Título do Trabalho: Um Estudo de Caso de Especificações Ativas de Requisitos de Software

Autor 1: Marcelo de Freitas Andrade

Tema: Engenharia de Software

Total de páginas: 21

1. Título do Trabalho: Um Estudo de Caso de Especificações Ativas de Requisitos de Software

2. Nomes dos autores completos e sem abreviaturas:
Autor 1: Marcelo de Freitas Andrade

3. Tema: Engenharia de Software

4. Resumo do Trabalho: Este trabalho vem apresentar um estudo de caso de utilização das chamadas especificações ativas de requisitos de software no contexto de um ambiente corporativo de desenvolvimento software. Especificações ativas são basicamente a própria documentação do comportamento esperado para o software a ser desenvolvido e que, com o devido tratamento (exemplificado neste trabalho) pode ser utilizada para se identificar o status do andamento das atividades ao longo do desenvolvimento do software; isto tudo sem intervenção humana e com baixo custo. Dessa forma, este trabalho visa mostrar um exemplo de como o uso adequado de ferramentas livres aliado a práticas de desenvolvimento dirigido a comportamento podem impactar positivamente no processo de construção do software, permeando todas as fases do ciclo de desenvolvimento, resultando em vários benefícios agregados como rastreabilidade direta entre requisitos e implementação, execução automática de testes, monitoramento em tempo real das atividades de desenvolvimento, atualização constante e melhoria da qualidade da documentação do software, menor inserção de bugs no código, dentre outros; tudo isso resultando em aumento da eficiência no desenvolvimento de software como um todo.

5. Lista de palavras-chave para catalogação bibliográfica:
Especificações ativas. Desenvolvimento orientado a comportamento. BDD. Automatização. Rastreabilidade. Requisitos de software. Testes funcionais.

Total de páginas: 21

CURRÍCULOS DOS AUTORES

Autor 1: **Marcelo de Freitas Andrade**

Bacharel em Ciência da Computação pelo Centro de Universitário do Pará, CESUPA;

Especialista em Desenvolvimento de Aplicações para Internet e Mestrando em Ciência da Computação pela Universidade Federal do Pará, UFPA;

Analista de sistemas do SERPRO.

SUMÁRIO

1. INTRODUÇÃO

Empresas que trabalham com desenvolvimento de software frequentemente apresentam dificuldades de várias ordens para a realização das atividades de testes em seus projetos. Seja pela escassez de recursos humanos alocados para o planejamento, projeto e execução destas atividades ou pela especificação de requisitos incompletos e ambíguos que prejudicam a condução dos testes.

Segundo [Grubb e Takang, 2003], cerca de 40 a 70% de todos os custos envolvidos durante o ciclo de vida de um sistema de software são referentes às manutenções realizadas pós-entrega. Desta forma. o investimento em testes pode agregar maior qualidade ao produto que está sendo construído, pois se aplicado sistematicamente desde o início do processo de desenvolvimento permite a descoberta de defeitos antes que o software seja homologado pelo cliente, promovendo o aumento da satisfação do mesmo e a diminuição dos custos do projeto como um todo, pois futuramente será menor o investimento em futuras manutenções corretivas.

Principalmente após o período conhecido como “crise do software”, intensificaram-se as buscas por técnicas visando aprimorar a qualidade do software, dentre elas uma metodologia de desenvolvimento que visava aplicar fortemente todas aquelas práticas que heuristicamente contribuíam para o sucesso de um projeto de software, a chamada Programação eXtrema, ou XP. Sendo o teste de software algo que por si só reconhecidamente contribui para a qualidade do software, XP então recomendava que estes fossem renegados às últimas etapas do ciclo de vida do software, mas que fossem criados sempre, e o quanto antes no processo de desenvolvimento, servindo até mesmo como forma de planejar e orientar toda a implementação a ser feita. Esta prática ficou conhecida como Desenvolvimento Dirigido por Testes, ou TDD.

Com o tempo, percebeu-se que os benefícios de TDD relativos à melhoria do projeto, à especificação do funcionamento e até mesmo à documentação de código podiam ser extrapolados para uso não apenas a nível de unidades de implementação como também à própria especificação das funcionalidades esperadas para o software e das expectativas do cliente. Com o auxílio de ferramentas adequadas, os princípios da desta técnica podem ser utilizados para definir requisitos de software de maneira clara, concreta e não-ambígua como forma de código executável, assim promovendo melhor a comunicação e seu entendimento pelos implementadores, dando melhor visibilidade

sobre o trabalho em andamento e efetivamente direcionando todo o desenvolvimento. Aos artefatos com essas características que capturam as funcionalidades desejadas para o software são chamados de documentação ativa ou especificações ativas.

O objetivo deste trabalho é apresentar um relato na prática de como, com o auxílio de ferramentas adequadas, a utilização de especificações ativas podem ajudar a resolver algumas das dificuldades citadas anteriormente para a realização de testes funcionais automatizados. Tudo isso no contexto real de um projeto em andamento dentro do SERPRO. Os conceitos advindos de TDD, como o Desenvolvimento Dirigido por Testes de Aceitação (ATDD) e Desenvolvimento Dirigido por Comportamento (BDD), também serão abordados.

Este conceitos relacionados sobre as técnicas envolvidas, bem como a classificação dos testes de software quanto à sua capacidade de automatização além da definição do escopo deste trabalho são mostrados na seção 2.

As ferramentas livres utilizadas no estudo de caso objeto deste trabalho e a caracterização do propósito de cada uma no contexto deste trabalho são apresentadas na seção 3. Já a seção 4 traz o relato de experiência, apresentando brevemente os problemas enfrentados num cenário atual e a descrição da solução para automatização de testes funcionais com uso de especificações ativas no contexto de um projeto real.

Por fim, não seção 5 são tecidas algumas conclusões e considerações finais sobre a solução adotada além dos possíveis próximos passos a serem seguidos para evolução da solução.

2. CONCEITOS PRINCIPAIS

Nesta seção são apresentados alguns dos conceitos principais relacionados à criação de especificações ativas.

2.1. Validação e verificação

Para [Pfleeger, 2004], a **validação** é todo procedimento que busca assegurar que o sistema de fato implementou os requisitos que se espera que estejam implementados, enquanto que a **verificação** é o que visa garantir que cada função do software está operando conforme o esperado.

Comumente diz-se, em engenharia de software, que a validação indica se está-se construindo o software correto, ao passo que a verificação já nos diz se está-se construindo o software corretamente.

2.2. Teste de software

Segundo a norma ANSI/IEEE 729 *apud* [Koscianki e Soares, 2004], **teste de software** é "o processo de se avaliar um sistema ou um componente de um sistema por meios manuais ou automáticos para verificar se ele satisfaz os requisitos especificados ou identificar diferenças entre resultados esperados e obtidos".

Esta definição da ANSI/IEEE 729 basicamente conceitua teste de software em termos de verificação e validação mas não menciona nada com relação à inteligência. O grau de inteligência envolvido é uma característica importante e que diferencia o que seja conferência do que seja teste em si. Para [Bolton, 2009], **conferência** é algo feito com o intuito apenas de confirmar crenças existentes, sendo relativa a confirmação, verificação e validação, ao passo que **teste** é um procedimento mais amplo, realizado com o objetivo de se encontrar novas informações, estando mais relacionado a exploração, descoberta, investigação e aprendizado.

Neste contexto, Bolton afirma que enquanto conferências são meros atestes que resultam em respostas binárias entre passou ou não passou e que dependem de máquina, teste de software é uma atividade intelectual que demandando certo grau de inteligência por necessitar de conhecimento sobre o universo do sistema e interpretação de resultados, como toda atividade gnóstica.

Esta visão também é compartilhada por [Rothman, 2009] em sua abordagem sobre

as habilidades necessárias para um testador de software. Já [Dinwiddie, 2009] vai além, e considera ainda que inteligência é um atributo necessário tanto à conferência quanto teste de software em si.

2.3. TDD. ATDD e BDD

TDD, ou Desenvolvimento Dirigido por Testes, é uma técnica de desenvolvimento de software na qual os desenvolvedores escrevem testes acerca de uma dada funcionalidade antes mesmo de codificá-las [Teles, 2004].

Angela Li [Li, 2009] assevera que os benefícios advindos do uso de TDD podem ser agrupados em três grandes benefícios de alto nível: *a)* melhoria da qualidade do código, *b)* melhoria da qualidade da aplicação e *c)* melhoria da produtividade dos desenvolvedores. As Tabelas 1, 2 e 3 enumeram os diversos benefícios em baixo nível apontados pela literatura para seus respectivos macro-benefícios.

Tabela 1. Benefícios relacionados à qualidade de código apontados pela literatura (adaptado de [Li, 2009])

Benefício de alto nível	*Melhoria da qualidade do código*
Significados	• *Design simples.* • *Código limpo.* • *Sem duplicação de código.* • *Baixo acoplamento entre módulos de código.* • *Alta coesão entre módulos de código.* • *Maior facilidade de compreensão.*
Fatores que contribuem	• *Os desenvolvedores desenvolvem conhecimento mais preciso sobre o escopo das funcionalidades.*

Tabela 2. Benefícios relacionados à qualidade da aplicação apontados pela literatura (adaptado de [Li, 2009])

Benefício de alto nível	*Melhoria da qualidade da aplicação*
Significados	• *Design simples.* • *Baixas taxas de defeito.* • *Altas taxas de satisfação do cliente com a aplicação.* • *Melhora a passagem da aplicação nos testes funcionais.*
Fatores que contribuem	• *Melhor compreensão dos requisitos.* • *TDD incentiva execução de verificação e validação mais frequentemente.* • *TDD potencializa o aumento da testabilidade.* • *Aumento da cobertura de testes.*

Tabela 3. Benefícios relacionados à produtividade dos desenvolvedores apontados pela literatura (adaptado de [Li, 2009])

Benefício de alto nível	*Melhoria da produtividade dos desenvolvedores*
Significados	• *Menor esforço para desenvolver o produto final do começo ao fim.*
Fatores que contribuem	• *Menor retrabalho nas etapas de teste e correção de erros.* • *Economia no esforço de desenvolvimento inicial.*

Importante ressaltar que TDD é mais uma técnica de codificação, e não de testes [Astels, 2010]. Considerando TDD como uma técnica na qual o programador primeiro especifica e delimita previamente o que deve ser codificado, de forma que a especificação seja utilizada como base para as atividades de codificação, pode-se fazer uma analogia desta prática com a própria especificação de requisitos de software.

[Hendrickson, 2008] conceitua o chamado Desenvolvimento Dirigido por Testes de Aceitação (ou **ATDD**[1]) como uma técnica bem similar, sendo que enquanto em TDD os testes enfocam em resultados da execução esperada de métodos ou trechos de código, em ATDD os mesmos são utilizados em um nível mais elevado para representar as próprias expectativas do cliente quanto ao comportamento que o software deve ter.

Ao possibilitar a utilização de código de testes para representar os requisitos do software, que podem ser capturados de validados junto ao cliente, passou a ser importante fazê-lo também em linguagem natural, ao invés do uso de código de programação. Dan North exemplifica isto muito bem [North, 2006], mostrando o impacto desta mudança de enfoque e na simples forma de escrita do código de testes na evolução da técnica. Ao mapear código de testes a partir de sentenças em linguagem natural, tem-se a técnica que Dan batizou[2] de Desenvolvimento Dirigido por Comportamento, ou **BDD** na sigla em inglês.

Deve-se notar que, além de ser uma evolução de TDD, ao aprimorar a comunicação e poder propiciar melhor colaboração com o cliente e mais interação entre os próprios desenvolvedores, a técnica de BDD também se mostra ainda mais alinhada aos princípios do Manifesto Ágil[3].

1 Há outras siglas para o mesmo conceito. Por exempĺo, Martin Fowler se refere a ele como Desenvolvimento Dirigido por Testes de Histórias, ou STDD (*Story Test Driven Development*).
2 *http://wiki.rubyonrails.org/getting-started/glossary*
3 *http://www.manifestoagil.com.br*

2.4. Critérios de aceite e testes de aceitação

Um conceito também bastante importante é o de critérios de aceite. Segundo [Jain, 2008], **critérios de aceite** são um conjunto de condições que uma representação de requisitos de software deve atender para ser aceita como concluída. Um critério de aceite deve, obrigatoriamente, descrever: o ator, o comportamento e o resultado observável.

Jain ainda destaca que critérios de aceite não são testes. Quando se têm definidos os critérios de aceite de uma funcionalidade além de exemplos concretos de utilização (dados e cenários), tem-se então o que se chama de **testes de aceitação**. A comunidade ágil comumente costuma descrever isto também como **especificação baseada em exemplos**.

2.5. Classificação de testes

A disciplina de teste de software é bastante abrangente. Há várias maneiras de se classificar testes de software, por exemplo. Para os propósitos deste trabalho, na Figura 1 é são apresentados os Quadrantes de Marick[4].

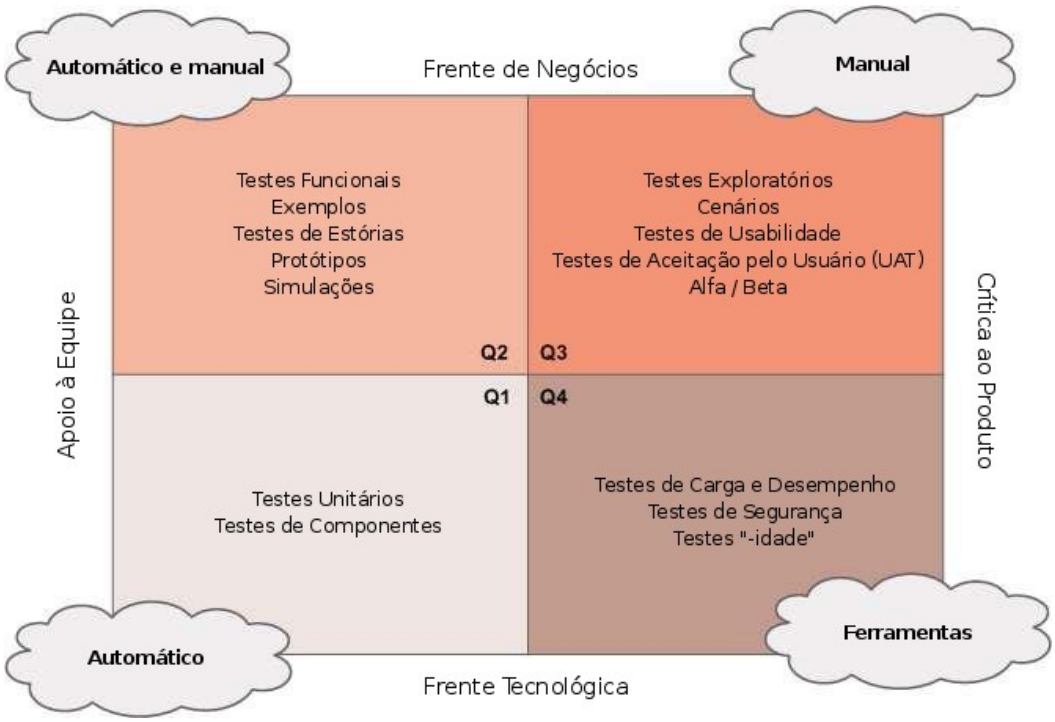


Figura 1. Quadrantes de Marick
[adaptado de Crispim e Gregory, 2009]

Na figura, o eixo horizontal delimita, as validações, acima ---testes descritos como frente de negócios--- das verificações, abaixo ---chamados de testes de frente

4 Crispim e Gregory também descrevem a figura como “Quadrantes de Testes Ágeis”.

tecnológica.

Noutra dimensão, o eixo das ordenadas relaciona, à sua esquerda os testes com finalidade de apoiar o trabalho dos desenvolvedores; e à sua direita, testes cujo foco está exercitar características do produto.

Desta forma, estes quadrantes definidos categorizam toda a miríade de testes possíveis, relacionando-os também à sua capacidade de automatização. Por exemplo, no quadrante Q3, testes exploratórios, execução de cenários, testes de usabilidade, testes de aceitação pelo usuário e testes alfa/beta, todas validações que exercitam características do produto, são testes que, em essência, só podem ser executados de forma manual, ou seja, por um testador humano capacitado.

Na figura também vê-se que testes carga, desempenho e segurança, apenas podem ser realizados com auxílio de ferramentas, o que faz sentido pela características destes testes.

Automatização de testes só pode ser executada em benefício da equipe de implementação. Neste conjunto, depreende-se que testes de frente tecnológica só podem ser executados de forma automática, o que é de fato o caso para testes unitários e de componentes.

Os testes do quadrante Q2 também apoiam o trabalho da equipe de desenvolvimento, mas diferente dos do quadrante Q1, são voltados para a visão do cliente, definindo a qualidade externa e as expectativas destes com relação ao produto.

2.5.1. Escopo do trabalho

O escopo deste trabalho encontra-se precisamente neste quadrante Q2. Mais precisamente nos **testes funcionais** e especificações baseadas em **exemplos**.

Ainda que esta categoria de testes possam ser executados tanto de forma automática quanto manualmente, a abordagem objetivo deste trabalho é, como já dito, na utilização de ferramentas adequadas para sua execução automatizada.

As ferramentas em questão são apresentadas na seção a seguir.

3. FERRAMENTAS UTILIZADAS

Esta seção descreve brevemente as principais ferramentas que são base para a solução abordada neste trabalho.

3.1. JUnit[5]

Segundo sua própria definição, JUnit é um framework simples para escrita de testes repetíveis. É um projeto de código aberto, criado por Erich Gamma (autor de um clássico livro sobre padrões de projeto) e Kent Beck (um dos idealizadores da metodologia XP).

A implementação da biblioteca JUnit está de acordo com o padrão para definição de testes unitários chamado xUnit. O objetivo de uma biblioteca para testes unitários é possibilitar ao implementador definir programaticamente suas expectativas com relação ao retorno esperado de um método de classe ou um determinado trecho de código.

JUnit é uma biblioteca bastante popular, sendo bastante utilizada como ferramenta de apoio à execução de TDD em projetos Java. Para outras linguagens de programação há também outras bibliotecas para testes unitários, como TestNG também para Java, CppUnit para C++, SimpleTest e PHPUnit para PHP, PyUnit para Python, RSpec para Ruby, nUnit (não-livre) para .NET, e outras. Uma listagem mais completa das diversas bibliotecas para execução de testes unitários pode ser encontrada em [Wikipedia, 2010].

3.2. Selenium[6]

Selenium, na verdade, é um grande projeto de software livre, desenvolvido originalmente por uma equipe da empresa ThoughWorks, que abrange um conjunto de várias ferramentas para automatizar o teste de aplicações web em diferentes plataformas.

O Selenium disponibiliza um conjunto rico de funções de teste para atender especificamente às necessidades de teste de uma aplicação web, tais como diversas opções para localizar elementos de interface com o usuário (UI) e comparar o comportamento atual da aplicação web com o que seria o esperado.

O projeto Selenium é composto por diversas ferramentas, dentre as quais:

- **Selenium IDE**

  Uma ferramenta, implementada como plugin do navegador Mozilla Firefox, para

5 *http://junit.sf.net*
6 *http://seleniumhq.org*

gravação de *scripts* de teste capaz de registrar todas as ações básicas feitas por um agente humano em um navegador web (tais como clicar em botões, selecionar opções ou digitar em caixas de texto) para possível posterior reexecução.

- **Selenium Remote Control (RC)**

  É uma ferramenta cliente-servidor multiplataforma que disponibiliza todas as ações capturáveis em um navegador web para serem executadas de forma programática inclusive com a adição de lógica de programação. O servidor é uma aplicação *stand-alone* que é capaz de interagir diretamente com o executável de um navegador web, executando nele as ações programadas a partir de uma biblioteca cliente para uma das diversas linguagens de programação suportadas. Uma representação da arquitetura do Selenium RC pode ser vista na Figura 2.

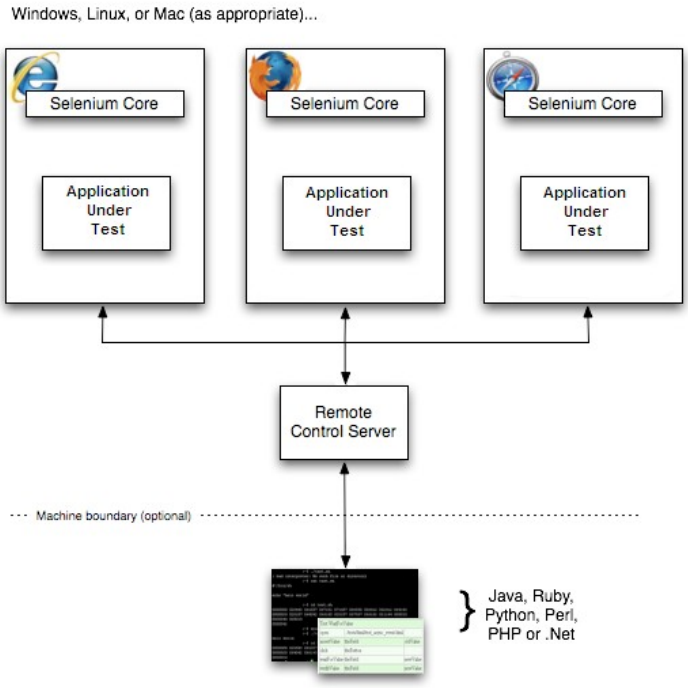


Figura 2. Arquitetura do Selenium RC

- **Selenium Core**

  O Selenium Core é o mecanismo básico que executa as ações na interação com o navegador web. Implementado em HTML e Javascript, está presente internamente tanto no Selenium IDE quanto no RC.

A relação completa das ferramentas que compõem o Selenium está disponível no site do projeto. Um rol de ferramentas de propósito similar em diversos aspetos pode ser visto em [SoftwareQATest, 2010].

3.3. Concordion[7]

Concordion é uma biblioteca de código aberto para auxílio à escrita de testes de aceitação automatizados em Java e algumas outras linguagens de programação.

Na linguagem Java, o Concordion é meramente uma ferramenta que estende a biblioteca JUnit aumentar seu enfoque e representar testes de aceitação em código. É uma ferramenta bastante simples, mas também muito poderosa.

Seu objetivo é possibilitar a escrita de testes de aceitação altamente legíveis, diretamente a partir de especificações funcionais escritas essencialmente em linguagem natural. Por padrão, Concordion obtém como entrada documentos escritos formato HTML.

Concordion é capaz de gerar especificações ativas, sendo uma ferramenta adequada à aplicação de BDD. Concordion oferece a membros da equipe de implementação a capacidade de mapeamento das especificações de funcionalidade do software em código executável de teste, que podem ser escritos desde o princípio, direcionando toda a atividade de desenvolvimento. Com isso, Concordion também permite a rastreabilidade desde a especificação até o código-fonte relativo à implementação correspondente, já incentivando sua cobertura por testes.

Documentos de especificação tratados pelo Concordion são especificações ativas por possibilitarem total visibilidade aos clientes e especialistas em negócio sobre o andamento das atividades, dando um indicativo de forma clara e inequívoca sobre que funcionalidades especificadas já se encontram implementadas.

Um exemplo de uma especificação de funcionalidade tratada pelo Concordion e identificada como especificação ativa é mostrada no estudo de caso na seção 4 a seguir. Algumas ferramentas similares ao Concordion incluem Fit/FitNesse (Java), Robot Framework (Java), Cucumber (Ruby), dentre outras. Ressaltando apenas Concordion também tem implementações em Ruby, Python e .NET.

7 *http://concordion.org*

4. JUNTANDO TUDO: RELATO DE UTILIZAÇÃO EM UM CENÁRIO REAL

Esta seção apresenta um relato de utilização de uma solução para automatização de testes utilizando as ferramentas Concordion e Selenium (e JUnit, indiretamente) no contexto de um dado projeto dentro do SERPRO[8].

4.1. Contexto

O processo que culminou neste trabalho teve início em meados de 2009 com a criação de uma equipe de qualidade de processo e produto. A escassez de recursos humanos para execução de tarefas de teste de software impulsionou iniciativas de automatização de teste. Após alguns meses de tentativas, foi selecionada uma funcionalidade dentro de um projeto piloto conduzido à época no departamento.

A ideia era experimentar o uso das técnicas para automatização com relativamente pouco esforço, cuja experiência pudesse ser relatada para servir de guia para futuras iniciativas, mas também sem prejudicar o andamento do projeto. A partir de algumas poucas experiências próprias, os critérios definidos para seleção de uma funcionalidade em um projeto-piloto foram:

- Regras de negócio já bem conhecidas pela equipe de implementação – para facilitar o esclarecimento de eventuais dúvidas pelos testadores;
- Funcionalidade em teste com relativa simplicidade – para evitar desvirtuar o foco entre o aprendizado das técnicas e ferramentas e o aprendizado de regras de negócio complexas;
- Projeto sem atrasos ou prazos críticos – para minimizar eventuais impactos no projeto e dar certa liberdade enquanto ainda não se tinha maturidade na aplicação das técnicas;
- Casos de teste já projetados – para facilitar o mapeamento das ações em código, uma vez que casos de teste nada mais são do que especificações baseadas em exemplos;
- Interface com o usuário bem definida e simples – para simplificar o desenvolvimento e evitar problemas com eventuais limitações das ferramentas.

4.2. Ações realizadas

Foi selecionado um caso de uso de um dado projeto que atendia bem a todos os critérios definidos. Basicamente tratava-se de uma funcionalidade de pesquisa na qual o usuário deveria consultar um dado CNPJ ou CPF, selecionando uma dada opção e

8 Para não invalidar o trabalho, detalhes de identificação do projeto foram omitidos.

conferir depois os diversos campos de dados retornados pelo sistema. Um esboço do que seria a tela de pesquisa pode ser visto na Figura 3.

Figura 3. Esboço da tela da funcionalidade de pesquisa.

Primeiramente, os primeiros casos de teste que inadequadamente não continham dados foram devidamente tratados. Alguns CNPJs foram gerados e arbitrariamente atribuídos a cada um dos cenários de teste. Depois, o conteúdo de um dos casos de teste foi reescrito como um documento HTML, em um arquivo nomeado de acordo com as convenções do Concordion. Após isto, o arquivo foi apropriadamente tratado, num processo que no jargão da biblioteca é chamado de **instrumentalização**[9].

Apesar de a ferramenta Concordion se propor a mapear testes a partir de texto escrito livremente em HTML, o fato de termos feito isto sobre o caso de teste estruturado facilitou bastante todo o processo (idealmente isto teria sido feito sobre texto livre elaborado a partir de ou em conjunto com o próprio cliente).

Então foi hora de partir para programação. A partir do documento HTML instrumentalizado, criou-se o **fixture** correspondente. Fixture, neste caso, é uma classe Java que mapeia as ações instrumentalizadas no documento no código-fonte executável. Em Concordion, que segue uma abordagem de convenção sobre configuração, para fazer este mapeamento basta deixar o nome da classe Java com o mesmo nome do documento HTML a que se refere, mais o sufixo “Test” (p.ex., supondo que o arquivo do documento HTML neste caso tivesse o nome *Pesquisa.html*, o fixture correspondente deveria se chamar *PesquisaTest.java*). Para seguir o padrão JUnit 3, esta classe deve estender de *ConcordionTestCase*. Já para JUnit 4 o teste não precisa ser subclasse de nenhuma outra, mas deve conter uma anotação com o *runner* adequado, conforme especificado no site.

Note-se que a classe Java do fixture continha como atributo uma instância da classe

9 A instrumentalização em Concordion está descrita em http://concordion.org/Tutorial.html#basics

DefaultSelenium[10] da biblioteca cliente Java (chamado de navegador programável) no qual é feita toda a interação com o navegador para simular as ações do usuário advindas das especificações mapeadas pelo Concordion.

O Concordion essencialmente mapeia métodos do fixture a partir de chamadas descritas em marcações HTML com os seguintes possíveis atributos instrumentalizados:

- **concordion:assertEquals** – verifica se o texto delimitado pela marcação HTML em questão é igual ao retorno do método do fixture apontado como valor do atributo. (p.ex., `<span concordion:assertEquals="retornaBomDia()">Bom dia! </span>`). Há também as versões *-assertTrue* e *-assertFalse*.
- **concordion:set** – atribui uma variável na especificação com o valor de retorno do método do fixture descrito.
- **concordion:execute** – executa um método arbitrário do fixture. Também pode ser usado em tabelas.
- **concordion:verifyRows** – confere o retorno com uma coleção de resultado retornados do fixture como dados presentes nas linhas da tabela onde estiver presente na especificação.

Os comandos disponíveis no Concordion estão documentados (também como especificações ativas) no seguinte endereço: *http://concordion.org/dist/1.3.1/test-output/concordion/spec/concordion/command/Command.html .*

4.3. Ações realizadas

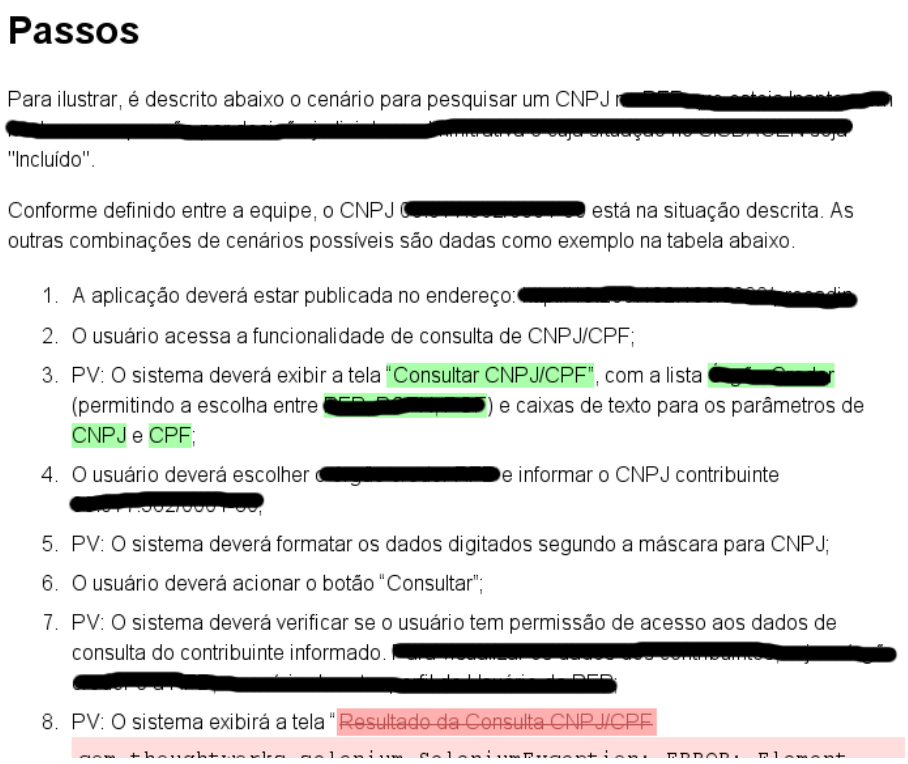
Nesta solução utiliza-se o Selenium RC, que faz todas as interações com o navegador, e que deve estar instalado na máquina que executará a especificação. Havendo um servidor Selenium no ar[11], e se tudo estiver correto, ao se executar a classe Java do fixture como um teste JUnit, na máquina onde o servidor Selenium estiver rodando, navegador com o qual o *DefaultSelenium* foi configurado deve ser aberto, acessar a URL da aplicação em teste, e executar o script conforme definido no corpo dos métodos da classe de fixture.

Ao final da execução, o Concordion exibe no terminal o caminho da versão ativa da

10 O navegador programável do Selenium é instanciado com 4 parâmetros: a URL da aplicação em teste (AUT), o IP ou nome da máquina onde o servidor Selenium está rodando bem como a porta na qual ele escuta, e uma referência a qual navegador deve ser utilizado no teste.

11 Como dito, o servidor Selenium é uma aplicação Java presente no Selenium RC e que pode ser executada, via terminal, com o comando: *java -jar selenium-server.jar*

especificação em HTML executada (o padrão é o mesmo nome de arquivo na pasta temporária do sistema). É uma mesma versão da especificação, mas ao abrir este arquivo informado no navegador web, percebe-se que as marcações em HTML que foram instrumentalizadas com *concordion:assertEquals* (ou alguma das outras asserções) e que correspondiam ao esperado, ficarão com um fundo verde. Já aquelas que divergiram do esperado ou cuja execução gerou alguma exceção também não esperada, estarão destacadas com um fundo vermelho, conforme exemplo abaixo:



Passos

Para ilustrar, é descrito abaixo o cenário para pesquisar um CNPJ [redacted] "Incluído".

Conforme definido entre a equipe, o CNPJ [redacted] está na situação descrita. As outras combinações de cenários possíveis são dadas como exemplo na tabela abaixo.

1. A aplicação deverá estar publicada no endereço: [redacted]
2. O usuário acessa a funcionalidade de consulta de CNPJ/CPF;
3. PV: O sistema deverá exibir a tela "Consultar CNPJ/CPF", com a lista [redacted] (permitindo a escolha entre [redacted]) e caixas de texto para os parâmetros de CNPJ e CPF;
4. O usuário deverá escolher [redacted] e informar o CNPJ contribuinte [redacted];
5. PV: O sistema deverá formatar os dados digitados segundo a máscara para CNPJ;
6. O usuário deverá acionar o botão "Consultar";
7. PV: O sistema deverá verificar se o usuário tem permissão de acesso aos dados de consulta do contribuinte informado. [redacted]
8. PV: O sistema exibirá a tela "~~Resultado da Consulta CNPJ/CPF~~ [redacted]

Figura 4. Exemplo de especificação ativa

5. CONCLUSÕES E PRÓXIMOS PASSOS

Ainda que os benefícios possam ser colhidos ao se aplicar com foco na execução de testes (sendo o principal, o aproveitamento racional de recursos humanos disponíveis para execução de testes funcionais), segundo James Shore [Shore, 2010], o verdadeiro objetivo da técnica de BDD é como técnica para melhoria da comunicação aprimorando a colaboração com o cliente.

Assim, percebe-se que o uso da técnica no cenário dado dentro da empresa ainda tem muito potencial não explorado.

Também nota-se a importância da especificações baseadas em exemplos, isto é, com dados concretos sobre execução em cada caso. Representou uma facilidade manter as ações a serem realizadas já com os respectivos dados a serem utilizados. No entanto, ainda há carência na infraestrutura de gestão de banco de dados para permitir se redefinir a situação do banco de dados de volta à situação esperada mais facilmente e também de forma automática. Em outros ambientes de programação, p.ex. Ruby on Rails, isto pode ser feito facilmente com recursos chamados de *migrations*[12] de banco de dados.

12 *http://guides.rubyonrails.org/migrations.html*

6. REFERÊNCIAS BIBLIOGRÁFICAS

Astels, D. (2010) “A New Look at Test Driven Development”, The Curmudgeoglast, http://techblog.daveastels.com/2005/07/05/a-new-look-at-test-driven-development/ [acesso em 29/08/2010]

Beck, K. (2002) “*Test-Driven Development: By Example*”, Addison-Wesley.

Beck, K. (2004) “*Programação Extrema Explicada: Acolha as Mudanças*”, Bookman, Porto Alegre - RS – Brasil.

Bolton, M. “*Blog: Testing vs. Checking*”. DevelopSense, Michael Bolton. http://www.developsense.com/2009/08/testing-vs-checking.html [acesso em 29/08/2010].

Crispim, L., Gregory, J. (2009) “*Agile Testing: A Pratical Guide For Testers And Agile Teams*”. Addison-Wesley.

Dinwiddie, G. (2009) “*It's Not the Script, It's How You Do It*”, George Dinwiddie's Blog, http://blog.gdinwiddie.com/2009/12/07/its-not-the-script-its-how-you-do-it/ [acesso em 29/08/2010]

Grubb, P., Takang, A. (2003) “*Software Maintenance – Concepts and Pratice*”, second edition.

Hendrickson, E. (2008) “*Dirigindo o Desenvolvimento com Testes: ATDD e TDD*”, In STANZ '08, Software Testing Australia/New Zealand.

Koscianski, A., Soares, M. S. (2006) “*Qualidade de Software: Aprenda as Metodologias e Técnicas Mais Modernas para o Desenvolvimento de Software*”, Novatec, ISBN 85-7522-085-3, São Paulo - SP - Brasil, Maio de 2006.

Li, L. A., (2009) “*Understanding the Efficacy of Test Driven Development*”, Auckland University of Technology, School of Computing and Mathematical Sciences.

Meszaros, G. (2007) “*xUnit Test Patterns: Refactoring Test Code*”, Addison-Wesley.

Naresh, J. “*Acceptance Test Driven Development*”, Slideshare present yourself, http://www.slideshare.net/nashjain/acceptance-test-driven-development-350264 [acesso em 29/08/2010].

North, D. (2006) “*Introducing BDD*”, DanNorth.net It's all behaviour, http://blog.dannorth.net/introducing-bdd/ [acesso em 29/08/2010]

Pettichord, B. (2001) “*Seven Steps to Test Automation Success*”, In. STAR West Conference, San Jose.

Pfleeger, S. L. (2004) “*Engenharia de Software: Teoria e Prática*”, 2ª edição, Prentice Hall, ISBN 85-87918-31-1, São Paulo - SP - Brasil.

Rothman, J. (2009) “*Does Exploratory Testing Have A Place On Agile Teams?*”, In

StickyMinds.com, http://bit.ly/eMfqK [acesso em 29/08/2010]

Shore, J. (2010) "*The Problems with Acceptance Testing*", The Art of Agile(sm) James Shore, http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html [acesso em 29/08/2010]

SoftwareQATest, (2010) "*Web Site Test Tools and Site Management Tools*", http://www.softwareqatest.com/qatweb1.html#FUNC [acesso em 29/08/2010]

Teles, V. M. (2004) "*Extreme Programming*", Novatec. São Paulo - SP - Brasil.

Wikipedia (2010) "*List of Unit Test Frameworks*", Wikipedia the Free Enciclopedia, http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks [acesso em 29/08/2010].