

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE

Samuel Valerio Ozorio Dutra

**ANÁLISE DO USO DE TESTES AUTOMATIZADOS NO DESENVOLVIMENTO DE
SOFTWARE**

NATAL
2008

SAMUEL VALERIO OZORIO DUTRA

ANÁLISE DO USO DE TESTES AUTOMATIZADOS NO DESENVOLVIMENTO DE
SOFTWARE

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do RN, como requisito obrigatório à obtenção do título de Especialista em Desenvolvimento de Sistemas Corporativos.

Orientador:

David Ricardo do Vale Pereira, M.Sc.

Natal
2008

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Dutra, Samuel Valerio Ozorio

Análise do uso de testes automatizados no desenvolvimento no desenvolvimento de software / Samuel Valerio Ozorio Dutra. – Natal: FARN, 2008.

72f.

Orientador: David Ricardo do Vale Pereira (Msc.)

Monografia (Especialização em Desenvolvimento de Sistemas Corporativos) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Software– Monografia. 2. Desenvolvimento de Software – Monografia.
I. Título.

RN/FARN/BC

CDU 004.4



SAMUEL VALERIO OZORIO DUTRA

ANÁLISE DO USO DE TESTES AUTOMATIZADOS NO DESENVOLVIMENTO DE
SOFTWARE

Monografia apresentada ao Curso de Especialização em Desenvolvimento de
Sistemas Corporativos da Faculdade para o Desenvolvimento do RN, e aprovada
em ____ de ____ de ____ pela banca examinadora composta pelos
seguintes membros:

David Ricardo do Vale Pereira

David Ricardo do Vale Pereira, M. Sc.
Orientador

Membro Examinador

Membro Examinador

RESUMO

Este trabalho tem como objetivo a realizar uma análise sobre o uso de testes automatizados no desenvolvimento de software, com ênfase na técnica de desenvolvimento guiado por testes (Test Driven Development - TDD). Serão apresentados alguns os problemas resultantes de falhas em software e as formas de minimizá-las; técnicas existentes para testes: caixa branca e caixa preta, seus fundamentos, objetivos e um comparativo entre as mesmas. Apresenta-se também algumas estratégias de testes: unidade, integração e aceitação, são exibidos seus conceitos e exemplifica-se cada uma delas. Como forma de validação, um estudo de caso é apresentado no qual o TDD foi utilizado para a criação de um componente para emissão de guias de arrecadação, obtendo-se resultados positivos como: testes executáveis que integram-se ao software efetivamente agregando valor ao produto; confiabilidade do desenvolvedor durante o desenvolvimento e as refatorações; produção de um código extremamente limpo, coeso e fracamente acoplado.

Palavras Chave: Testes de Software. Desenvolvimento Guiado por Testes. Componente.

ABSTRACT

The purpose of this work is to analyse the use of automated tests in the software development, with emphasis on Test Driven Development technique (TDD). The causes of possible software failures will be exposed and how to minimize them; tests techniques: white box and black box, its concepts, goals and a comparative about both. Some software tests strategies will be exposed: unit, integration and acceptance, its concepts will be described and examples exposed. As a validation, a case study will be demonstrated with TDD to create a component for issuance of bills, yielding positive results as executable tests that integrates software actually adding value to products; high reliability of the developer during the development and refactorings; production of a code very clean, cohesive and weakly coupled.

Key Words: Software Tests. Test Driven Development. Component.

Lista de figuras

Figura 1 - Código fonte da classe Calculadora.....	19
Figura 2 - Código fonte da classe Calculadora com trechos de impressão.....	19
Figura 3 - Classe Calculadora com ponto de parada.....	20
Figura 4 - Exibição das variáveis no ponto de parada da classe Calculadora.....	21
Figura 5 - Classe Calculadora com método main().....	22
Figura 6 - Classe TestadorDaCalculadora.....	23
Figura 7 - Introdução de falha na Calculadora.....	23
Figura 8 - Mensagem de erro da Calculadora.....	24
Figura 9 - Teste da adição da Calculadora com JUnit.	25
Figura 10 - Resultado do primeiro teste com JUnit da Calculadora.....	25
Figura 11 - Detalhes da falha introduzida na Calculadora.....	26
Figura 12 - Resultado do teste da Calculadora após correção.....	26
Figura 13 - Criação do teste do GerenciadorDeLivros que retorna uma lista vazia.....	28
Figura 14 - Criação da classe Livro sem corpo.....	29
Figura 15 - Criação da classe GerenciadorDeLivros sem corpo.....	29
Figura 16 - Criação do método obterListaAlfabeticaDeLivros() no GerenciadorDeLivros.....	29
Figura 17 - Remoção dos erros de compilação do teste que retorna a lista vazia do GerenciadorDeLivros.....	29
Figura 18 - Teste da lista vazia do GerenciadorDeLivros sem erros de compilação.....	30
Figura 19 - Método obterListaAlfabeticaDeLivros() modificado para retornar uma nova lista.....	30
Figura 20 - Nova execução do teste após alterar obterListaAlfabeticaDeLivros()..	31
Figura 21 - Criação do teste para um livro.....	31
Figura 22 - Criação de construtor e método no Livro.....	32
Figura 23 - Criação do método adiciona() em GerenciadorDeLivros.....	32
Figura 24 - Remoção dos erros de compilação do teste de um livro.....	32
Figura 25 - Execução do teste após remoção dos erros de compilação.....	33
Figura 26 - Criação da lista de livros dentro do GerenciadorDeLivros.....	33
Figura 27 - Nova execução após adição da lista de livros no GerenciadorDeLivros.....	34
Figura 28 - Criação do teste de múltiplos livros.....	35
Figura 29 - Resultado após criação do teste de múltiplos livros.....	35
Figura 30 - Adição do método sort() na lista de livros.....	35
Figura 31 - Adição dos testes para comparação de Livros.....	36
Figura 32 - Implementação de Comparable pelo Livro.....	37
Figura 33 - Execução dos testes após o Livro se tornar comparável.....	38
Figura 34 - Implementação correta do compareTo() do Livro.....	38
Figura 35 - Execução após implementação correta do compareTo() do Livro.....	38
Figura 36 - Execução final do TestadorDoGerenciadorDeLivros.....	39
Figura 37 - Criação da tabela Usuario.....	40
Figura 38 - Criação do entrada.xml.....	40
Figura 39 - Criação do TestadorDoDbUnit.....	41
Figura 40 - Inclusão do usuário Fernando da Silva.....	42
Figura 41 - Seleção de todos os usuários.....	42
Figura 42 - Criação do teste do DbUnit.....	42
Figura 43 - Execução do teste do DbUnit.....	42

Figura 44 - Confirmação do comportamento do DbUnit.....	43
Figura 45 - Formulário padrão da guia de arrecadação.....	46
Figura 46 - Representação numérica e gráfica do código de barras.....	47
Figura 47 - Criação do TestadorDoCodigoDeBarras.....	47
Figura 48 - Criação do CodigoDeBarras.....	47
Figura 49 - Execução do teste do número de posições do CodigoDeBarras.....	48
Figura 50 - Método getRepresentacaoNumerica() retornando String vazia.....	48
Figura 51 - Execução do teste após representação numérica retorna String vazia.....	49
Figura 52 - Adição do teste em que a representação numérica deve conter apenas números.....	49
Figura 53 - Falha no teste da representação numérica conter apenas números.....	49
Figura 54 - Modificação para que a representação numérica contenha apenas números.....	49
Figura 55 - Execução dos testes com sucesso após modificação para que a representação numérica contenha apenas números.....	50
Figura 56 - Campo identificação do produto.....	50
Figura 57 - Criação do testador do produto.....	51
Figura 58 - Criação do enum de produto.....	51
Figura 59 - Falha na execução do primeiro teste do produto.....	51
Figura 60 - Mudança no produto para que o teste não falhe mais.....	51
Figura 61 - Execução do teste do produto sem falhas.....	52
Figura 62 - Novos testes no testador do código de barras após criação do produto.....	53
Figura 63 - Modificação no código de barras para que ele passe no teste do produto.	53
Figura 64 - Especificação do segmento.....	54
Figura 65 - Primeira parte dos testes do segmento.....	54
Figura 66 - Segunda parte dos testes do segmento.....	55
Figura 67 - Segmento após modificações para atender os testes.....	55
Figura 68 - Criação do inicializa() no TestadorDoCodigoDeBarras.....	56
Figura 69 - Adição do teste do segmento.....	56
Figura 70 - Modificação no CodigoDeBarras para passar no teste do segmento.....	56
Figura 71 - Especificação do tipo de valor.....	57
Figura 72 - Testes do tipo de valor.....	58
Figura 73 - Versão final do enum TipoDeValor.....	58
Figura 74 - Mudança no inicializa() para suporte tipo de valor.....	59
Figura 75 - Criação do teste para o tipo de valor.....	59
Figura 76 - Mudanças em CodigoDeBarras para passar no teste de tipo de valor..	59
Figura 77 - Adição dos testes do valor.....	60
Figura 78 - Após modificação para passar no teste referente ao valor do CodigoDeBarras.....	61
Figura 79 - Especificação da empresa ou órgão.....	61
Figura 80 - Primeira parte dos testes da identificação da empresa ou órgão.....	62
Figura 81 - Segunda parte dos testes da identificação da empresa ou órgão.....	62
Figura 82 - Modificação no CodigoDeBarras para passar nos testes da identificação da empresa ou órgão.....	63
Figura 83 - Testes do campo livre.....	64
Figura 84 - Implementação resultante dos testes do campo livre.....	64
Figura 85 - Teste para gerar o dígito verificador pelo módulo 10.....	65
Figura 86 - Implementação para passar no testes do dígito verificador pelo módulo 10.....	66

Figura 87 - Teste do dígito verificador pelo módulo 11.....	66
Figura 88 - Implementação para o dígito verificador módulo 11.....	67
Figura 89 - Teste da representação numérica completa.....	67
Figura 90 - Código de barras de uma conta da CAERN.....	68
Figura 91 - Teste para o simular a mesma conta da CAERN.....	68
Figura 92 - Criação da classe GuiaDeArrecadacao apenas para agrupar dados.....	69

SUMÁRIO

1 INTRODUÇÃO.....	7
2 FUNDAMENTOS E TÉCNICAS DE TESTE.....	10
2.1 FUNDAMENTOS.....	10
2.2 OBJETIVOS.....	13
2.3 CASOS DE TESTE.....	13
2.4 TÉCNICAS DE TESTE.....	14
2.4.1 Teste de caixa branca.....	14
2.4.2 Teste de caixa preta.....	15
2.4.3 Comparativo entre as técnicas de testes.....	16
3 ESTRATÉGIAS DE TESTE.....	18
3.1 TESTES DE UNIDADE.....	18
3.1.1 Depuração através de impressão na saída padrão.....	19
3.1.2 Execução da aplicação em modo de depuração.....	20
3.1.3 Escrita de código a parte exclusivamente para testes.....	21
3.1.4 Test-Driven Development (TDD).....	26
3.1.4.1 Exemplo.....	28
3.2 TESTES DE INTEGRAÇÃO.....	39
3.2.1 Exemplo.....	39
3.5 TESTES DE ACEITAÇÃO.....	43
4 ESTUDO DE CASO COM TDD.....	45
5 CONCLUSÕES.....	70
REFERÊNCIAS.....	71

1 INTRODUÇÃO

Todo processo que conte com a intervenção humana está sujeito a falhas. Os erros pertencem ao cotidiano humano. Quanto mais complexo o processo e maior a dependência de humanos, eleva-se a vulnerabilidade a erros.

Os humanos buscam, constantemente, minimizar a possibilidade de ocorrência de erros em seus processos. Para gradativamente amplificar a qualidade e conseqüentemente a produtividade do procedimento em questão.

O desenvolvimento de software é complexo e altamente dependente de humanos. Portanto, está demasiadamente exposto à falhas. Isso requer um especial cuidado na criação dos mesmos, principalmente em sistemas de considerável relevância. Há vários exemplos verídicos de softwares que trouxeram grandes prejuízos devido a falhas em sua criação:

- Ariane 5: Em 1996, menos de um minuto após o seu lançamento, o foguete francês se autodestruíu. A comissão que analisou o acidente indicou um erro no software de controle como a origem da falha do lançamento. O problema ocorreu graças a um estado que o software atingiu e não era esperado o que levou o sistema a falhar por completo. (HINKEL, 1998).
- Therac 25: Entre junho de 1985 e janeiro de 1987, a máquina de radioterapia, controlada por computador, provocou seis acidentes envolvendo overdoses, que resultaram em mortes e graves injúrias a pacientes. Concluíram que a falha ocorreu devido a erros no desenvolvimento do software que controlava a máquina, como a falta de testes e documentação. (TURNER; LEVESON, 1993).

Uma das formas de reduzir essas possíveis falhas futuras é otimizar o processo de construção do software de maneira que o mesmo tenha mais qualidade. A definição de qualidade de software é frequentemente debatida e não há um consenso. Ela é estabelecida por Pressman (1995, p.724) como:

Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que

são esperadas de todo software profissionalmente desenvolvido.

Isso é problemático para os desenvolvedores pois eles devem manter tanto os requisitos do cliente (eficiência, confiabilidade) quanto os dos desenvolvedores (alta coesão, baixo acoplamento, reusabilidade, legibilidade). Além disso, as especificações são geralmente imperfeitas, ambíguas, incompletas e até mesmo inconsistentes.

A falha em um software, de acordo com Patton (2002), pode ser encontrada nas situações onde o software:

- Não faz algo que a especificação diz que ele deveria fazer;
- Faz algo que a especificação diz que ele não deveria fazer;
- Faz algo que a especificação não menciona;
- Não faz algo que a especificação não menciona porém deveria fazê-lo mesmo assim;
- É difícil de entender;
- É complicado de usar;
- É lento.

Kaner; Falk; Nguyen (1993) ressaltam que uma definição comumente utilizada para um erro de software é quando ele não satisfaz a sua especificação. Ele recomenda que tal definição não seja usada, pois isso só ocorrerá quando a especificação existir e estiver correta. Um programa que segue uma especificação incorreta será incorreto, e não correto.

Mesmo após uma implementação adequada, gradativamente o software perde a qualidade com o uso e tempo, devido a vários fatores:

- Um programa executando continuamente por um longo período de tempo deve funcionar cada vez mais lento e pode até parar: Graças a vazamento¹ ou fragmentação² da memória. Pode requerer o reinício do servidor.
- A performance cai com o crescente número de usuários e volume de

1 Ocorre quando blocos de memória possuem dados que poderiam ser liberados por serem inacessíveis e que, por descuido na construção da aplicação, ainda são referenciados no código mesmo sem estarem sendo usados, não podendo ser liberados.

2 É um fenômeno no qual o espaço é usado de forma ineficiente causando o desperdício do mesmo.

dados: Implica em futuros upgrades de software ou hardware.

- A manutenibilidade diminui com o tempo: Necessidade de manutenção preventiva (migração para novas tecnologias por exemplo).
- O software se torna obsoleto rapidamente: Em virtude da célere evolução das tecnologias. Isso requer a freqüente inovação e evolução do software.

Durante o desenvolvimento de um software, há uma série de atividades que buscam agregar qualidade ao software, visando minimizar futuros problemas. Conforme ressalta Deutsch (1979, p.329):

O desenvolvimento de sistemas de software envolve uma série de atividades de produção em que as oportunidades de injeção de falhas humanas são enormes... Por causa da incapacidade que os seres humanos têm de executar e comunicar com perfeição, o desenvolvimento de software é acompanhado por atividades de garantia de qualidade.

Dentre as atividades existentes que prezam pela qualidade do software, podemos relacionar:

- Testes: técnica dinâmica que consiste em provocar e observar comportamentos no software para descobrir defeitos;
- Revisões: técnica estática que visa a análise do sistema através do código-fonte e documentação para encontrar problemas;
- Medições: uso de ferramentas que mensuram componentes selecionados do software para um melhor controle e predição sobre o mesmo.

Este trabalho tem como objetivo analisar o uso de testes automatizados no desenvolvimento de software. Para isso, iremos introduzir os fundamentos e técnicas de testes no capítulo 2, relacionar as estratégias de testes existentes no capítulos 3 com foco na demonstração de uma delas: *Test-Driven Development* (TDD). Em seguida essa estratégia será aplicada em um estudo de caso no capítulo 5. Por fim, as conclusões serão feitas no capítulo 6.

2 FUNDAMENTOS E TÉCNICAS DE TESTE

Vimos que os testes de software podem representar uma parcela considerável do processo de qualidade de software. Pressman (1995, p.786) afirma que eles representam a última revisão de especificação, projeto e codificação. É crescente o esforço empreendido nos testes haja vista sua importância. Principalmente em sistemas críticos, nos quais falhas podem resultar em danos graves e irreparáveis.

O presente capítulo procura descrever os fundamentos, objetivos e principais técnicas para o projeto dos testes de software.

2.1 FUNDAMENTOS

Durante o desenvolvimento de um software, parte-se de um conceito abstrato para uma implementação tangível. Em seguida, deve-se testar essa implementação. Para fazê-lo, segundo Pressman (1995, p.787), o desenvolvedor cria uma série de casos de teste que têm a intenção de "demolir" o software recém construído.

A elaboração e execução exaustiva dos testes visam minimizar os custos provenientes da correção de falhas no software pois o custo para consertar um erro é exponencialmente crescente. De acordo com Patton (2002), se o custo médio para arrumar um erro na especificação é de dez centavos de dólar, na implantação este custo subiria para cem dólares. Ainda segundo Patton (2002), conforme mostra o gráfico 1, o custo cresce em média dez vezes a cada etapa do desenvolvimento:

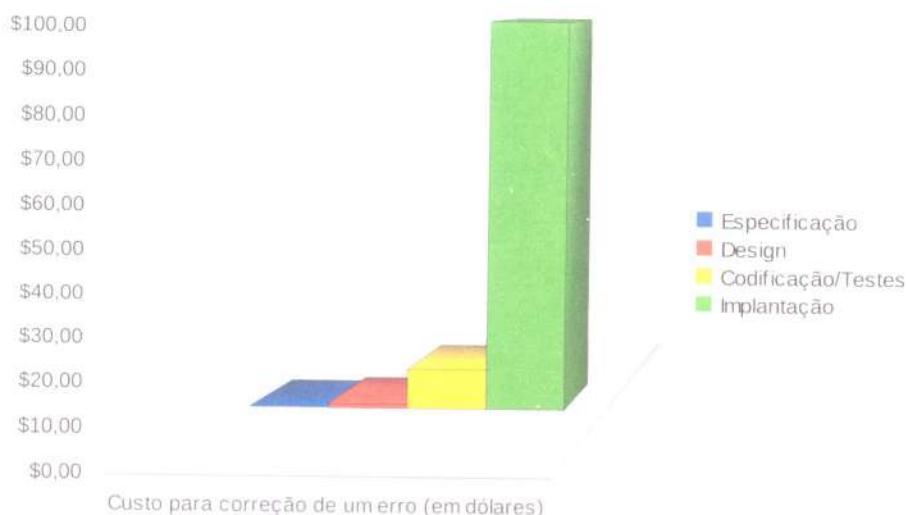


Gráfico 1 - Custo para correção de um erro (em dólares).
Fonte: Patton (2002).

Os custos de qualidade são únicos quando planejados, isto é, na situação em que foram previstos e estão associados aos planos, execução e revisão dos testes. Já os custos que não foram planejados são divididos da seguinte forma:

- Custos antes da *release*³: Custo de isolar, reportar e corrigir. É considerado relativamente pequeno. Refere-se ao lado esquerdo do gráfico 1.
- Custos após a liberação da *release*: Se os erros forem expostos ao cliente, serão bem mais custosos pois culminarão em suporte técnico e possíveis correções, retestes e liberação de uma nova *release*. Nos piores casos podem ocorrer danos irreparáveis ao consumidor. É representado pela etapa de implantação da figura do gráfico 1.

Crosby (1979) afirma que os custos planejados somados aos custos não planejados antes da *release* são bem menores do que os custos em não planejados após a liberação da *release*.

Segundo Patton (2002), a maior fonte dos *bugs*⁴ de software está na especificação dos requisitos, conforme demonstra o gráfico 2:

³ É a distribuição, pública ou privada, de uma versão inicial ou atualizada de um software.

⁴ É um erro em um programa de computador que resulta em um comportamento inesperado do mesmo.

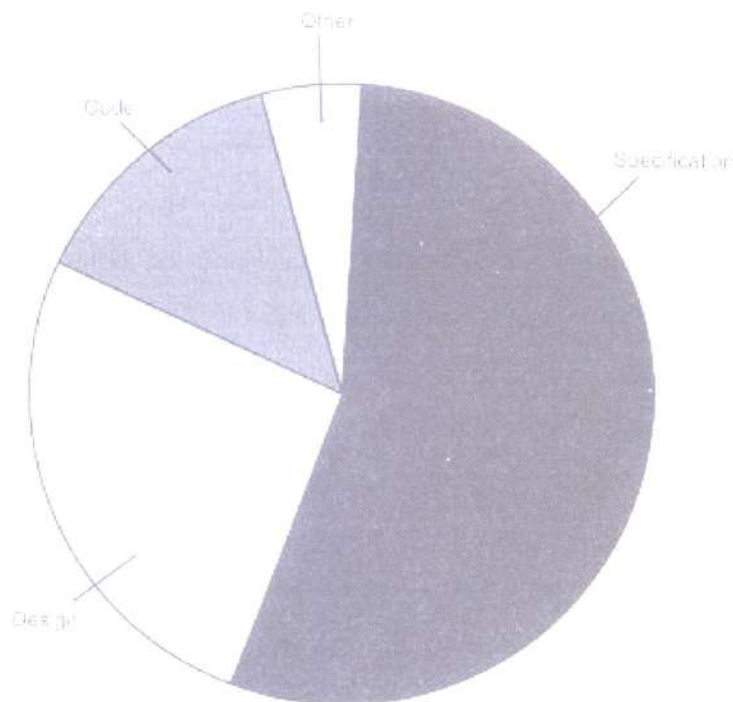


Gráfico 2 - Distribuição em etapas de falhas que resultam em bugs.

Fonte: Patton (2002).

Comparando-se os gráficos 1 e 2, percebe-se que os bugs surgem desde a especificação e são extremamente custosos após a implantação. Isto é, há um bom espaço de tempo no processo de desenvolvimento para descobri-los e corrigi-los.

O desenvolvedor ao testar um software, de certa forma, está assumindo que pode ter errado em sua construção, o que é perfeitamente plausível, considerando a complexidade envolvida no desenvolvimento de um software. Conforme destaca Beizer (1990, p.1):

Há um mito segundo o qual, se fôssemos realmente bons para programar, não haveria 'bugs' a ser procurados. Se pudéssemos realmente nos concentrar, se todos usassem programação estruturada, projeto 'top-down', tabelas de decisão, se tivéssemos as balas de prata certas, então não haveria 'bugs'.

Massol; Husted (2003, p.4) afirmam que não há desenvolvimento de software sem a presença de testes. Sempre há a codificação seguida dos testes, ou os testes seguidos da codificação. Ao testar-se, é comum a descoberta de problemas, principalmente na primeira execução. Então, codifica-se e testa-se

novamente, até que o software apresente o comportamento desejado.

2.2 OBJETIVOS

Diferentemente do que muitos pensam, um teste bem-sucedido é aquele que encontra erros. Baseando-se em Patton (2002), podemos relacionar as regras norteiam os testes:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro e nunca prova a ausência de defeitos;
- Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto;
- Um teste bem-sucedido é aquele que revela um erro ainda não descoberto. Para que seja corrigido o mais rápido possível, e garanta que ele está solucionado.

Segundo Pressman (1995, p.789), como um benefício secundário a atividade de teste demonstra que os casos testados estão trabalhando de acordo com as especificações. Isso proporciona uma boa indicação da confiabilidade do software e alguma indicação da qualidade do dele como um todo.

2.3 CASOS DE TESTE

Seguindo as regras para atingir os objetivos dos testes, os casos de testes devem ser projetados com a intenção de descobrir novos erros no software. É uma atividade tão desafiadora quanto o próprio projeto inicial do produto.

Pressman (1995, p.791) define duas maneiras de se testar um produto:

- Conhecendo-se a função específica que um produto projetado deve executar, testes podem ser realizados para demonstrar que cada função é operacional;
- Conhecendo-se o funcionamento interno de um produto, testes podem ser realizados para garantir que “todas as engrenagens se

encaixam", ou seja, que a operação interna do produto tem um desempenho de acordo com as especificações e que os componentes internos foram adequadamente postos à prova.

A primeira abordagem é chamada teste de caixa preta e a segunda, teste de caixa branca.

2.4 TÉCNICAS DE TESTE

2.4.1 Teste de caixa branca

Também conhecida como teste estrutural (Beizer, 1995), é uma técnica de projeto de casos de teste na qual leva-se em consideração o mecanismo interno de um sistema ou componente (IEEE, 1990). O conhecimento sobre a estrutura do software é utilizado para identificar casos de teste adicionais.

A conotação de caixa branca indica a existência de uma visibilidade total sobre a estrutura interna do programa, especialmente, a estrutura lógica do código. Portanto, é dependente da linguagem de programação.

Conforme Pressman (1995, p.793), o desenvolvedor pode derivar casos de teste que (1) garantam que todos os caminhos independentes dentro de um módulo tenham sido exercitados pelo menos uma vez; (2) exercitem todas as decisões lógicas para valores falsos ou verdadeiros; (3) executem todos os laços em suas fronteiras e dentro de seus limites operacionais; e (4) exercitem as estruturas de dados internas para garantir a sua validade.

Os motivos que nos levam a usar o teste de caixa branca residem na natureza dos defeitos de software, relacionados por Jones (1981):

- *Erros lógicos e pressuposições incorretas são inversamente proporcionais à probabilidade de que um caminho de programa seja executado.* É comum o surgimento de erros quando projetamos ou implementamos algo que esteja fora da função principal. O processamento cotidiano tende a ser bem entendido (e bem esmiuçado), enquanto o processamento de um

"caso especial" tende a "cair por terra";

- *Muitas vezes acreditamos que um caminho lógico não tem a probabilidade de ser executado quando, de fato, ele pode ser executado regularmente.* Nossas pressuposições sobre o fluxo de controle muitas vezes não condizem com a realidade.
- *Erros tipográficos são aleatórios.* Estamos sujeitos aos erros de digitação. Eles podem ser descobertos por mecanismos de verificação de sintaxe. Entretanto, um erro tipográfico que afeta um caminho lógico por exemplo, só será descoberto durante os testes.

Os testes de caixa preta, por mais minuciosos que sejam, podem não revelar os tipos de erros citados acima. Os testes de caixa branca são mais adequados para descobri-los.

2.4.2 Teste de caixa preta

Também conhecido como teste comportamental, essa técnica não requer o conhecimento interno da estrutura do programa pelo testador. Por exemplo, em um teste de caixa preta o testador necessita apenas conhecer as entradas e as saídas esperadas. Não importando como essas saídas são geradas.

Essa técnica concentra-se nos requisitos funcionais do software. Ela permite que o testador derive conjuntos de condições de entrada que exercitem completamente todos os requisitos funcionais de um programa. O testador não examina o código, ele conhece apenas as especificações do programa.

O teste de caixa preta não é uma alternativa ao de caixa branca. Pelo contrário, trata-se de uma abordagem complementar que tem maior probabilidade de descobrir erros de origens diferentes daqueles dos métodos de caixa branca.

Segundo Pressman (1995,p.816) os testes de caixa preta procuram descobrir erros nas seguintes categorias: (1) funções incorretas ou ausentes; (2) erros de interface; (3) erros nas estruturas de dados ou no acesso a banco de dados externos; (4) erros de desempenho; e (5) erros de inicialização e término da aplicação.

Algumas vantagens desta técnica de teste são relacionadas:

- O teste é imparcial pois o implementador e o testador são independentes;
- O testador não precisa conhecer linguagens de programação;
- O teste é feito do ponto de vista do usuário, não do implementador;
- Os casos de teste podem ser projetados tão cedo quanto as especificações estejam completas.

As desvantagens também podem ser destacadas:

- O teste pode ser redundante se o implementador já rodou o caso de teste;
- Os testes são difíceis de projetar;
- Testar todas as possíveis entradas é irreal pois levaria muito tempo. Isso leva a vários caminhos do programa não serem testados.

Percebe-se claramente que os testes de caixa preta complementam os de caixa branca. Ambos são necessários para obter um software com menos bugs.

2.4.3 Comparativo entre as técnicas de testes

O desenvolvedor possui mais controle sobre o teste de caixa branca que o testador sobre o de caixa preta. Se você executa um caso de teste de caixa branca, você sabe exatamente quais linhas de código você executou. Portanto, caso o teste falhe, o você saberá exatamente quais linhas de código deve analisar baseado no caso de teste que falhou. Devido a este controle, remover defeitos nos testes de caixa branca é mais econômico que nas fases posteriores no ciclo de desenvolvimento. (WILLIAMS, 2006)

Ainda segundo Williams (2006), os erros encontrados através dos testes de caixa preta, geralmente exigem mais esforço para serem corrigidos. Pois, neste caso, um caso de testes não revela um problema (e um local aproximado de onde o problema deve ser corrigido). Diferentemente do de caixa branca, um caso de testes de caixa preta falho revela o sintoma de um problema. Pode ser difícil, consumir tempo e tomar um tempo imprevisível encontrar a principal causa do sintoma (o

problema que causou a falha) para que o desenvolvedor modifique o código. Por conseguinte, você deve testar o quanto for possível através dos testes de caixa branca para só então projetar e executar os testes de caixa preta. O quadro 1, elaborado por Williams (2006), compara as duas técnicas de teste:

Características	Caixa branca	Caixa preta
Visibilidade do testador	Possui visibilidade do código e escreve casos de teste baseados no mesmo	Não enxerga o código e escreve casos de teste baseado nas entradas e saídas documentadas nas especificações e/ou requisitos
O que um caso de teste falho revela	Um problema	O sintoma de um problema
É facilmente controlável	Sim, sabe-se exatamente quais as linhas de código envolvidas	Não, pode ser difícil encontrar a causa do problema

Quadro 1 - Comparativo entre técnicas de teste.

Fonte: Williams (2006).

Ambas as técnicas são importantes e objetivam encontrar diferentes tipos de falhas. Simples falhas são encontradas nos testes de caixa preta se os testes de caixa branca não foram feitos adequadamente. Deve-se procurar sempre remover a maior quantidade de defeitos possível usando os testes de caixa branca, momento em que a identificação das falhas é mais controlável.

3 ESTRATÉGIAS DE TESTE

Uma estratégia de teste integra técnicas de projeto de casos de teste em uma série bem-definida de passos que resultam na construção bem-sucedida do software. Qualquer estratégia de teste deve incorporar planejamento de teste, projeto de casos de teste, execução de testes e a resultante coleta e avaliação de dados. (PRESSMAN, 1995)

No presente capítulo serão apresentadas algumas das estratégias existentes, exemplificando-se cada uma delas.

3.1 TESTES DE UNIDADE

Os testes de unidade consistem em verificar as menores unidades de código possíveis, denominadas unidades de trabalho. Diferentemente dos testes de integração e aceitação, que visam analisar como os vários componentes do sistema interagem. De acordo com Massol; Husted (2003, p.6), em uma aplicação java a unidade geralmente é um método. O teste de unidade é um teste de caixa branca realizado pelo desenvolvedor que pode ser automatizado através da criação de pequenos programas.

Alguns desenvolvedores consideram a escrita de testes automatizados como uma parte essencial do processo de desenvolvimento: "Um componente não pode provar que funciona até que passe em uma série de testes" (MASSOL; HUSTED; 2003, p.4).

Os testes de caixa branca feitos pelo desenvolvedor durante a construção de uma aplicação seguem as seguintes abordagens:

- Depuração através de impressão na saída padrão;
- Execução da aplicação em modo de depuração;
- Escrita de código a parte exclusivamente para testes.

Cada uma das abordagens será apresentada a seguir. Para isso, considere o método *adiciona()* da classe *Calculadora* como a unidade a ser testada. Este exemplo foi criado com base no feito por Massol; Husted (2003, p.7) em *Junit in*

Action. A classe *Calculadora* é exibida na figura 1:

```
package br.farn.sisecorp.tccsv.dominio;

class Calculadora {
    int adiciona(int primeiraParcela, int segundaParcela) {
        int soma;
        soma = primeiraParcela + segundaParcela;
        return soma;
    }
}
```

Figura 1 - Código fonte da classe *Calculadora*.

Fonte: O próprio autor. (2008)

3.1.1 Depuração através de impressão na saída padrão

Pela extrema simplicidade, é a forma mais utilizada por desenvolvedores iniciantes. Consiste em adicionar trechos de código que imprimam na saída padrão os valores das variáveis, indicadores do que está ocorrendo e das posições no código.

Para o método *adiciona()* da classe *Calculadora*, adicionaríamos os trechos de impressão na saída padrão conforme as linhas 5,6,7,9,11,12 e 13 da figura 2:

```
1 package br.farn.sisecorp.tccsv.dominio;
2
3 class Calculadora {
4     int adiciona(int primeiraParcela, int segundaParcela) {
5         System.out.println("Entre na Calculadora.adiciona()");
6         System.out.println("primeiraParcela=" + primeiraParcela);
7         System.out.println("segundaParcela=" + segundaParcela);
8         int soma;
9         System.out.println("Adicionando as duas parcelas...");
10        soma = primeiraParcela + segundaParcela;
11        System.out.println("Duas parcelas adicionadas com sucesso.");
12        System.out.println("soma=" + soma);
13        System.out.println("Prestes a sair da Calculadora.adiciona()");
14        return soma;
15    }
16 }
```

Figura 2 - Código fonte da classe *Calculadora* com trechos de impressão.

Fonte: O próprio autor. (2008)

Em seguida, o desenvolvedor compilaria, executaria novamente a aplicação e verificaria o caminho percorrido, o processamento realizado e os valores

das variáveis. Localizado o ponto em que deseja adicionar um comportamento ou alterar o pré-existente, modificaria o código neste ponto, compilaria e executaria novamente até que o programa funcione conforme desejado. Então, todos os códigos dos trechos de impressão seriam removidos.

Nessa abordagem faz-se necessária uma cuidadosa análise e interpretação do desenvolvedor sobre os trechos impressos. Com a escrita de código temporário apenas para localizar o ponto em que deseja alterar o comportamento da aplicação, existe o risco do desenvolvedor esquecer de apagar algum trecho temporários e deixar código desnecessário que poderá poluir os *logs* da aplicação quando ela estiver em produção.

Outra desvantagem é o caráter temporário do teste e com isso a impossibilidade de reutilização do mesmo.

3.1.2 Execução da aplicação em modo de depuração

Outra possibilidade seria executar a aplicação em modo de depuração (*debug*) com auxílio de alguma ferramenta. Os ambientes integrados de desenvolvimento – *Integrated Development Environment* (IDE), utilizados pela maioria dos desenvolvedores, já possuem mecanismo de depuração integrado.

Com a execução da aplicação em modo de depuração, pontos de interrupção (*breakpoints*) são definidos. No exemplo da Calculadora, foi estabelecido um ponto de interrupção na linha 6 da classe *Calculadora* (figura 3):

```
package br.farn.sisocorp.tccsv.dominio;

class Calculadora {
    int adiciona(int primeiraParcela, int segundaParcela) {
        int soma;
        soma = primeiraParcela + segundaParcela;
        return soma;
    }
}
```

Figura 3 - Classe Calculadora com ponto de parada.
Fonte: O próprio autor. (2008)

A partir dos pontos de interrupção é possível analisar o valor das variáveis e o caminho percorrido pela lógica do programa instrução a instrução. A figura 4 apresenta o valor das variáveis no ponto de interrupção definido na classe

Calculadora.



Name	Value
this	Calculadora (id=16)
primeiraParcela	3
segundaParcela	2

Figura 4 - Exibição das variáveis no ponto de parada da classe Calculadora.

Fonte: O próprio autor. (2008)

Nesta abordagem não há escrita de código temporário como os trechos de impressão da estratégia anterior. Todavia, o teste continua dependendo de uma análise minuciosa do desenvolvedor. Outra desvantagem que persiste é o caráter temporário e a impossibilidade de reuso do teste por outros desenvolvedores.

3.1.3 Escrita de código a parte exclusivamente para testes

Objetivando minimizar ao máximo a dependência da análise humana, altamente vulnerável a erros, e a característica temporária do teste, possibilitando que o mesmo seja reutilizado futuramente, surgiu a idéia da escrita de código a parte exclusivamente para testes.

Em um primeiro momento, o desenvolvedor poderia implementar o teste adicionando um método *main()*⁵ na própria classe possuidora do método, como na figura 5.

⁵ É similar a função *main()* em C e C++; é o ponto de entrada para a aplicação e conseqüentemente irá invocar os outros métodos necessários para execução da mesma. (MYCROSYSTEMS, 1995)

```

1 package br.farn.sisecorp.tccsv.dominio;
2
3 class Calculadora {
4     int adiciona(int primeiraParcela, int segundaParcela) {
5         int soma;
6         soma = primeiraParcela + segundaParcela;
7         return soma;
8     }
9
10    static void testaAdicao() {
11        Calculadora calculadora = new Calculadora();
12        int primeiraParcela = 2;
13        int segundaParcela = 3;
14        int somaEsperada = 5;
15        int somaObtida = calculadora.adiciona(primeiraParcela, segundaParcela);
16        if (somaObtida != somaEsperada) {
17            System.err.println("Falha no Calculadora.adiciona():");
18            System.err.println("Resultado inadequado");
19            System.err.println("somaEsperada=" + somaEsperada);
20            System.err.println("somaObtida=" + somaObtida);
21        }
22    }
23
24    public static void main(String[] args) {
25        testaAdicao();
26    }
27 }

```

Figura 5 - Classe Calculadora com método main().

Fonte: O próprio autor. (2008)

Porém, tal ato feriria o princípio da coesão, visto que testar não é responsabilidade da classe em questão. Como solução deste problema, a escrita deste trecho de código poderia ser feita em uma classe separada, como na figura 6:

```

1 package br.farn.siscorp.tocsv.dominio;
2
3 class TestadorDaCalculadora {
4     public static void main(String[] args) {
5         testaAdicao();
6     }
7
8     static void testaAdicao() {
9         Calculadora calculadora = new Calculadora();
10        int primeiraParcela = 2;
11        int segundaParcela = 3;
12        int somaEsperada = 5;
13        int somaObtida = calculadora.adiciona(primeiraParcela, segundaParcela);
14        if (somaObtida != somaEsperada) {
15            System.err.println("Falha no Calculadora.adiciona():");
16            System.err.println("Resultado inadequado ");
17            System.err.println("somaEsperada=" + somaEsperada);
18            System.err.println("somaObtida=" + somaObtida);
19        }
20    }
21 }

```

Figura 6 - Classe TestadorDaCalculadora.

Fonte: O próprio autor. (2008)

A classe *TestadorDaCalculadora* cria uma instância da calculadora, passa as duas parcelas como parâmetro e verifica o resultado. Se o resultado não for o esperado, imprime-se uma mensagem na saída padrão.

Caso o código da figura 6 seja executado, nada será impresso na saída padrão. Agora imagine se alterarmos o código da classe *Calculadora* de tal forma que o teste falhe, como na figura 7:

```

1 package br.farn.siscorp.tocsv.dominio;
2
3 class Calculadora {
4     int adiciona(int primeiraParcela, int segundaParcela) {
5         int soma;
6         soma = primeiraParcela + segundaParcela + 1;
7         return soma;
8     }
9 }

```

Figura 7 - Introdução de falha na Calculadora.

Fonte: O próprio autor. (2008)

Então, ao compilarmos e executarmos novamente a classe *TestadorDaCalculadora*, obtém-se a mensagem na saída padrão da figura 8:

```
Falha no Calculadora.adiciona():
Resultado Inadequado!
somaEsperada=5
somaObtida=6
```

Figura 8 - Mensagem de erro da Calculadora.

Fonte: O próprio autor. (2008)

É evidente ainda o fato de o desenvolvedor ter que analisar se algo foi impresso na saída para constatar se há alguma falha no software. O processo de teste pode ser otimizado de tal forma que um *framework* auxiliar execute tal tarefa.

Segundo Massol; Husted (2003) há um conjunto de boas práticas que os frameworks de teste devem seguir:

- Cada teste unitário deve ser executado independentemente dos demais;
- Os erros devem ser detectados e reportados para cada teste;
- Deve ser simples definir quais testes serão executados.

O framework de teste *JUnit* provê a independência dos testes pelo carregamento de cada um deles em *classloaders*⁶ distintos. Adicionalmente, reporta os erros individualmente para cada teste e utiliza mecanismos de introspecção para selecionar quais os métodos de teste serão executados.

A partir da versão quatro do *JUnit*, para indicarmos que um método é um teste basta adicionarmos uma anotação *@Test* antes do mesmo. Além disso, o próprio *framework* fornece métodos para a construção das assertivas desejadas.

Adaptando a classe *TestadorDaCalculadora* para ser utilizada pelo *JUnit*, seria apenas necessário adicionar a anotação *@Test* antes do método que testa a adição e utilizar um dos métodos para construção das assertivas. Como na figura 9:

⁶ Objetos responsáveis pelo carregamento das classes. Toda classe possui referência ao classloader que a definiu. São criados automaticamente conforme necessário pelo ambiente de execução Java.

```

1 package br.farn.siscorp.tccsy.dominio;
2
3 import org.junit.Test;
4
5 import static org.junit.Assert.*;
6 import static org.hamcrest.CoreMatchers.*;
7
8 public class TestadorDaCalculadora {
9     @Test
10     public void testaAdicao() {
11         Calculadora calculadora = new Calculadora();
12         int primeiraParcela = 2;
13         int segundaParcela = 3;
14         int somaEsperada = 5;
15         int somaObtida = calculadora.adiciona(primeiraParcela, segundaParcela);
16         assertThat(somaObtida, is(somaEsperada));
17     }
18 }

```

Figura 9 - Teste da adição da Calculadora com JUnit.
Fonte: O próprio autor. (2008)

É perceptível o ganho de legibilidade no teste. Executando-o novamente com o auxílio da IDE e o do *JUnit*, o resultado exibido na figura 10 é obtido.

Finished after 0,032 seconds

Runs: 1/1

Errors: 0

Failures: 1

br.farn.siscorp.tccsy.dominio.TestadorDaCalculadora [Runner: JUnit 4]
testaAdicao

Figura 10 - Resultado do primeiro teste com JUnit da Calculadora.
Fonte: O próprio autor. (2008)

Após a execução dos testes, são apresentadas as informações:

- Duração;
- Quantos foram executados do total existente;
- Quantos apresentaram erros em sua execução;
- Quantos falharam em sua execução.

Um erro é caracterizado pelo lançamento de uma exceção inesperada pela execução do testes. Já uma falha ocorre quando as assertivas do teste não são atendidas.

No caso do *TestadorDaCalculadora*, o resultado do teste é exibido com

falha em virtude das modificações feitas na figura 7. A própria IDE exibe detalhes sobre a falha do teste, como na figura 11:

```

Failure Trace
java.lang.AssertionError:
  Expected: is <5>
    got: <6>

at br.farn.siscorp.tccsv.dominio.TestadorDaCalculadora.testaAdicao(TestadorDaCalculadora.java:17)

```

Figura 11 - Detalhes da falha introduzida na Calculadora.

Fonte: O próprio autor. (2008)

Com a remoção da instrução que provocava a falha e a execução novamente do teste, o resultado da figura 12 é apresentado.

```

Finished after 0,016 seconds

Runs: 1/1           Errors: 0           Failures: 0

br.farn.siscorp.tccsv.dominio.TestadorDaCalculadora [Runner: JUnit 4]
  testaAdicao

```

Figura 12 - Resultado do teste da Calculadora após correção.

Fonte: O próprio autor. (2008)

Seguindo os mesmos passos vários outros testes poderiam ser adicionados. Obtém-se, conseqüentemente, uma vasta coleção de testes para o software em construção.

Comparando esta abordagem com as duas anteriores, há um esforço maior na escrita do teste pelo desenvolvedor. Entretanto, elimina-se a necessidade de um análise cuidadosa humana e o caráter temporário do teste. Adicionalmente, este passa a agregar valor definitivamente ao software.

Considerando que os testes deixam de ser do desenvolvedor e passam a integrar concretamente ao software, prevalece o espírito colaborativo que culmina em um software menos sujeito à falhas. Amplia-se a produtividade nas futuras modificações do software pois com a existência dos testes automatizados, o desenvolvedor torna-se mais confiante para efetuar alterações no código.

3.1.4 Test-Driven Development (TDD)

Partindo das vantagens de se descobrir as falhas o mais cedo possível

através dos testes de caixa branca e o uso dos testes automatizados com auxílio de *frameworks* específicos, surgiu o conceito de desenvolvimento guiado por testes – *Test-Driven Development* (TDD).

Parafraseando Ron Jeffrie, o TDD possui como objetivo principal o "código limpo que funcione". Segundo Beck (2003), para atingí-lo devemos guiar o desenvolvimento com testes automatizados, seguindo duas simples regras:

- Escreva novo código apenas se um teste automatizado falhou;
- Elimine a duplicação.

Baseado em tais regras, criou-se o mantra do TDD:

- Vermelho: escreva um pequeno teste que não funcione e que provavelmente nem compilará em um primeiro momento;
- Verde: faça o teste funcionar rapidamente;
- Refatoração: elimine toda código duplicado de tal forma que o teste continue funcionando.

Ainda de acordo com Beck (2003), o TDD melhora a gerência do medo durante a codificação. Pois, com ele, o desenvolvedor tem mais controle e conhecimento sobre o que está construindo.

Guiar o desenvolvimento por testes, significa escrevê-los antes mesmo do software. Os testes deixam de ser apenas verificadores e validadores das unidades e passam a guiar a construção do software. O próprio design da aplicação evolui durante as refatorações. Objetiva-se ser o mais simples e rápido possível, produzindo código de alta qualidade e pouco vulnerável a falhas.

Castro em seu artigo "Refatoração e TDD, feitos um para o outro" na Java Magazine expõe outras vantagens do uso desta técnica. Segundo ele, o desenvolvimento guiado por testes segue o caminho da prevenção que resulta um custo menor na correção de falhas por dois motivos:

- O teste expõe o erro assim que ele entra no sistema, poupando tempo pois não são necessárias depurações demoradas;
- Caso um erro seja introduzido em uma parte diferente da que se está trabalhando no momento, os testes acusam o erro de maneira precisa e rápida, permitindo que ele seja corrigido rapidamente.

Outra vantagem do uso do TDD é a possibilidade de usar os testes como

documentação do software produzido. Como eles representam exemplos de utilizações do código, podem ser bastante úteis para auxiliar na compreensão deste. Possuem, adicionalmente, o diferencial de estarem sempre sincronizados com o código, o que muitas vezes não ocorre com a documentação através de diagramas e textos.

3.1.4.1 Exemplo

O exemplo será baseado no feito por Brian Button (2007). Suponha que há vários livros desorganizados sobre o chão. Não é possível relacionar quais os livros existentes. Necessita-se de um repositório de livros. Neste exemplo, este será chamado de gerenciador de livros. Objetiva-se construí-lo e o mesmo deve apresentar a lista dos livros ordenados alfabeticamente pelo título. Conforme o quadro 2:

Id do livro	Título	Gênero	Já foi lido
1	Seja Líder de Si Mesmo	Auto-ajuda	S
2	Casais Inteligentes Enriquecem Juntos	Negócios	N
3	Técnicas de Cozinha Profissional	Culinária	N
4	Vale Tudo : Tím Maia	Biografia	S

Quadro 2 - Lista de Livros.

Fonte: O próprio autor.

O primeiro passo é construir o teste que crie o gerenciador de livros e garanta que ele retorna uma lista vazia visto que nenhum livro foi adicionado. Como na figura 13.

```

1 public class TestadorDoGerenciadorDeLivros {
2     @Test
3     public void gerenciadorRecenteRetornaListaVaziaDeLivros() {
4         GerenciadorDeLivros gerenciador = new GerenciadorDeLivros();
5         List<Livro> listaAlfabeticaDeLivros = gerenciador
6             .obtemListaAlfabeticaDeLivros();
7         assertNotNull("a lista não pode ser nula.", listaAlfabeticaDeLivros);
8         assertTrue("a lista deve estar vazia.", listaAlfabeticaDeLivros.isEmpty());
9     }
10 }

```

Figura 13 - Criação do teste do GerenciadorDeLivros que retorna uma lista vazia.

Fonte: O próprio autor. (2008)

O compilador automaticamente acusa a inexistência das classes *GerenciadorDeLivros* e *Livro*. Visando a escrita apenas do código estritamente

necessário, as classes serão criadas sem corpo. Conforme nas figuras 14 e 15:

```
class Livro {
```

```
}
```

Figura 14 - Criação da classe Livro sem corpo.
Fonte: O próprio autor. (2008)

```
class GerenciadorDeLivros {
```

```
}
```

Figura 15 - Criação da classe GerenciadorDeLivros sem corpo.
Fonte: O próprio autor. (2008)

As classes *Livro* e *GerenciadorDeLivros* passarão a ser reconhecidas. No entanto a classe continua com erro de compilação devido à inexistência do método *obtemListaAlfabeticaDeLivros()*. Portanto, este será criado apenas para que o teste compile.

```
class GerenciadorDeLivros {
    Collection<Livro> obtenListaAlfabeticaDeLivros() {
        return null;
    }
}
```

Figura 16 - Criação do método *obtemListaAlfabeticaDeLivros()* no *GerenciadorDeLivros*.
Fonte: O próprio autor. (2008)

Após a criação das classes *Livro* e *GerenciadorDeLivros*; e do método *obtemListaAlfabeticaDeLivros()* o teste não apresenta mais erro de compilação na figura 17:

```
10 public class TestadorDoGerenciadorDeLivros {
11     @Test
12     public void gerenciadorRecemCriadoRetornaListaVaziaDeLivros() {
13         GerenciadorDeLivros gerenciador = new GerenciadorDeLivros();
14         List<Livro> listaAlfabeticaDeLivros = gerenciador
15             .obtemListaAlfabeticaDeLivros();
16         assertNotNull("a lista não pode ser nula.", listaAlfabeticaDeLivros);
17         assertTrue("a lista deve estar vazia.", listaAlfabeticaDeLivros.isEmpty());
18     }
19 }
```

Figura 17 - Remoção dos erros de compilação do teste que retorna a lista vazia do *GerenciadorDeLivros*.
Fonte: O próprio autor. (2008)

Após as modificações para que o teste compile sem erros, deve-se executá-lo observando-se seu resultado, como na figura 18.

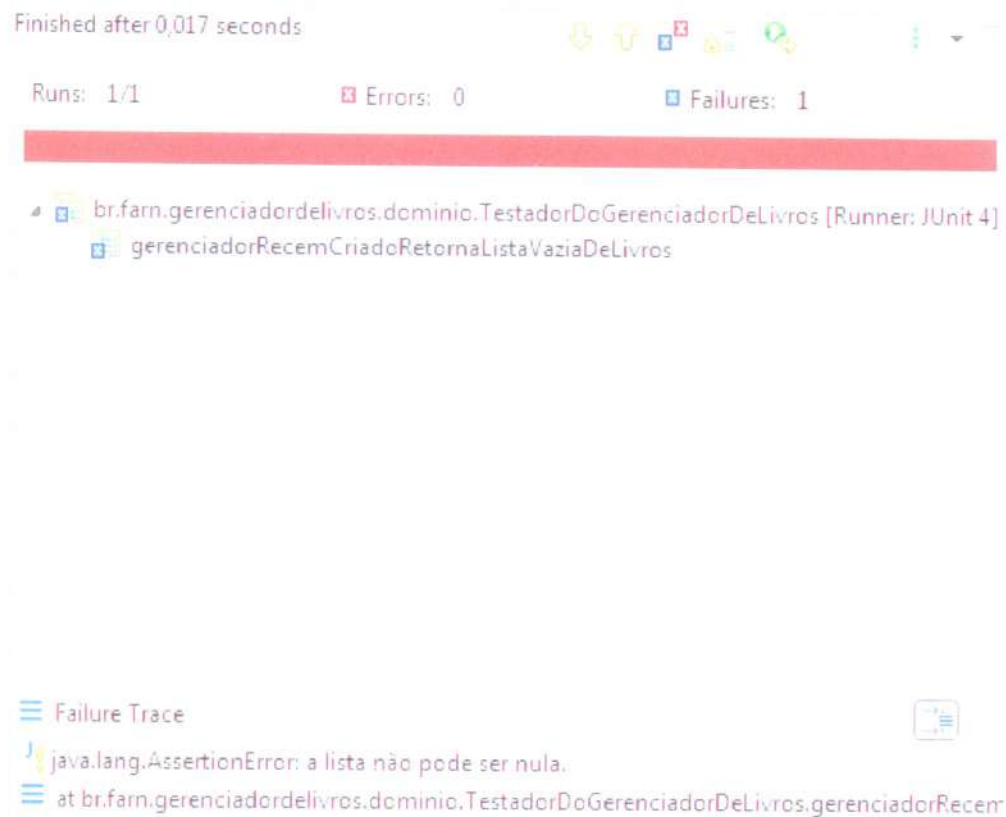


Figura 18 - Teste da lista vazia do GerenciadorDeLivros sem erros de compilação.
Fonte: O próprio autor. (2008)

Ao executar o teste, uma falha é apresentada e obtém-se a cor vermelha. O próximo passo é efetuar as modificações necessárias para que o teste seja executado sem falhas e atinja-se a cor verde. Para correção da falha obtida, o método *obtemListaAlfabeticaDeLivros()* será alterado para que retorne uma nova lista vazia, como na figura 19.

```
class GerenciadorDeLivros {

    List<Livro> obtenListaAlfabeticaDeLivros() {
        return new ArrayList<Livro>();
    }
}
```

Figura 19 - Método *obtemListaAlfabeticaDeLivros()* modificado para retornar uma nova lista.
Fonte: O próprio autor. (2008)

A figura 20 mostra a nova execução.

Finished after 0,013 seconds

Runs: 1/1

Errors: 0

Failures: 0

br.fam.gerenciadordelivros.dominio.TestadorDoGerenciadorDeLivros [Runner: JUnit 4]

Figura 20 - Nova execução do teste após alterar `obtemListaAlfabeticaDeLivros()`.

Fonte: O próprio autor. (2008)

O teste é executado sem falhas. É válido ressaltar que o teste representa uma afirmação de como o código deve agir quando a lista estiver vazia. Esta afirmação é válida agora e terá que ser válida para sempre, caso contrário o teste não será executado sem falhas. Apesar de ser uma situação extremamente trivial, se não houvesse a escrita deste teste não se teria nenhuma garantia de como o sistema reagiria em uma situação desta. A figura 21 mostra a adição de um novo teste.

```

20  @Test
21  public void umLivroAdicionadoEhRetornadoNaListaOrdenada() {
22      GerenciadorDeLivros gerenciador = new GerenciadorDeLivros();
23      String tituloDoLivro = "Seja Líder de Si Mesmo";
24      Livro sejaLiderDeSiMesmo = new Livro(tituloDoLivro);
25      gerenciador.adiciona(sejaLiderDeSiMesmo);
26
27      List<Livro> listaAlfabeticaDeLivros = gerenciador
28          .obtemListaAlfabeticaDeLivros();
29      assertEquals("o tamanho da lista deve ser um.", 1,
30          listaAlfabeticaDeLivros.size());
31      assertEquals("o primeiro livro da lista deve ter o mesmo título"
32          + " do recém adicionado.", tituloDoLivro,
33          listaAlfabeticaDeLivros.get(0).obtenhaTitulo());
34  }

```

Figura 21 - Criação do teste para um livro.

Fonte: O próprio autor. (2008)

Para corrigir os erros de compilação, serão criados o construtor e o método `obtenhaTitulo()` na classe `Livro` e o método `adiciona()` na classe `GerenciadorDeLivros`. As figuras 22 e 23 exibem tais modificações.

```

class Livro {
    private String titulo;
    Livro(String titulo) {
        this.titulo = titulo;
    }
    String obtenhaTitulo() {
        return titulo;
    }
}

```

Figura 22 - Criação de construtor e método no Livro.

Fonte: O próprio autor. (2008)

```

class GerenciadorDeLivros {

    List<Livro> obtenListaAlfabeticaDeLivros() {
        return new ArrayList<Livro>();
    }

    void adiciona(Livro livro) {

    }
}

```

Figura 23 - Criação do método adiciona() em GerenciadorDeLivros.

Fonte: O próprio autor. (2008)

Por conseguinte, os erros de compilação do novo teste serão removidos como na figura 24.

```

20- }
21- public void umLivroAdicionadoEhRetornadoNaListaOrdenada() {
22-     GerenciadorDeLivros gerenciador = new GerenciadorDeLivros();
23-     String tituloDoLivro = "Seja Líder de Si Mesmo";
24-     Livro sejaLiderDeSiMesmo = new Livro(tituloDoLivro);
25-     gerenciador.adiciona(sejaLiderDeSiMesmo);
26-
27-     List<Livro> listaAlfabeticaDeLivros = gerenciador
28-         .obtemListaAlfabeticaDeLivros();
29-     assertEquals("o tamanho da lista deve ser um.", 1,
30-         listaAlfabeticaDeLivros.size());
31-     assertEquals("o primeiro livro da lista deve ter o mesmo título"
32-         + " do recém adicionado.", tituloDoLivro,
33-         listaAlfabeticaDeLivros.get(0).obtenhaTitulo());
34- }

```

Figura 24 - Remoção dos erros de compilação do teste de um livro.

Fonte: O próprio autor. (2008)

A figura 25 mostra a nova execução do teste.

Finished after 0,018 seconds

Runs: 2/2

Errors: 0

Failures: 1

br.fam.gerenciadordelivros.dominio.TestadorDoGerenciadorDeLivros [Runner: JUnit
 gerenciadorRecemCriadoRetornaListaVaziaDeLivros
 umLivroAdicionadoEhRetornadoNaListaOrdenada

Failure Trace

java.lang.AssertionError: o tamanho da lista deve ser um, expected:<1> but was:<0>
 at br.fam.gerenciadordelivros.dominio.TestadorDoGerenciadorDeLivros.umLivroAdicio

Figura 25 - Execução do teste após remoção dos erros de compilação.

Fonte: O próprio autor. (2008)

Obtém-se uma falha na linha 28 da classe *TestadorDoGerenciadorDeLivros* dentro do teste *umLivroAdicionadoEhRetornadoNaListaOrdenada*. Isso ocorre pois o método *obtemListaAlfabeticaDeLivros()* retorna uma lista vazia quando deveria possuir um elemento. Para a remoção da falha, deve-se criar uma lista de livros dentro do gerenciador e modificar os métodos *adiciona()* e *obtemListaAlfabeticaDeLivros()*. A figura 26 mostra essas alterações.

```

class GerenciadorDeLivros {

    private List<Livro> listaDeLivros = new ArrayList<Livro>();

    List<Livro> obterListaAlfabeticaDeLivros() {
        return listaDeLivros;
    }

    void adiciona(Livro livro) {
        listaDeLivros.add(livro);
    }
}

```

Figura 26 - Criação da lista de livros dentro do GerenciadorDeLivros.
Fonte: O próprio autor. (2008)

Posteriormente, os testes devem ser executados novamente. O resultado desta nova execução é exibido na figura 27.

Finished after 0,014 seconds

Runs: 2/2

Errors: 0

Failures: 0

```

br.farn.gerenciadordelivros.dominio.TestadorDoGerenciadorDeLivros [Runner: JUnit]
  gerenciadorRecemCriadoRetornaListaVaziaDeLivros
  umLivroAdicionadoEhRetornadoNaListaOrdenada

```

Figura 27 - Nova execução após adição da lista de livros no GerenciadorDeLivros.
Fonte: O próprio autor. (2008)

Os testes são executados sem falhas. Então, deve-se adicionar um novo teste. A figura 28 mostra o novo teste criado para múltiplos livros adicionados em ordem não alfabética.

```

36  @Test
37  public void multiplosLivrosAdicionadosSaoRetornadosOrdenadosAlfabeticamentePeloTitulo() {
38      GerenciadorDeLivros gerenciador = new GerenciadorDeLivros();
39      String tituloDoPrimeiroLivro = "Seja Líder de Si Mesmo";
40      String tituloDoSegundoLivro = "Casais Inteligentes Enriquecem Juntos";
41      gerenciador.adiciona(new Livro(tituloDoPrimeiroLivro));
42      gerenciador.adiciona(new Livro(tituloDoSegundoLivro));
43
44      List<Livro> listaAlfabeticaDeLivros = gerenciador
45          .obtemListaAlfabeticaDeLivros();
46      assertEquals("o tamanho da lista deve ser dois.", 2,
47          listaAlfabeticaDeLivros.size());
48      assertEquals("o título do primeiro livro da lista ordenada deve ser"
49          + " o do segundo livro adicionado", tituloDoSegundoLivro,
50          listaAlfabeticaDeLivros.get(0).obtemTitulo());
51      assertEquals("o título do segundo livro da lista ordenada deve ser"
52          + " o do primeiro livro adicionado", tituloDoPrimeiroLivro,
53          listaAlfabeticaDeLivros.get(1).obtemTitulo());
54  }
55  }

```

Figura 28 - Criação do teste de múltiplos livros.

Fonte: O próprio autor. (2008)

Em seguida, executam-se os testes novamente e os resultados são apresentados na figura 29.

Finished after 0.022 seconds

Runs: 3/3

Errors: 0

Failures: 1

br.fam.gerenciadorde livros.dominio.TestadorDoGerenciadorDeLivros [Runner: JUnit 4]

- gerenciadorRecemCriadoRetornaListaVaziaDeLivros
- umLivroAdicionadoEhRetornadoNaListaOrdenada
- multiplosLivrosAdicionadosSaoRetornadosOrdenadosAlfabeticamentePeloTitulo

Failure Trace

adicionado expected: <[Casais Inteligentes Enriquecem Juntos]> but was: <[Seja Líder de Si Mesmo]>
 sSaoRetornadosOrdenadosAlfabeticamentePeloTitulo(TestadorDoGerenciadorDeLivros.java:48)

Figura 29 - Resultado após criação do teste de múltiplos livros.

Fonte: O próprio autor. (2008)

Verifica-se que o método *obtemListaAlfabeticaDeLivros()* da classe *GerenciadorDeLivros* não retornou a lista em ordem alfabética e conseqüentemente

o teste falhou. Objetivando-se ordenar a lista alfabeticamente para só então retorná-la, será utilizado o método `sort()` da classe *Collections* do próprio java. Esta mudança é feita na linha 12 da figura 30.

```

class GerenciadorDeLivros {
    private List<Livro> listaDeLivros = new ArrayList<Livro>();

    List<Livro> obterListaAlfabeticaDeLivros() {
        Collections.sort(listaDeLivros);
        return listaDeLivros;
    }

    void adiciona(Livro livro) {
        listaDeLivros.add(livro);
    }
}

```

Figura 30 - Adição do método `sort()` na lista de livros.

Fonte: O próprio autor. (2008)

O compilador acusa um erro: para que a classe *Collections* ordene uma coleção, os itens contidos nesta devem ser comparáveis. Portanto, como é uma lista de livros, deve-se tornar o livro um elemento comparável, o que exigirá implementação específica na classe *Livro*. Conforme Button (2007, p.127), ao atingir-se uma situação destas, deve-se escrever uma nova classe de testes: “Um cenário para uma classe não deve forçá-lo a adicionar código em uma classe distinta. Escreva um cenário separado. O primeiro cenário descreve a necessidade; o segundo, guia a implementação.”

Neste caso, será criada uma nova classe para testes sobre a classe *Livro*, denominada *TestadorDoLivro*. Em seu interior serão adicionados testes para verificarem o método de comparação. A figura 31 mostra os métodos adicionados.

```

1 public class TestadorDoLivro {
2     //Teste
3     public void livroComTituloAnteriorAoSegundoDeveRetornarUmValorNegativoNaComparacao() {
4         Livro anterior = new Livro("Casais Inteligentes Enriquecem Juntos");
5         Livro posterior = new Livro("Sega Lider de Sa Mesmo");
6
7         assertTrue(anterior.compareTo(posterior) < 0);
8     }
9
10    //Teste
11    public void livroComTituloPosteriorAoSegundoDeveRetornarUmValorPositivoNaComparacao() {
12        Livro posterior = new Livro("Sega Lider de Sa Mesmo");
13        Livro anterior = new Livro("Casais Inteligentes Enriquecem Juntos");
14
15        assertTrue(posterior.compareTo(anterior) > 0);
16    }
17
18    //Teste
19    public void livroComMesmoTituloDeveRetornarZeroNaComparacao() {
20        Livro anterior = new Livro("Casais Inteligentes Enriquecem Juntos");
21        Livro posterior = new Livro("Casais Inteligentes Enriquecem Juntos");
22
23        assertEquals(0, anterior.compareTo(posterior));
24    }
25 }

```

Figura 31 - Adição dos testes para comparação de Livros.

Fonte: O próprio autor. (2008)

Visando a correção do erro de compilação, a figura 32 implementa a interface *Comparable* do próprio Java e conseqüentemente o método *compareTo()*.

```

1 class Livro implements Comparable<Livro> {
2     private String titulo;
3     Livro(String titulo) {
4         this.titulo = titulo;
5     }
6     String obtenhaTitulo(){
7         return titulo;
8     }
9     //Método da interface
10    public int compareTo(Livro o) {
11        return 0;
12    }
13 }

```

Figura 32 - Implementação de Comparable pelo Livro.

Fonte: O próprio autor. (2008)

A figura 33 exibe a nova execução dos testes do *TestadorDoLivro*.

Finished after 0,021 seconds

Runs: 3/3

Errors: 0

Failures: 2

```
br.farn.gerenciadordelivros.dominio.TestadorDoLivro [Runner: JUnit 4]
  livroComTituloAnteriorAoSegundoDeveRetornarUmValorNegativoNaComparacao
  livroComTituloPosteriorAoSegundoDeveRetornarUmValorPositivoNaComparacao
  livroComMesmoTituloDeveRetornarZeroNaComparacao
```

Failure Trace



mTituloAnteriorAoSegundoDeveRetornarUmValorNegativoNaComparacao(TestadorDoLivro.java:13)

Figura 33 - Execução dos testes após o Livro se tornar comparável.

Fonte: O próprio autor. (2008)

Em virtude do método *compareTo()* ainda não implementar nenhum comportamento, dois testes são apresentam falha em sua execução. Para corrigi-las, implementa-se o comportamento necessário na figura 34.

```
class Livro implements Comparable<Livro> {
    private String titulo;
    Livro(String titulo) {
        this.titulo = titulo;
    }
    String obtenhaTitulo() {
        return titulo;
    }
    @Override
    public int compareTo(Livro outroLivro) {
        return this.titulo.compareTo(outroLivro.titulo);
    }
}
```

Figura 34 - Implementação correta do *compareTo()* do Livro.

Fonte: O próprio autor. (2008)

Em sequência, executa-se novamente os testes da classe *TestadorDoLivro*. O resultado é mostrado na figura 35.

Finished after 0,015 seconds

Runs: 3/3

Errors: 0

Failures: 0

```
br.fam.gerenciadordelivros.dominio.TestadorDoLivro [Runner: JUnit 4]
  livroComTituloAnteriorAoSegundoDeveRetornarUmValorNegativo
  livroComTituloPosteriorAoSegundoDeveRetornarUmValorPositivo
  livroComMesmoTituloDeveRetornarZeroNaComparacao
```

Figura 35 - Execução após implementação correta do `compareTo()` do `Livro`.

Fonte: O próprio autor. (2008)

Os testes sobre a classe *Livro* são executados com sucesso. Neste ponto, os erros de compilação da classe *GerenciadorDeLivros* já foram eliminados pois os elementos da lista de livros são comparáveis. O próximo passo é a execução dos testes do *TestadorDoGerenciadorDeLivros*. A figura 36 exhibe este passo.

Finished after 0,015 seconds

Runs: 3/3

Errors: 0

Failures: 0

```
br.fam.gerenciadordelivros.dominio.TestadorDoGerenciadorDeLivros [F
  gerenciadorRecemCriadoRetornaListaVaziaDeLivros
  umLivroAdicionadoEhRetornadoNaListaOrdenada
  multiplosLivrosAdicionadosSaoRetornadosOrdenadosAlfabeticame
```

Figura 36 - Execução final do *TestadorDoGerenciadorDeLivros*.

Fonte: O próprio autor. (2008)

Após a criação de seis testes, três para verificar o método de comparação da classe *livro* e outros três para a classe *GerenciadorDeLivros*, finaliza-se o desenvolvimento com um modelo extremamente simples e pouco vulnerável à falhas. Uma suíte de testes foi criada e faz parte do produto, poderá ser reutilizada e executada por outros desenvolvedores da equipe.

3.2 TESTES DE INTEGRAÇÃO

Diferentemente dos testes de unidade que testam as menores unidades de código independentemente, os testes de integração verificam a integração entre os módulos que constituem a aplicação com suas reais dependências, através de um ambiente próximo ao de produção.

É importante ressaltar que apesar das denominações semelhantes, os

testes de integração diferem do conceito de integração contínua, prática adotada pela programação extrema – *Extreme Programming* (XP) que consiste em integrar o código dos desenvolvedores de uma equipe no repositório de forma segura várias vezes ao dia.

3.2.1 Exemplo

Objetivando-se persistir as informações, uma dependência comumente encontrada nos sistemas corporativos é a dos bancos de dados relacionais. Por conseguinte, geralmente há uma camada responsável pela persistência que comunica-se com o banco.

O *DbUnit*, uma extensão do *JUnit*, é um *framework* que facilita o teste da camada de persistência. Este permite que se defina um estado para o banco de dados que não será modificado entre a execução dos testes. Isso previne que um teste influencie no estado do banco de dados de tal maneira que os testes subsequentes falhem, garantindo a independência entre os mesmos. Ele também pode ser utilizado para verificar se os dados existentes no banco conferem com um conjunto de valores.

Para o exemplo, será utilizado o banco de dados HSQL. Após o banco em execução, a figura 37 mostra a criação da tabela a ser testada através do *DbUnit*, denominada *Usuario*.

```
CREATE TABLE Usuario (  
    id bigint not null identity,  
    nome varchar(100) not null,  
    login varchar(20) not null,  
    senha varchar(20) not null,  
    primary key (id) )
```

Figura 37 - Criação da tabela *Usuario*.

Fonte: O próprio autor. (2008)

Em seguida, será criado o arquivo *entrada.xml* que representará o estado desejado para o banco durante os testes. Conforme a figura 38:

```

<?xml version="1.0" encoding="UTF-8"?>

<dataset>
  <!-- senha e o md5 de 32 -->
  <usuario id="1" nome="Samuel Valerio" login="samuelvalerio"
    senha="202cb962ac59075b964b07152d234b70" />
  <usuario id="2" nome="Luiz Pereira" login="luizpereira"
    senha="202cb962ac59075b964b07152d234b70" />
</dataset>

```

Figura 38 - Criação do entrada.xml.

Fonte: O próprio autor. (2008)

Especificou-se que na tabela *Usuario* devem existir dois registros. Então, a classe *TestadorDoDbUnit* será criada com a estrutura para que seja possível a utilização do *DbUnit*. Esta criação pode ser visualizada na figura 39.

```

1 public class TestadorDoDbUnit {
2
3     private IDatabaseTester testadorDoBanco;
4
5     @Before
6     public void inicializa() throws Exception {
7         testadorDoBanco = new JdbcDatabaseTester("org.hsqldb.jdbcDriver",
8             "jdbc:hsqldb:hsqldb://localhost/meubd", "sa", "");
9         testadorDoBanco.setDataSet(new FlatXmlDataSet(new FileInputStream(
10             "entrada.xml")));
11         testadorDoBanco.onSetup();
12     }
13
14     @After
15     public void finaliza() throws Exception {
16         testadorDoBanco.onTearDown();
17     }
18 }

```

Figura 39 - Criação do TestadorDoDbUnit.

Fonte: O próprio autor. (2008)

Um método anotado com *@Before* no *JUnit 4* será executado antes da execução de cada teste existente. Nele é feita a inicialização do conjunto de dados que será utilizado pelos testes.

A figura 39 em sua linha 13 possui interface *IDatabaseTester*. Esta representa o responsável pela manipulação dos *datasets*. No exemplo em questão, o banco HSQLDB será acessado via driver jdbc, logo a implementação indicada para este caso é o *JdbcDatabaseTester*.

Após a definição da classe que manipulará os *datasets*, deve-se definir quais serão os *datasets* manipulados pelo caso de teste. Isto é feito através do método *setDataSet()* na linha 19 ainda da figura 39.

Para finalizar a inicialização, invoca-se o método *onSetup()* na linha 21 da figura 39 que irá provocar o comportamento padrão do *DbUnit* para um teste. Isto equivale ao comandos de limpar todo o conteúdo das tabelas envolvidas no *dataset* e incluir os dados definidos no *entrada.xml*, mantendo as tabelas envolvidas apenas com os registros necessários para a execução dos testes.

Também anota-se com *@After* o método *finaliza()* para que o *JUnit 4* execute-o sempre ao término de cada teste. O comportamento do método *onTearDown()* padrão do *DbUnit* não modifica os dados que foram inseridos de acordo com o *dataset* do caso de teste.

Para a visualização da modificação nos dados existentes na tabela *Usuario*, será incluído o usuário "Fernando da Silva" que inexiste no *dataset* do caso de teste a ser executado. A figura 40 mostra essa inclusão.

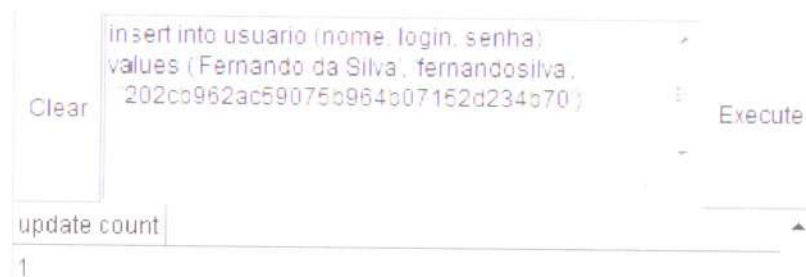


Figura 40 - Inclusão do usuário Fernando da Silva.
Fonte: O próprio autor. (2008)

Para a conferência sobre a inclusão do usuário, na figura 41 executa-se o comando de seleção sobre os registros da tabela *Usuario*.

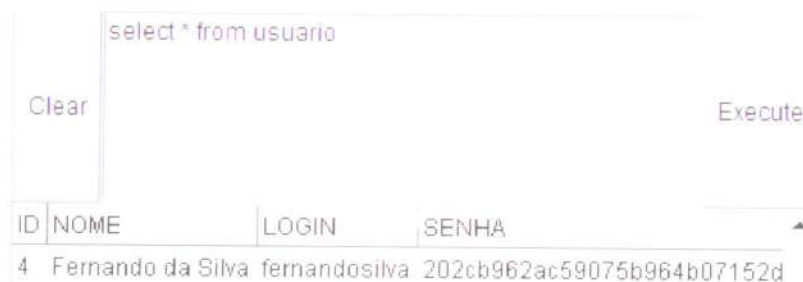


Figura 41 - Seleção de todos os usuários.
Fonte: O próprio autor. (2008)

Então, um teste será adicionado visando apenas verificar se os dois registros configurados no *dataset* foram adicionados corretamente. Esta adição pode ser vista na figura 42.

```

19  @Test
20  public void comparaSeOsDadosNaTabelaSaoOsMesmosDaEntradaDoDbUnit()
21      throws DataSetException {
22      assertNotNull("o dataset não pode ser nulo", testadorDoBanco
23          .getDataSet());
24      assertEquals("a tabela deve possuir dois usuários.", 2, testadorDoBanco
25          .getDataSet().getTable("usuario").getRowCount());
26  }

```

Figura 42 - Criação do teste do DbUnit.

Fonte: O próprio autor. (2008)

A figura 43 mostra o resultado da nova execução do teste.

Finished after 0,188 seconds

Runs: 1/1

Errors: 0

Failures: 0

br.farm.siscorp.tccsv.TestadorDoDbUnit [Runner: JUnit 4]
 comparaSeOsDadosNaTabelaSaoOsMesmosDaEntradaDoDbUnit

Figura 43 - Execução do teste do DbUnit.

Fonte: O próprio autor. (2008)

O teste foi executado sem falhas. Considerando o comportamento padrão do *DbUnit* de apagar os registros existentes nas tabelas envolvidas no *dataset* e a posterior inclusão dos dados definidos no mesmo, executa-se novamente o comando para listar todos os registros existentes na tabela *usuario*. Conforme exibido na figura 44.

select * from usuario			
Clear			
Execute			
ID	NOME	LOGIN	SENHA
1	Samuel Valerio	samuelvalerio	202cb962ac59075b964b07152d21
2	Luiz Pereira	luizpereira	202cb962ac59075b964b07152d21

Figura 44 - Confirmação do comportamento do DbUnit.

Fonte: O próprio autor. (2008)

Conforme esperado, o usuário "Fernando da Silva" foi removido e os usuários "Samuel Valerio" e "Luiz Pereira" foram adicionados, comprovando o funcionamento do teste através do *DbUnit*. Este tipo de teste, pode ser utilizado para verificar o comportamento de Objetos de Acesso a Dados – *Data Access Objects* (DAO), comumente implementados na camada de persistência de aplicações corporativas.

3.5 TESTES DE ACEITAÇÃO

Os testes de aceitação, também denominados funcionais, são criados a partir das especificações das funcionalidades. São executados do ponto de vista do cliente. Ele deve especificar os cenários que serão testados quando uma funcionalidade foi corretamente implementada.

Estes testes são de caixa preta. Cada teste representa algum resultado esperado pelo usuário. Os clientes são os responsáveis por verificar a corretude dos testes de aceitação e revisá-los. Também são utilizados como testes de regressão antes da liberação da *release*.

A automatização deste tipo de teste é extremamente simplificada com o uso de ferramentas como o *Selenium*. Utilizando-a, os testes rodam diretamente no navegador, simulando um usuário. Há dois componentes do *Selenium* que podem ser utilizados para automatizar os testes de aceitação.

O *Selenium Remote Control* (RC) no qual os testes são realizados através de um programa que comanda o navegador por um *proxy*. Este programa pode ser escrito em Java, C#, Perl, PHP e Ruby. Exige a programação dos testes. Em contrapartida, é bastante flexível e poderoso.

Já o *Selenium IDE* os testes são realizados através de um plugin instalado no navegador *Firefox*. Este permite a gravação da navegação do usuário para executá-la posteriormente. É extremamente simples a sua utilização. No entanto, perde-se um pouco do controle sobre o teste e possui limitações para testes mais complexos. Os testes gerados podem ser exportados em algumas linguagens. Devido à extrema simplicidade de uso da ferramenta, sua utilização não será demonstrada neste trabalho.

A automatização dos testes de aceitação, podem aumentar significativamente a produtividade dos testadores vez que os testes são reutilizados. Além de passarem a integrar o software, podendo ser executados a qualquer momento, agregando mais qualidade conseqüentemente ao produto.

4 ESTUDO DE CASO COM TDD

Este trabalho apresentará o estudo de caso sobre o projeto JRimum, fruto da iniciativa conjunta entre o Grupo JRimum e a empresa Nordeste Fomento Mercantil objetivando a criação de componentes reutilizáveis java com foco no domínio de negócios do Brasil.

Um dos componentes já disponibilizados é o Bopepo: uma biblioteca Java que tem por finalidade a geração de boletos bancários, documentos que representam um título de cobrança utilizados por empresas da iniciativa privada, seguindo uma especificação criada pela Federação Brasileira de Bancos - FEBRABAN.

Almeja-se, através de TDD, construir outro componente só que para a emissão de guias de arrecadação, geralmente utilizadas por empresas e órgãos de cunho público para o recebimento de tributos/taxas estaduais e municipais e contas de concessionárias de serviços públicos, seguindo outra especificação da mesma federação.

De acordo com a especificação, a guia de arrecadação deve seguir o formulário padrão ilustrado na figura 45.

COMPROVANTE - CONSUMIDOR

USO DA CONCESSIONÁRIA

100 MM

INSTRUÇÕES PARA RECEBIMENTO

10 MM

VENCIMENTO VALOR

10 MM

AUTENTICAÇÃO MECÂNICA

20 MM

COMPROVANTE - EMPRESA

LOGOTIPO DA EMPRESA

84610000000 5, 24610029110 2, 00546033900 4, 534588506108 0

IDENTIFICAÇÃO PARA PAGAMENTO EM TERMINAIS ELTRÔNICOS

VENCIMENTO VALOR

AUTENTICAÇÃO MECÂNICA

20 MM

Figura 45 - Formulário padrão da guia de arrecadação.
Fonte: Especificação da FEBRABAN. (2004)

Constata-se a existência de dois comprovantes: o da empresa e o do consumidor. No do consumidor constam apenas informações que deverão ser fornecidas por quem estiver utilizando o componente como instruções para recebimento, vencimento e valor. Já no comprovante da empresa, além do logotipo da mesma, vencimento e valor há a representação numérica e gráfica do código de barras.

A representação numérica da guia, denominada código de barras, reúne as informações necessárias para permitir o arrecadação da guia por algum agente arrecadador, como um banco por exemplo. Além disto, ela permite a geração da representação gráfica através de barras. Com a existência desta, é possível automatizar o procedimento de leitura com o uso de algum leitor óptico e também

elimina-se a possibilidade de erros de digitação.

Para o início do TDD, devemos identificar o primeiro comportamento a ser testado. O único comportamento detectado na guia em um primeiro momento é a lógica para a geração do código de barras. O restante são apenas informações que serão passadas por quem estiver utilizando o componente e que serão exibidas na visualização da guia.

A figura 46 exibe o código de barras em sua representação numérica e gráfica. Considerando que existem vários *frameworks* que geram a representação gráfica a partir da numérica, o software a ser desenvolvido pode gerar apenas esta e delegar a responsabilidade de geração daquela para algum *framework*.



Figura 46 - Representação numérica e gráfica do código de barras.

Fonte: Especificação FEBRABAN. (2004)

A partir deste ponto, já há condições da codificação do primeiro teste. Deverá ser criada a classe *CodigoDeBarras* e o seu comportamento para gerar sua representação numérica que não poderá ser nula e deve possuir quarenta e quatro posições. A figura 47 exibe-o.

```
1 public class TestadorDoCodigoDeBarras {
2     Test()
3     {
4         public void representacaoNumericaNuncaDeveSerNulaEDevePossuirQuarentaEQuatroPosicoes() {
5             CodigoDeBarras codigoDeBarras = new CodigoDeBarras();
6             assertNotNull(codigoDeBarras.getRepresentacaoNumerica());
7             assertEquals(44, codigoDeBarras.getRepresentacaoNumerica().length());
8         }
9     }
10 }
```

Figura 47 - Criação do TestadorDoCodigoDeBarras.

Fonte: O próprio autor. (2008)

A figura 48 exibe a criação da classe *CodigoDeBarras* e do método necessários para que o teste compile.

```
1 class CodigoDeBarras {
2
3     String getRepresentacaoNumerica() {
4         return null;
5     }
6 }
7 }
```

Figura 48 - Criação do CodigoDeBarras.

Fonte: O próprio autor. (2008)

Após a compilação e execução do teste, obtém-se a falha exibida na figura 49.

Runs: 1/1  Errors: 0  Failures: 1



rdeguias.TestadorDoCodigoDeBarras [Runner: JUnit 4]
NuncaDeveSerNulaEDevePossuirQuarentaEQuatroPosicoes

 Failure Trace

rentaEQuatroPosicoes(TestadorDoCodigoDeBarras.java:11)

Figura 49 - Execução do teste do número de posições do CodigoDeBarras.
Fonte: O próprio autor. (2008)

Modifica-se o comportamento do método *getRepresentacaoNumerica()* de forma que o teste seja executado com sucesso. A figura 50 ilustra esta modificação.

```
class CodigoDeBarras {
    String getRepresentacaoNumerica () {
        return " ";
    }
}
```

Figura 50 - Método *getRepresentacaoNumerica()* retornando String vazia.
Fonte: O próprio autor. (2008)

A ilustração 51 exhibe o resultado da nova execução do teste.

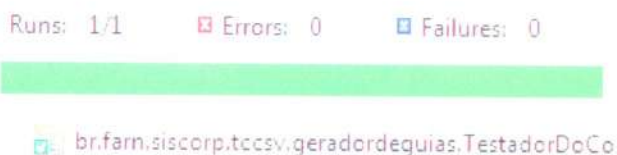


Figura 51 - Execução do teste após representação numérica retorna String vazia.

Fonte: O próprio autor. (2008)

Outro teste válido é verificar se a representação numérica é formada só por números. A figura 52 exibe a adição deste novo teste.

```

@Test
public void representacaoNumericaDeveConterApenasNumeros() {
   CodigoDeBarras codigoDeBarras = new CodigoDeBarras();
    Pattern p = Pattern.compile("[0-9]{44}");
    Matcher m = p.matcher(codigoDeBarras.getRepresentacaoNumerica());
    assertTrue(m.matches());
}

```

Figura 52 - Adição do teste em que a representação numérica deve conter apenas números.

Fonte: O próprio autor. (2008)

Com a execução do teste recém-adicionado, é apresentado um erro, ilustrado pela figura 53.

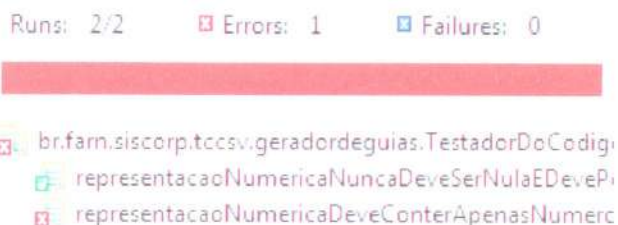


Figura 53 - Falha no teste da representação numérica conter apenas números.

Fonte: O próprio autor. (2008)

Objetivando-se a execução dos testes com sucesso, altera-se novamente o método na figura 54.

```

String getRepresentacaoNumerica() {
    return "123456789012345678901234567890123456789012345678901234";
}

```

Figura 54 - Modificação para que a representação numérica contenha apenas números.

Fonte: O próprio autor. (2008)

Após a modificação, os testes são executados novamente com sucesso, conforme a figura 55.

Runs: 2/2 Errors: 0 Failures: 0



br.farn.siscorp.tccsv.geradordeguias.TestadorDoCo

Figura 55 - Execução dos testes com sucesso após modificação para que a representação numérica contenha apenas números.

Fonte: O próprio autor. (2008)

A partir deste ponto, os testes de composição do código de barras serão adicionados. O quadro 3, extraído da especificação FEBRABAN (2004), detalha essa composição.

Posição	Tamanho	Conteúdo
01-01	1	Identificação do Produto
02-02	1	Identificação do Segmento
03-03	1	Identificação do valor real ou referência
04-04	1	Dígito verificador geral (módulo 10 ou 11)
05-15	11	Valor
16-19	4	Identificação da Empresa/Órgão
20-44	25	Campo livre de utilização da Empresa/Órgão
16-23	8	CNPJ/MF
24-44	21	Campo livre de utilização da Empresa/Órgão

Quadro 3 - Layout do código de barras.

Fonte: Especificação FEBRABAN. (2004)

O primeiro campo a ser considerado é a identificação do produto, que possui tamanho um e ocupado apenas a posição um. O preenchimento deste campo é explicado na figura 56.

Identificação do Produto	Constante "8" para identificar arrecadação
--------------------------	--

Figura 56 - Campo identificação do produto.

Fonte: Especificação da FEBRABAN. (2004)

Como a especificação não menciona outros possíveis produtos fora a arrecadação, será criado apenas o teste para este caso. O teste ocorrerá sobre outra unidade, logo deve ser criado um caso separado como na figura 57.

```

public class TestadorDoProduto {
    @Test
    public void deveRetornarOitoQuandoForArrecadacao() {
        assertEquals("8", Produto.ARRECADACAO.getIdentificacao());
    }
}

```

Figura 57 - Criação do testador do produto.
Fonte: O próprio autor. (2008)

Para que o teste compile, na figura 58 será criado o Enum de produto com o método que retorne a identificação.

```

enum Produto {
    ARRECADACAO("");
    Produto(String identificacao) {
    }

    String getIdentificacao() {
        return null;
    }
}

```

Figura 58 - Criação do enum de produto.
Fonte: O próprio autor. (2008)

Como esperado, o teste em sua execução apresentará uma falha. A figura 59 exibe-a.

Runs: 1/1 Errors: 0 Failures: 1

br.farn.siscorp.tccsv.geradordeguias.TestadorDoPro
 deveRetornarOitoQuandoForArrecadacao

Figura 59 - Falha na execução do primeiro teste do produto.
Fonte: O próprio autor. (2008)

Então, objetivando-se a execução do teste com sucesso, modifica-se o enum *Produto* conforme a figura 60.

```

enum Produto {
    ARRECADACAO("8");
    private String identificacao;
    Produto(String identificacao) {
        this.identificacao = identificacao;
    }

    String getIdentificacao() {
        return identificacao;
    }
}

```

Figura 60 - Mudança no produto para que o teste não falhe mais.
Fonte: O próprio autor. (2008)

Com tal alteração, o teste do *Produto* será executado sem falhas. Seu resultado é apresentado na figura 61.



Figura 61 - Execução do teste do produto sem falhas.
Fonte: O próprio autor. (2008)

Após a construção do produto e sua identificação, possibilita-se a adição de um novo teste para o código de barras. Na situação onde o produto é a arrecadação, a primeira posição da representação numérica deve ser obrigatoriamente oito.

Como a definição do produto é obrigatória em todo código de barras de guias, cria-se o teste com esta definição feita através de um construtor ao invés de um método *setter*. Então, será necessária uma refatoração no teste para eliminar os erros de compilação que surgirão e a duplicação da instanciação do *CodigoDeBarras*. A figura 62 ilustra essas modificações no *TestadorDoCodigoDeBarras*.

```

1 public class TestadorDoCodigoDeBarras {
2
3     private CodigoDeBarras codigoDeBarras;
4
5     //...
6     public void inicializa() {
7         codigoDeBarras = new CodigoDeBarras(Produto.ARRACAÇAO);
8     }
9
10    //...
11    public void representacaoNumericaNuncaDeveSerNulaEDevePossuirQuarentaEQuatroPosicoes() {
12        assertNotNull(codigoDeBarras.getRepresentacaoNumerica());
13        assertEquals(44, codigoDeBarras.getRepresentacaoNumerica().length());
14    }
15
16    //...
17    public void representacaoNumericaDeveConterApenasNumeros() {
18        Pattern p = Pattern.compile("[0-9]{44}");
19        Matcher m = p.matcher(codigoDeBarras.getRepresentacaoNumerica());
20        assertTrue(m.matches());
21    }
22
23    //...
24    public void testaPrimeiraPosicaoDaRepresentacaoNumerica() {
25        assertEquals("1", codigoDeBarras.getRepresentacaoNumerica().substring(
26            0, 1));
27    }
28 }

```

Figura 62 - Novos testes no testador do código de barras após criação do produto.
Fonte: O próprio autor. (2008)

Após a nova execução dos testes, uma nova falha é apresentada no teste recém adicionado. Efetua-se as alterações necessárias na classe *CodigoDeBarras* para que o novo teste seja executado com sucesso. A figura 63 mostra a classe modificada.

```

1 class CodigoDeBarras {
2
3     final private Produto produto;
4
5     CodigoDeBarras(Produto produto) {
6         this.produto = produto;
7     }
8
9     String getRepresentacaoNumerica() {
10        return produto.getIdentificacao() + "1234567890123456789012345678901234";
11    }
12 }

```

Figura 63 - Modificação no código de barras para que ele passe no teste do produto.
Fonte: O próprio autor. (2008)

Como foi utilizado um enum para definir o produto, elimina-se a possibilidade do usuário do componente passar algum produto inválido para o contexto das guias de arrecadação. Esta medida também torna o componente menos vulnerável à falhas.

O segundo campo que integrará o código de barras é o segmento da

empresa ou órgão. A especificação lista os segmentos existentes na figura 64.

Identificação do Segmento	Identificar o segmento e a forma de identificação da Empresa/Órgão
	1. Prefeituras; 2. Saneamento; 3. Energia Elétrica e Gás; 4. Telecomunicações; 5. Órgãos Governamentais; 6. Carnes e Aparentados ou demais Empresas. Órgãos que serão identificados através do CNPJ 7. Multas de trânsito; 8. Uso exclusivo do banco

Figura 64 - Especificação do segmento.

Fonte: Especificação da FEBRABAN. (2004)

Seguindo os mesmos passos feitos para o campo do produto, serão criados o *TestadorDoSegmento* e o enum *Segmento* com cada uma das opções listadas na especificação. Como os passos são muito parecidos aos do produto, as figuras 65, 66 e 67 exibirão apenas as versões finais da classe de teste e do enum.

```

1 public class TestadorDoSegmento {
2     //Test
3     public void deveRetornarUmQuandoForPrefeitura() {
4         assertEquals("1", Segmento.PREFEITURAS.getIdentificacao());
5     }
6
7     //Test
8     public void deveRetornarDoisQuandoForSaneamento() {
9         assertEquals("2", Segmento.SANEAMENTO.getIdentificacao());
10    }
11
12    //Test
13    public void deveRetornarTresQuandoForEnergiaElétricaEGas() {
14        assertEquals("3", Segmento.ENERGIA_ELETRICA_E_GAS.getIdentificacao());
15    }
16
17    //Test
18    public void deveRetornarQuatroQuandoForTelecomunicacoes() {
19        assertEquals("4", Segmento.TELECOMUNICACOES.getIdentificacao());
20    }
21
22    //Test
23    public void deveRetornarCincoQuandoForOrgaosGovernamentais() {
24        assertEquals("5", Segmento.ORGaos_GOVERNAMENTAIS.getIdentificacao());
25    }
26 }

```

Figura 65 - Primeira parte dos testes do segmento.

Fonte: O próprio autor. (2008)

```

43  @Test
44  public void deveRetornarSeisQuandoForCarnesEAssemelhadosOuDemaisIdentificadosPeloCnpj() {
45      assertEquals(
46          "6",
47          Segmento.CARNES_E_ASSEMBELHADOS_OU_DEMAIS_IDENTIFICADOS_PELo_CNPJ
48              .getIdificacao());
49  }
50
51  @Test
52  public void deveRetornarSeteQuandoForMultasDeTransito() {
53      assertEquals("7", Segmento.MULTAS_DE_TRANSITO.getIdificacao());
54  }
55
56  @Test
57  public void deveRetornarNoveQuandoForDeUsoExclusivoDoBanco() {
58      assertEquals("9", Segmento.USO_EXCLUSIVO_DO_BANCO.getIdificacao());
59  }
60  }

```

Figura 66 - Segunda parte dos testes do segmento.

Fonte: O próprio autor. (2008)

```

enum Segmento {
    PREFEITURAS("1"),
    SANEAMENTO("2"),
    ENERGIA_ELETRICA_E_GAS("3"),
    TELECOMUNICACOES("4"),
    ORGAOS_GOVERNAMENTAIS("5"),
    CARNES_E_ASSEMBELHADOS_OU_DEMAIS_IDENTIFICADOS_PELo_CNPJ("6"),
    MULTAS_DE_TRANSITO("7"),
    USO_EXCLUSIVO_DO_BANCO("9");

    private String identificacao;

    Segmento(String identificacao) {
        this.identificacao = identificacao;
    }

    String getIdificacao() {
        return identificacao;
    }
}

```

Figura 67 - Segmento após modificações para atender os testes.

Fonte: O próprio autor. (2008)

Após a criação do testes e dos segmentos, possibilita-se a adição de um novo teste ao testador do código de barras. Como a definição do segmento é obrigatória assim como o produto, ela será feita também através do construtor. Isso exige primeiramente uma refatoração na classe *TestadorDoCodigoDeBarras*, conforme as figuras 68 e 69.

```

1  public void inicializa() {
2      codigoDeBarras = new CodigoDeBarras(Produto.ARECADACAO, Segmento.PREFEITURAS);
3  }

```

Figura 68 - Criação do inicializa() no TestadorDoCodigoDeBarras.

Fonte: O próprio autor. (2008)

```

1  public void testaSegundaPosicaoDaRepresentacaoNumericaQuandoPrefeitura() {
2      assertEquals("1", codigoDeBarras.getRepresentacaoNumerica().substring(1, 2));
3  }

```

Figura 69 - Adição do teste do segmento.

Fonte: O próprio autor. (2008)

O último teste adicionado apresenta uma falha quando executado. Para evitar esta falha, a classe *CodigoDeBarras* é modificada e exibida na figura 70.

```

1  class CodigoDeBarras {
2
3      final private Produto produto;
4      final private Segmento segmento;
5
6      CodigoDeBarras(Produto produto, Segmento segmento) {
7          this.produto = produto;
8          this.segmento = segmento;
9      }
10
11      String getRepresentacaoNumerica() {
12          return produto.getIdentificacao() + segmento.getIdentificacao()
13              + "345678901234567890123456789012345678901234";
14      }
15  }

```

Figura 70 - Modificação no CodigoDeBarras para passar no teste do segmento.

Fonte: O próprio autor. (2008)

O próximo campo a ser integrado ao código de barras é o identificador de valor efetivo ou referência. A figura 71 exhibe a especificação do mesmo.

Identificador de Valor Efetivo ou Referência	Este campo terá: "6" - Valor a ser cobrado efetivamente em reais com dígito verificador calculado pelo módulo 10 na quarta posição do Código de Barras e valor com 11 posições (versão 2 e posteriores) sem qualquer alteração. "7" - Quantidade de moeda Zeros - somente na impossibilidade de utilizar o valor; Valor a ser reajustado por um índice com dígito verificador calculado pelo módulo 10 na quarta posição do Código de Barras e valor com 11 posições (versão 2 e posteriores). "8" - Valor a ser cobrado efetivamente em reais com dígito verificador calculado pelo módulo 11 na quarta posição do Código de Barras e valor com 11 posições (versão 2 e posteriores) sem qualquer alteração. "9" - Quantidade de moeda Zeros - somente na impossibilidade de utilizar o valor; Valor a ser reajustado por um índice com dígito verificador calculado pelo módulo 11 na quarta posição do Código de Barras e valor com 11 posições (versão 2 e posteriores).
--	---

Figura 71 - Especificação do tipo de valor.

Fonte: Especificação da FEBRABAN. (2004)

Os passos para a criação do teste para o tipo de valor e o respectivo enum serão também bastante parecidos com os feitos para o produto e o segmento. Portanto, serão exibidos apenas o resultado final dos testes e do enum nas figuras 72 e 73.

```

public class TestadorDoTipoDeValor {
    // ...

    public void devePossuirIdentificacaoSeisQuandoEfetivoComDigitoVerificadorCalculadoPeloModuloDez() {
        assertEquals(
            "6",
            TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_DEZ
                .getIdentificacao();
        );
    }

    public void devePossuirIdentificacaoSeisQuandoReferenciaComDigitoVerificadorCalculadoPeloModuloDez() {
        assertEquals(
            "06",
            TipoDeValor.REFERENCIA_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_DEZ
                .getIdentificacao();
        );
    }

    public void devePossuirIdentificacaoOitoQuandoEfetivoComDigitoVerificadorCalculadoPeloModuloOnze() {
        assertEquals(
            "8",
            TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_ONZE
                .getIdentificacao();
        );
    }

    public void devePossuirIdentificacaoNoveQuandoReferenciaComDigitoVerificadorCalculadoPeloModuloOnze() {
        assertEquals(
            "08",
            TipoDeValor.REFERENCIA_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_ONZE
                .getIdentificacao();
        );
    }
}

```

Figura 72 - Testes do tipo de valor.

Fonte: O próprio autor. (2008)

```

enum TipoDeValor {
    EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_DEZ("6"),
    REFERENCIA_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_DEZ("06"),
    EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_ONZE("8"),
    REFERENCIA_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_ONZE("08");

    private String identificacao;

    TipoDeValor(String identificacao) {
        this.identificacao = identificacao;
    }

    String getIdentificacao() {
        return identificacao;
    }
}

```

Figura 73 - Versão final do enum TipoDeValor.

Fonte: O próprio autor. (2008)

Como a definição do tipo de valor também é obrigatória, ela será testada através do construtor do código de barras. As figura 74 e 75 mostram as mudanças no testador do código de barras.

```

//define
public void inicializa() {
    codigoDeBarras = new CodigoDeBarras(
        Produto.ARRECADACAO,
        Segmento.PREFEITURAS,
        TipoDeValor.EFETIVO_CCM_DIGITO_VERIFICADOR_CALCULADO_PELC_MODULO_DEZ);
}

```

Figura 74 - Mudança no inicializa() para suporte tipo de valor.

Fonte: O próprio autor. (2008)

```

//=
public void testaTerceiraPosicaoDaRepresentacaoNumericaQuandoValorEfetivoEDigitoVerificadorComModuloDez() {
    assertEquals("6", codigoDeBarras.getRepresentacaoNumerica().substring(
        2, 3));
}

```

Figura 75 - Criação do teste para o tipo de valor.

Fonte: O próprio autor. (2008)

A figura 76 exibe as mudanças necessárias na classe *CodigoDeBarras* para que o novo teste seja executado sem falhas.

```

class CodigoDeBarras {
    //
    final private Produto produto;
    final private Segmento segmento;
    final private TipoDeValor tipoDeValor;

    CodigoDeBarras(Produto produto, Segmento segmento, TipoDeValor tipoDeValor) {
        this.produto = produto;
        this.segmento = segmento;
        this.tipoDeValor = tipoDeValor;
    }

    String getRepresentacaoNumerica() {
        return produto.getIdentificacao() + segmento.getIdentificacao()
            + tipoDeValor.getIdentificacao()
            + "456789012345678901234567890123456789012345";
    }
}

```

Figura 76 - Mudanças em CodigoDeBarras para passar no teste de tipo de valor.

Fonte: O próprio autor. (2008)

Conforme a composição do código de barras, o próximo campo a ser testado seria o dígito verificador, também conhecido como dígito de auto-conferência (DAC). Como seu cálculo depende do preenchimento dos demais campos que constituem o código de barras, ele será testado e implementando apenas após os demais campos.

O próximo campo constituinte do código de barras é o valor. Segundo a especificação, ele deve possuir no máximo onze dígitos. São feitos testes simples apenas para garantir que o software aceita somente valores numéricos com onze

dígitos. Os testes e o código resultante são respectivamente exibidos nas figuras 77 e 78.

```

@Test
public void testaQuintaPosicaoDaRepresentacaoNumericaQuandoValorPossuiOnzeDigitos() {
    CódigoDeBarras codigoDeBarras = new CódigoDeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_PEL_MODULO_DEZ,
        "12345678901");
    assertEquals("12345678901", codigoDeBarras.getRepresentacaoNumerica()
        .substring(4, 15));
}

@Test(expected = IllegalArgumentException.class)
public void testeQuintaPosicaoDaRepresentacaoNumericaQuandoValorPossuiMaisDeOnzeDigitos() {
    new CódigoDeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_PEL_MODULO_DEZ,
        "123456789012");
}

@Test(expected=IllegalArgumentException.class)
public void testeQuintaPosicaoDaRepresentacaoNumericaQuandoValorEhPassadoComLetras() {
    new CódigoDeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_PEL_MODULO_DEZ,
        "1234567890a");
}

```

Figura 77 - Adição dos testes do valor.
Fonte: O próprio autor. (2008)

```

final private Produto produto;
final private Segmento segmento;
final private TipoDeValor tipoDeValor;
final private String valor;

CodigoDeBarras(Produto produto, Segmento segmento, TipoDeValor tipoDeValor,
    String valor) {
    this.produto = produto;
    this.segmento = segmento;
    this.tipoDeValor = tipoDeValor;
    verificaSeOValorEhValido(valor);
    this.valor = valor;
}

void verificaSeOValorEhValido(String valor) {
    Pattern p = Pattern.compile("[0-9]{11}");
    Matcher m = p.matcher(valor);
    if (!m.matches())
        throw new IllegalArgumentException("O valor passado é inválido.");
}

```

Figura 78 - Após modificação para passar no teste referente ao valor do CodigoDeBarras.
 Fonte: O próprio autor. (2008)

Após testar e implementar o valor, o próximo campo é a identificação da empresa ou órgão. Sua especificação pode ser vista na figura 79.

Identificação da Empresa Órgão	O campo identificação da Empresa Órgão terá uma codificação especial para cada segmento. Será um código de quatro posições atribuído e controlado pela Febraban, ou as primeiras oito posições do cadastro geral de contribuintes do Ministério da Fazenda. E através desta informação que o banco identificará a quem repassar as informações e o crédito. Se for utilizado o CNPJ para identificar a Empresa Órgão, haverá uma redução no seu campo livre que passará a conter 21 posições. No caso de utilização do Segmento 9, este campo deverá conter o código de compensação do mesmo, com quatro dígitos. Cada banco definirá a forma de identificação da empresa a partir da 10ª posição.
--------------------------------	--

Figura 79 - Especificação da empresa ou órgão.
 Fonte: Especificação da FEBRABAN. (2004)

A especificação do campo identificação da empresa ou órgão afirma que este campo pode assumir quatro ou oito posições, dependendo de quem esteja emitindo a guia. Os testes são implementados de tal forma que obriguem o software a só aceitar identificadores de quatro ou oito dígitos e no caso de utilização do segmento nove, obrigatoriamente o campo terá aceitar apenas quatro dígitos. Estes

testes são exibidos nas figuras 80 e 81.

```

@Test
public void testeDaRepresentacaoNumericaQuandoAIdentificacaoPodeTerQuatroDigitos() {
    CodigodeBarras codigodeBarras = new CodigodeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_99,
        "12345678901", "1234");
    assertEquals("1234", codigodeBarras.getRepresentacaoNumerica()
        .substring(15, 19));
}

@Test
public void testeDaRepresentacaoNumericaQuandoAIdentificacaoPodeTerOitoDigitos() {
    CodigodeBarras codigodeBarras = new CodigodeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_99,
        "12345678901", "12345678");
    assertEquals("12345678", codigodeBarras.getRepresentacaoNumerica()
        .substring(15, 23));
}

```

Figura 80 - Primeira parte dos testes da identificação da empresa ou órgão.
Fonte: O próprio autor. (2008)

```

@Test(expected=IllegalArgumentException.class)
public void testeDaRepresentacaoNumericaQuandoAIdentificacaoNaoPodeTerSeisDigitos() {
    new CodigodeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_99,
        "12345678901", "123456");
}

@Test
public void testeDaRepresentacaoNumericaQuandoAIdentificacaoDeveTerQuatroDigitosPoisOSegmentoEnNove() {
    new CodigodeBarras(
        Produto.ARRECADACAO,
        Segmento.USO_EXCLUSIVO_DO_BANCO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_99,
        "12345678901", "1234");
    assertEquals("1234", codigodeBarras.getRepresentacaoNumerica()
        .substring(15, 19));
}

@Test(expected=IllegalArgumentException.class)
public void testeDaRepresentacaoNumericaQuandoAIdentificacaoNaoPodeSerDiferenteDeQuatroDigitosPoisOSegmentoEnNove() {
    new CodigodeBarras(
        Produto.ARRECADACAO,
        Segmento.USO_EXCLUSIVO_DO_BANCO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_Pelo_MODULO_99,
        "12345678901", "12345678");
}

```

Figura 81 - Segunda parte dos testes da identificação da empresa ou órgão.
Fonte: O próprio autor. (2008)

As modificações necessárias na classe do código de barras para que os novos testes sejam executados sem falhas são exibidas na ilustração 82.

```

CodigoDeBarras(Produto produto, Segmento segmento, TipoDeValor tipoDeValor,
    String valor, String identificacaoDaEmpresaOuOrgao) {
    this.produto = produto;
    this.segmento = segmento;
    this.tipoDeValor = tipoDeValor;
    verificaSeOValorEhValido(valor);
    this.valor = valor;
    verificaSeAIdentificacaoEhValida(identificacaoDaEmpresaOuOrgao);
    this.identificacaoDaEmpresaOuOrgao = identificacaoDaEmpresaOuOrgao;
}

void verificaSeAIdentificacaoEhValida(String identificacaoDaEmpresaOuOrgao) {
    Pattern p4 = Pattern.compile("[0-9]{4}");
    Pattern p8 = Pattern.compile("[0-9]{8}");
    Matcher m4 = p4.matcher(identificacaoDaEmpresaOuOrgao);
    Matcher m8 = p8.matcher(identificacaoDaEmpresaOuOrgao);

    if (segmento == Segmento.USO_EXCLUSIVO_DO_BANCO && !m4.matches())
        throw new IllegalArgumentException("A identificação da empresa " +
            "ou órgão quando o segmento é o nove " +
            "deve conter quatro dígitos.");

    if (!m4.matches() && !m8.matches())
        throw new IllegalArgumentException("A identificação da empresa ou " +
            "órgão deve conter quatro ou oito dígitos.");
}

```

Figura 82 - Modificação no CodigoDeBarras para passar nos testes da identificação da empresa ou órgão.
 Fonte: O próprio autor. (2008)

O próximo elemento que integra-se ao código de barras é o campo livre. Como o próprio nome sugere, é um campo a ser utilizado livremente pelo cliente. Os únicos testes a serem criados são os que definem que o campo livre deve ser formado exclusivamente por números; deve ocupar vinte e cinco posições quando a identificação da empresa ocupar quatro e deve ocupar vinte e uma posições quando a identificação da empresa ocupar oito. Os testes que especificam tais comportamentos são mostrados na figura 83.

Primeiramente, será testado o cálculo do dígito verificador com módulo dez. Segundo a especificação FEBRABAN, para cálculo do dígito verificador módulo dez, que deverá constar na quarta posição do código de barras, deverá se feita a seguinte montagem:

1. Definir uma área auxiliar de 43 posições subdividida em dois campos. O primeiro de três posições deverá conter, o identificador do produto, identificação do segmento e identificador do valor efetivo ou referência. O segundo campo deverá conter as 40 posições restantes;
2. Calcular o módulo 10 das 43 posições;
3. Montar o campo para impressão no Código de Barras, com as três primeiras posições, o DAC já calculado, e as 40 posições restantes;
4. A representação numérica do Código de Barras, deverá ser montada após o cálculo do dígito verificador.

De acordo com o primeiro passo, infere-se a existência de um comportamento adicional no código de barras que receberá dois argumentos. Baseando-se em um exemplo existente na própria especificação e outro extraído de uma conta de água, cria-se o respectivo teste na figura 85.

```

@Test
public void geraDigitoVerificadorModuloDez() {
    assertEquals("1", codigoDeBarras.geraDigitoVerificadorModuloDez("822",
        "00002150452009741232201540396290109608940"));
    assertEquals("8", codigoDeBarras.geraDigitoVerificadorModuloDez("826",
        "0000000896700062030123456000011200880008"));
}

```

Figura 85 - Teste para gerar o dígito verificador pelo módulo 10.
Fonte: O próprio autor. (2008)

Após a implementação e algumas refatorações na classe *CodigoDeBarras*, obteve-se a adição de três métodos para que o teste executasse sem falhas. Os métodos são exibidos na figura 86.

```

String geraDigitoVerificadorModuloDez(String tresPrimeirosCampos,
    String camposRestantes) {
    final int somaMod10 = Modulo.calculaSomaSequencialMod10(
        tresPrimeirosCampos + camposRestantes, 1, 2);
    final byte restoMod10 = calculaRestoMod10(somaMod10);
    byte digitoVerificadorMod10 = calculaDVMod10(restoMod10);
    return String.valueOf(digitoVerificadorMod10);
}

private byte calculaRestoMod10(int somaMod10) {
    final byte restoMod10;
    if (somaMod10 < 10)
        restoMod10 = (byte) somaMod10;
    else
        restoMod10 = (byte) (somaMod10 % 10);
    return restoMod10;
}

private byte calculaDVMod10(byte restoMod10) {
    final byte digitoVerificadorMod10;
    if (restoMod10 == 0)
        digitoVerificadorMod10 = 0;
    else
        digitoVerificadorMod10 = (byte) (10 - restoMod10);
    return digitoVerificadorMod10;
}

```

Figura 86 - Implementação para passar no testes do dígito verificador pelo módulo 10.
Fonte: O próprio autor. (2008)

Segundo a mesma especificação, os mesmos passos são feitos para o cálculo do dígito verificador através do módulo onze. A figura 87 mostra a criação de um teste baseado em um exemplo da própria especificação.

```

@Test
public void geraDigitoVerificadorModuloOnze() {
    assertEquals("0", codigoDeBarras.geraDigitoVerificadorModuloOnze("822",
        "00002150492009741232231154099291109636941"));
}

```

Figura 87 - Teste do dígito verificador pelo módulo 11.
Fonte: O próprio autor. (2008)

Para que o teste seja executado sem falha, implementa-se a lógica necessária em seguida aplica-se diversas refatorações visando um código mais legível. A figura 88 exibe o código após estes passos.

RN - CAERN tomada como exemplo na figura 90.



Figura 90 - Código de barras de uma conta da CAERN.

Fonte: Conta emitida pela CAERN. (2008)

O teste construído é exibido na figura 91.

```

@Test
public void testaRepresentacaoNumericaDivididaEmBlocosComDigitosVerificadores() {
    CodigoDeBarras codigoDeBarras = new CodigoDeBarras(
        Produto.ARRECADACAO,
        Segmento.SANEAMENTO,
        TipoDeValor.EFETIVO_COM_DIGITO_VERIFICADOR_CALCULADO_PELO_MODULO_DEZ,
        "000000000000", "00062130", "123456000011200880003");
    assertEquals("826800000000", codigoDeBarras.getPrimeiroBloco());
    assertEquals("0", codigoDeBarras.getDVDoPrimeiroBloco());
    assertEquals("39670005203", codigoDeBarras.getSegundoBloco());
    assertEquals("9", codigoDeBarras.getDVDoSegundoBloco());
    assertEquals("01234560000", codigoDeBarras.getTerceiroBloco());
    assertEquals("6", codigoDeBarras.getDVDoTerceiroBloco());
    assertEquals("11200880003", codigoDeBarras.getQuartoBloco());
    assertEquals("2", codigoDeBarras.getDVDoQuartoBloco());
}

```

Figura 91 - Teste para o simular a mesma conta da CAERN.

Fonte: O próprio autor. (2008)

Desta forma, apenas visando facilitar o agrupamento dos dados do código de barras com o restante das informações de uma guia de arrecadação, uma classe representando a guia poderia ser criada como na figura 92.

```

public class GuiaDeArrecadacao {
    private String instrucaoParaORecebimento;
    private String vencimento;
    private String valor;
    private String caminhoParaOLogoDaEmpresa;
    private CodigodeBarras codigodeBarras;

    public GuiaDeArrecadacao(String instrucaoParaORecebimento,
        String vencimento, String valor, String caminhoParaOLogoDaEmpresa,
        CodigodeBarras codigodeBarras) {
        super();
        this.instrucaoParaORecebimento = instrucaoParaORecebimento;
        this.vencimento = vencimento;
        this.valor = valor;
        this.caminhoParaOLogoDaEmpresa = caminhoParaOLogoDaEmpresa;
        this.codigodeBarras = codigodeBarras;
    }

    public String getInstrucaoParaORecebimento() {
        return instrucaoParaORecebimento;
    }

    public String getVencimento() {
        return vencimento;
    }

    public String getValor() {
        return valor;
    }

    public String getCaminhoParaOLogoDaEmpresa() {
        return caminhoParaOLogoDaEmpresa;
    }

    public CodigodeBarras getCodigodeBarras() {
        return codigodeBarras;
    }
}

```

Figura 92 - Criação da classe GuiaDeArrecadacao apenas para agrupar dados.
 Fonte: O próprio autor. (2008)

Com uma instância da classe *GuiaDeArrecadacao* o usuário do componente poderia exibir a guia em várias visualizações como em uma página HTML ou um PDF por exemplo.

5 CONCLUSÕES

Este trabalho destacou os problemas que as falhas em um software podem causar, tais como gastos superiores aos orçamentos previstos e danos irreparáveis em softwares críticos. O trabalho propôs a utilização de testes automatizados como uma maneira de minimizar a vulnerabilidade a falhas de um software.

Foram apresentadas estratégias e técnicas de testes, ressaltando-se a importância de maximizar os esforços em testes mais controláveis, como os de caixa branca praticados pelo próprio desenvolvedor. Mostrou-se que embora muitos considerem estes testes como um esforço custoso e desnecessário, há razões sólidas que justificam a adoção dos mesmos como a produção de um código mais coeso e menos acoplado.

O estudo de caso desenvolveu um novo componente para um projeto pré-existente com a utilização do desenvolvimento guiado por testes (TDD). Obteve-se um total de trinta e dois testes para apenas uma classe e três enums, dados comuns ao utilizar-se desta técnica. Durante o desenvolvimento deste componente, percebeu-se claramente os benefícios do uso de TDD como maior confiança no que está sendo feito e a escrita de código apenas para cumprir com os requisitos.

Trabalhos futuros poderiam ser feitos dando continuidade a este trabalho: desenvolver o mesmo componente sem a utilização de testes automatizados e comparar-se a qualidade do código produzido; ou estudar o uso dos testes automatizados aqui produzidos com algumas ferramentas que tendem a agregar ainda mais qualidade como as de integração contínua e análise de cobertura de código.

REFERÊNCIAS

- BECK, Kent. **Test-Driven Development By Example**. [s.l.]: Addison-Wesley, 2003.
- _____. **Black box testing**. New York: John Wiley & Sons, 1995.
- BEIZER, B. **Software testing techniques**. 2.ed. [s.l.]: Van Nostrand Reinhold, 1990.
- BUTTON, Brian. **Another TDD example – Part 1**. [s.l.], 2007. Disponível em: <<http://agileprogrammer.com/oneagilecoder/archive/2007/11/03/23700.aspx>>. Acesso em: 29 out. 2008.
- CASTRO, Vinicius Almeida. Refatoração e TDD, feitos um para o outro. **Java Magazine**. Disponível em: <<http://www.devmedia.com.br/articles/viewcomp.asp?comp=4444>>. Acesso em: 26 out. 2008.
- CROSBY, Philip B. **Quality is free: the art of making quality certain: how to manage quality - so that it becomes a source of profit for your business**. [s.l.]: McGraw-Hill Companies, 1979.
- DEUTSCH, M. **Verification and validation in software engineering**. [s.l.]: Prentice-Hall, 1979.
- FEBRABAN. Layout Padrão de Arrecadação/Recebimento com Utilização do Código de Barras. [s.l.], v.4, 2004. Disponível em <http://www.febraban.org.br/Acervo1.asp?id_texto=195&id_pagina=173&palavra=>>. Acesso em: 03 nov. 2008.
- HINKEL, José Nivaldo. Ariane 5: um erro numérico (overflow) levou à falha no primeiro lançamento. **Boletim SBMAC**. São Paulo, v.2, 1998. Disponível em: <<http://www.sbmec.org.br/bol/bol-2/artigos/ariane5.html>>. Acesso em: 19 jun. 2008.
- IEEE. IEEE Standard 610.12-1990. **IEEE standard glossary of software engineering terminology**. [s.l.]:1990.
- JONES, T. C. **Programming productivity issues for the 80s**. [s.l.]: IEEE Computer Society Press, 1981.

KANER, Cem; FALK, Jack; NGUYEN, Hung Quoc. **Testing computer software**. 2.ed. [s.l.]: International Thomson Computer Press, 1993.

MASSOL, Vicent; HUSTED, Ted. **Junit in action**. [s.l.]: Manning, 2003.

MICROSYSTEMS, Sun. **The java tutorials**. [s.l.], 1995. Disponível em: <http://java.sun.com/docs/books/tutorial/getStarted/application/index.html#MAIN>>. Acesso em: 11 out. 2008.

PATTON, Ron. **Software testing**. [s.l.]: Sams, 2002.

PRESSMAN, Roger S. **Engenharia de software**. São Paulo: Pearson Makron Books, 1995.

TURNER, Clark S.; LEVESON, Nancy. **An investigation of the Therac-25 accidents**. IEEE Computer, v.26, n.7, p.18-41, jul. 1993. Disponível em: http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html>. Acesso em: 21 jun. 2008.

WILLIAMS, Laure. **White box testing**. [s.l.], 2006. Disponível em: <http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf>>. Acesso em: 05 out. 2008.