecimento do funcionamento e eficácia dos testes de regressão e do programa de controle de versões.

3.2.1 Sistema Utilizado

O sistema utilizado para essa validação trata-se de um sistema de automação de mercados, que inicialmente apresenta uma tela de *login* que evita o acesso de pessoas que não são cadastradas.

Existe também no sistema um processo de compra que consiste em selecionar a melhor proposta, das inseridas pelos fornecedores, para gerar o pedido. Posteriormente envia um e-mail com o pedido criado para o representante selecionado.

Quando os produtos do pedido solicitado chegam, o usuário lança no sistema esses produtos para atualização do estoque e, posteriormente, gera uma fatura contendo os itens deste pedido e o representante responsável.

Além disso, o sistema permite ao usuário fazer uma listagem dos produtos que estão abaixo do mínimo. Outra funcionalidade é de listar os produtos indicados para se fazer uma promoção.

O restante do sistema consiste em cadastros, sendo eles, cadastro de produtos, usuários, fornecedores, representantes e clientes.

3.2.2 Erro Gerado Devido Uma Alteração

Com a utilização do sistema proposto foi criado um exemplo para amostragem de erros ocasionados a partir de uma alteração solicitada por um usuário fictício do sistema. Com isso esta subseção tem por finalidade representar que realmente podem

ser gerados erros em outras partes do sistema devido a uma alteração realizada, e que a aplicação da abordagem proposta pode auxiliar muito na solução desse problema.

Imaginando-se que tal usuário faz uma solicitação de alteração na tela de cadastro de produtos do sistema, onde essa alteração tem por objetivo permitir que se efetue o cadastramento dos produtos sem informar um valor no campo referente ao estoque máximo. O motivo de tal alteração, segundo o cliente, é que sua empresa começou a trabalhar com produtos novos, os quais ele não tem o conhecimento de quanto será a saída e precisa de dois meses para analisar e definir a quantidade máxima para o estoque desses produtos.

Atualmente o sistema não permite o cadastramento de qualquer produto sem que se informe seu estoque máximo, pois esse campo foi definido como obrigatório, sendo assim, quando se tenta efetuar o cadastro do produto sem informar o valor do estoque máximo o sistema retorna a mensagem ilustrada na figura 6.

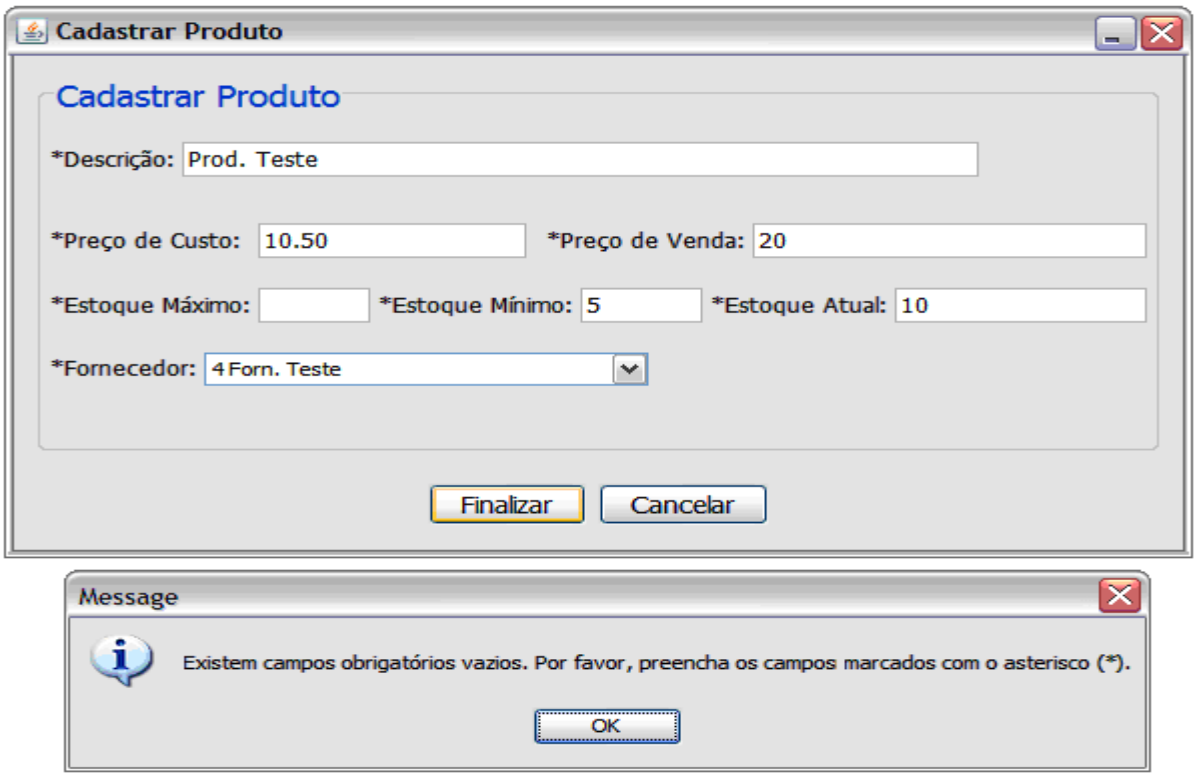

Figura 5 - Mensagem de obrigatoriedade retornada pelo sistema

A empresa entendendo a necessidade do cliente, repassou a alteração solicitada para sua equipe de desenvolvimento que prontamente efetuou a alteração

solicitada. Após ter sido feita a alteração, o sistema passa a permitir que se faça o cadastro dos produtos sem informar o valor do campo estoque máximo, o que resulta em uma mensagem positiva do sistema, como mostra a figura 7.

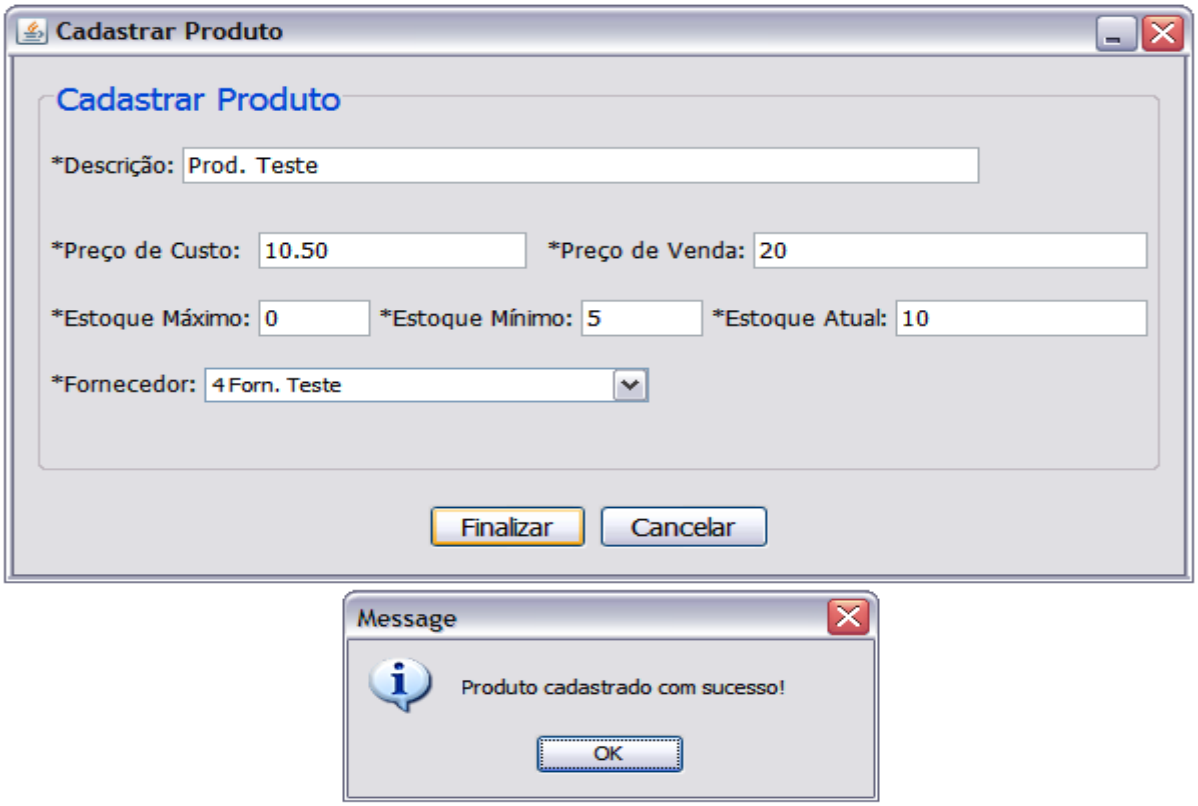

Figura 6 - Mensagem de sucesso na operação

Agora o sistema foi alterado e trabalha de acordo com que o usuário solicitou, mas existe um porém, uma outra funcionalidade do sistema é listar os produtos que são indicados para se criar uma promoção e um dos valores utilizados para o cálculo desta listagem é exatamente o valor de estoque máximo para o produto, informado no cadastro do produto.

O cálculo para a listagem desses produtos consiste em fazer a verificação em todos os produtos, um por vez, para constatar se o estoque atual do produto verificado ultrapassou o estoque máximo do mesmo, indicando assim se determinado produto é propício a ser designado para uma promoção ou não.

Como foi visto anteriormente um produto foi cadastrado sem valor no estoque máximo, onde o sistema automaticamente irá jogar zero no valor desse campo. O

resultado desse cadastramento sem valor no estoque máximo do produto vai ser que o produto cadastrado, erroneamente aparecerá na listagem dos produtos indicados a promoção, como apresentado na figura 8.

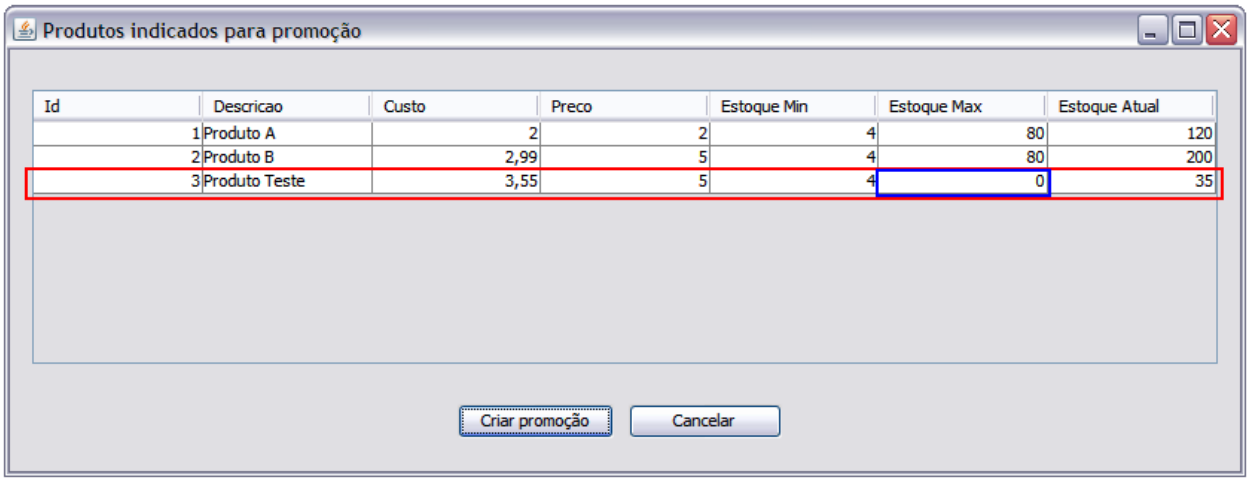

Figura 7 - Produto erroneamente listado juntamente com os indicados a promoção

Perceba que o produto cadastrado consta na listagem juntamente com os produtos que realmente estão propícios a uma promoção, pois de fato ele apresenta o seu estoque atual em maior número que seu estoque máximo, devido a ultima alteração realizada.

Se esse sistema de automação de mercados estivesse fazendo uso da abordagem proposta este erro gerado não seria problema. O problema ainda seria gerado pela alteração efetuada, mas no momento da execução da suíte de testes, a qual faz uso de todos os testes já criados que estão armazenados, o teste criado para unidade responsável pela seleção desses produtos que devem ser indicados à promoção, barraria o produto em questão pois esse produto seria identificado como pertencente a uma classe inválida como ilustrado na figura 9.

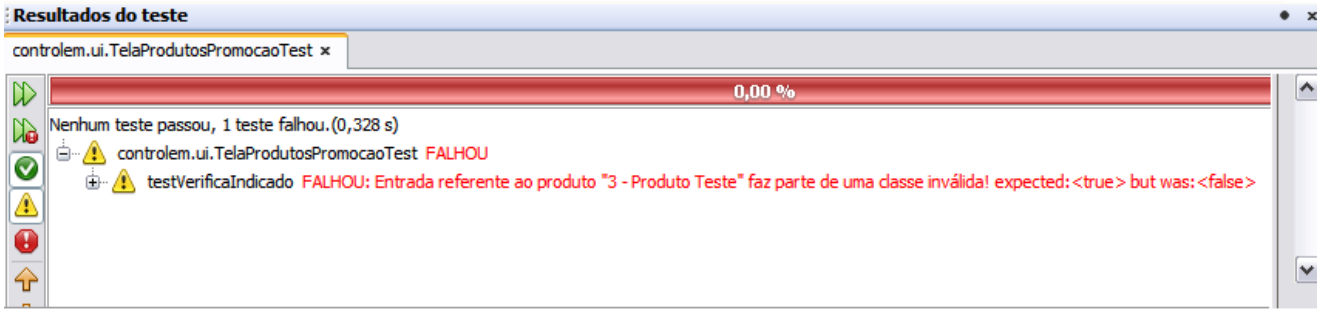

Figura 8 – Inconsistência, no produto, barrada pelo teste criado

3.2.3 Resultados e Discussões

Contudo, a abordagem se mostra muito útil para as empresas de desenvolvimento de software que optam por melhorar a qualidade do seu trabalho, deixando seu software mais confiável, consistente, e muito menos suscetível a erros.

Os benefícios citados podem ser alcançados com a correta aplicação de cada um dos passos da abordagem, que submeterão o sistema dessas empresas a testes periódicos, os testes de regressão. Isso se torna possível através da utilização dos testes criados para as unidades, que são armazenados em um repositório em vez de serem criados novos testes manualmente toda vez que ocorre um erro no sistema, o que seria muito mais trabalhoso, por conseguinte, mais demorado.

Mais um ponto importante é que esses testes também revelam erros e inconsistências ocultas, como mostrado na subseção 3.2.2. Com isso, evita-se que sejam liberadas versões falhas aos clientes e, dependendo do caso, causando transtornos e perca de tempo para solucionar o problema, uma vez que a origem exata do erro não é conhecida.

Outra consideração a ser feita é a prática do controle de versão apresentada no terceiro passo, que evita que os testes das unidades fiquem desatualizados pois sempre que a unidade é alterada o responsável pelo desenvolvimento dos testes é informado para que atualize esses testes o mais rápido possível, diminuindo assim ainda mais a chance de erros ocorrerem ou passarem despercebidos.

4 CONCLUSÃO

Mediante pesquisa exploratória e aspectos levantados sobre testes de software, em especial testes de regressão, apresentados ao longo deste trabalho, foi possível visualizar os benefícios que podem ser gerados com a adoção e implantação de tais testes na rotina de desenvolvimento das empresas de software.

O teste, como foi apresentado por autores renomados da área, é uma etapa de validação do sistema e detecção de erros visando aumentar a qualidade do mesmo. E isso foi constatado no transcorrer do trabalho, tanto na boa fundamentação de idéias trabalhadas quanto na apresentação da abordagem.

Na explicação da abordagem é possível verificar os benefícios trazidos por sua utilização, por apresentar testes bem fundamentados e garantidos, que tem um padrão de criação baseado em um dos critérios mais conhecidos entre os testadores, o que torna o trabalho muito mais profissional. Também por armazenar esses testes em um repositório o que permite e facilita sua reutilização e organização. Esses benefícios também se devem ao constante monitoramento desses testes pelo programa proposto, o que permite que eles estejam sempre atualizados, e finalmente pela re-execução dos testes a cada atualização efetuada.

Todos os processos da abordagem mostram o bom funcionamento da mesma, sua eficácia e como ela pode colaborar deixando mais prático e ágil o serviço dos testadores e desenvolvedores, poupando tempo e diminuindo ocorrências de erros, o que se transforma em qualidade do sistema e satisfação do cliente.

REFERÊNCIAS

ARISTIDES, Vicente de Paula Neto. *Criando Testes com JUnit*. Centro de Informática – Universidade Federal de Pernambuco. Disponível em: <http://javafree.uol.com.br/dependencias/tutoriais/testes_junit.pdf>. Acesso em: 27 julho 2012.

BRUNELI, Marcos V. de Queiroz. *A utilização de uma metodologia de teste no processo de melhoria da qualidade de software*. Disponível em: <http://www.bibliotecadigital.unicamp.br/document/?code=vtls000423676&fd=y>. Acesso em: 29 julho 2012.

BRUNO, Elisângela Andrade; SILVA, Paulo César Barreto da; ALVES, Thiago Salhab. *Testes Funcionais de Software*. Disponível em: <http://www.devmedia.com.br/websys.4/webreader.asp?cat=48&revista=esmagazine_45#a-4328>. Acesso em: 22 julho 2012.

CARDILLI, Danilo. *Uma visão técnica de teste de caixa branca* . Disponível em: < http://www.devmedia.com.br/uma-visao-da-tecnica-de-teste-de-caixa-branca/15610 >. Acesso em: 29 julho 2012.

MALDONADO, J. C.; et al.; Introdução ao Teste de Software, Instituto de Matemática e de Computação, v. 2004 – 1, n. 65, 2004.

Oliveira, R. B. Framework *functest:* Aplicando padrões de software na automação de testes funcionais. Dissertação de Mestrado, Universidade de Fortaleza, Fortaleza, 2007.

DIAS, Adriane Pedroso; DALCIN, Sabrina Borba; D'ORNELLAS, Marcos Cordeiro. *Aplicando testes em XP com o framework JUnit*. UFSM - Universidade Federal de Santa Maria - PPGEP - Programa de Pós-Graduação em Engenharia de Produção. Disponível em: < http://www.dcc.ufla.br/infocomp/artigos/v5.3/art09.pdf>. Acessado em: 25 julho de 2012.

MYERS, Glenford J.. *The Art of Software Testing*. Disponível em:

<http://noqualityinside.com.ar/nqi/nqifiles/The%20Art%20of%20Software%20Testing%20-%20Second%20Edition.pdf>. Acesso em: 10 julho 2012.

MIRANDA JUNIOR, Prof. Pasteur O. de. *Testes de Caixa Branca e Caixa Preta*. PUC Minas. Disponível em: <http://www.pasteurjr.blogspot.com.br/>. Acesso em: 15 julho 2012.

PFLEEGER, Shari Lawrence. *Engenharia de Software*: Teoria e Prática. 2ª ed. São Paulo: Pearson Education, 2004.

PRESSMAN, Roger S. *Engenharia de Software*. 6ª Ed. McGraw-Hill Interamericana do Brasil.

PONTES, Melissa Barbosa. *Atividades Aliadas dos Testes de Software*. Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-20-atividades-aliadas-dos-testes-de-software/15463>. Acesso em: 18 julho 2012.