**UNIVERSIDADE DE CAXIAS DO SUL**

**CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA**

**CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

FERNANDO CICCONETO

# Implantação de um Processo de Testes na Empresa Promob

TRABALHO DE CONCLUSÃO DE CURSO

Caxias do Sul

2015

FERNANDO CICCONETO

# Implantação de um Processo de Testes na Empresa Promob

Relatório final apresentado à Universidade de Caxias do Sul, como parte dos requisitos necessários à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Daniel Luís Notari

Caxias do Sul
2015

# Agradecimentos

Agradeço à minha família, amigos, professores e colegas que me apoiaram durante toda a minha formação.

Agradeço também aos meus colegas de equipe da Promob, que me oportunizaram a aplicação deste trabalho na empresa.

# Resumo

TDD é a sigla de Test-Driven Development, ou Desenvolvimento Guiado por Testes (BECK, 2010). É uma técnica de desenvolvimento de software, apresentada como parte dos métodos ágeis de desenvolvimento de software (BECK et al., 2001), onde os testes automatizados são escritos antes do código funcional. A aplicação do TDD visa vários objetivos, tais como possibilitar o entendimento do sistema através da leitura dos testes, não desenvolver código desnecessário, não escrever código sem teste, e facilitar os testes de regressão (FOGGETTI, 2014). Este trabalho tem como objetivo aplicar o TDD no cenário da empresa Promob, de Caxias do Sul (RS), através da elaboração de um processo, que será utilizado por uma equipe de desenvolvimento que já aplica métodos ágeis em sua rotina de trabalho.

**Palavras-chave:** TDD, desenvolvimento ágil, *Scrum,* testes de software, automação de testes.

# Lista de Figuras

# Lista de Tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| ERP | Enterprise Resource Planning (Planejamento de Recursos Corporativos) |
| IDE | Integrated Development Environment (Ambiente de Desenvolvimento Integrado) |
| PBI | Product Backlog Item (Item do Backlog do Produto) |
| PO | Product Owner (Dono do Produto) |
| SQL | Structured Query Language (Linguagem de Consulta Estruturada) |
| TDD | Test-Driven Development (Desenvolvimento Dirigido por Testes) |
| TI | Tecnologia da Informação |
| WIQL | Work Item Query Language (Linguagem de Consulta de Itens de Trabalho) |
| XML | eXtensible Markup Language (Linguagem de Marcação Extensível) |
| XP | eXtreme Programming (Programação Extrema) |

Sumário

# 1 Introdução

O Manifesto Ágil para Desenvolvimento de Software define um conjunto de valores aplicáveis ao desenvolvimento de software. Estes valores devem ser vistos como preferências, e não como alternativas, de forma a encorajar o enfoque de certas áreas, mas sem eliminar outras (AMBLER, 2004, p. 23 e 24). Estes valores são os seguintes (BECK et al., 2001):

- **Indivíduos e interações** valem mais que processos e ferramentas;
- **Um software funcionando** vale mais que documentação extensiva;
- **A colaboração do cliente** vale mais que a negociação de contrato;
- **Responder a mudanças** vale mais que seguir um plano.

Estes valores formam a base das chamadas Metodologias Ágeis de Desenvolvimento de Software, e entre estas metodologias está o *Scrum*. O *Scrum* é um método de desenvolvimento ágil de software baseado em iterações, chamadas de *sprints*. O método, definido por Schwaber e Sutherland (2014), não prescreve o uso de práticas de programação, mas sim fornece um *framework* de gerenciamento de projetos (SOMMERVILLE, 2011, p. 50). O *Scrum* ainda prega que toda a equipe de desenvolvimento deve ser auto-organizada (SCHWABER; SUTHERLAND, 2014, p. 6).

O desenvolvimento dirigido por testes (TDD, do inglês *Test-Driven Development*), definido por Beck (2010), é uma abordagem para o desenvolvimento de software apresentada como parte dos métodos ágeis. No TDD, a escrita de testes automatizados e o desenvolvimento de código funcional são intercalados (SOMMERVILLE, 2011, p. 155), de forma que não existe código sem teste (FOGGETTI, 2014, p. 91).

A automação de testes, por sua vez, é uma abordagem de testes de software onde os testes são embutidos em um programa separado que os executa e invoca o sistema que está sendo testado. Utilizando esta abordagem, é possível rodar centenas de testes em poucos segundos (SOMMERVILLE, 2011, p. 156).

Por fim, testar um software, de forma automatizada ou não, é um procedimento que tem por objetivo encontrar defeitos em um software. Os resultados dos testes podem ser utilizados para medir a qualidade do programa (SOMMERVILLE, 2011, p. 145).

## 1.1 Problema de pesquisa

A Promob Software Solutions é uma empresa sediada em Caxias do Sul (RS) que desenvolve soluções em software para o mercado moveleiro. O contexto deste trabalho está em uma das áreas de desenvolvimento da Promob.

Até o início do ano de 2014, não havia uma metodologia fixa de desenvolvimento nesta área. O principal problema estava na priorização de requisitos, que era pouco eficiente, de forma que a maior parte dos requisitos que chegavam à equipe era considerada urgente, e os requisitos não urgentes acabavam não sendo desenvolvidos, mesmo que fossem importantes para o produto.

Durante o ano de 2014, foi implantado um processo de desenvolvimento baseado em *Scrum*. Este processo abrange:

1. A priorização e seleção dos requisitos a serem desenvolvidos pela equipe;
2. A criação de tarefas, que determinam como estes requisitos são desenvolvidos e entregues;
3. O período de liberações do software;
4. A pré-documentação[1] dos requisitos entregues a cada liberação.

As seguintes características do *Scrum* foram utilizadas:

- *Sprints* fixados em duas semanas;
- *Backlog* de produto;
- Planejamento de *sprints*;
- *Product Owner* e equipe de desenvolvimento;
- Equipe de desenvolvimento auto-organizável.

A partir da implantação deste processo, tornou-se possível desenvolver melhorias[2] que agregaram valor ao produto, aumentando a satisfação e a produtividade dos usuários da ferramenta. Além disto, a cada liberação, é possível coletar *feedbacks* dos usuários e já desenvolver novas melhorias em um curto prazo.

Durante a implantação do processo, a equipe definiu um procedimento de testes de software, onde cada requisito é testado de forma manual pelos desenvolvedores.

[1] Na Promob, a área de suporte técnico recebe esta pré-documentação para escrever o documento de *relase notes*.

[2] Estas melhorias são os requisitos não-urgentes, porém importantes do produto.

Este procedimento é o alvo deste trabalho, pois tem se mostrado pouco eficiente, por dois motivos:

1. Pelo fato de os testes serem manuais, estes testes são demorados para executar. Por consequência, é executado um número insuficiente de testes;
2. Não há um levantamento de casos de teste. A escolha dos testes que serão executados depende da vontade e inspiração de cada desenvolvedor.

A ausência de um procedimento mais eficiente é sentida pela equipe e pelos usuários a cada liberação, onde ocorrem muitos defeitos, sendo vários deles recorrentes.

## 1.2 Questão de pesquisa

Neste cenário, a utilização da técnica TDD no processo de desenvolvimento poderia diminuir a quantidade de defeitos liberados para produção?

## 1.3 Objetivo geral

Definir um processo de testes baseado na técnica TDD que seja aplicável dentro do processo de desenvolvimento da Promob, e desenvolver uma ferramenta para apoiar o processo de testes.

## 1.4 Objetivos específicos

- Identificar o percentual de defeitos que pode ser reduzido através da implantação do novo processo;
- Definir o processo para uso da técnica TDD;
- Definir o novo planejamento de testes;
- Executar um estudo de caso na área de desenvolvimento da empresa Promob;
- Comparar os indicadores de defeitos após o estudo de caso do novo processo.

## 1.5 Estrutura do texto

Este trabalho está organizado em seis capítulos:

- Capítulo 2: revisão dos tópicos que devem ser entendidos para a compreensão deste trabalho;

- Capítulo 3: é detalhado o contexto onde foi realizado este trabalho. Depois, é apresentada a proposta de solução para o problema de pesquisa;
- Capítulo 4: aspectos da implementação da proposta de solução;
- Capítulo 5: estudo de caso de aplicação da proposta de solução;
- Capítulo 6: conclusão do trabalho.

# 2 Revisão Bibliográfica

Este capítulo aborda os tópicos necessários para o entendimento do trabalho. São detalhados os seguintes assuntos: *Scrum,* testes de software, automação de testes de software, TDD, e os trabalhos relacionados.

## 2.1 *Scrum*

O Manifesto Ágil foi lançado em 2001, e é a base ideológica das Metodologias Ágeis de Desenvolvimento de Software, tais como o XP (*eXtreme Programming*) e o *Scrum*. Este manifesto é o resultado de uma reunião entre alguns desenvolvedores de software, realizada com o objetivo de discutir metodologias de desenvolvimento. O Manifesto para Desenvolvimento Ágil de Software (BECK et al., 2001) prega que:

> Estamos descobrindo melhores maneiras de desenvolver software, fazendo-o e ajudando outros a fazê-lo. Ao longo deste trabalho, começamos a valorizar:
>
> - **indivíduos e interações** em vez de processos e ferramentas;
> - **software funcional** em vez de documentação extensiva;
> - **colaboração com o cliente** em vez de negociação de contrato;
> - **respostas à mudança** em vez de seguimento de um plano.
>
> Ou seja, mesmo havendo valor nos itens da direita, valorizamos mais os itens da esquerda.

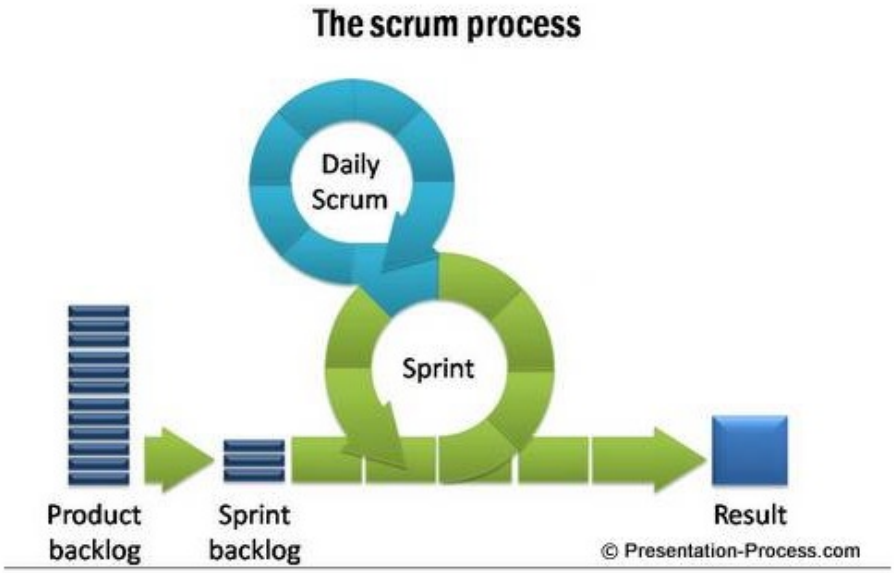
Entretanto, o *Scrum* já havia sido idealizado antes do lançamento do manifesto. Pham e Pham (2011, p. 41) afirmam que “a pedido da Object Management Group (OMG) em 1995, Jeff [Sutherland] e Ken [Schwaber] trabalharam em conjunto para resumir o que haviam aprendido ao longo dos anos; eles criaram uma nova metodologia, que foi chamada de *Scrum*, descrita em um artigo de Schwaber”.

Segundo Sommerville (2011), o *Scrum* “é um método ágil geral, mas seu foco está no gerenciamento do desenvolvimento iterativo”. O desenvolvimento iterativo é uma das principais características do processo.

As iterações do *Scrum* são chamadas de *sprints*. Os *sprints* são iterações planejadas, onde, ao final das mesmas, há a entrega de requisitos. Estas iterações têm um tempo fixo de duração, geralmente entre uma e quatro semanas (PHAM; PHAM, 2011). Com isto, no *Scrum* é possível realizar entregas parciais de um produto em desenvolvimento para os *stakeholders*[1].

[1] Os usuários propriamente ditos, ou os representantes dos usuários.

**Figura 1 – Processo do *Scrum***




Fonte: https://presentation-process-img.s3.amazonaws.com/SCRUM%20POWERPOINT/scrum-process.JPG

A lista de requisitos a serem desenvolvidos está contida no artefato conhecido como *backlog* do produto. Sommerville (2011, p. 50) define que o *backlog* do produto "é a lista do trabalho a ser feito no projeto". Cada um dos itens do *backlog* é chamado de *Product Backlog Item* (PBI). A qualquer momento, podem ser incluídos novos PBI, pois estas inclusões não afetam os *sprints* em andamento.

O *backlog* do produto é de responsabilidade do *Product Owner* (PO). De acordo com Audy (2015, p. 46), as responsabilidades do PO são:

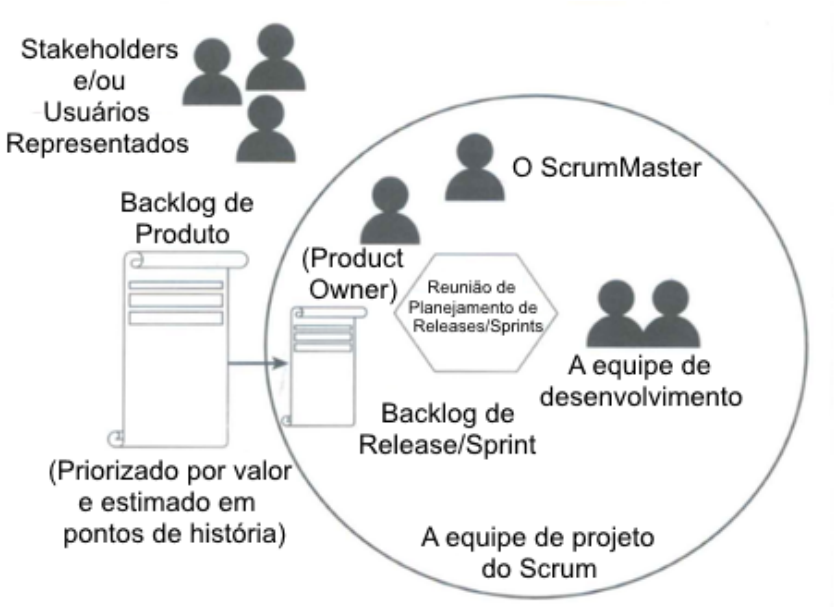
- Registrar com clareza os itens do Backlog do produto;
- Ordenar assertivamente os itens do Backlog do produto;
- Garantir que o Backlog do produto seja visível e claro a todos;
- Garantir que a equipe entenda 100% os itens do Sprint backlog;
- Garantir o valor do trabalho desempenhado pela equipe.

O PO pode ser visto como um gerente de produto, que comunica-se com os usuários, levanta necessidades e define prioridades.

O grupo composto pelas pessoas responsáveis pelo desenvolvimento do produto é conhecido como *equipe de desenvolvimento*. Segundo Schwaber e Sutherland (2014, p. 6), "o time de desenvolvimento consiste de profissionais que realizam o trabalho

de entregar uma versão usável que potencialmente incrementa o produto pronto ao final de cada *sprint*". Ainda de acordo com Pham e Pham (2011, p. 44), "a equipe de desenvolvimento tentará identificar tarefas a partir das histórias previamente escolhidas e deduzir quanto tempo a equipe levará para transformar essas tarefas em incrementos de produto potencialmente entregáveis". Em outros termos, a equipe é responsável por realizar a análise e o desenvolvimento do produto.

**Figura 2 – Atores e artefatos do *Scrum***




Fonte: Pham e Pham (2011, p. 42)

Neste ponto, é importante deixar clara a diferença entre PBI e tarefa. Um PBI é um requisito de usuário, e sempre é descrito como tal. Pham e Pham (2011, p. 43) definem que "os requisitos de usuário para o *backlog* do produto do *Scrum* costumam ser coletados como histórias de usuário curtas". Já uma tarefa é um registro de como e em quanto tempo a equipe irá atender a um PBI. As tarefas são geradas pela equipe a partir de um PBI, sendo que para um PBI pode ser gerada uma ou várias tarefas.

Existe ainda uma pessoa conhecida como *ScrumMaster*, que tem a função de gerenciar o *Scrum* propriamente dito. De acordo com Sommerville (2011, p. 51), "o *ScrumMaster* é um facilitador, que organiza reuniões diárias, controla o backlog de trabalho, registra decisões e mede o progresso comparado ao backlog". É o responsável por implementar e manter o *Scrum*; para isto, ele ensina o *Scrum* para as pessoas envolvidas e responde a eventuais dúvidas. O *ScrumMaster* também atua como líder do processo e das pessoas, resolvendo conflitos que possam atrapalhar o andamento dos *sprints* e do processo. Outra função do *ScrumMaster* é organizar e mediar as reuniões do *Scrum*.

As reuniões que acontecem no *Scrum* são as seguintes (SCHWABER; SUTHERLAND, 2014):

- Planejamento do *sprint*: define-se quais PBI serão atendidos no *sprint*. Logo depois, a equipe de desenvolvimento cria as tarefas e as estima;
- Reunião diária: reuniões rápidas de aproximadamente 15 minutos. Servem para verificar o progresso da equipe durante o *sprint*;
- Revisão do *sprint*: revisa-se o trabalho desenvolvido, bem como verifica-se o que foi desenvolvido e o que não foi desenvolvido;
- Retrospectiva do *sprint*: levanta-se quais foram as lições aprendidas pela equipe durante o *sprint*.

Um aspecto importante do *Scrum* diz respeito à equipe de desenvolvimento. No *Scrum*, a equipe é auto-gerenciável e auto-organizável, ou seja, a equipe decide por conta própria como vai realizar o seu trabalho. Sommerville (2011, p. 51) define que "toda a equipe deve ter poderes para tomar as suas decisões". Schwaber e Sutherland (2014, p. 6) ainda afirmam que "ninguém diz ao time de desenvolvimento como transformar o *backlog* do produto em incrementos de funcionalidades potencialmente utilizáveis". Desta forma, todos os aspectos de análise de sistema e codificação são feitos de forma autônoma pela equipe, incluindo as decisões sobre quem irá realizar quais tarefas.

A equipe de desenvolvimento também é multidisciplinar, ou seja, possui membros para executar qualquer tarefa designada. Desta forma, a equipe muitas vezes é composta por programadores, analistas de sistemas, designers de interface de usuário, DBAs, etc., embora Pham e Pham (2011, p. 42) admitam que frequentemente muitas pessoas da equipe não são membros permanentes e sim "emprestadas dos diferentes departamentos aos quais pertencem".

## 2.2 Testes de software

Antes de iniciar qualquer qualquer estudo sobre testes de software, é importante clarificar a finalidade do ato de testar software. Dijkstra et al. (1972, apud Sommerville (2011, p. 145)) afirma que "os testes podem mostrar apenas a presença de erros, e não a sua ausência". Tendo isto em mente, é possível afirmar que o objetivo de um testador de software é encontrar defeitos. De acordo com Sommerville (2011, p. 144), defeitos são tudo aquilo que provoca "situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações".

Para que o testador atinja o seu objetivo, é necessário que seja elaborada uma estratégia referente à execução dos testes. Uma estratégia de teste de software, por exemplo, pode acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem como testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente (PRESSMAN, 2011, p. 402).

A elaboração da estratégia de testes passa pela escolha dos tipos de testes que serão realizados. Existem vários tipos de testes, que podem ser aplicados em diferentes momentos, por diferentes atores e utilizando diferentes técnicas. Molinari (2008, p. 58) separou os testes de acordo com suas dimensões. Estas dimensões, bem como os tipos de testes, são apresentados a seguir.

### 2.2.1 Primeira dimensão: estado ou momento do teste

Esta dimensão determina qual é o estado de funcionamento do sistema no momento do teste. O sistema pode estar desmembrado em pequenas unidades ou trechos de código, pode estar parcialmente "montado" com alguns de seus módulos, ou ainda pode estar pronto para a implantação (MOLINARI, 2008).

- Teste de unidade: teste em nível de componente ou classe. É o teste cujo objetivo é uma parte do código. Pressman (2011, p. 408) ainda complementa que "o teste de unidade normalmente é considerado um auxiliar para a etapa de codificação";
- Teste de integração: garante que um ou mais componentes, quando combinados, funcionem corretamente;
- Teste de sistema: a aplicação, composta por todos os seus componentes, tem de funcionar como um todo.

### 2.2.2 Segunda dimensão: técnica do teste

Esta dimensão enumera algumas técnicas que podem ser utilizadas para realizar os testes (MOLINARI, 2008).

- Teste operacional: garante que a aplicação pode executar durante muito tempo sem falhar;
- Teste de regressão: garante que uma nova funcionalidade ou alteração não produziu falhas nas funcionalidades já existentes;
- Teste de caixa-preta: quando se utiliza as especificações do sistema para planejar os testes;

- Teste de caixa-branca: quando se utiliza o código-fonte do sistema para planejar os testes;
- Teste alfa: os usuários do software trabalham com a equipe de desenvolvimento para testar o software no local do desenvolvedor;
- Teste beta: testar a aplicação em produção.

### 2.2.3 Terceira dimensão: metas do teste

Esta dimensão diz respeito a quais são os objetivos do testes (MOLINARI, 2008).

- Teste funcional: testar se as funcionalidades presentes na documentação funcionam como especificado, incluindo regras de negócio;
- Teste de interface: verifica se a navegabilidade e os objetos de tela funcionam corretamente, em conformidade com os padrões vigentes;
- Teste de performance: verifica se o tempo de resposta é o desejado para o "momento" de utilização da aplicação e suas respectivas telas envolvidas;
- Teste de carga: verifica se a aplicação suporta a quantidade de usuários simultâneos requeridos;
- Teste de estresse: testa a aplicação em situações inesperadas;
- Teste de configuração: testa se a aplicação funciona corretamente em diferentes ambientes de hardware e software;
- Teste de integridade: testar a integridade dos dados armazenados.

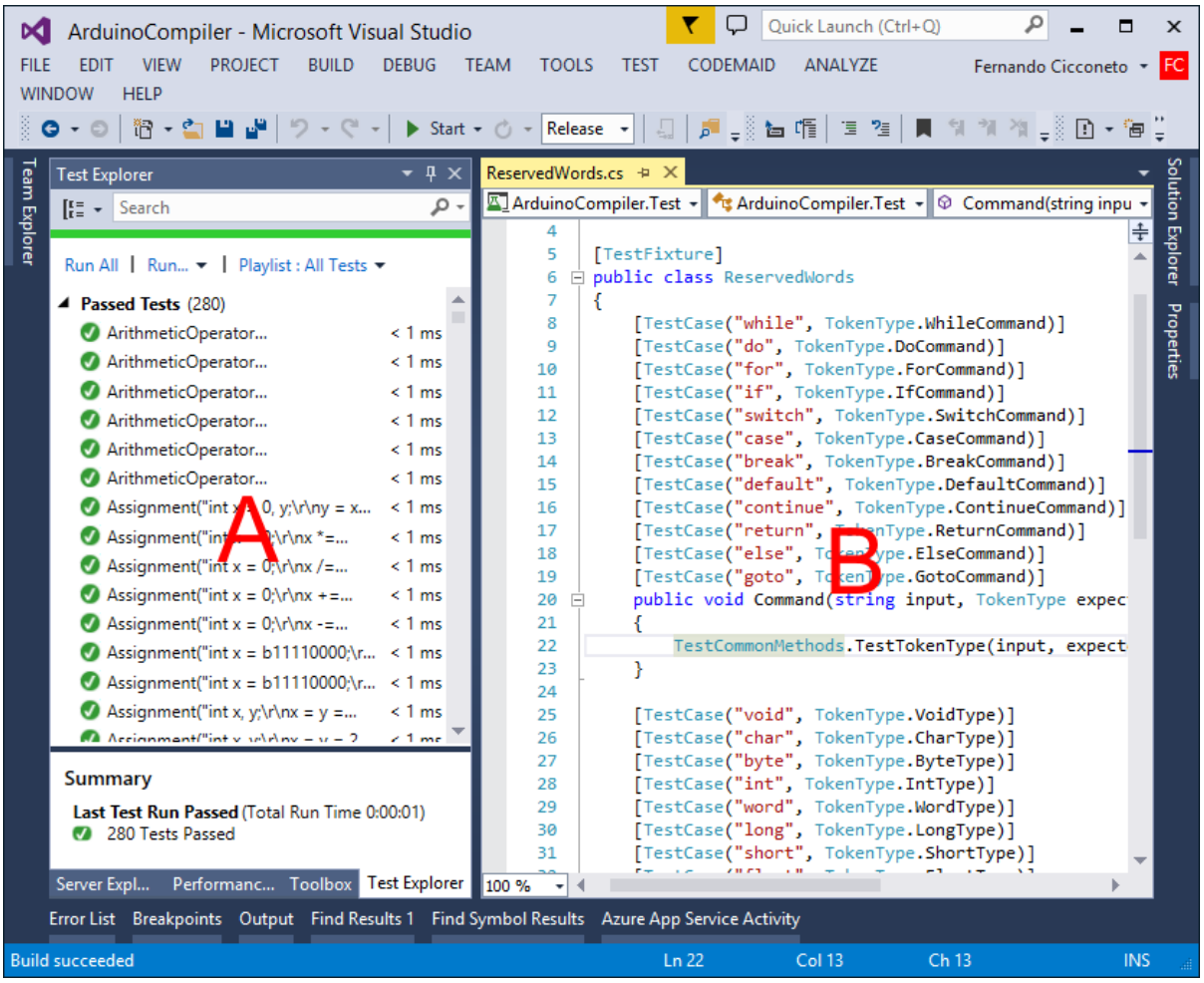
## 2.3 Automação de testes

A automação de testes é uma prática de testes que consiste em codificar um programa que é executado a cada vez que o sistema em desenvolvimento é testado. Essa forma geralmente é mais rápida e assertiva que o teste manual, especialmente quando envolve testes de regressão (SOMMERVILLE, 2011, p. 147).

Sommerville (2011, p. 147) acrescenta que "na prática, o processo de testes geralmente envolve uma mistura de testes manuais e testes automatizados, já que os testes automáticos só podem verificar se um programa faz aquilo a que é proposto". Teste de interface e de configuração, para citar alguns exemplos, podem ser realizados somente através de testes manuais.

Existem *frameworks* de automação de testes que auxiliam o programador na tarefa de elaborar e executar testes automatizados. Estas ferramentas oferecem, entre várias facilidades, a possibilidade de visualizar os resultados dos testes por meio de uma interface gráfica. Desta forma, a cada alteração realizada no sistema, é possível executar os testes rapidamente, podendo-se ainda ver os resultados e analisá-los em caso de falha. Um exemplo destes *frameworks* é o NUnit, ilustrado na figura 3.

**Figura 3 – Microsoft Visual Studio 2013 com o *framework* NUnit**




Fonte: elaborado pelo autor

Na figura 3, é possível visualizar o Microsoft Visual Studio 2013 com o *framework* NUnit em funcionamento. O NUnit é responsável por executar os códigos de testes e exibir, na janela da esqueda (A), os resultados destes testes. Há também um código-fonte de testes aberto (B).

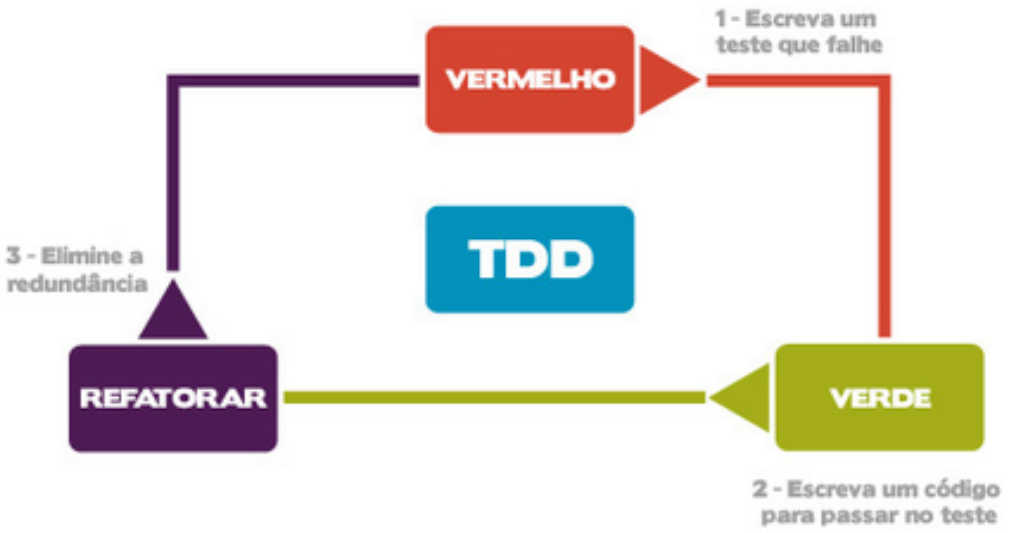
## 2.4 TDD

TDD é a sigla de *Test-Driven Development*, ou Desenvolvimento Guiado por Testes (BECK, 2010). É uma técnica de desenvolvimento de software onde os testes automatizados são escritos antes do código funcional.

São seguidas três tarefas de programação, nesta ordem (BECK, 2010):

1. Vermelho - Escrever um pequeno teste que falhe e que talvez nem mesmo compile inicialmente;
2. Verde - Fazer rapidamente o teste ser aprovado, mesmo cometendo algum pecado necessário no processo;
3. Refatorar - Eliminar todas as duplicatas que foram criadas apenas para que o teste fosse aprovado.

**Figura 4 – Ciclo do TDD**




Fonte: http://arquivo.devmedia.com.br/artigos/Fabio_Gomes_Rocha/TDD/TDD_1.jpg

Estes três passos formam um ciclo, que é repetido até que o desenvolvimento em questão tenha sido concluído.

Foggetti (2014, p. 91) enumera alguns motivos para aplicar o TDD:

- O entendimento do sistema pode ser feito com a leitura dos testes;
- Não é desenvolvido um código desnecessário;
- Não existe código sem teste;
- Uma vez que o teste funciona, ele pode ser usado sempre, inclusive como teste de regressão.

A prática da refatoração no TDD, além de eliminar redundâncias, “visa manter o código simples” (FOGGETTI, 2014, p. 92) e “torna o software mais fácil de entender e modificar” (FOWLER, 2004, p. 53). A refatoração é fácil de ser realizada no TDD, pois é fácil saber se o procedimento gerou algum defeito no sistema, visto que os testes sempre cobrem 100% do código a ser refatorado.

## 2.5 Trabalhos relacionados

Milanez (2014), em sua tese de mestrado, aborda um ponto que será utilizado neste trabalho, que é a forma como os casos de teste no TDD são criados. Milanez (2014, p. 14) argumenta que “o TDD não dispõe nem recomenda critérios para capturar e analisar requisitos de usuários”. Com base nisto, o autor propõe que seja utilizada uma abordagem baseada em casos de uso, objetivando “esclarecer se a inserção de etapas de análise e captura de requisitos, realizadas previamente por meio de casos de uso, enriquece a utilização do TDD” (MILANEZ, 2014, p. 15). A abordagem de Milanez pode ser adaptada para este trabalho, substituindo os casos de uso pelas histórias de usuário contidas nos PBIs.

## 2.6 Considerações finais

Neste capítulo, foram revisados os tópicos fundamentais para a compreensão deste trabalho.

O *Scrum* já é aplicado na empresa Promob, conforme será aprofundado no capítulo 3, enquanto os tópicos de testes de software, automação de testes e TDD serão aplicados a partir do estudo de caso apresentado neste trabalho.

# 3 Proposta de Solução

Este capítulo inicia descrevendo o cenário da empresa Promob. São abordados aspectos do produto desenvolvido, da equipe de desenvolvimento e do processo de desenvolvimento. É detalhada também a questão dos testes, que originou o problema de pesquisa.

Depois, é apresentada uma proposta de processo de testes, e quais ferramentas são utilizadas para apoiar este novo processo.

Por último, é feita a modelagem da ferramenta que será desenvolvida para apoiar a execução do processo.

## 3.1 Cenário atual

A Promob Software Solutions é uma empresa brasileira do ramo de Tecnologia da Informação (TI) sediada na cidade de Caxias do Sul, no estado do Rio Grande do Sul. Possui unidades operacionais localizadas em várias outras cidades brasileiras, como Bento Gonçalves, Florianópolis, Curitiba, São Paulo, Belo Horizonte e Salvador. Possui também filiais em outros países: México, Colômbia e Argentina, além de um revendedor localizado na Espanha. Em todas as suas unidades, a Promob soma mais de 200 colaboradores.

A Promob desenvolve softwares para o mercado moveleiro, oferecendo várias soluções, dentre elas:

- Promob: projeto e orçamento de interiores totalmente em 3D;
- Promob Catalog: cadastro de produtos para serem utilizados pelo Promob;
- Promob Builder: planejamento de fabricação de móveis integrado ao Promob;
- Manager: gestão de lojas de móveis;
- Promob ERP: gestão de fábricas.

A Promob possui uma grande equipe de desenvolvimento, que é dividida em várias áreas, cada uma responsável por desenvolver um segmento de produtos. Cada área tem total autonomia para definir suas tecnologias, arquitetura, plataformas e processos de desenvolvimento.

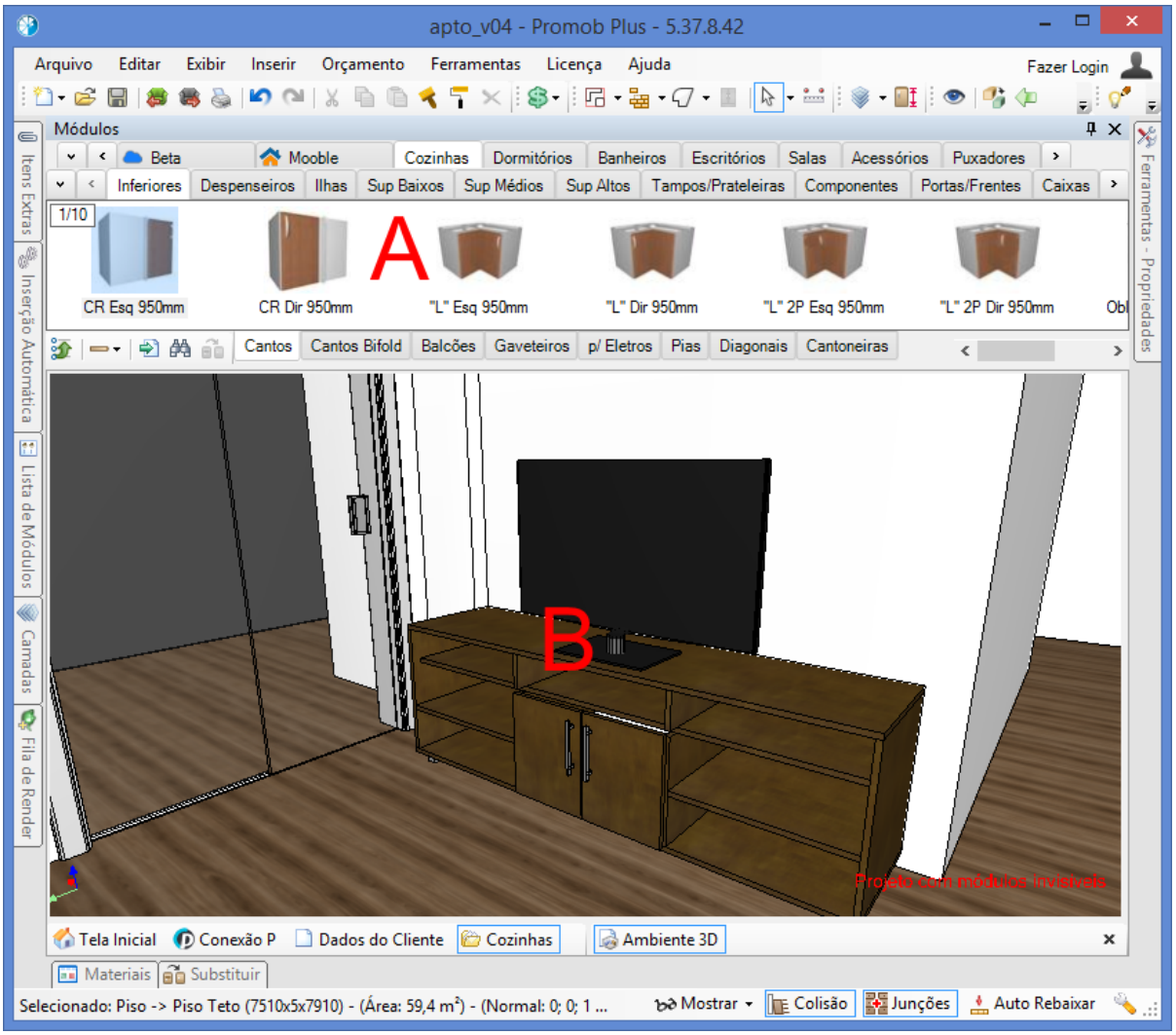
Este trabalho tem como contexto de pesquisa a área responsável por desenvolver o Promob Catalog.

### 3.1.1 Promob Catalog

O Promob Catalog, também conhecido como Catalog, é uma ferramenta de cadastro de produtos para o software Promob.

O Promob é um software de projeto e orçamento de interiores, totalmente em 3D, utilizado por projetistas de interiores, arquitetos, marceneiros, entre outros profissionais da área moveleira.

**Figura 5 – Tela principal de projeto do Promob**



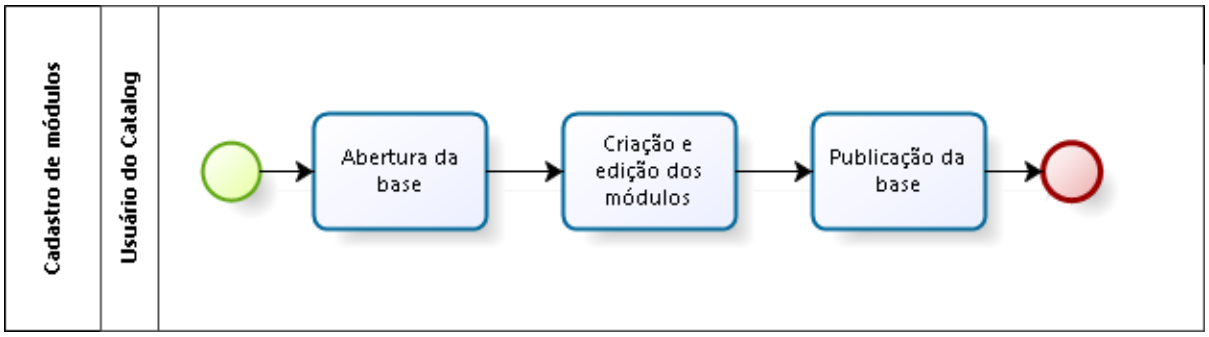

Fonte: elaborado pelo autor

Com o Promob, é possível projetar ambientes internos de forma rápida, através da inserção de módulos prontos no ambiente. Na figura 5, é possível visualizar a tela principal de projeto do Promob, onde os módulos disponíveis estão na parte superior (A) da tela e o ambiente está na parte central (B). Para inserir um módulo qualquer no ambiente, é necessário arrastá-lo de A para B, ou dar um duplo clique sobre o mesmo.

O Catalog, por sua vez, é a ferramenta utilizada para cadastrar estes módulos no Promob. É utilizado tanto de forma interna pela Promob, quanto por fabricantes de

móveis que compram o Catalog para cadastrar seus próprios módulos.

**Figura 6 – Processo de cadastro de módulos com o Promob Catalog**




Fonte: elaborado pelo autor

A figura 6 demonstra como funciona o processo de cadastro dos módulos com o Catalog. Primeiramente, é necessário entender o que é uma base[1]: trata-se de um conjunto de módulos, materiais 3D, tabelas de preços, entre outras configurações. Existem inúmeras bases para o Promob: algumas são desenvolvidas pela própria Promob, como é o caso do Promob Plus, que é uma base voltada para marceneiros; outras são bases próprias dos fabricantes que compram o Catalog. Quando um cliente compra uma licença de Promob, ele escolhe qual base deseja utilizar, dependendo do fabricante com o qual trabalha[2].

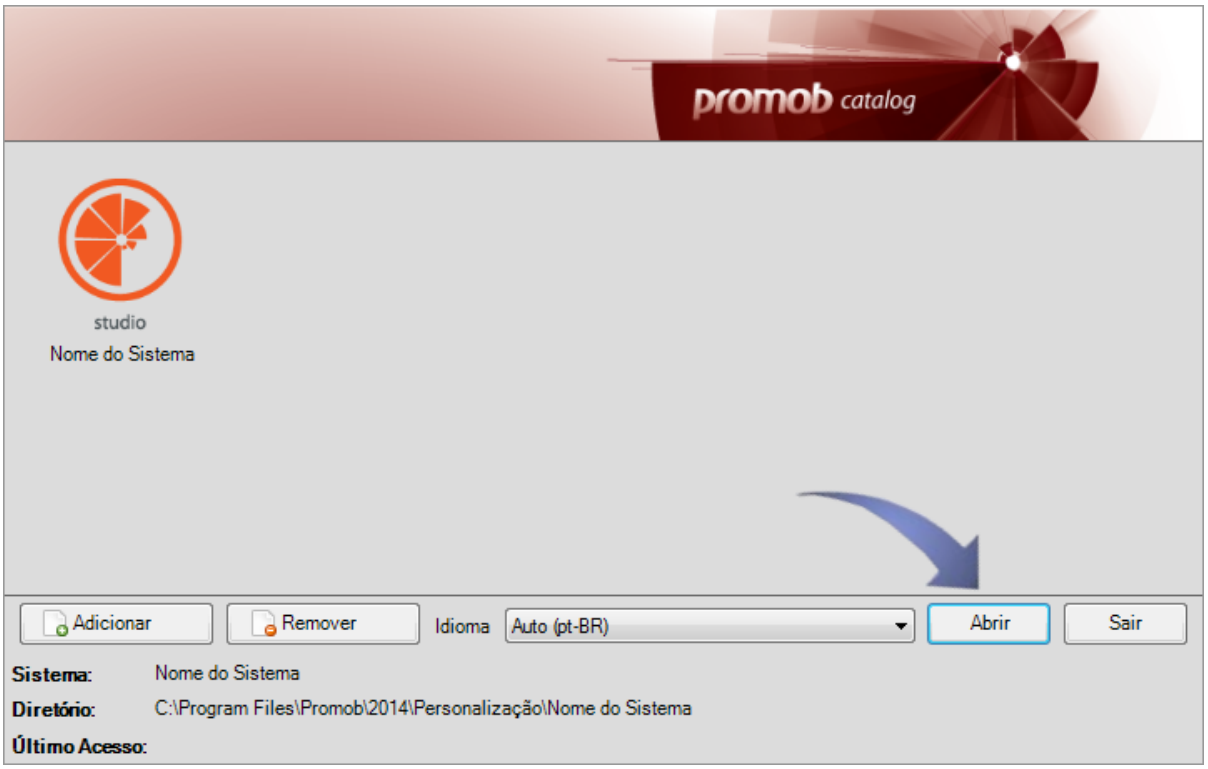
O usuário do Catalog executa os passos da figura 6, que são os seguintes:

1. Abertura da base: o usuário seleciona a base que deseja editar, através da tela ilustrada na figura 7, que é exibida ao abrir o Catalog. O usuário deve ter uma cópia da base que deseja editar em seu computador;
2. Criação e edição dos módulos: após o carregamento da base, o usuário tem acesso a diversas telas de edição, como a que é ilustrada na figura 8;
3. Publicação da base: o Catalog é acompanhado de outra ferramenta desenvolvida pela Promob, que é o Publisher, com o qual o usuário pode publicar a base, tornando as suas alterações disponíveis para os usuários de Promob[3]. O Publisher é ilustrado na figura 9.

O Catalog é totalmente escrito na linguagem de programação C#, utilizando o .NET Framework 4.0. Para o desenvolvimento do Catalog, é utilizada a IDE Microsoft Visual Studio 2013. As bases estão contidas em diretórios que contém arquivos, em sua maioria, XML, e não utilizam bancos de dados.

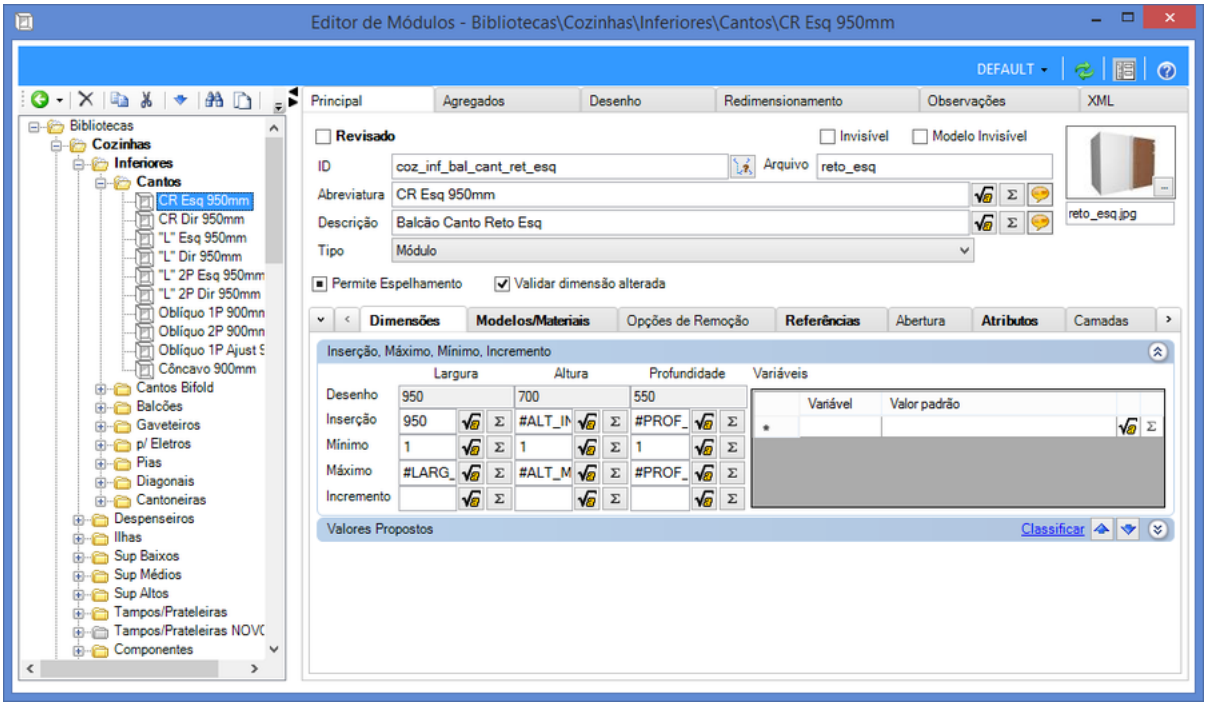
[1] "Base", ou ainda "sistema", são termos definidos pela Promob.
[2] De uma forma geral, marceneiros compram o Promob Plus, arquitetos compram o Promob Arch e lojas de móveis compram bases de fabricantes.
[3] O Promob é auto-atualizado nos clientes a partir do momento em que a base é publicada

**Figura 7 – Tela inicial do Promob Catalog**



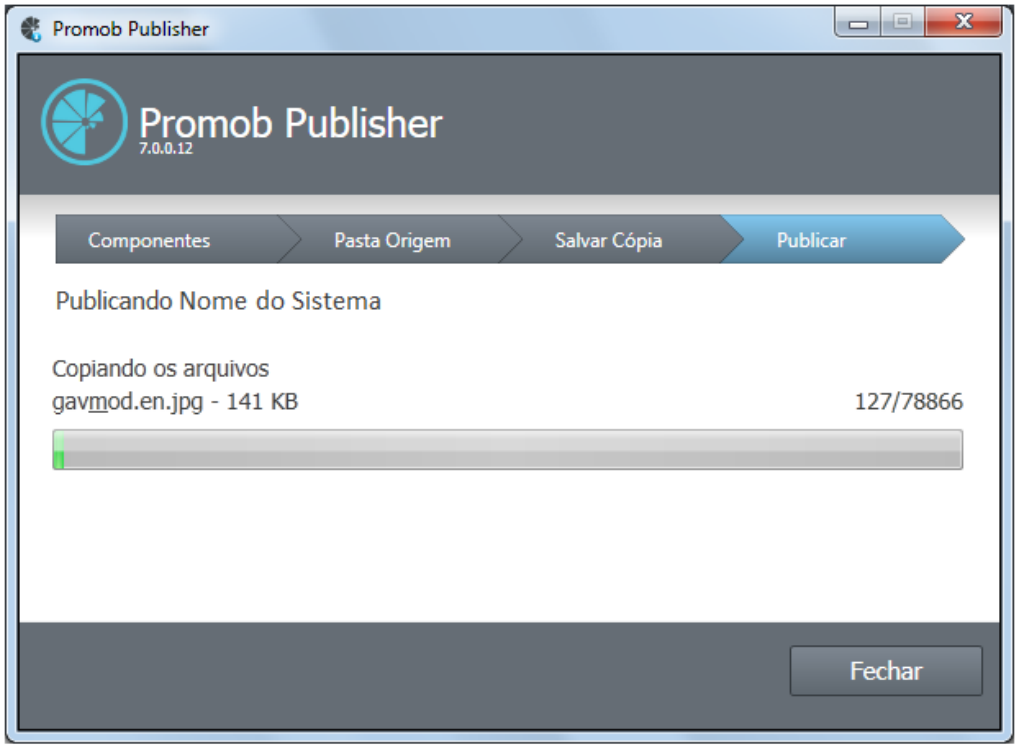

Fonte: http://kb.promob.com/support/pt-br/KB%20%20Externa/Catalog/Inicio/Como_realizar_abertura_do_sistema/Imagem-03.png

**Figura 8 – Tela do editor de módulos do Promob Catalog**




Fonte: elaborado pelo autor

**Figura 9 – Tela do Publisher durante a publicação de uma base**




Fonte: http://kb.promob.com/support/pt-br/KB%20%20Externa/Catalog/Publisher/Como_publicar_sistema/1%20Imagem02.png

### 3.1.2 Equipe do Catalog

A equipe do Catalog é composta pelos seguintes integrantes:

- Um gerente, que é responsável pela gestão da equipe e do produto. É a pessoa que decide o que deve ser feito pela equipe, repassando os objetivos definidos pela diretoria da empresa;
- Um analista de sistemas, que auxilia o gerente nas tomadas de decisão sobre o produto, além de fazer estimativas, liberar versões, definir questões de arquitetura de software, repassar tarefas para os programadores e orientá-los;
- Dois programadores, responsáveis pela codificação do Catalog.

A equipe do Catalog conta ainda com o apoio de uma equipe de suporte técnico, responsável por atender chamadas de clientes de Catalog, pela documentação das alterações realizadas e pela elaboração de documentos de ajuda[4].

[4] A empresa Promob possui uma base de conhecimento on-line que pode ser acessada pelos usuários.

Cabe ressaltar também que apenas os dois programadores trabalham 100% do tempo dedicados ao Catalog; o analista e o gerente também trabalham em outros projetos da área à qual pertencem.

### 3.1.3 Processo de desenvolvimento atual

Durante o ano de 2014, a equipe de desenvolvimento do Catalog adotou um processo de desenvolvimento. Até então, não trabalhava-se com uma metodologia fixa, que definisse processos para a priorização e desenvolvimento de requisitos.

Este aspecto era muito problemático, tanto para a equipe quanto para o produto Catalog e seus usuários. Quando um requisito entrava para a equipe, havia somente duas opções:

1. Se o requisito fosse constatado como defeito, ou então uma necessidade específica de algum cliente forte no mercado, era desenvolvido com urgência;
2. Caso contrário, o requisito era colocado na lista de melhorias sem prazo. Toda a equipe e as pessoas envolvidas acreditavam que um dia o requisito seria desenvolvido, o que raramente acontecia.

Assim, muitos requisitos que eram importantes para o produto e poderiam entregar valor para todos os usuários do Catalog eram colocados em segundo plano. Além disto, as entregas realizadas eram muito defeituosas, pois a urgência do desenvolvimento impedia que fossem feitas refatorações e testes adequados no software. Também não havia um cronograma de liberações, que eram realizadas com muita frequência por falta de planejamento.

A nova metodologia, adotada em 2014, mudou drasticamente a forma como se desenvolve o Promob Catalog. Esta nova forma de trabalhar é baseada no *Scrum*, e passou a definir a maior parte dos procedimentos adotados pela equipe.

O gerente da equipe, por ser encarregado de gerir o Promob Catalog enquanto produto, assumiu o papel de PO. Já o analista e os dois programadores formaram a equipe de desenvolvimento definida pelo *Scrum*. A divisão de papéis na equipe de desenvolvimento continua a mesma: o analista é responsável por fazer apenas a análise do sistema, sem programar, e os programadores são responsáveis pelo código-fonte.

A figura 10 ilustra como ocorre a criação e aprovação de um PBI. As equipes de desenvolvimento e suporte do Promob Catalog têm autonomia para criar novos PBI. Quando um novo PBI é criado, o PO analisa o mesmo, e pode aprová-lo ou reprová-lo. Caso aprove, o PO define um valor numérico de prioridade e coloca o PBI no *backlog* do produto; caso contrário, o PBI é devolvido para o seu autor.

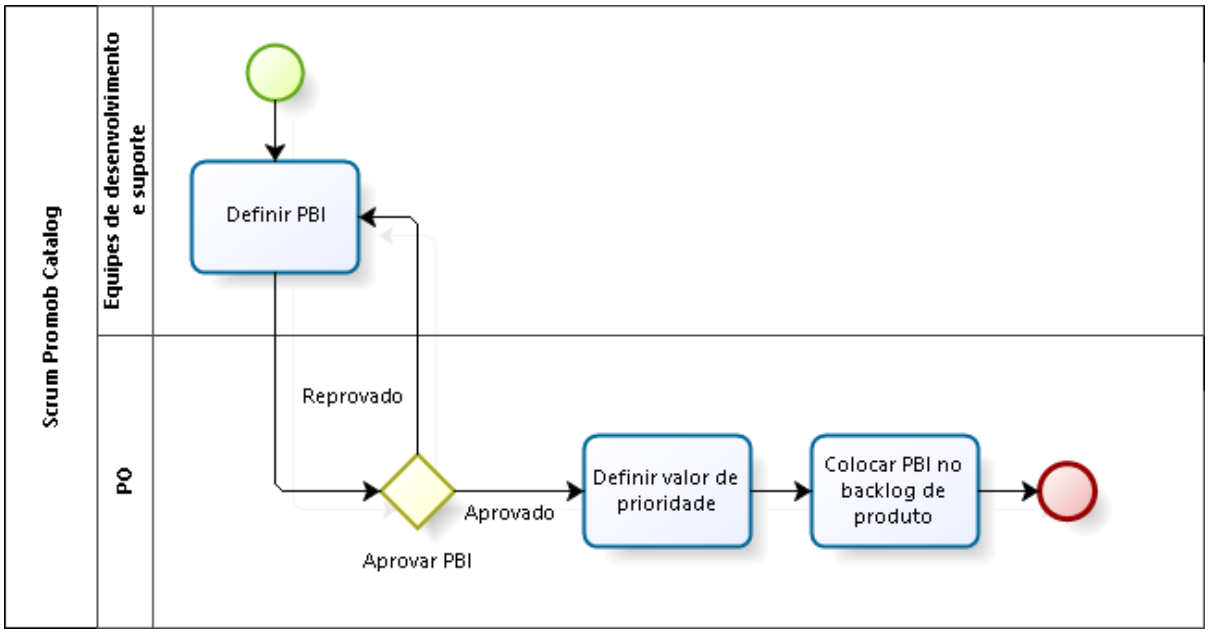
A forma como os *sprints* são definidos é ilustrada na figura 11. Os *sprints* do Promob Catalog têm duração de duas semanas. Antes da data de início do *sprint*, o PO seleciona os PBIs que serão desenvolvidos. Então, a equipe faz as estimativas de cada PBI em pontos de complexidade e verifica se o *sprint* ficou com a soma de pontos correta. Caso faltem ou sobrem pontos de complexidade, o PO irá adicionar ou remover PBIs. No caso em que são adicionados PBIs, a equipe deverá estimar estes novos itens, formando um ciclo que se repete até que o *sprint* tenha o total de pontos de complexidade correto. A partir da data de início do *sprint*, os PBIs são desenvolvidos pela equipe, até que o *sprint* termine.

Os pontos de complexidade são equivalentes a horas de desenvolvimento da equipe. Por exemplo: na figura 12, é possível visualizar um exemplo de PBI com oito pontos de complexidade. Isto significa que foi estimado que este PBI demorará oito horas, ou um dia de trabalho, para ser desenvolvido.

A soma de pontos de complexidade por *sprint* é: número de horas diárias x dias úteis do *sprint* x número de programadores. Isto geralmente resulta em: 8 x 10 x 2 = 160 pontos por *sprint*.

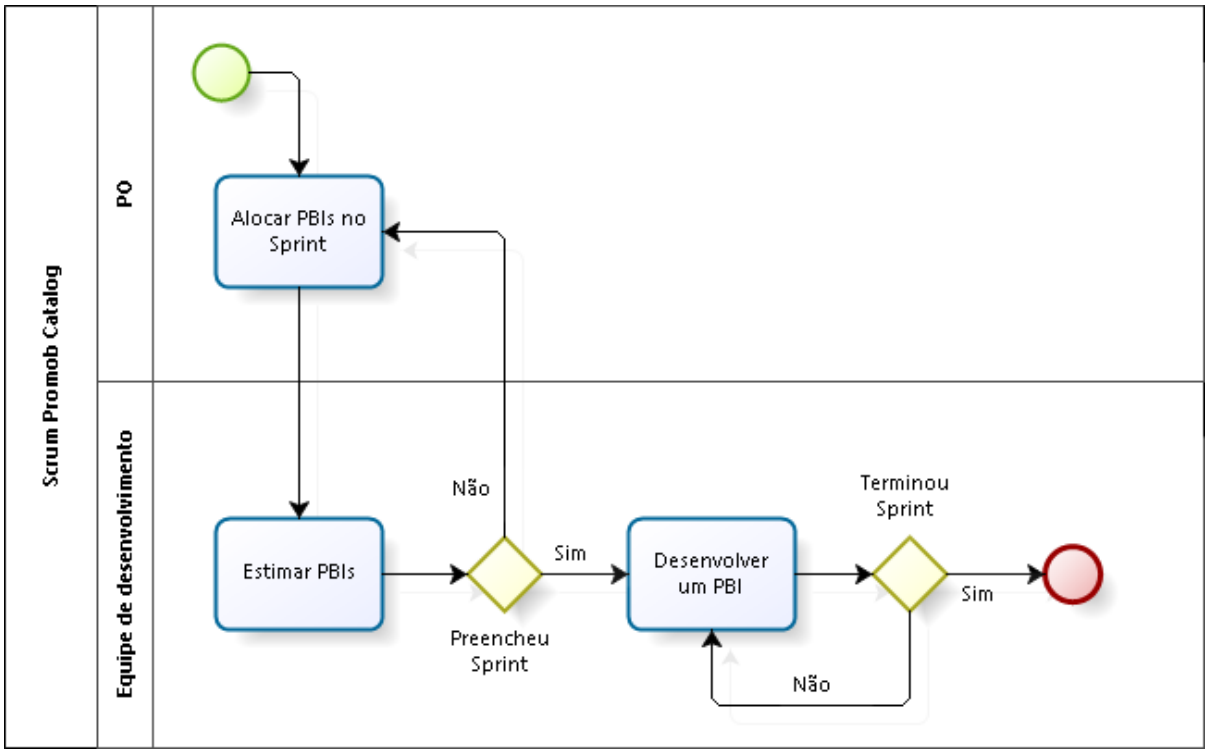
Cada PBI pode ser desenvolvido por um único programador. Ao início do *sprint*, nenhum PBI tem responsável alocado, e a alocação será feita pela equipe ao decorrer do *sprint*.

**Figura 10 – Procedimento de criação e aprovação de um PBI**



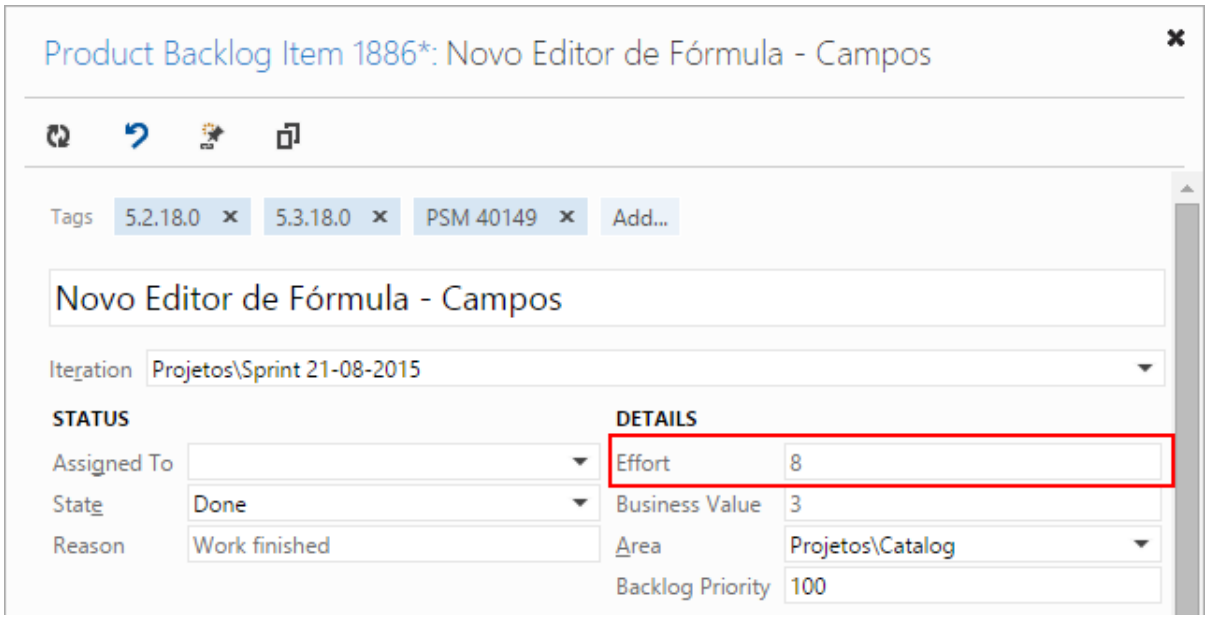

Fonte: elaborado pelo autor

**Figura 11 – Planejamento e execução dos *sprints***



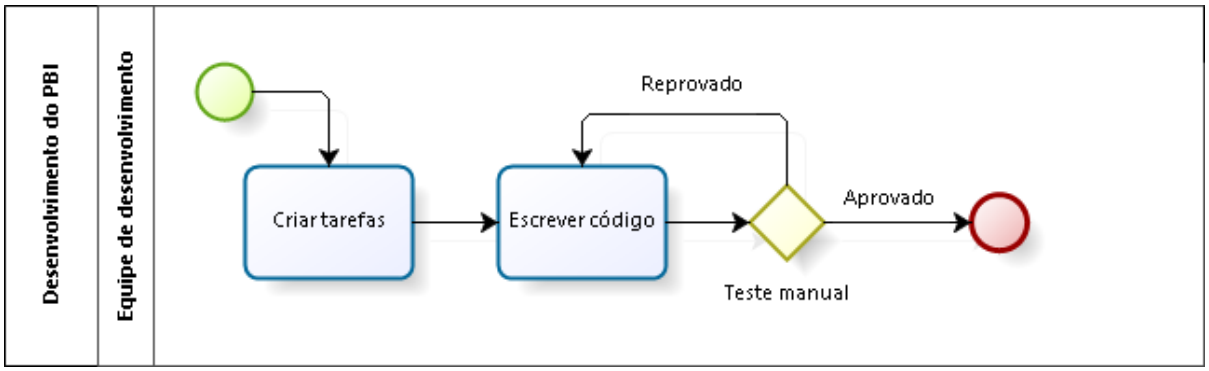

Fonte: elaborado pelo autor

**Figura 12 – Exemplo de PBI com oito pontos de complexidade**




Fonte: elaborado pelo autor

**Figura 13 – Desenvolvimento de um PBI**




Fonte: elaborado pelo autor

A figura 13 ilustra como ocorre o desenvolvimento de um PBI. Ao iniciar o desenvolvimento, o programador responsável cria as tarefas, que são atividades técnicas que levarão ao atendimento dos requisitos descritos no PBI. Após a criação de todas as tarefas do PBI, o programador inicia a codificação. Ao terminar de escrever o código e encerrar as tarefas, ocorre um procedimento de testes.

Os testes sempre são executados pela equipe de desenvolvimento, e caracterizam-se como testes de sistema, pois o Catalog é executado como se estivesse em produção. Com o Catalog em execução, são feitos ao menos três testes: testes funcionais, testes de interface e testes de integridade de dados. Todos estes testes são manuais, e é a própria equipe quem determina, após a codificação, quais testes serão executados, de acordo com a necessidade de cada PBI.

Caso os testes resultem em “aprovado”, o PBI é dado como concluído. Caso contrário, o programador responsável pelo PBI volta a escrever código para corrigir a situação de erro, e os testes são executados novamente, até que os mesmos resultem em “aprovado”.

A exceção ao planejamento dos *sprints* está nos PBIs de correção de defeitos: estes devem ser atendidos com urgência, caracterizando-se como itens não-planejados. Portanto, quando um PBI de correção é registrado, o mesmo é colocado no *sprint* atual e deve ser atendido imediatamente, sendo corrigido diretamente na versão de produção do Catalog. Isto possibilita a liberação imediata da correção.

Outra característica do *Scrum* presente na metodologia do Promob Catalog é a equipe auto-organizável, pois os integrantes da equipe têm autonomia para definir: modelagem de classes e de dados, padrões de codificação, ferramentas para auxiliar no desenvolvimento, etc. O PO define o que deve ser entregue pela equipe; já a equipe de desenvolvimento define como irá realizar as suas entregas.

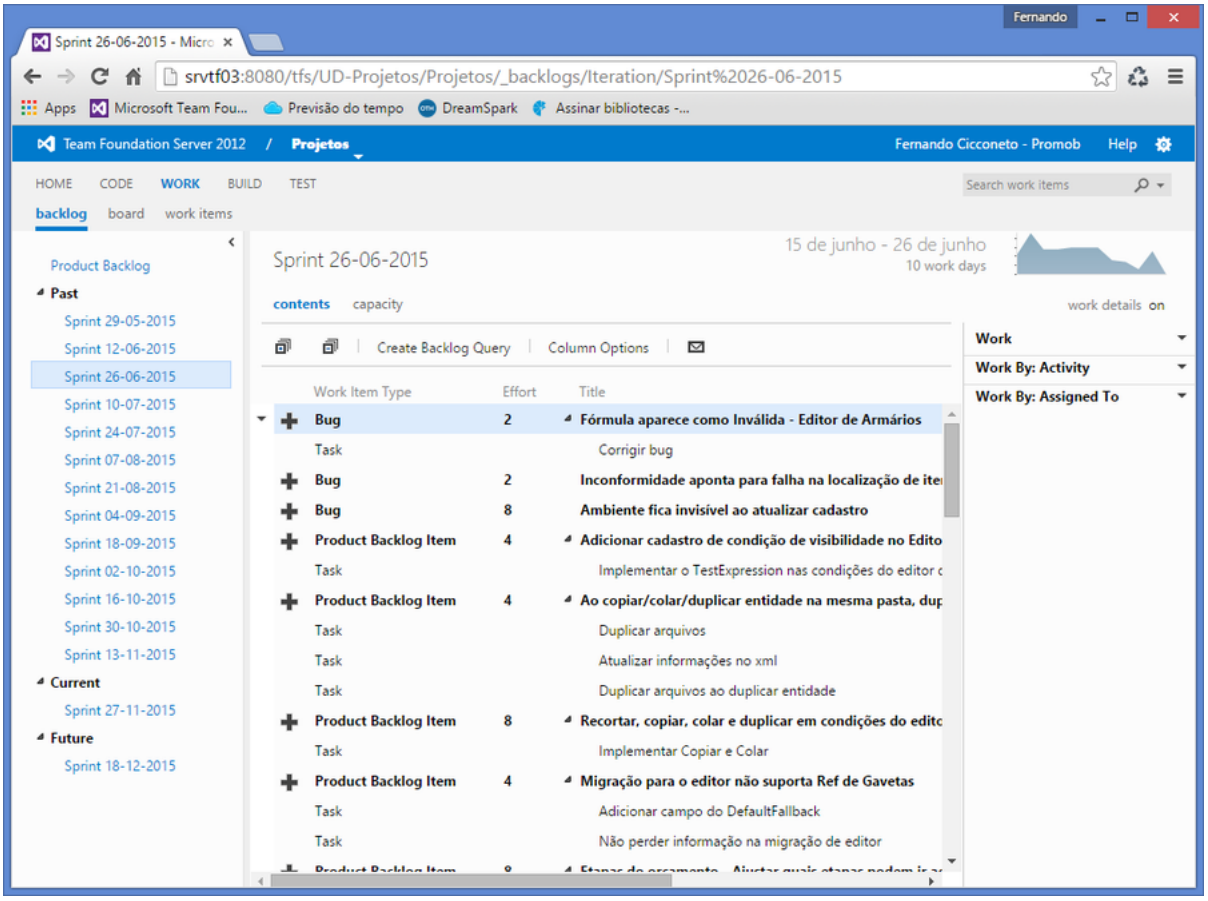
### 3.1.4 Ferramentas do processo

Para gerenciar a metodologia, a equipe do Catalog utiliza a ferramenta do Microsoft Team Foundation Server 2012, que é baseada no *Scrum*. Esta ferramenta fornece todas as funcionalidades que a equipe precisa para realizar a metologia de desenvolvimento:

- *Backlog* de produto;
- Descrição detalhada de requisitos nos PBIs;
- Planejamento e gerenciamento dos *sprints*;
- Criação de tarefas;
- Integração com versionamento de código, ou seja, é possível vincular uma alteração no código-fonte a uma ou várias tarefas.

Todos os fluxos descritos neste tópico são executados com o auxílio do Team Foundation Server.

**Figura 14 – Microsoft Team Foundation Server 2012**




Fonte: elaborado pelo autor

Na figura 14, é possível visualizar a lista de PBIs de um *sprint*. Para cada PBI, é possível visualizar também a lista de tarefas.

### 3.1.5 A questão dos testes

A aplicação da metodologia do Promob Catalog foi bem-sucedida, pois resolveu a maioria dos problemas para os quais foi proposta. Com os *sprints*, tornou-se possível estabelecer um cronograma de liberação de melhorias. O *backlog* do produto tornou possível a priorização de requisitos e a realização de grandes melhorias, que acrescentaram valor e aumentaram a satisfação dos usuários com o produto. O PO está em contato constante com os usuários, coletando *feedbacks* e acrescentado novos requisitos ao *backlog*, possibilitando a melhoria contínua do produto. Já a equipe está estudando formas de melhorar o seu trabalho a todo momento.

Após a consolidação da metodologia, percebeu-se a necessidade de melhorar o processo de testes da equipe. Atualmente, são corrigidos cerca de dez defeitos a cada *sprint*, conforme é possível visualizar na tabela 1.

**Tabela 1 – Quantidade de defeitos corrigidos a cada sprint**

| Data de início | Data de finalização | Defeitos corrigidos |
|---|---|---|
| 18/05/2015 | 29/05/2015 | 10 |
| 01/06/2015 | 12/06/2015 | 11 |
| 15/06/2015 | 26/06/2015 | 8 |
| 29/06/2015 | 10/07/2015 | 12 |
| 13/07/2015 | 24/07/2015 | 11 |
| 27/07/2015 | 07/08/2015 | 12 |
| 10/08/2015 | 21/08/2015 | 12 |
| 24/08/2015 | 04/09/2015 | 10 |
| 08/09/2015 | 18/09/2015 | 9 |

Fonte: elaborado pelo autor

A equipe elaborou um levantamento das causas dos defeitos, que é possível visualizar na tabela 2. De acordo com o levantamento, aproximadamente três em cada quatro defeitos ocorrem por conta de falhas no desenvolvimento e nos testes dos requisitos.

**Tabela 2 – Classificação dos defeitos corrigidos por causa**

| Causa | Quantidade de defeitos | % do total |
|---|---|---|
| Erro na análise ou interpretação do requisito | 21 | 22,1% |
| **Erro no desenvolvimento ou nos testes** | **70** | **73,7%** |
| Outros | 4 | 4,2% |

Fonte: elaborado pelo autor

Então, pode-se inferir que, através da implantação de melhorias no processo de desenvolvimento e testes, teoricamente é possível reduzir em até 73,7% a quantidade de defeitos corrigidos a cada *sprint* pela equipe do Catalog.

Após analisar a situação, a equipe levantou duas falhas no atual processo de desenvolvimento e testes:

1. Pelo fato de os testes serem manuais, estes testes são demorados para executar. Por consequência, é executado um número insuficiente de testes;
2. Não há um levantamento de casos de teste. Embora os requisitos sejam claros e fáceis de testar, não há uma forma tão clara de determinar os testes de regressão.

O foco deste trabalho é propor uma solução para diminuir a quantidade de defeitos corrigida por *sprint*, através da elaboração de um processo de testes que determine uma forma rápida de executar testes funcionais, tanto para os requisitos do *sprint* atual quanto para os testes de regressão.

## 3.2 O novo processo de testes

O novo processo utiliza testes de unidade automatizados para executar os testes funcionais no Catalog, pelo motivo visto na revisão bibliográfica, de que são uma forma mais rápida e assertiva que o teste manual, especialmente quando envolve testes de regressão (SOMMERVILLE, 2011, p. 147).

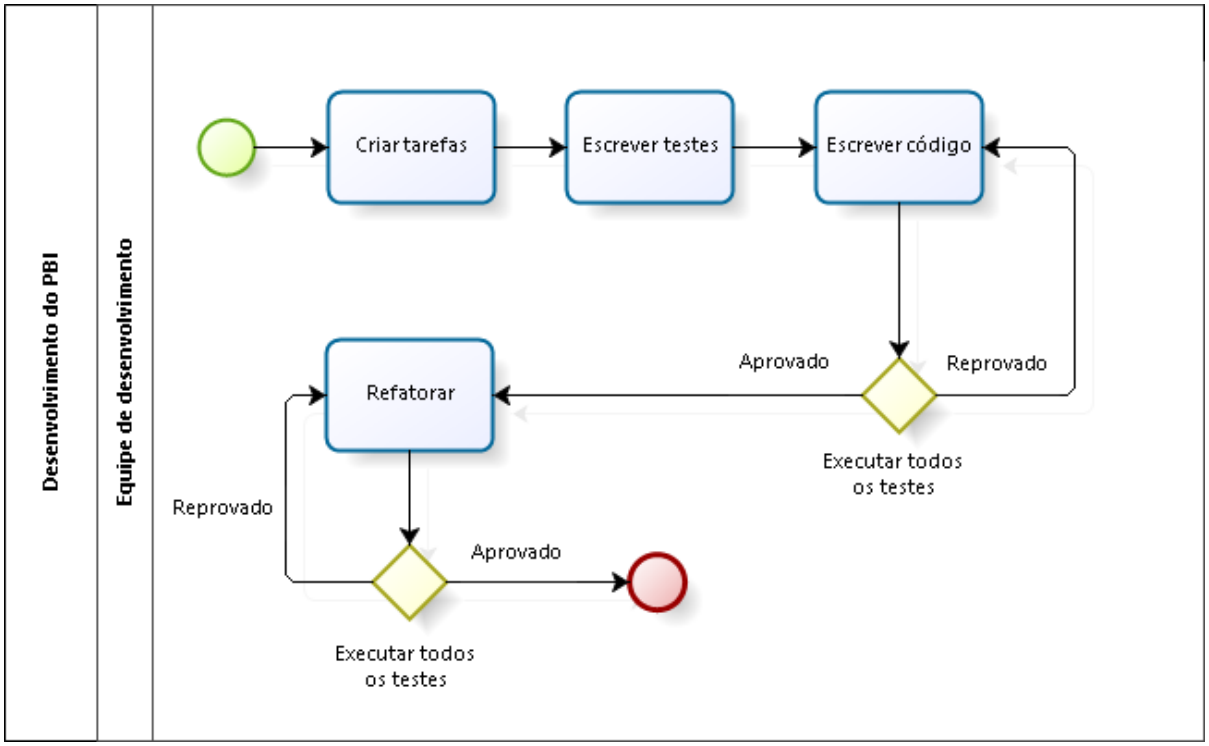
Este novo processo prevê a escrita de testes para os requisitos além do código funcional. Desta forma, todos os requisitos desenvolvidos a partir da implantação do processo passam a ser testados automaticamente, facilitando a testagem do Catalog a médio prazo.

Na figura 15, é possível observar o novo fluxo previsto pelo processo. A escrita dos testes automatizados ocorre logo após a criação das tarefas do PBI e antes da escrita do código funcional (vermelho do TDD, na figura 4). Desta forma, durante a escrita do código funcional (verde do TDD, na figura 4), o programador já possui os

testes dos quais necessita para validar rapidamente o seu atual desenvolvimento. Isto também incorpora as vantagens do TDD ao processo, que são (FOGGETTI, 2014, p. 91):

- O entendimento do sistema pode ser feito com a leitura dos testes;
- Não é desenvolvido um código desnecessário;
- Não existe código sem teste;
- Uma vez que o teste funciona, ele pode ser usado sempre, inclusive como teste de regressão.

**Figura 15 – Novo processo de desenvolvimento de PBIs, com escrita de testes**




Fonte: elaborado pelo autor

O fluxo também prevê uma refatoração ao final do desenvolvimento, assim como no TDD; isto objetiva simplificar o código funcional escrito, além de facilitar manutenções e alterações futuras.

Este fluxo é utilizado tanto no desenvolvimento de requisitos quando na correção de defeitos, que são os itens não-planejados da metologia da equipe.

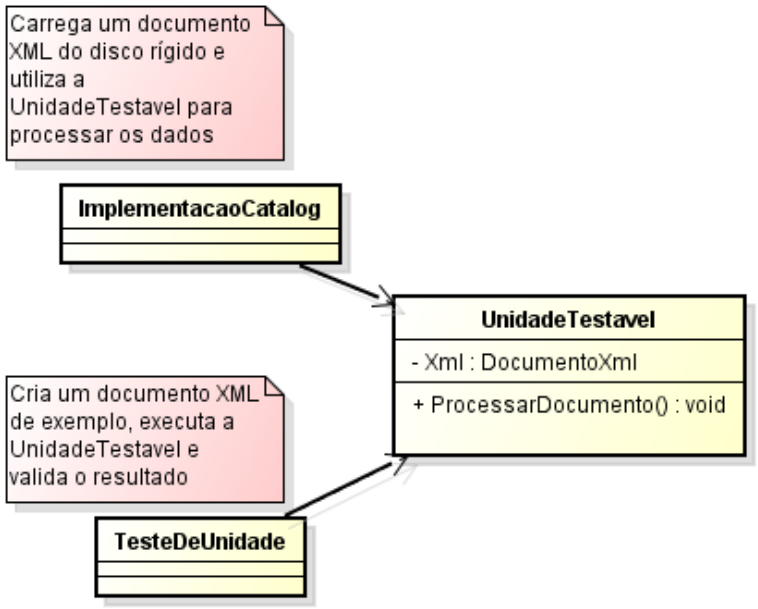
### 3.2.1 Planejamento dos testes

Os testes devem ser escritos de acordo com os requisitos de cada PBI, sempre buscando testar as funcionalidades previstas (testes funcionais) e a integridade dos dados gerados pelo Catalog (teste de integridade). Por exemplo, ao incluir um novo campo em alguma tela de cadastro, o teste deverá validar se a alteração do campo funciona e se os dados são gerados de forma correta no documento XML.

Os testes automatizados previstos no processo caracterizam-se como testes de unidade, que testam exclusivamente a unidade criada para atender a funcionalidade do requisito. Desta forma, cada unidade deve ser modelada de tal forma que possa ser testada pelo ambiente de testes de unidade. Isto inclui evitar operações que acessem recursos externos ao programa, como o disco rígido.

Caso a unidade necessite de dados externos, estes dados devem ser passados como parâmetro para a unidade. Assim, é possível utilizar dados de exemplo nos testes, enquanto que no sistema em funcionamento são utilizados os dados reais. A figura 16 propõe um padrão de modelagem para este tipo de situação.

**Figura 16 – Padrão de modelagem proposto para suportar testes de unidade**




Fonte: elaborado pelo autor

### 3.2.2 Medição dos resultados e expectativas

A medição dos resultados do novo processo será feita através da anotação da quantidade de defeitos corrigidos por *sprint* e as causas dos defeitos, a exemplo do que já foi feito nas tabelas 1 e 2.

Com o novo processo, espera-se que a quantidade de defeitos ocorridos por falhas no desenvolvimento e nos testes seja reduzida. Conforme visto na seção 3.1.5, 73,7% da quantidade total de defeitos corrigidos no produto ocorrem por conta destas falhas.

Acredita-se que esta diminuição deverá ser gradativa, pelo fato de os testes serem escritos apenas para os requisitos e correções desenvolvidos a partir do início da aplicação do novo processo.

### 3.2.3 Ferramentas do novo processo

Para o gerenciamento do *Scrum*, continuará sendo utilizado o Microsoft Team Foundation Server 2012. A forma como esta ferramenta é usada não será modificada.

Para a automação e execução dos testes baseados no TDD, será utilizado o *framework* NUnit, que é totalmente integrado ao Microsoft Visual Studio 2013.

Existem algumas necessidades que não serão atendidas por estas ferramentas:

1. Obter de forma fácil a informação de quantas e quais tarefas estão sendo testadas, para a validação do processo. Isto é possível através de controle de versionamento do Team Foundation Server, porém verificar o versionamento associado a cada tarefa, uma a uma, é um procedimento oneroso;
2. Ao ler uma tarefa, acessar facilmente o teste de unidade correspondente, também para a validação do processo. Também é possível, porém oneroso, via versionamento do Team Foundation Server.

Para atender a estas necessidades, será desenvolvida uma ferramenta de apoio, que funcionará como um *front end* do novo processo, reunindo informações do Team Foundation Server e dos testes de unidade em uma única tela, e também oferecendo atalhos para realizar alguns procedimentos comuns do processo.

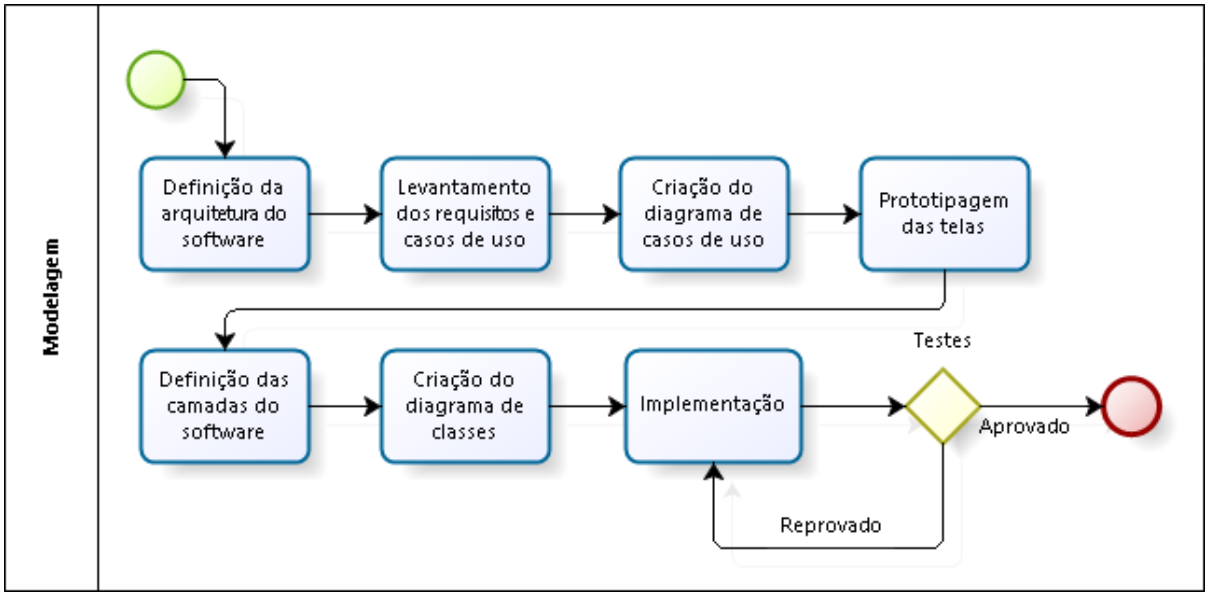
## 3.3 Desenvolvimento da ferramenta de apoio

O desenvolvimento da ferramenta de apoio será feito de acordo com o fluxograma da figura 17. Os passos do desenvolvimento serão os seguintes:

1. **Definição da arquitetura do software**: definição de qual será o ambiente, a linguagem de programação, as bibliotecas e *frameworks* utilizados;
2. **Levantamento dos requisitos e casos de uso**: definição dos requisitos e casos de uso que a ferramenta deve atender;

3. **Criação do diagrama de casos de uso**: elaboração do diagrama de acordo com os casos de uso levantados anteriormente;
4. **Prototipagem das telas**: criação dos protótipos de tela da ferramenta;
5. **Definição das camadas do software**: quais serão as camadas e como funcionará a interação entre as mesmas;
6. **Criação do modelo de classes**: definição das principais classes do sistema, e como as mesmas interagem entre si;
7. **Implementação**: codificação do software;
8. **Testes**: validação da codificação do software.

**Figura 17 – Fluxo do desenvolvimento da ferramenta de apoio**

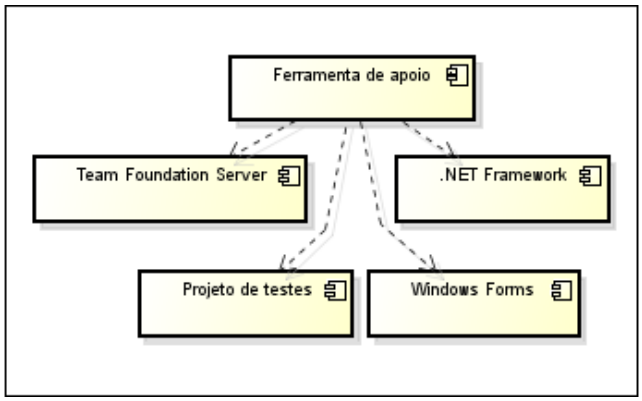



Fonte: elaborado pelo autor

### 3.3.1 Definição da arquitetura do software

A ferramenta será totalmente escrita em linguagem C#. Serão utilizados o .NET Framework para a construção da ferramenta e o Windows Forms para a elaboração da interface com o usuário. A ferramenta se comunicará com o Team Foundation Server e com o projeto de testes do Catalog. A arquitetura da ferramenta é representada na figura 18.

**Figura 18 – Arquitetura da ferramenta de apoio**




Fonte: elaborado pelo autor

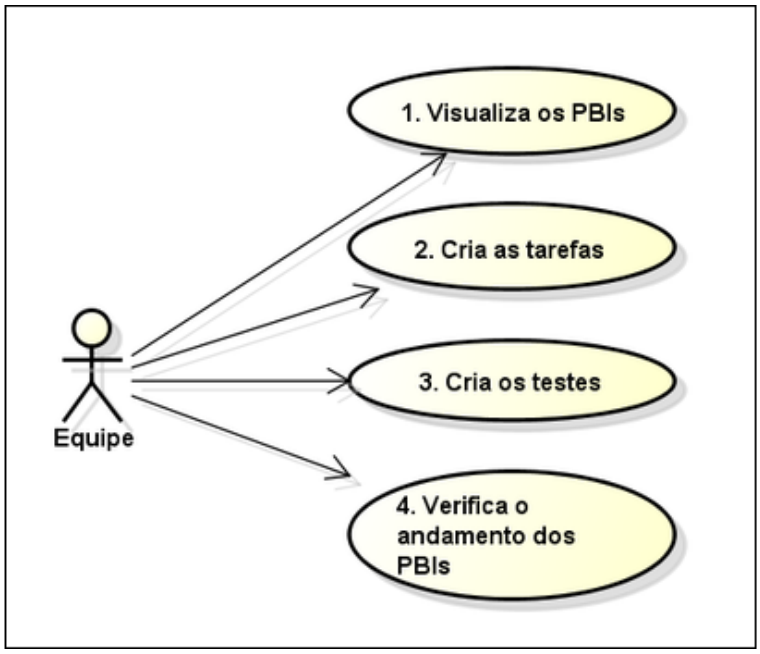
### 3.3.2 Requisitos e casos de uso da ferramenta

Os requisitos da ferramenta têm como objetivo dar apoio ao processo de testes definido anteriormente. São os seguintes:

1. **Visualização de PBIs**: a ferramenta deve disponibilizar uma tela com a lista de PBIs organizados por *sprint*, dando ênfase ao *sprint* atual. A lista de PBIs será requisitada ao Team Foundation Server;
2. **Atalho para criação de tarefas**: ao selecionar um PBI, deve haver uma opção que direcione o usuário à página do Team Foundation Server onde são criadas as tarefas para o PBI. Depois que as tarefas são criadas, a ferramenta deve trazer estas tarefas e mostrá-las na tela;
3. **Atalho para criação de testes**: ao selecionar uma tarefa, deve haver uma opção que direcione o usuário ao projeto de código-fonte do Visual Studio que contém os testes automatizados. Deverá ser disponibilizada uma biblioteca no formato DLL para vincular os métodos de teste desenvolvidos no Visual Studio à tarefa correspondente no Team Foundation Server. Depois que os testes são criados, a ferramenta deve mostrar que a tarefa contém testes através da pesquisa de dados dos testes automatizados, além de apresentar um atalho para o arquivo-fonte que contém o teste;
4. **Verificação do andamento dos PBIs e tarefas**: a ferramenta deve apresentar, de forma clara, o status das tarefas e PBIs, de forma que a equipe de desenvolvimento possa identificar facilmente o andamento dos mesmos. É importante também que seja mostrada a presença ou ausência de testes automatizados;

O diagrama de casos de uso da figura 19 foi gerado a partir dos requisitos levantados anteriormente.

**Figura 19 – Diagrama de casos de uso da ferramenta de apoio**




Fonte: elaborado pelo autor

Também foram desenvolvidos os protótipos de tela, que estão anexados a este trabalho no apêndice A.

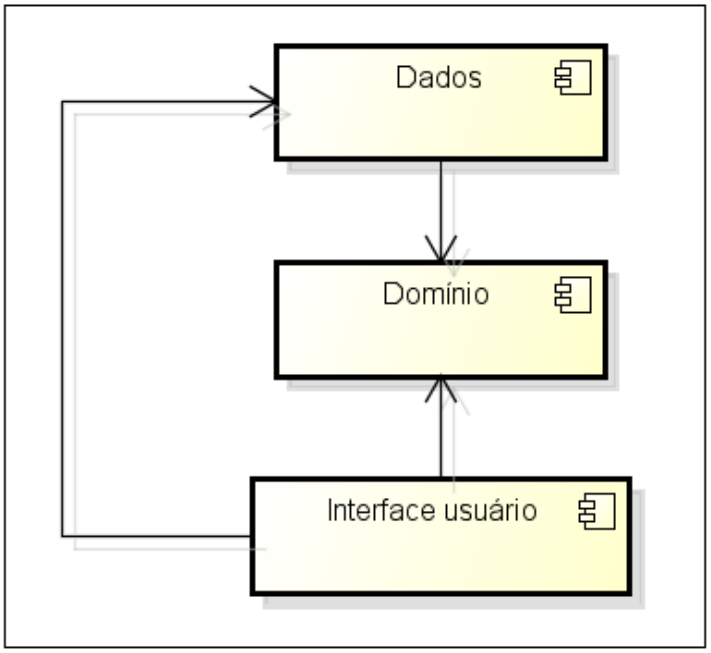
### 3.3.3 Camadas e classes da ferramenta

O software será desenvolvido em três camadas:

1. **Domínio**: contém as classes que representam os dados da ferramenta. Eventualmente pode conter algumas operações, desde que dependam apenas dos dados do domínio.
2. **Dados**: contém as classes que carregam os dados do domínio, através da consulta ao Team Foundation Server, ao projeto-fonte de teste e a outras fontes que possam surgir futuramente.
3. **Interface com o usuário**: contém as telas da ferramenta e o controle da interação com o usuário.

A representação das camadas pode ser visualizada na figura 20. Já o modelo da figura 21 representa as principais classes do software, e como funcionam as interações entre elas. O pacote *Domain* é a camada de domínio; o pacote *Repositories* é a camada de dados; e os pacotes *Controllers* e *UserInterface* são a camada de interface.

**Figura 20 – Representação das camadas da ferramenta de apoio**



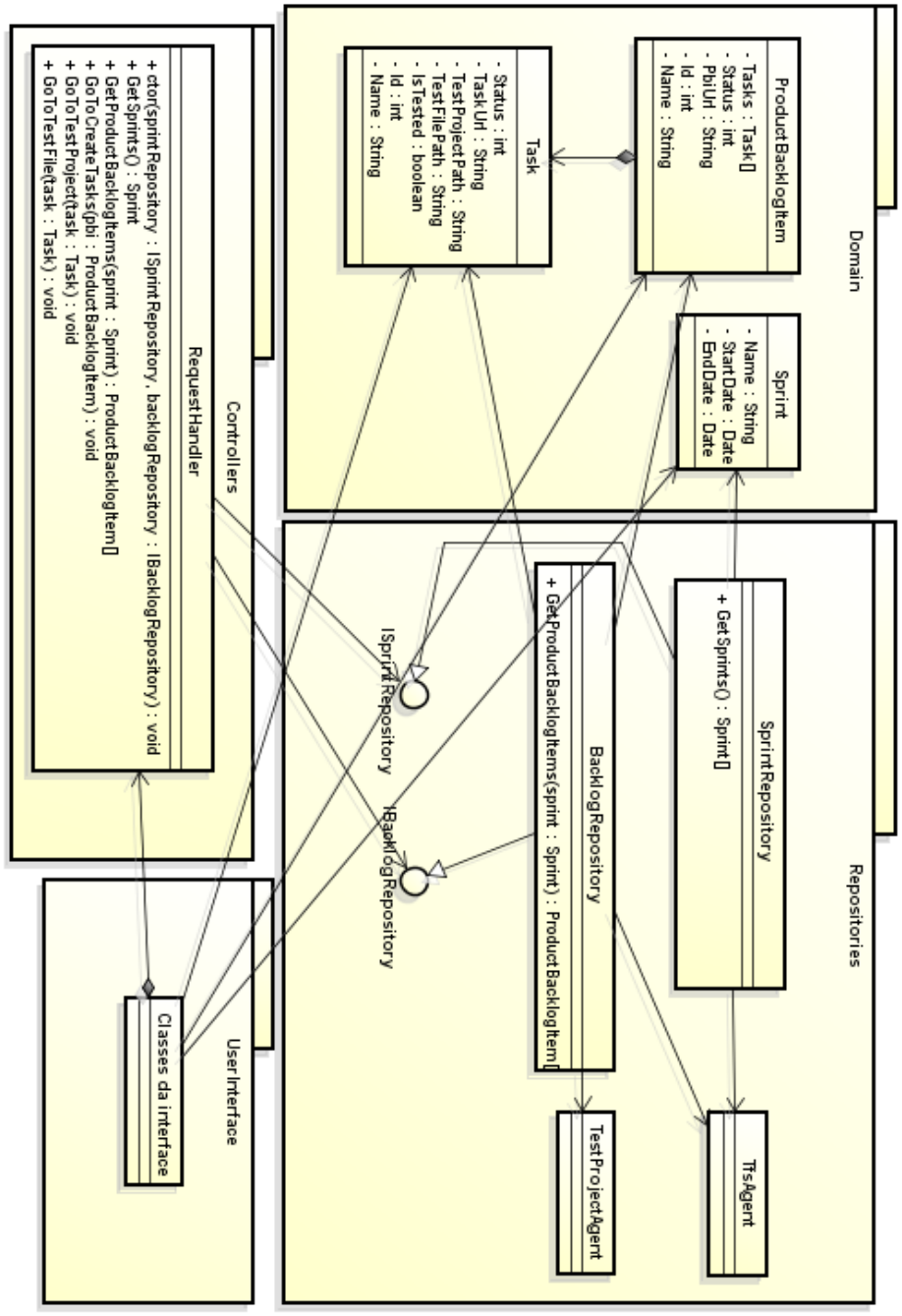

Fonte: elaborado pelo autor

As funções das classes da figura 21 são:

- *ProductBacklogItem*: representa um PBI e contém os dados e tarefas do mesmo;
- *Task*: representa uma tarefa e contém os dados da mesma;
- *Sprint*: representa um *sprint* e contém os dados do mesmo;
- *ISprintRepository*: define um repositório de *sprints*;
- *IBacklogRepository*: define um repositório de PBIs;
- *SprintRepository*: implementação do *ISprintRepository*, responsável por buscar *sprints* do Team Foundation Server utilizando a classe *TfsAgent*;
- *BacklogRepository*: implementação do *IBacklogRepository*, responsável por buscar tarefas do Team Foundation Server e complementá-las com os dados dos testes, utilizando as classes *TfsAgent* e *TestProjectAgent*;
- *TfsAgent*: classe auxiliar que conecta-se ao Team Foundation Server e obtém dados do servidor;
- *TestProjectAgent*: classe auxiliar que obtém dados do projeto de testes;
- *RequestHandler*: lida com as requisições do usuário na interface gráfica;

- Classes da interface: conjunto de classes responsável por desenhar as interfaces gráficas na tela e receber entradas do usuário.

**Figura 21 – Modelo de classes da ferramenta de apoio**




Fonte: elaborado pelo autor

Os métodos que implementam os casos de uso são os seguintes:

1. *GetSprints* e *GetProductBacklogItems*, da classe *RequestHandler*. O primeiro é chamado para obter a lista de *sprints*, enquanto o segundo é chamado para obter

a lista de PBIs de um *sprint*;

2. *GoToCreateTasks*, da classe *RequestHandler*;
3. *GoToTestProject*, da classe *RequestHandler*;
4. *GetSprints* e *GetProductBacklogItems*, da classe *RequestHandler*. Novamente estes métodos são chamados, pois eles devolverão a lista de PBIs atualizada.

## 3.4 Considerações finais

Neste capítulo, foi apresentada a visão do cenário onde foi estudado o problema de pesquisa, detalhando aspectos do projeto Catalog e do contexto onde o projeto está inserido. Foi visto que o Catalog é um software de cadastro de móveis, e que a equipe de desenvolvimento do Catalog utiliza uma metodologia baseada em *Scrum*. Depois, foi apresentada a dificuldade presente nos testes do Catalog, que originou o problema de pesquisa.

Com base neste problema, foi proposta uma alteração nesta metodologia, inserindo um novo processo de testes no desenvolvimento do Catalog. Este processo é baseado em TDD, visto que é uma técnica que deverá facilitar o trabalho da equipe de desenvolvimento.

Espera-se que este novo processo reduza a quantidade de defeitos corrigidos pela equipe de desenvolvimento durante os *sprints* do Catalog, através da maior facilidade com que serão realizados os testes, tanto para os testes funcionais dos requisitos em desenvolvimento quanto para os testes de regressão.

# 4 Implementação

Este capítulo descreve como ocorreu a implementação da proposta de solução. É dividido em duas seções: a primeira referente à preparação para utilizar os testes automatizados no Promob Catalog, e a segunda referente à implementação da ferramenta de apoio aos testes.

A implementação destes dois itens torna possível a aplicação do novo processo de testes, previsto na proposta de solução.

## 4.1 Testes automatizados

Para aplicar o processo ilustrado na figura 14, foi necessário conhecer o comportamento do sistema em um ambiente de testes de unidade, bem como levantar quais componentes deveriam ser carregados ao executar os testes. Depois disto, foram feitas algumas adaptações para executar o Catalog neste ambiente de forma adequada.

Estes procedimentos são fundamentais para a aplicação do novo processo de testes, visto que a utilização de testes automatizados é parte fundamental deste processo.

Para fazer estas análises, foram criados os casos de testes pilotos apresentados a seguir, de tal forma que o novo processo foi simulado e as necessidades foram levantadas.

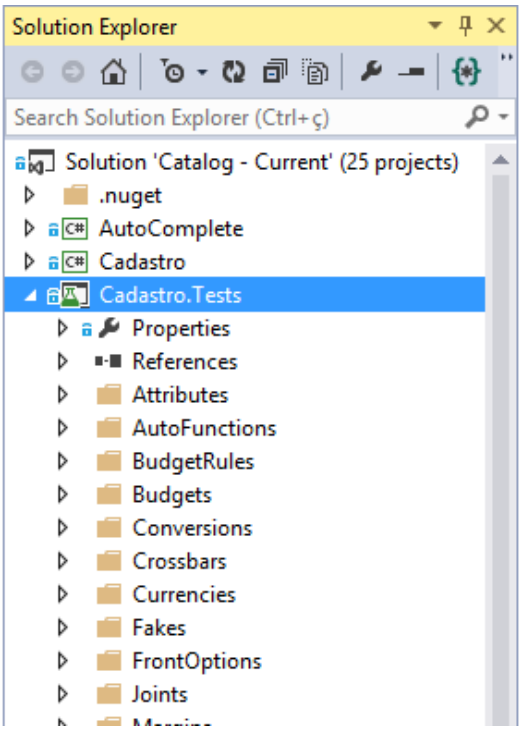
### 4.1.1 Definição do projeto de testes

Para utilizar testes automatizados em alguma solução de código-fonte, é necessário existir um projeto de testes, que nada mais é do que um agrupamento que contém códigos de testes.

Assim sendo, foi necessário definir o projeto de testes para o Visual Studio, antes mesmo da escrita dos casos de testes pilotos. Este projeto é onde serão escritos todos os testes automatizados do novo processo.

Já existiam alguns testes automatizados para pontos específicos do Catalog, portanto o projeto de testes já existia e foi reaproveitado. A figura 22 ilustra o projeto de testes aberto no Microsoft Visual Studio.

**Figura 22 – Projeto de testes aberto no Microsoft Visual Studio**




Fonte: elaborado pelo autor

### 4.1.2 Caso de teste piloto 1

O primeiro piloto escrito é um teste de unidade que altera o valor de um campo em uma das telas do Catalog, validando se o documento XML foi alterado de forma correta. Trata-se de um piloto que simula a criação de um novo campo em tela no novo processo. O teste trabalha apenas com o *back end* responsável pelo carregamento do valor do campo e pela alteração do mesmo. Na figura 23, é possível visualizar o código do teste de unidade, que executa quatro passos:

1. Carrega o documento XML da figura 25. Este documento foi criado para o teste e está armazenado internamente no projeto de testes, portanto, trata-se de dados fictícios;

2. Carrega o *TreeNode*, que é a unidade responsável por controlar o *back end* da tela do Catalog, que está ilustrada na figura 24;

3. Utilizando a unidade *TreeNode*, altera o valor da propriedade *Description*, que contém o valor do campo Descrição, destacado em vermelho na figura 24;

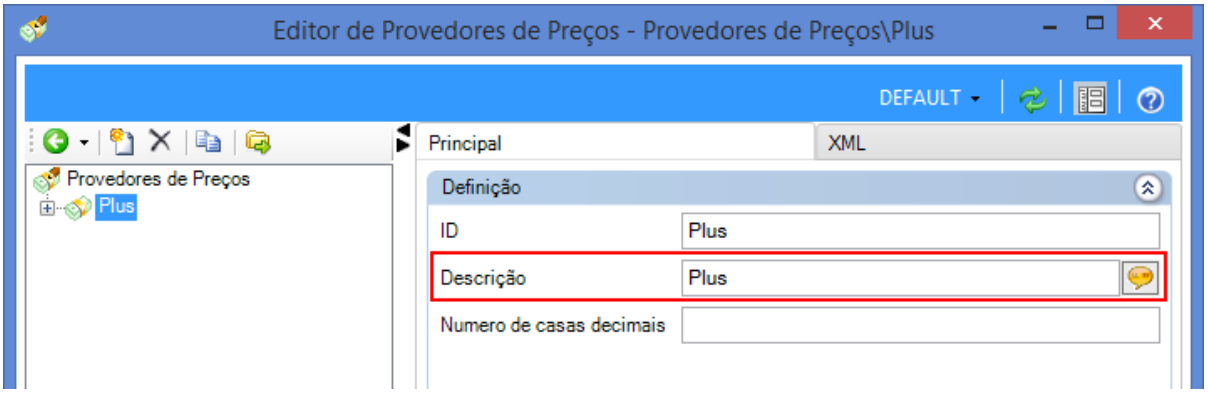
4. Finalmente, valida se o documento XML foi alterado corretamente. Esta é a chamada que determina se o resultado do teste é aprovado ou reprovado.

**Figura 23 – Código-fonte do caso de teste piloto 1**

```
/// <summary>
/// Seta o valor de uma propriedade do editor de provedores de preços e testa se o XML foi alterado
/// </summary>
[TestCase]
public void SetPriceProviderField()
{
    // Instancia um Nodo a partir de um resource do projeto de testes
1   var xml = TestUtils.ParseXmlNode(Resources.ExamplePriceProvider);

    // Carrega o TreeNode
2   var treeNode = (TreeNodePriceProvider)TreeNodeFactory.Instance.Build(null, xml);

    // Seta o valor do campo Descrição
3   var properties = treeNode.GetService<PriceProviderProperties>();
    properties.Description = "Exemplo2";

    // Testa se alterou o XML
4   xml.Atributos["DESCRIPTION"].Valor.Should().Be("Exemplo2");
}
```

Fonte: elaborado pelo autor

**Figura 24 – Tela correspondente ao caso de teste piloto 1**




Fonte: elaborado pelo autor

**Figura 25 – Documento XML correspondente ao caso de teste piloto 1**

```
<PRICEPROVIDER ID="ex" DESCRIPTION="Plus">
  <SOURCES>...</SOURCES>
</PRICEPROVIDER>
```

Fonte: elaborado pelo autor

O teste da figura 23 não carrega a tela ilustrada na figura 24, mas, pelo fato de a tela utilizar a unidade *TreeNode*, o teste garante que o campo está funcional e há integridade nos dados do campo.

### 4.1.3 Caso de teste piloto 2

O segundo piloto escrito é um teste de unidade que simula o carregamento de uma tela de edição do Catalog, validando se os objetos de *back end* do editor foram carregados corretamente. Este teste, diferentemente do anterior, não envolve alterações em campos ou qualquer tipo de dados, testando somente o carregamento dos objetos envolvidos. Trata-se de um piloto que simula a criação de uma nova tela no Catalog, sendo apenas um entre tantos testes que podem ser escritos para esta situação.

O código da figura 26 executa quatro passos:

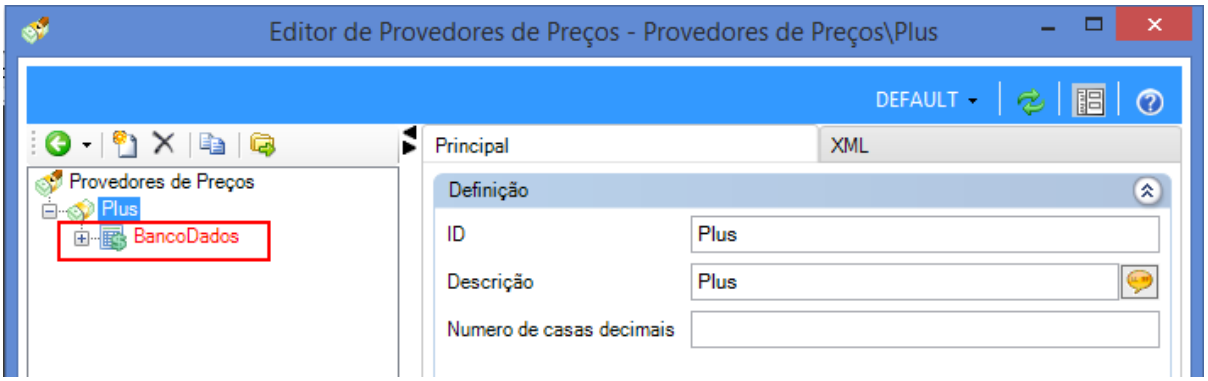
1. Carrega o documento XML da figura 28, da mesma forma que o teste anterior;
2. Carrega a unidade *TreeNode*, também da mesma forma que o teste anterior;
3. Executa um método da unidade, responsável por expandir a árvore destacada na figura 27;
4. Valida se a quantidade e tipo dos nós carregados corresponde aos dados do documento XML.

**Figura 26 – Código-fonte do caso de teste piloto 2**

```
/// <summary>
/// Testa se está carregando os filhos de um TreeNode do editor de provedores de preços
/// </summary>
[TestCase]
public void LoadPriceProviderChildren()
{
    // Instancia um Nodo a partir de um resource do projeto de testes
1   var xml = TestUtils.ParseXmlNode(Resources.ExamplePriceProvider);

    // Carrega o TreeNode
2   var treeNode = (TreeNodePriceProvider)TreeNodeFactory.Instance.Build(null, xml);

    // Carrega os filhos do TreeNode
3   treeNode.ConstruirFilhosSobDemanda();

    // Testa se tem 1 filho (mesma quantidade de fontes de preços do resource)
    // e se o filho é uma fonte de preços
4   treeNode.Nodes.Count.Should().Be(1);
    treeNode.Nodes[0].Should().Be.OfType<TreeNodeSource>();
}
```

Fonte: elaborado pelo autor

**Figura 27 – Tela correspondente ao caso de teste piloto 2**




Fonte: elaborado pelo autor

**Figura 28 – Documento XML correspondente ao caso de teste piloto 2**

```
<PRICEPROVIDER ID="ex" DESCRIPTION="Plus">
  <SOURCES>
    <SOURCE ID="BancoDados"
            CLASS="ProCAD.Precos.Fontes.FontePrecoBancoDados"
            DLL="ProCAD.Precos.dll"
            DATABASE="%PastaPrice%\ex\ex.mdb"
            TABLE="Prices"
            COLUMNPRICE="Price"
            COLUMNCODE="Code"
            CANMODIFY="Y"
            SEARCHFIRSTPERSONALBASE="N"
            SEARCHONLYPERSONALBASE="N" />
  </SOURCES>
</PRICEPROVIDER>
```

Fonte: elaborado pelo autor

Novamente, somente a unidade de *back end* é carregada, sem as telas correspondentes, validando os objetos que são utilizados pelas telas em tempo de execução do Catalog.

### 4.1.4 Componentes, ajustes e considerações

Através da elaboração dos testes pilotos, foi percebido que não é necessário carregar nenhum dos componentes do sistema que são carregados em tempo de execução, e sim apenas as unidades testadas. Isto ocorreu porque as unidades testadas já possuem toda a funcionalidade de que necessitam para executar, bastando carregar um documento XML e fornecê-lo para a unidade.

Foram descobertos dois comportamentos que impediam o funcionamento das unidades no ambiente de testes. Tratavam-se de duas exceções que ocorriam na unidade *TreeNode*, na ausência do componente de idiomas do Catalog. A equipe entendeu que o componente de idiomas não é fundamentalmente necessário para

a execução das unidades, portanto ambas as situações foram consideradas como defeitos e corrigidas. Durante a aplicação do novo processo, eventuais defeitos que ocorram somente nos testes poderão ser corrigidos.

Eventualmente, caso haja alguma ocorrência de código de *back end* implementado diretamente em tela, deverá ser feita uma refatoração para mover este código ao *back end* de fato, tornando-o testável. No desenvolvimento de novas funcionalidades, elas deverão ser desenvolvidas de forma testável, fato que reforça a importância de escrever os testes antes do código funcional.

### 4.1.5 Atributo *TaskId*

Para realizar a vinculação dos casos de testes às tarefas do Team Foundation Server, foi criada uma DLL, chamada *TfsTestManager.Tools*, que contém um atributo chamado *TaskId*. Esta DLL foi referenciada pelo projeto de testes, de forma que o atributo será utilizado nos casos de testes do projeto. Um exemplo de utilização do atributo é demonstrado na figura 29.

**Figura 29 – Teste de unidade vinculado à tarefa com ID 2142**

```
[TestCase, TaskId(2142)]
public void DuplicateMaterialTreeNode01()...
```

Fonte: elaborado pelo autor

Para vincular um caso de teste a uma tarefa do Team Foundation Server, o desenvolvedor deve decorar[1] o método do caso de teste desejado com o atributo *TaskId*, informando no atributo o ID da tarefa vinculada.

## 4.2 Gerenciador de testes do Team Foundation Server

O gerenciador de testes do Team Foundation Server (nome do projeto: *TfsTestManager*) é o programa que implementa a ferramenta de apoio ao novo processo, modelada no capítulo 3.

Esta ferramenta tem a função de mostrar todos os PBIs e tarefas presentes no Team Foundation Server do projeto Catalog, através da consulta das informações de cada PBI e tarefa, além de mostrar se há algum teste vinculado a cada tarefa e onde está este teste, através do atributo *TaskId*.

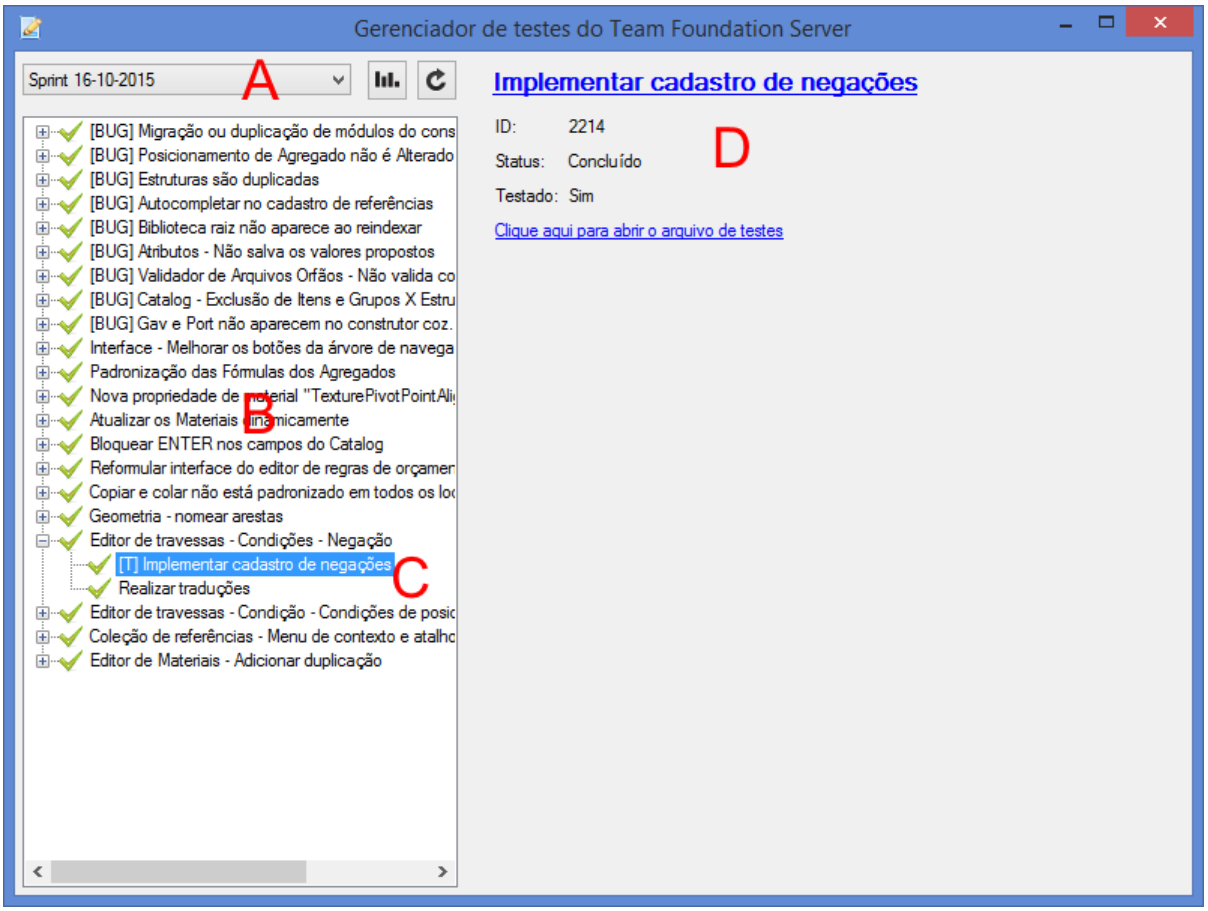
A hierarquia das informações vindas do Team Foundation Server é a seguinte (figura 30):

1. Lista de *sprints* (A);

[1] Termo utilizado nas linguagens do .NET Framework para a utilização de atributos.

2. Lista de PBIs do *sprint* selecionado (B);
3. Cada PBI pode ser expandido para visualizar as tarefas (C);
4. Informações do PBI ou tarefa selecionada (D).

**Figura 30 – Gerenciador de testes em funcionamento**




Fonte: elaborado pelo autor

A tela da ferramenta exibe o status de cada PBI e tarefa através dos ícones ao lado esquerdo de cada item. Também há atalhos para executar alguns procedimentos comuns ao processo:

- Acesso aos PBIs e tarefas, para visualização e edição de detalhes;
- Acesso à tela para criar tarefas a partir de cada PBI;
- Acesso ao projeto de testes a partir de cada tarefa;
- Acesso ao arquivo que contém o teste de unidade a partir de cada tarefa, caso a tarefa o tenha.

A ferramenta funciona como um *front end* para o novo processo. Com isto, torna-se possível verificar quantas e quais tarefas estão sendo testadas, possibilitando levantar se os testes estão sendo escritos, conforme levantado na proposta de solução do capítulo anterior.

O código-fonte da ferramenta foi escrito na linguagem de programação C#, seguindo o modelo de classes definido na figura 21, e está anexado a este trabalho. A implementação dos casos de uso também segue o modelo do capitulo 3. A implementação dos protótipos de tela pode ser visualizada no apêndice B.

Para funcionar, a ferramenta precisa coletar dados de dois locais: o servidor do Microsoft Team Foundation Server, e o projeto de testes utilizado para o Catalog. A forma como a ferramenta busca os dados destes dois locais é descrita nas próximas duas seções.

### 4.2.1 *Microsoft.TeamFoundation*

Para consultar as informações do Team Foundation Server, foi utilizada uma biblioteca presente no .NET Framework 4.5, chamada Microsoft.TeamFoundation. Esta biblioteca fornece algumas DLLs que encapsulam o acesso ao servidor e a consulta de informações:

- *Microsoft.TeamFoundation.Client*;
- *Microsoft.TeamFoundation.Common*;
- *Microsoft.TeamFoundation.WorkItemTracking.Client*;
- *Microsoft.TeamFoundation.WorkItemTracking.Client.DataStoreLoader*;
- *Microsoft.TeamFoundation.WorkItemTracking.Common*.

Para consultar os *sprints*, é enviada uma requisição ao servidor em busca do ciclo de vida do projeto. O servidor retorna um documento XML contendo algumas informações (figura 31), entre elas a coleção de *sprints* do projeto, que é lida pela ferramenta e armazenada em memória. Esta requisição ocorre apenas uma vez a cada execução da ferramenta, durante a abertura da mesma.

Para consultar os PBIs e tarefas, é utilizada uma linguagem própria de consulta, semelhante ao SQL, chamada WIQL (do inglês: *Work Item Query Language*). A consulta utilizada pode ser vista na figura 32.

Esta consulta devolve todos os PBIs e tarefas do *sprint* nomeado “Sprint 16-10-2015”, pertencentes ao produto Catalog e que não foram removidas pela equipe. São trazidos: ID, título, status, tipo de item (se é um PBI de requisito ou de defeito) e produto.

Após a consulta, a ferramenta organiza os dados, de forma a colocar as tarefas em seus respectivos PBIs. Esta consulta é executada sob demanda, toda vez que algum *sprint* é selecionado.

**Figura 31 – Exemplo de documento XML contendo o ciclo de vida do projeto**

```
<Nodes xmlns="">

  <Node NodeID="vstfs:///Classification/Node/1394a6e2-56a5-4c60-825b-042816956e7c"
        Name="Iteration"
        Path="\Projetos\Iteration"
        ProjectID="vstfs:///Classification/TeamProject/656f2d14-8948-4147-8199-d176025cc1ef"
        StructureType="ProjectLifecycle">

    <Children>

      <Node NodeID="vstfs:///Classification/Node/093f9d9d-0337-491d-9f55-a54af716eae9"
            Name="Sprint 26-06-2015"
            ParentID="vstfs:///Classification/Node/1394a6e2-56a5-4c60-825b-042816956e7c"
            Path="\Projetos\Iteration\Sprint 26-06-2015"
            ProjectID="vstfs:///Classification/TeamProject/656f2d14-8948-4147-8199-d176025cc1ef"
            StructureType="ProjectLifecycle"
            StartDate="2015-06-15T00:00:00.000"
            FinishDate="2015-06-26T00:00:00.000" />

      <Node NodeID="vstfs:///Classification/Node/8bd18f68-5b23-4435-8e00-1e90f3d7920f"
            Name="Sprint 10-07-2015"
            ParentID="vstfs:///Classification/Node/1394a6e2-56a5-4c60-825b-042816956e7c"
            Path="\Projetos\Iteration\Sprint 10-07-2015"
            ProjectID="vstfs:///Classification/TeamProject/656f2d14-8948-4147-8199-d176025cc1ef"
            StructureType="ProjectLifecycle"
            StartDate="2015-06-29T00:00:00.000"
            FinishDate="2015-07-10T00:00:00.000" />

    </Children>

  </Node>

</Nodes>
```

Fonte: elaborado pelo autor

**Figura 32 – Modelo de consulta WIQL utilizado pelo gerenciador de testes**

```
Select [Id],
       [Title],
       [State],
       [Work Item Type],
       [Area Path]
From WorkItems
Where [IterationPath] = 'Projetos\Sprint 16-10-2015' And
      [Area Path] = 'Projetos\Catalog' And
      [State] != 'Removed'
```

Fonte: elaborado pelo autor

### 4.2.2 Roslyn

Para consultar a vinculação dos testes com as tarefas conforme demonstrado na figura 29, foi utilizada uma biblioteca da Microsoft chamada Roslyn.

**Figura 33 – Código que lê e analisa um arquivo C#**

```
// Lê o conteúdo do arquivo C#
var code = File.ReadAllText(file);

// Analisa o código
var tree = CSharpSyntaxTree.ParseText(code);
```

Fonte: elaborado pelo autor

**Figura 34 – Exemplo de código-fonte de testes com sua respectiva árvore sintática**


Fonte: elaborado pelo autor

A biblioteca Roslyn é disponibilizada separadamente do .NET Framework através

do NuGet[2], e tem a função de compilar código C#, tendo as opções de executar o processo completo ou apenas partes do mesmo. No caso do gerenciador de testes, são executadas as análises léxica e sintática da linguagem C# no projeto de testes. Estas análises têm o objetivo de encontrar métodos que contêm o atributo *TaskId* em suas declarações.

Para executar as análises léxica e sintática em um arquivo de código C#, basta ler o conteúdo do arquivo e executar a chamada contida na figura 33.

Esta chamada retorna a árvore de elementos sintáticos do arquivo. Na figura 34, é possível visualizar um exemplo de árvore sintática. Com esta árvore, é possível obter os atributos *TaskId* contidos no código-fonte com o parâmetro ID, possibilitando determinar quais arquivos de código apontam para quais tarefas.

## 4.3 Considerações finais

Neste capítulo, foi descrita a forma como ocorreu a implementação da proposta de solução. Foi detalhada a preparação do Catalog para receber os testes automatizados, bem como a implementação dos casos de teste pilotos. Logo após, foi detalhada também a implementação do gerenciador de testes, que é a ferramenta que irá apoiar a execução do processo.

Com a implementação concluída, é possível iniciar a aplicação do novo processo de testes na equipe de desenvolvimento do Catalog.

[2] Ferramenta on-line da Microsoft integrada ao Visual Studio, que disponibiliza pacotes de bibliotecas para serem adicionados aos projetos.

# 5 Estudo de caso

A aplicação do novo processo iniciou-se no dia 21/09/2015, coincidindo com o início de um *sprint* do Promob Catalog.

Este capítulo apresenta um estudo de caso, que é iniciado ao mesmo tempo que a aplicação do novo processo, estendendo-se durante quatro *sprints*, até a data de 13/11/2015. Os requisitos e objetivos destes *sprints* são detalhados no apêndice C.

Na primeira seção, é descrito como a equipe de desenvolvimento do Promob Catalog foi preparada para aplicar o novo processo. Na segunda seção, é descrito como o processo funcionou durante os quatro *sprints*. Finalmente, na terceira seção são descritos os resultados deste estudo de caso.

## 5.1 Preparação da equipe de desenvolvimento

Durante a semana anterior à data de início, o novo processo foi apresentado à equipe de desenvolvimento do Promob Catalog. Foram mostrados os aspectos do novo processo descritos ao decorrer deste trabalho, tais como: automação de testes, TDD, o fluxograma do novo processo (figura 15), os casos de teste pilotos do capítulo 4, o gerenciador de testes, entre outros. Finalmente, os integrantes da equipe foram orientados a seguir o novo processo.

De início, todos os integrantes da equipe mostraram-se abertos à ideia de mudar o processo de desenvolvimento e aplicar o novo método baseado na técnica TDD.

## 5.2 Utilização do novo processo

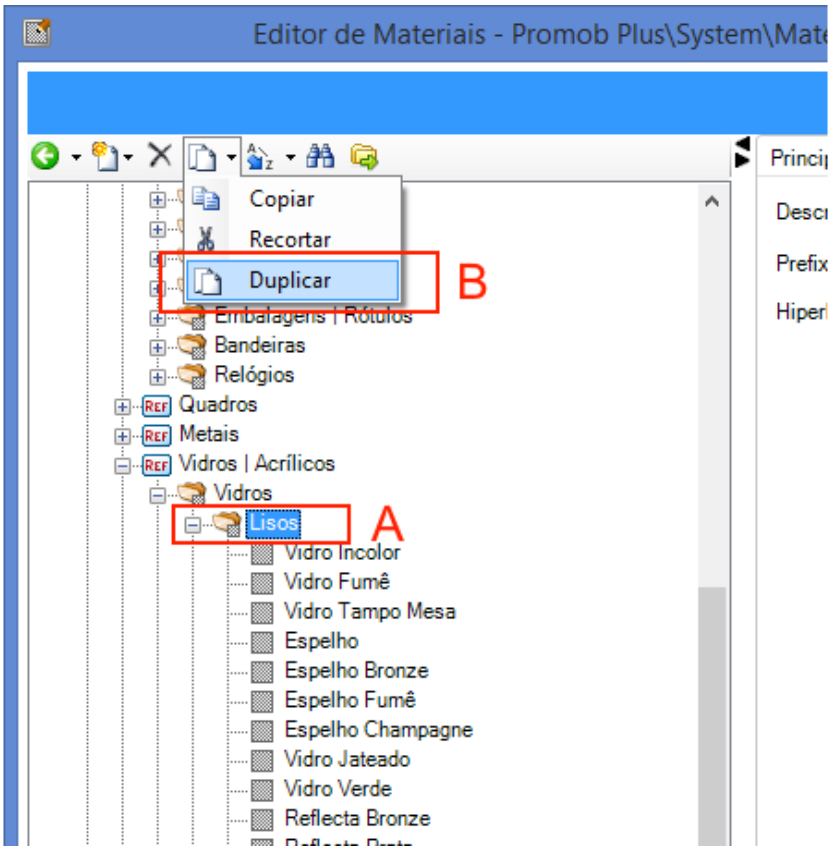
Durante os quatro *sprints* do estudo de caso, uma parte considerável dos requisitos foi desenvolvida com testes de unidade. A seguir, são mostrados dois exemplos de PBIs desenvolvidos de acordo com o novo processo.

### 5.2.1 Exemplo de utilização do novo processo 1: opção de duplicação

Como exemplo de utilização do novo processo, há o desenvolvimento do PBI 2230, que está no apêndice C.

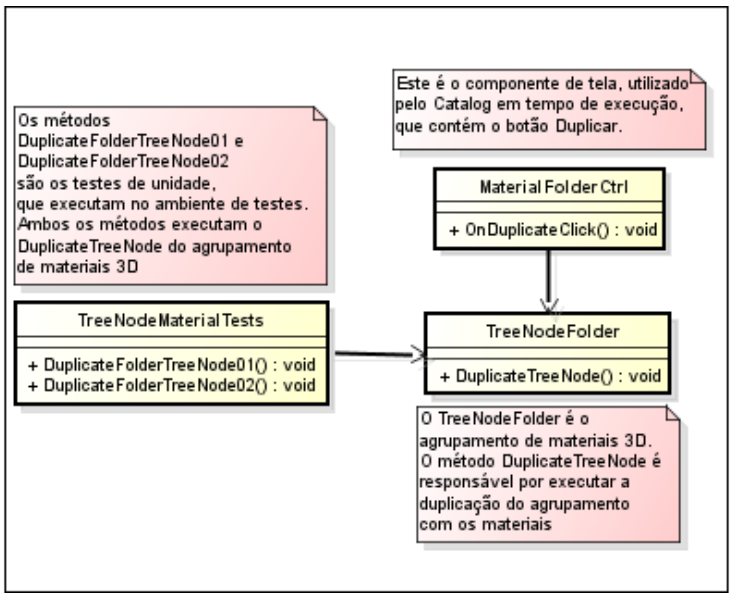
Este PBI, de forma resumida, consiste em criar um botão na tela para duplicar (B) agrupamentos de materiais 3D cadastrados (A), criando uma cópia do agrupamento selecionado em si e de todos os materiais 3D contidos no agrupamento. A tela é ilustrada na figura 35.

**Figura 35 – Tela de cadastro de materiais 3D com a opção de duplicar agrupamentos**




Fonte: elaborado pelo autor

**Figura 36 – Modelo de classes da duplicação de grupos de materiais 3D**

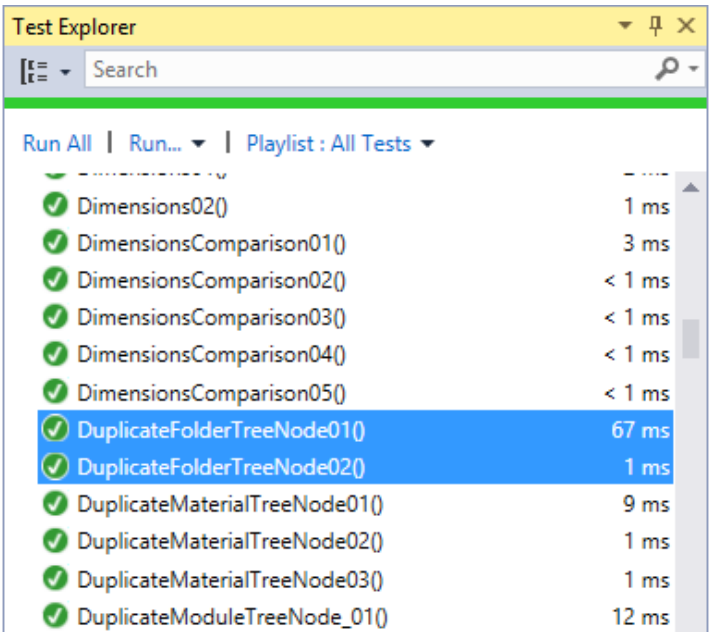



Fonte: elaborado pelo autor

Foram criadas duas tarefas para este PBI: a primeira refere-se à duplicação de grupos, e a segunda refere-se a duplicação de REFs, que são arquivos que contém grupos raízes da estrutura de materiais 3D. Para ambas as tarefas, foi seguida a sequência: escrever testes e codificar. As duas tarefas foram modeladas e desenvolvidas de forma muito semelhante, portanto, apenas a primeira tarefa é ilustrada neste trabalho.

O modelo de classes da primeira tarefa é ilustrado na figura 36. A unidade testável, que neste caso é o método *DuplicateTreeNode*, deve ser independente do restante do sistema, para possibilitar o teste da unidade. Já os testes de unidade, presentes na lista de testes de unidade que o Visual Studio mostra com o auxílio do NUnit, podem ser visualizados na figura 37.

**Figura 37 – Testes de unidade da duplicação de grupos de materiais 3D**




Fonte: elaborado pelo autor

Logo após concluir o modelo e escrever os testes, foi escrito o código do método *DuplicateTreeNode*, para ao final executar os testes, validando a escrita deste código. Não foi feita uma refatoração ao final do processo, pois a equipe avaliou o código e não julgou necessário este passo.

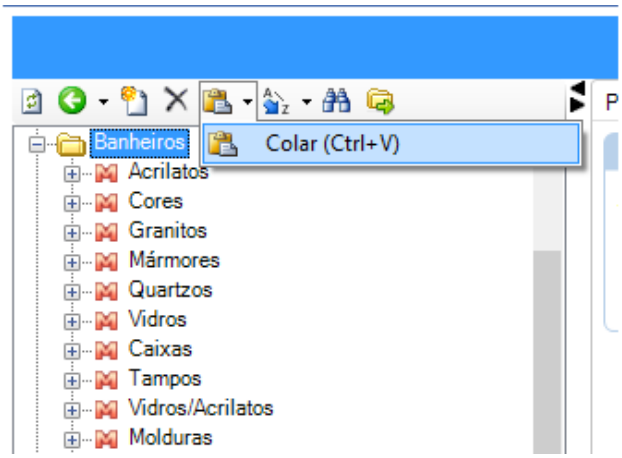
Um benefício imediato que a abordagem baseada em TDD trouxe está justamente na modelagem do código. O fato de a unidade precisar ser testada obriga a equipe a separar o código em, no mínimo, duas camadas: a interface com o usuário e o *back end* que contém a implementação. Antes, era comum cometer o erro de escrever partes da lógica de funcionamento do Catalog diretamente na interface, ou até de escrever blocos monolíticos na interface em alguns casos. De qualquer forma, o código funcionava. Agora, existe o objetivo de escrever unidades testáveis, ou seja, se as alterações não forem modeladas corretamente, não é possível testar, e isto acaba

melhorando a organização do código.

### 5.2.2 Exemplo de utilização do novo processo 2: copiar e colar

No PBI 2168, que está no apêndice C, era necessário fazer alterações em uma funcionalidade já existente: o copiar e colar. Estas alterações têm o sentido de padronizar o comportamento desta funcionalidade nas diferentes telas do Catalog.

**Figura 38 – Exemplo da funcionalidade colar**




Fonte: elaborado pelo autor

O copiar e colar é uma funcionalidade que consiste em replicar trechos de cadastro dentro do Promob Catalog. Alguns destes conjuntos de informação possuem um identificador único (ID), que deve ser alterado no trecho resultante da cópia. Estas alterações eram feitas de forma diferente em diversas partes do Catalog: em algumas, era gerado um novo identificador único; em outras, o identificador original era concatenado com "_1". O PBI consiste em fazer todos os identificadores serem concatenados.

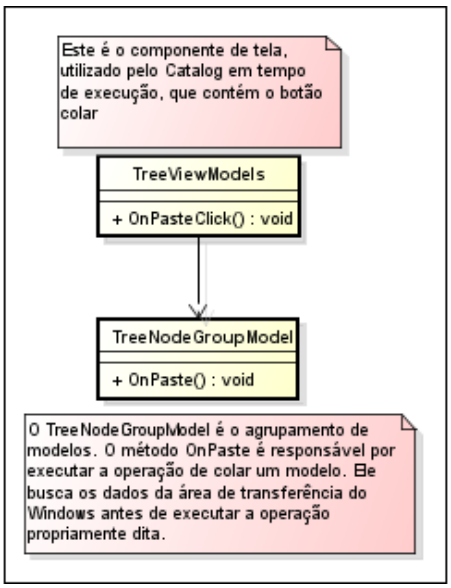
Para cada uma das partes que passaria por alterações, foi gerada uma tarefa. Como exemplo para este relato, é utilizada a tarefa referente à tela de cadastro de modelos, que são conjuntos de opções pré-definidas para os itens de modulação do Promob.

Nesta tela, o método de colagem era responsável por buscar os dados copiados da área de transferência do Windows, e a partir deles executar a colagem. Para testar isto, seria necessário colocar dados de testes na área de transferência. Isto implicaria que, ao executar o teste de unidade, o desenvolvedor iria ter a sua área de transferência alterada, o que não é um comportamento desejado.

Então, durante a modelagem da tarefa, foi feita também uma refatoração no método de colagem, dividindo-o dois métodos menores: um responsável por buscar os dados da área de transferência, e outro responsável por realizar a colagem. Assim, a interface em tempo de execução continua com o mesmo comportamento, enquanto

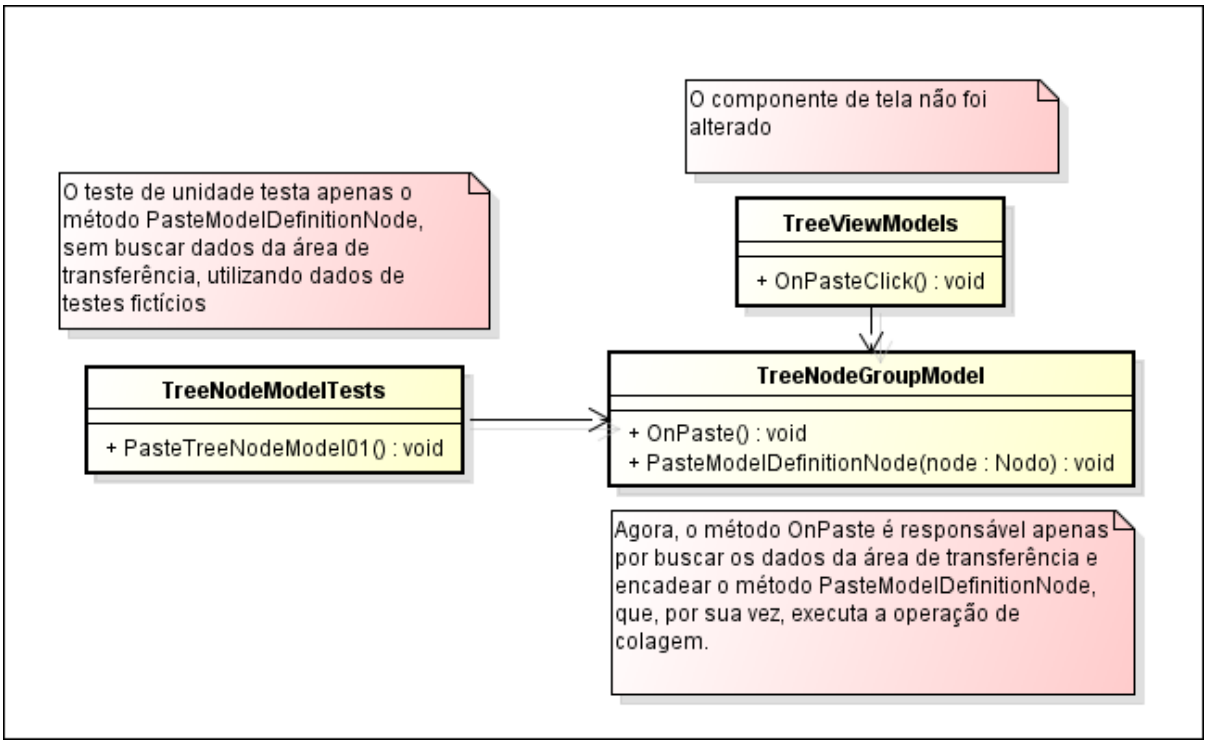
o teste de unidade testa apenas a operação de colagem utilizando dados fictícios de testes. As figuras 39 e 40 ilustram o modelo de classes antes e depois da refatoração, respectivamente.

**Figura 39 – Modelo de classes da funcionalidade colar, antes da refatoração**




Fonte: elaborado pelo autor

**Figura 40 – Modelo de classes da funcionalidade colar, depois da refatoração**




Fonte: elaborado pelo autor

Da mesma forma que no exemplo anterior, a unidade testável obriga a equipe a separar a camada de interface do *back end* que contém a implementação.

Após a refatoração prévia, o restante do processo seguiu normalmente.

### 5.2.3 Alterações realizadas

Durante a aplicação do processo, foram feitas algumas alterações, que são descritas a seguir.

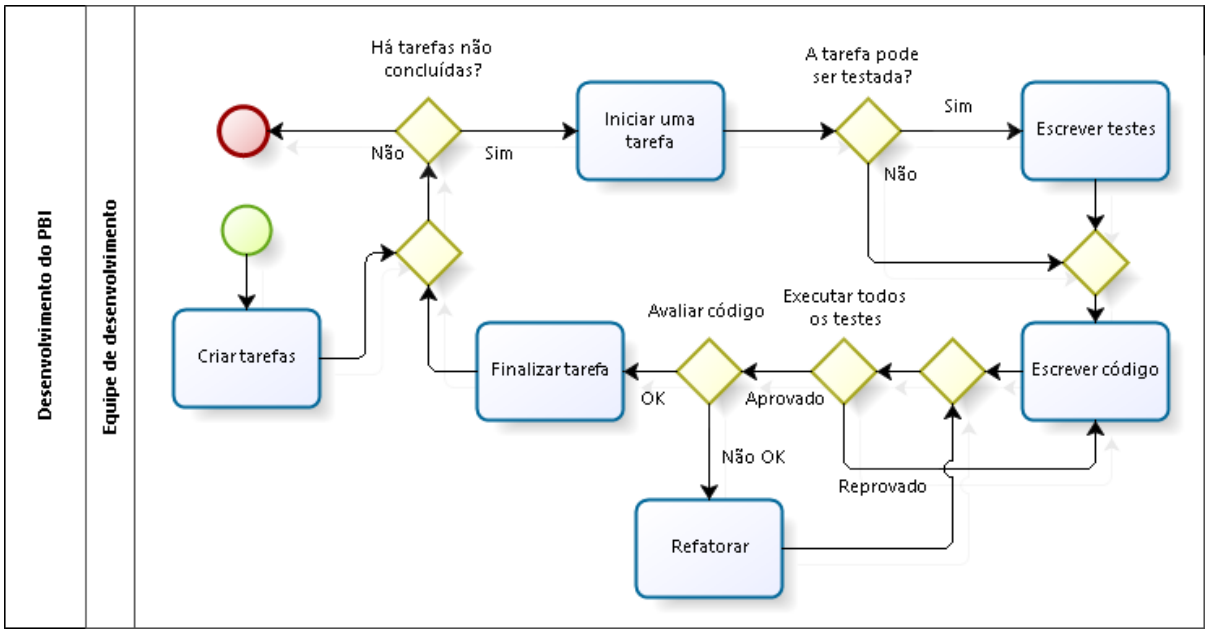
Não havia sido prevista a vinculação de um teste de unidade a várias tarefas. Algumas vezes, um único teste pode abranger várias tarefas que são muito relacionadas entre si. Com isto, o atributo TaskId foi alterado para TaskIds, aceitando vários IDs, conforme demonstrado na figura 41. Todavia, esta alteração não abre um precedente para escrever testes generalistas.

**Figura 41 – Atributo *TaskIds***

```
[TestCase]
[TaskIds(2122)]
public void DuplicateTreeNode_01()...

[TestCase]
[TaskIds(2122, 2123)]
public void DuplicateTreeNode_02()...
```

Fonte: elaborado pelo autor

**Figura 42 – Processo final de desenvolvimento dos PBIs**




Fonte: elaborado pelo autor

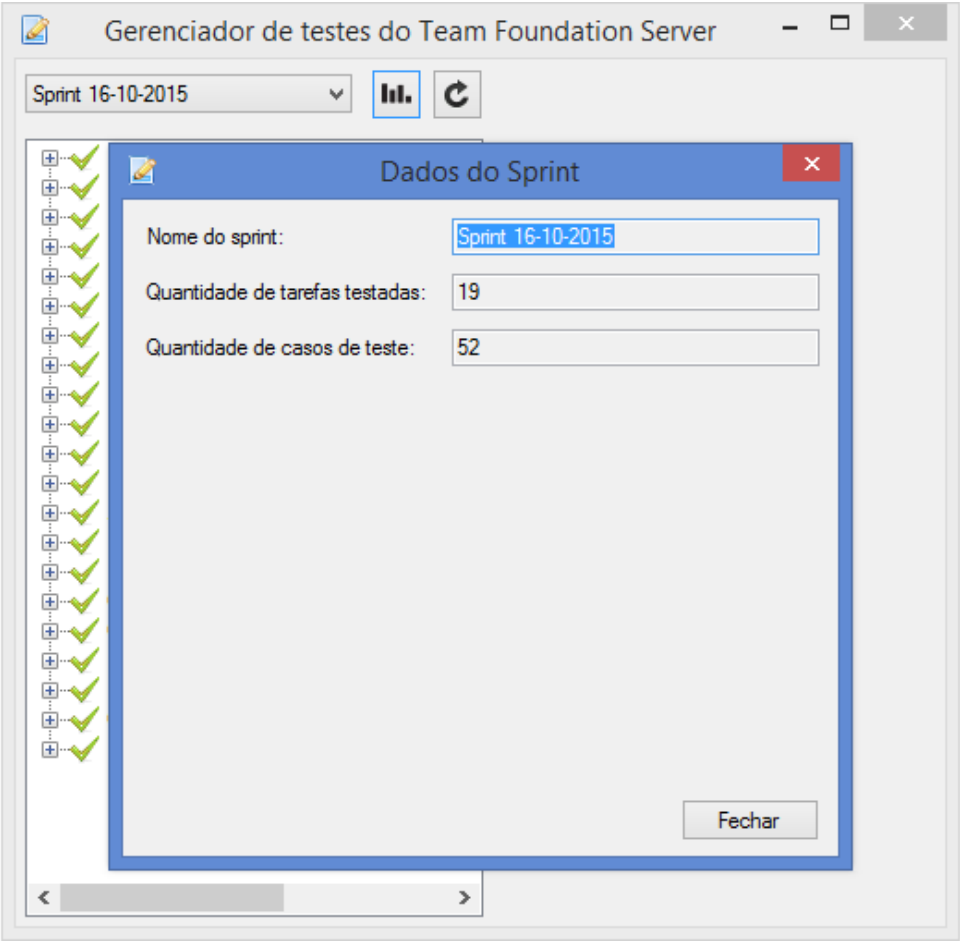
Como nem sempre o teste de uma tarefa pode ser automatizado (ex.: tarefas de interface com o usuário), e na proposta de solução não ficou claro como deveria

ser o processo para uma tarefa deste gênero, o fluxograma do novo processo foi redesenhado, conforme ilustrado na figura 42. É possível visualizar que os testes somente são escritos quando a tarefa pode ser testada de forma automatizada.

Também na figura 42, é possível visualizar que a refatoração tornou-se um passo opcional. Esta alteração foi feita porque a equipe de desenvolvimento entendeu que o ato de refatorar código é uma decisão que deve ser tomada para cada caso, dependendo da necessidade da tarefa e de algumas características do código que foi escrito, como: baixa coesão dos objetos desenvolvidos, presença de código duplicado, código pouco legível, etc.

Inicialmente, o gerenciador de testes procurava pelos testes de unidade apenas nos arquivos de código-fonte do projeto Cadastro.Tests, e não havia sido prevista a procura em outros projetos. Existe um módulo do Catalog, responsável pela exportação de listagens de itens das bases em arquivos-texto, que já era testado de forma automatizada antes da aplicação do novo processo. Porém, seus testes de unidade estavam em um outro projeto, o Promob.Catalog.LibraryLister.Tests. Este projeto foi incluído na busca do gerenciador de testes, passando a ser integrado ao novo processo também.

**Figura 43 – Funcionalidade de dados do *sprint* do gerenciador de testes**




Fonte: elaborado pelo autor

Foi adicionada a funcionalidade de dados do *sprint*, ilustrada na figura 43. Ao clicar no botão de dados do *sprint*, é aberta uma janela modal que exibe alguns dados sobre o *sprint* selecionado. Estes dados, que são a quantidade de tarefas testadas e a quantidade de casos de teste, são utilizados para a análise dos resultados do trabalho, sendo que com esta nova funcionalidade torna-se mais fácil obtê-los.

### 5.2.4 Dificuldades enfrentadas

A adesão aos testes automatizados foi parcial, no sentido de testar os itens do *backlog* de produto do Catalog. Dos requisitos que poderiam ser testados, excluindo os defeitos, 36% não foram testados de forma automatizada. Entre os defeitos testáveis corrigidos no período, 75% não foram testados. Estes dados foram coletados com o auxílio do gerenciador de testes desenvolvido neste trabalho.

A principal justificativa da equipe para não testar alguns dos itens do *backlog* foi a necessidade de fazer grandes refatorações para testar itens de baixa complexidade. De fato, a maior parte dos requisitos de maior complexidade (16 pontos ou mais) foi testada, enquanto não foi dada a mesma atenção ao requisitos menos complexos.

Há um outro fator importante presente em várias equipes de desenvolvimento da empresa Promob: não há uma cultura de desenvolver software dando importância aos testes, e são poucos os programadores que aplicam TDD, testes automatizados ou qualquer outra técnica que facilite os testes em suas rotinas de trabalho.

Conforme previsto na proposta de solução, este trabalho visa resultados a médio prazo, pois espera-se que haja um ganho real nos testes de regressão quando já houver uma quantidade considerável de testes de unidade, diminuindo assim a quantidade de defeitos. O fato de haver poucos resultados a curto prazo pode ter sido determinante para a adesão aos novos testes não ter sido imediata.

## 5.3 Resultados

Na tabela 3, é possível visualizar os defeitos corrigidos durante os quatro *sprints* em que o novo processo de testes foi aplicado.

Já na tabela 4, é possível observar as causas dos defeitos corrigidos durante este mesmo período.

**Tabela 3 – Quantidade de defeitos corrigidos a cada sprint, durante a aplicação do novo proceso de testes**

| Data de início | Data de finalização | Defeitos corrigidos |
|---|---|---|
| 21/09/2015 | 02/10/2015 | 8 |
| 05/10/2015 | 16/10/2015 | 8 |
| 19/10/2015 | 30/10/2015 | 8 |
| 03/11/2015 | 13/11/2015 | 9 |

Fonte: elaborado pelo autor

**Tabela 4 – Classificação dos defeitos corrigidos por causa, durante a aplicação do novo processo de testes**

| Causa | Quantidade de defeitos | % do total |
|---|---|---|
| Erro na análise ou interpretação do requisito | 8 | 24,2% |
| **Erro no desenvolvimento ou nos testes** | 25 | 75,8% |

Fonte: elaborado pelo autor

Conforme é possível observar, a quantidade média de defeitos não diminuiu de forma significativa durante a aplicação do novo processo de testes. Apesar disto, apenas três destes defeitos foram consequência de erros de desenvolvimento destes quatro *sprints*, sendo que neste período foram feitas duas liberações do produto.

Já era esperado que continuassem ocorrendo defeitos por consequência dos *sprints* anteriores, mas também era esperado que a quantidade destes defeitos diminuísse, pelo motivo de o Catalog estar em constante manutenção.

## 5.4 Considerações finais

No capítulo 2, foi citado que “os testes podem mostrar apenas a presença de erros, e não a sua ausência”, de acordo com Dijkstra et al. (1972, apud Sommerville (2011, p. 145)). Levando isto em conta, ainda não é possível afirmar se os *sprints* em que o novo processo foi aplicado ocasionarão defeitos no futuro, embora espere-se que não. Caso esta expectativa se confirme, a quantidade de defeitos deverá diminuir nos próximos *sprints*.

Um fato é que a introdução do novo processo de testes induziu a melhorias no desenvolvimento do Catalog. Já foi citado durante este capítulo que as modelagens das tarefas realizadas tornaram-se mais consistentes. Há a percepção de que a qualidade do código produzido pela equipe melhorou.

Há também os relatos dos integrantes da equipe, que contam que, ao menos nos maiores requisitos que desenvolveram, os testes de unidade foram muito úteis durante a escrita do código funcional, pois tornou-se muito mais fácil executar os testes automatizados do que os testes manuais.

Estes benefícios imediatos certamente contribuíram para que houvesse poucos defeitos decorrentes dos quatro primeiros *sprints* com o novo processo.

# 6 Conclusão

No início deste trabalho, foi determinado o objetivo geral, que era definir um novo processo de testes, baseado na técnica TDD, para ser aplicado na empresa Promob, além de uma ferramenta para apoiar este processo. Desta forma, seria possível responder à questão de pesquisa, que questionava se a técnica TDD poderia diminuir a quantidade de defeitos liberados para produção no cenário da empresa.

Ao decorrer do trabalho, o novo processo e a ferramenta de apoio foram implementados e foi feito um estudo de caso para analisar os resultados desta aplicação. De imediato, a quantidade de defeitos corrigidos a cada *sprint* não diminuiu de forma significativa, embora tenha sido visto que a maioria destes defeitos foram consequências de erros de desenvolvimento dos *sprints* anteriores.

Em algum momento, os defeitos mais antigos deverão cessar, visto que o produto está em constante manutenção. Desta forma, se os *sprints* do estudo de caso não geraram novos defeitos, a quantidade de defeitos corrigidos deverá diminuir.

A utilização do novo processo baseado em TDD melhorou a qualidade do desenvolvimento de forma não-quantitativa. Os exemplos de modelos de classes das figuras 36 e 40 ilustram que a modelagem do software está sendo feita de forma mais consistente, pois o TDD obrigou os programadores a tal. Além disto, a redução da dificuldade em testar e validar o desenvolvimento certamente contribuiu para a realização de liberações mais seguras.

Melo e Ferreira (2010) ressaltam que a implantação de métodos ágeis em uma organização é um processo lento e complexo, e que os primeiros resultados obtidos após a implantação motivam a organização a explorar novas possibilidades de trabalho. De fato, o TDD, sendo uma prática apresentada como parte dos métodos ágeis, representa uma quebra de paradigma que pôde ser sentida na prática durante o estudo de caso. Apenas após os primeiros resultados a equipe de desenvolvimento do Catalog sentiu-se incentivada a aplicar o novo processo com maior rigor, tornando a adesão ocorrer de forma gradativa. Mesmo após dois meses de aplicação do processo, há várias situações testáveis que não estão sendo testadas.

Para concluir este trabalho, cabe colocar que, de acordo com Audy (2015), um dos principais valores da abordagem ágil para desenvolvimento de software é a geração de oportunidades e ambiente com foco em melhoria contínua, privilegiando sempre a interação entre as pessoas, tanto internamente ao time, quanto do time com seus clientes, fornecedores, parceiros e demais envolvidos.

Então, é válido afirmar que uma das responsabilidades da equipe ágil é buscar

formas de melhorar os suas rotinas de desenvolvimento de forma constante, sendo que este trabalho é um exemplo deste valor em aplicação. Existe a possibilidade de tomar infinitas iniciativas na equipe que não se restringem ao TDD, e as iniciativas podem partir de todos os integrantes.

Algumas atividades futuras podem ser:

- Buscar o aprimoramento do desenvolvimento *test-first*, estudando outras formas eficientes de testar software além dos testes automatizados;
- Estudos em métodos estatísticos para análise de testes, oferecendo dados para os programadores melhorarem seu trabalho, tais como: onde eles mais erram, quais são as partes do software mais críticas, etc;
- Estudos na aplicação de padrões de codificação de sistemas, visando a utilização de testes automatizados de forma fácil.

# Referências


AMBLER, S. W. *Modelagem Ágil: Práticas eficazes para a Programação eXtrema e o Processo Unificado*. Porto Alegre: Bookman, 2004. Citado na página 10.

AUDY, J. *Scrum 360 - Um guia completo e prático de agilidade*. São Paulo: Casa do Código, 2015. Citado 2 vezes nas páginas 15 e 65.

BECK, K. *TDD: desenvolvimento guiado por testes*. Porto Alegre: Bookman, 2010. Citado 4 vezes nas páginas 3, 10, 20 e 21.

BECK, K. et al. *Manifesto para Desenvolvimento Ágil de Software*. 2001. Data de acesso: 22/11/2015. Disponível em: <http://agilemanifesto.org/iso/ptbr/>. Citado 3 vezes nas páginas 3, 10 e 14.

FOGGETTI, C. *Gestão ágil de projetos*. São Paulo: Education do Brasil, 2014. Citado 5 vezes nas páginas 3, 10, 21, 22 e 35.

FOWLER, M. *Refatoração: aperfeiçoando o projeto do código existente*. Porto Alegre: Bookman, 2004. Citado na página 22.

MELO, C. de O.; FERREIRA, G. R. M. Adoção de métodos ágeis em uma Instituição Pública de grande porte - um estudo de caso. In: *Agile Brazil*. [s.n.], 2010. Data de acesso: 23/11/2015. Disponível em: <http://agilcoop.org.br/files/WBMA_Melo_e_Ferreira.pdf>. Citado na página 65.

MILANEZ, M. V. *Test Driven Development: uma abordagem baseada em Use Cases*. Dissertação (Mestrado) — Pontifícia Universidade Católica de São Paulo, 2014. Data de acesso: 21/06/2015. Disponível em: <http://www.sapientia.pucsp.br/tde_arquivos/33/TDE-2014-10-28T12:49:48Z-15499/Publico/MarcusViniciusMilanez.pdf>. Citado na página 22.

MOLINARI, L. *Testes de Software: Produzindo Sistemas Melhores e Mais Confiáveis*. São Paulo: Érica, 2008. Citado 2 vezes nas páginas 18 e 19.

PHAM, A.; PHAM, P.-V. *Scrum em ação: gerenciamento e desenvolvimento Ágil de projetos de software*. São Paulo: Novatec Editora, 2011. Citado 3 vezes nas páginas 14, 16 e 17.

PRESSMAN, R. S. *Engenharia de software: uma abordagem profissional*. Porto Alegre: AMGH, 2011. Citado na página 18.

SCHWABER, K.; SUTHERLAND, J. *Guia do Scrum*. 2014. Data de acesso: 22/11/2015. Disponível em: <http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-Portuguese-BR.pdf>. Citado 3 vezes nas páginas 10, 15 e 17.

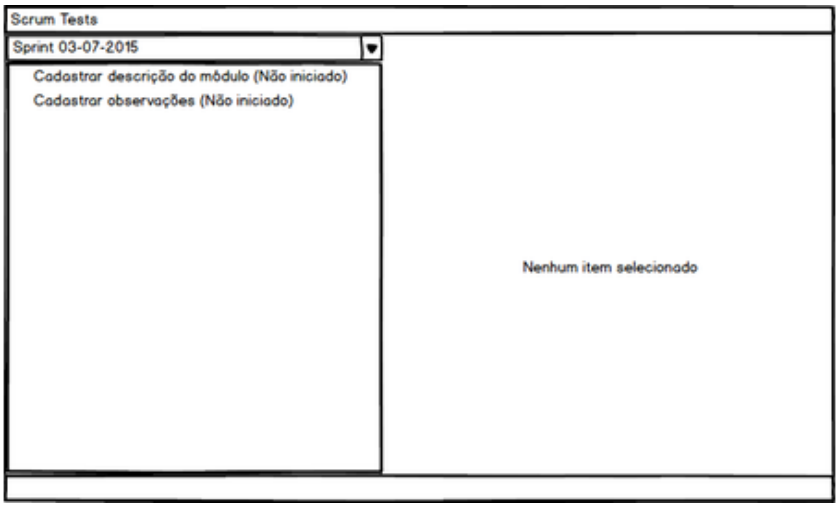
SOMMERVILLE, I. F. *Engenharia de Software*. São Paulo: Pearson Prentice Hall, 2011. Citado 8 vezes nas páginas 10, 14, 15, 16, 17, 19, 34 e 63.

# Apêndices

# APÊNDICE A – Protótipos de tela

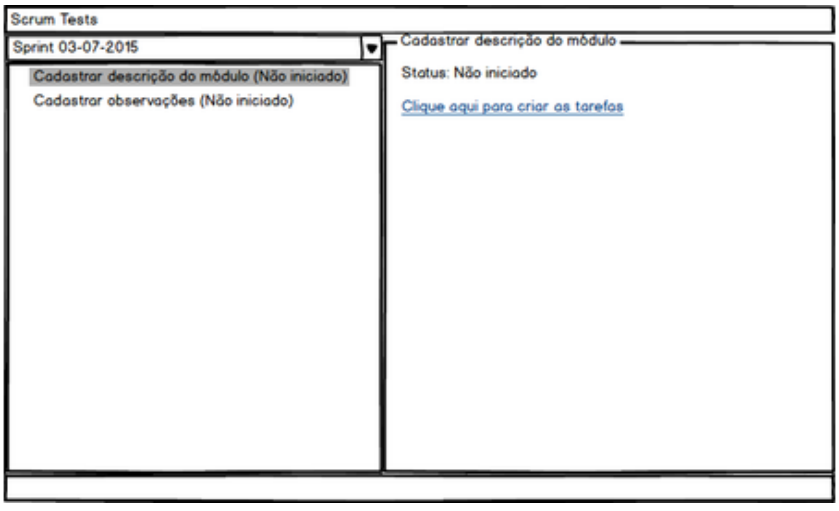
Neste apêndice, são apresentados os protótipos de tela definidos na proposta de solução (seção 3.3.2).

**Figura 44 – Tela principal (requisito 1)**



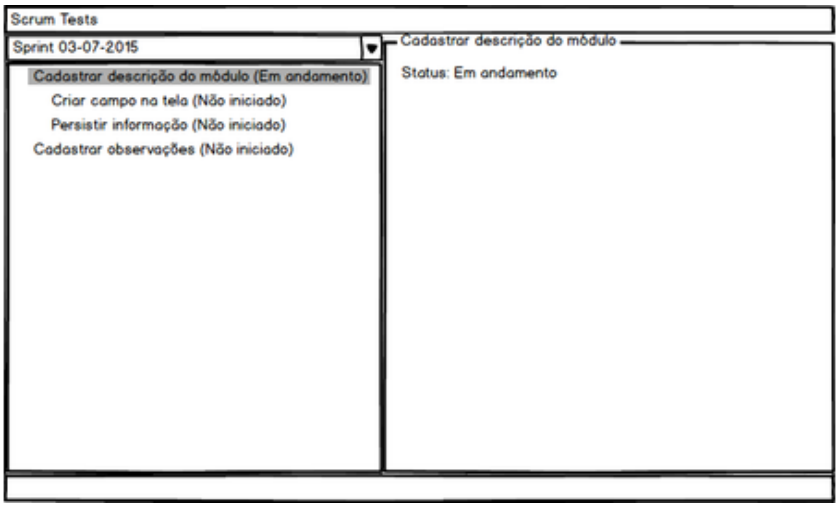

Fonte: elaborado pelo autor

**Figura 45 – Tela principal com um PBI selecionado (requisito 2)**



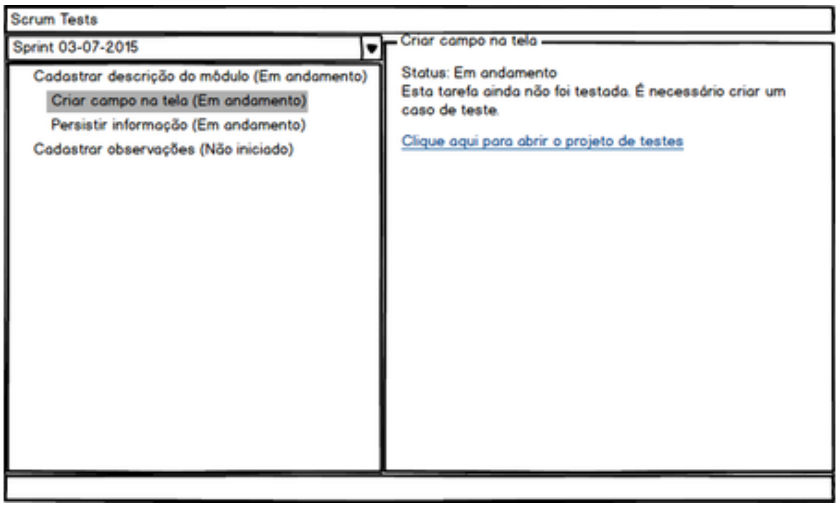

Fonte: elaborado pelo autor

**Figura 46 – Tela principal após a criação das tarefas de um PBI (requisito 2)**



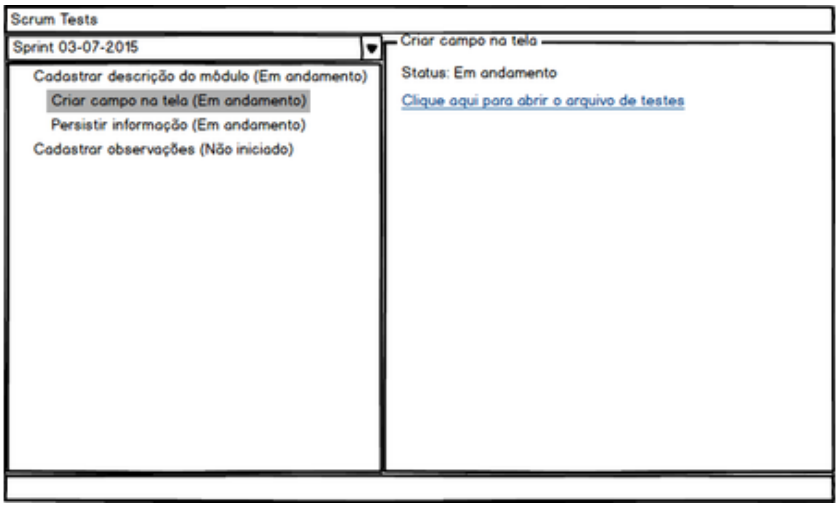

Fonte: elaborado pelo autor

**Figura 47 – Tela principal com uma tarefa selecionada (requisito 3)**



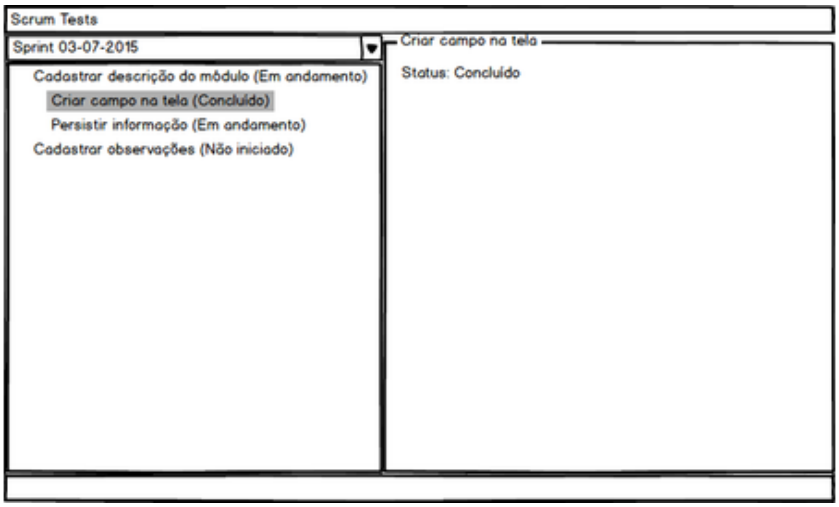

Fonte: elaborado pelo autor

**Figura 48 – Tarefa selecionada após a criação do teste (requisito 3)**




Fonte: elaborado pelo autor

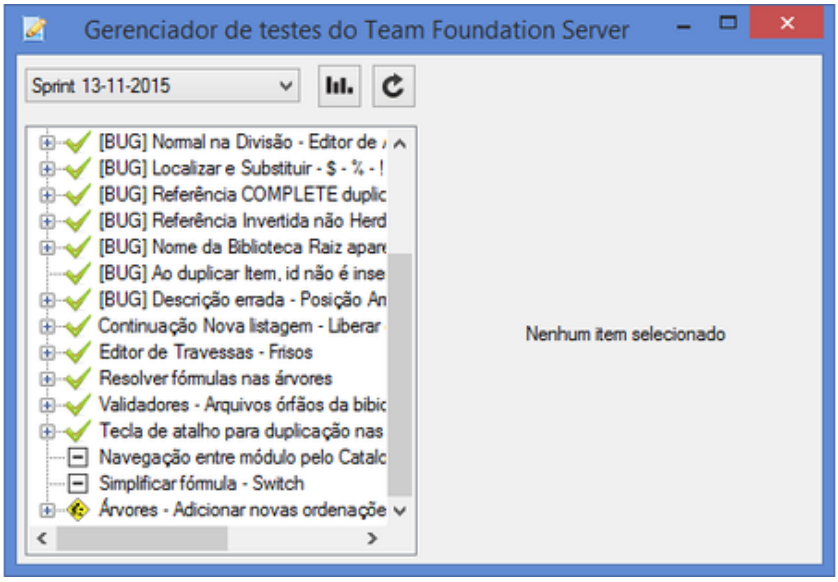
**Figura 49 – Tarefa selecionada após a conclusão (requisito 4)**




Fonte: elaborado pelo autor

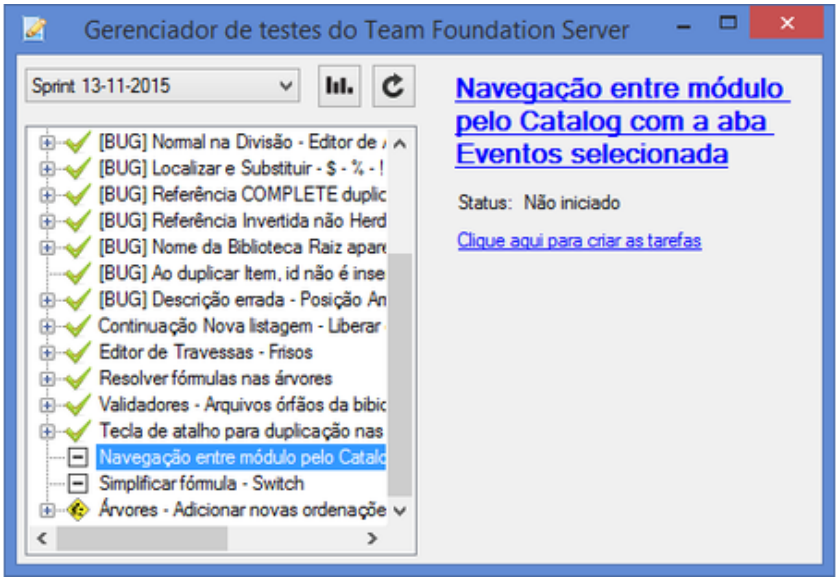
# APÊNDICE B – Implementação dos protótipos de tela

**Figura 50 – Implementação da tela principal (requisito 1)**



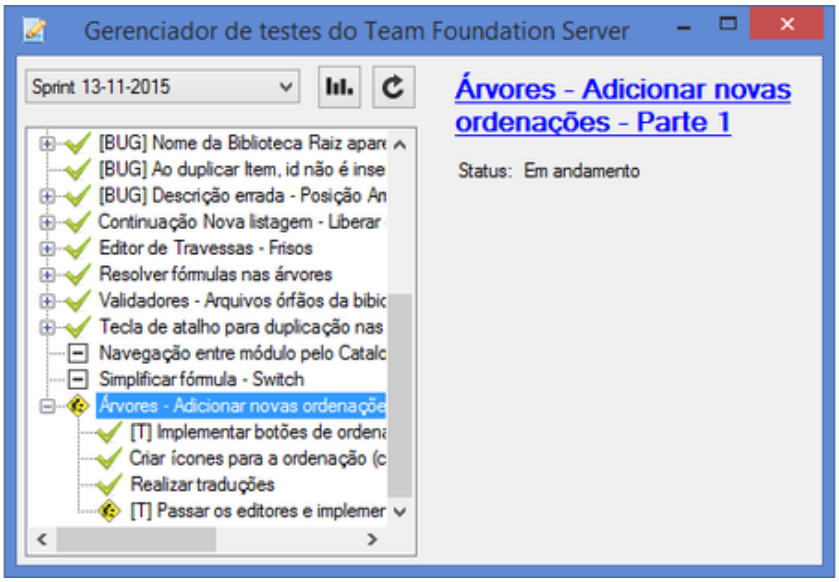

Fonte: elaborado pelo autor

**Figura 51 – Implementação da tela principal com um PBI selecionado (requisito 2)**



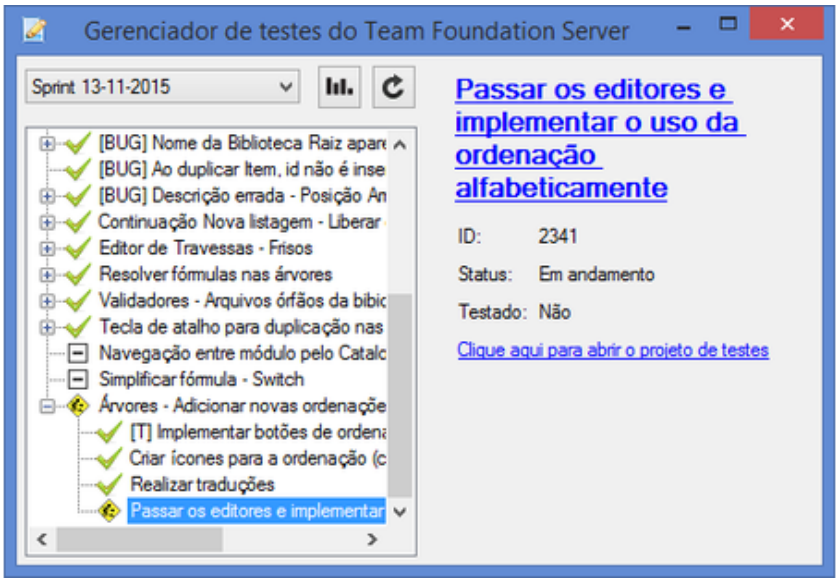

Fonte: elaborado pelo autor

**Figura 52 – Implementação da tela principal após a criação das tarefas de um PBI (requisito 2)**



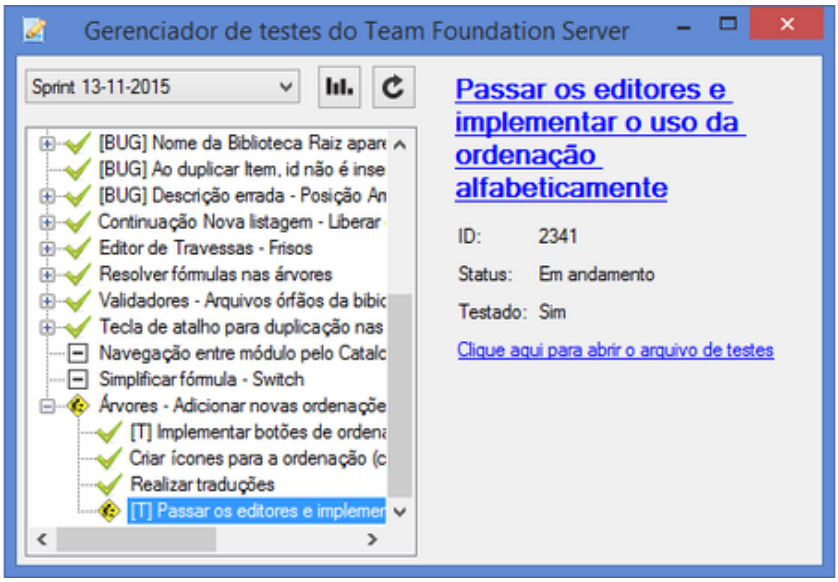

Fonte: elaborado pelo autor

**Figura 53 – Implementação da tela principal com uma tarefa selecionada (requisito 3)**



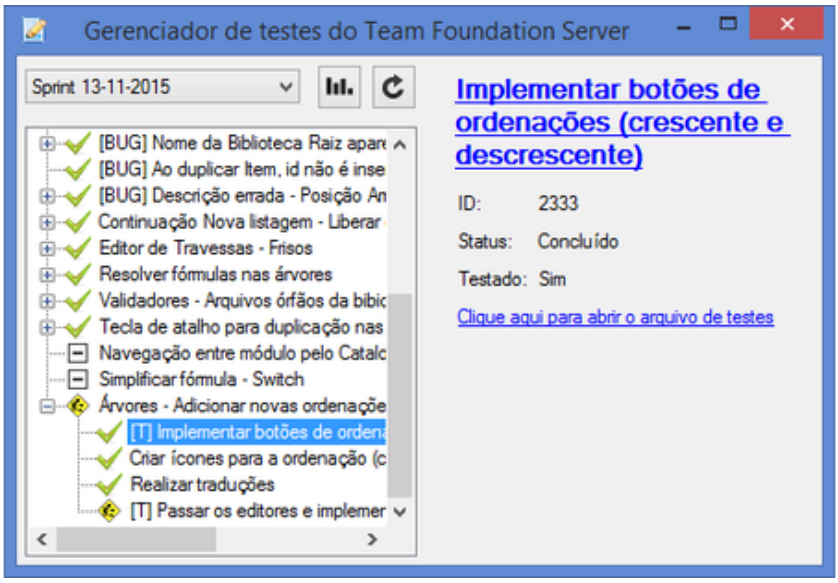

Fonte: elaborado pelo autor

**Figura 54 – Implementação da tarefa selecionada após a criação do teste (requisito 3)**




Fonte: elaborado pelo autor

**Figura 55 – Implementação da tarefa selecionada após a conclusão (requisito 4)**




Fonte: elaborado pelo autor

# APÊNDICE C – Tabelas de requisitos dos *sprints* com o novo processo

Neste apêndice, são apresentadas as listas de PBIs dos quatro *sprints* durante os quais foi feito o estudo de caso do capítulo 5.

Os quatro *sprints* têm o objetivo de fazer manutenções e ajustes gerais no Promob Catalog. A maioria dos requisitos destes *sprints* visam a melhoria de usabilidade, desempenho, e a criação de ferramentas que melhorem a produtividade do usuário. Há também alguns requisitos para atender necessidades específicas de alguns clientes da Promob.

Durante o estudo de caso, o planejamento dos *sprints* visava o atendimento de 140 pontos de complexidade por *sprint*, sendo 80 pontos atendidos pelo programador A e 60 pontos pelo programador B, que é estagiário.

**Tabela 5 – PBIs do Sprint 21/09/2015 - 02/10/2015**

| ID | Título | Tipo | Complexidade |
|---|---|---|---|
| 2112 | Ordenação de orçamentos incorreta | Defeito | 4 |
| 2113 | Montagem de Abreviatura - Dimensão de Inserção | Defeito | 8 |
| 2116 | Editor de Portas - Checkbox 'Editar' das Variáveis | Defeito | 4 |
| 2128 | Ao copiar pasta de materiais gera inconformidade | Defeito | 3 |
| 2132 | Opções de Remoção de modelos associados | Defeito | 4 |
| 2135 | Reindexar não salva as informações de versão e revisão | Defeito | 1 |
| 2137 | Gaveta interna dos armários inferiores sem categoria | Defeito | 2 |
| 2164 | SHIFT+TAB não funciona nos campos dos editores | Defeito | 1 |
| 2042 | Editor de Imagens - Fazer funcionar | PBI | 12 |
| 2061 | Idiomas | PBI | 4 |
| 2075 | Copiar/colar em grupos de Portas Deslizantes e Travessas | PBI | 8 |

| ID | Título | Tipo | Complexi-dade |
|---|---|---|---|
| 2090 | Continuação: AutoComplete - Montadores Find e Checked | PBI | 32 |
| 2096 | Editor de cadastro de itens do construtor | PBI | 4 |
| 2100 | Editor de Modelos - Duplicação de modelos definição | PBI | 4 |
| 2102 | Duplicar não está padronizado em todos os locais | PBI | 12 |
| 2107 | Melhoria da mensagem ao remover entidade | PBI | 1 |
| 2108 | Editor de Módulos - Atributos - box Atributos Editáveis no Ambiente | PBI | 2 |
| 2109 | Editor de Estruturas - Criar ícone Duplicar | PBI | 4 |
| 2174 | Migrar o Catalog para o .NET Framework 4.5 | PBI | 1 |

Fonte: elaborado pelo autor

**Tabela 6 – PBIs do Sprint 05/10/2015 - 16/10/2015**

| ID | Título | Tipo | Complexidade |
|---|---|---|---|
| 2088 | Migração ou duplicação de módulos do construtor ocasiona erro no resize | Defeito | 6 |
| 2167 | Posicionamento de Agregado não é Alterado | Defeito | 3 |
| 2169 | Estruturas são duplicadas | Defeito | 2 |
| 2182 | Autocompletar no cadastro de referências | Defeito | 8 |
| 2188 | Biblioteca raiz não aparece ao reindexar | Defeito | 1 |
| 2193 | Atributos - Não salva os valores propostos | Defeito | 2 |
| 2222 | Validador de Arquivos Orfãos - Não valida corretamente os Aggregates | Defeito | 1 |
| 2225 | Catalog - Exclusão de Itens e Grupos X Estruturas - Reaberta | Defeito | 1 |
| 2097 | Interface - Melhorar os botões da árvore de navegação | PBI | 8 |
| 2129 | Padronização das Fórmulas dos Agregados | PBI | 1 |
| 2131 | Nova propriedade de material “TexturePivotPointAlignment” | PBI | 8 |
| 2149 | Atualizar os Materiais dinamicamente | PBI | 8 |

| ID | Título | Tipo | Complexi-dade |
|---|---|---|---|
| 2161 | Bloquear ENTER nos campos do Catalog | PBI | 2 |
| 2166 | Reformular interface do editor de regras de orçamento | PBI | 24 |
| 2168 | Copiar e colar não está padronizado em todos os locais | PBI | 24 |
| 2171 | Geometria - nomear arestas | PBI | 8 |
| 2177 | Editor de travessas - Condições - Negação | PBI | 6 |
| 2178 | Editor de travessas - Condição - Condições de posicionamento do painel | PBI | 4 |
| 2179 | Coleção de referências - Menu de contexto e atalhos de teclado | PBI | 6 |
| 2230 | Editor de Materiais - Adicionar duplicação | PBI | 12 |

Fonte: elaborado pelo autor

**Tabela 7 – PBIs do Sprint 19/10/2015 - 30/10/2015**

| ID | Título | Tipo | Complexidade |
|---|---|---|---|
| 2248 | Traduções - Tradução é apresentada de forma errada | Defeito | 1 |
| 2254 | Estruturas - Inconformidade ao cancelar adição | Defeito | 1 |
| 2258 | Ao editar algumas informações no editor de travessas somem outras informações do XML | Defeito | 3 |
| 2283 | Valores de atributos não apresentados de forma correta | Defeito | 2 |
| 2289 | Duplicação de entidade não copia TAG | Defeito | 2 |
| 2292 | MOVIE - Fórmulas não são aceitas pelo Catalog | Defeito | 2 |
| 2305 | Exemplo de uso da função ROUND no Editor de Fórmulas | Defeito | 1 |
| 2320 | Editor de Travessas - Travessas na definição de travessas não são removidas no XML | Defeito | 1 |
| 2268 | Nova listagem - Retirar tela com formato de saída da criação de templates | PBI | 1 |
| 2269 | Nova listagem - Template Default | PBI | 4 |
| 2270 | Nova listagem - Enfileirar listagem | PBI | 24 |
| 2293 | Alterar a opção dos modelos irmãos pela referência | PBI | 2 |

| ID | Título | Tipo | Complexi-dade |
|---|---|---|---|
| 2245 | Nova listagem - Liberar e testar | PBI | 24 |
| 2255 | Autocompletar - Melhorar o FIND | PBI | 8 |
| 2260 | Ajustes gerais do Catalog | PBI | 8 |
| 2302 | Editor de travessas - Nova opção | PBI | 3 |
| 2266 | Movimentar e organizar coleções de Desenho | PBI | 30 |

Fonte: elaborado pelo autor

**Tabela 8 – PBIs do Sprint 03/11/2015 - 13/11/2015**

| ID | Título | Tipo | Complexidade |
|---|---|---|---|
| 2342 | Condição não traduzida | Defeito | 1 |
| 2344 | Coleção de desenhos não mostra os campos | Defeito | 1 |
| 2357 | Orçamento - Associar Ícones | Defeito | 1 |
| 2358 | Normal na Divisão - Editor de Armários | Defeito | 1 |
| 2360 | Localizar e Substituir - $ - % - ! | Defeito | 1 |
| 2361 | Referência COMPLETE duplicada | Defeito | 1 |
| 2362 | Referência Invertida não Herda da Normal | Defeito | 1 |
| 2365 | Nome da Biblioteca Raiz aparece Vazio ao Reindexar | Defeito | 2 |
| 2374 | Descrição errada - Posição Armário | Defeito | 1 |
| 2339 | Ajuste interface Editor de Modelos | PBI | 2 |
| 2340 | Alterar ícone do abrir desenho | PBI | 1 |
| 2327 | Continuação Nova listagem - Liberar e testar | PBI | 8 |
| 2332 | Editor de Travessas - Frisos | PBI | 2 |
| 2250 | Resolver fórmulas nas árvores | PBI | 16 |
| 2282 | Validadores - Arquivos órfãos da bibioteca | PBI | 24 |
| 2356 | Tecla de atalho para duplicação nas árvores | PBI | 4 |
| 2256 | Navegação entre módulo pelo Catalog com a aba Eventos selecionada | PBI | 4 |

| ID | Título | Tipo | Complexidade |
|---|---|---|---|
| 2272 | Árvores - Adicionar novas ordenações - Parte 1 | PBI | 36 |

Fonte: elaborado pelo autor