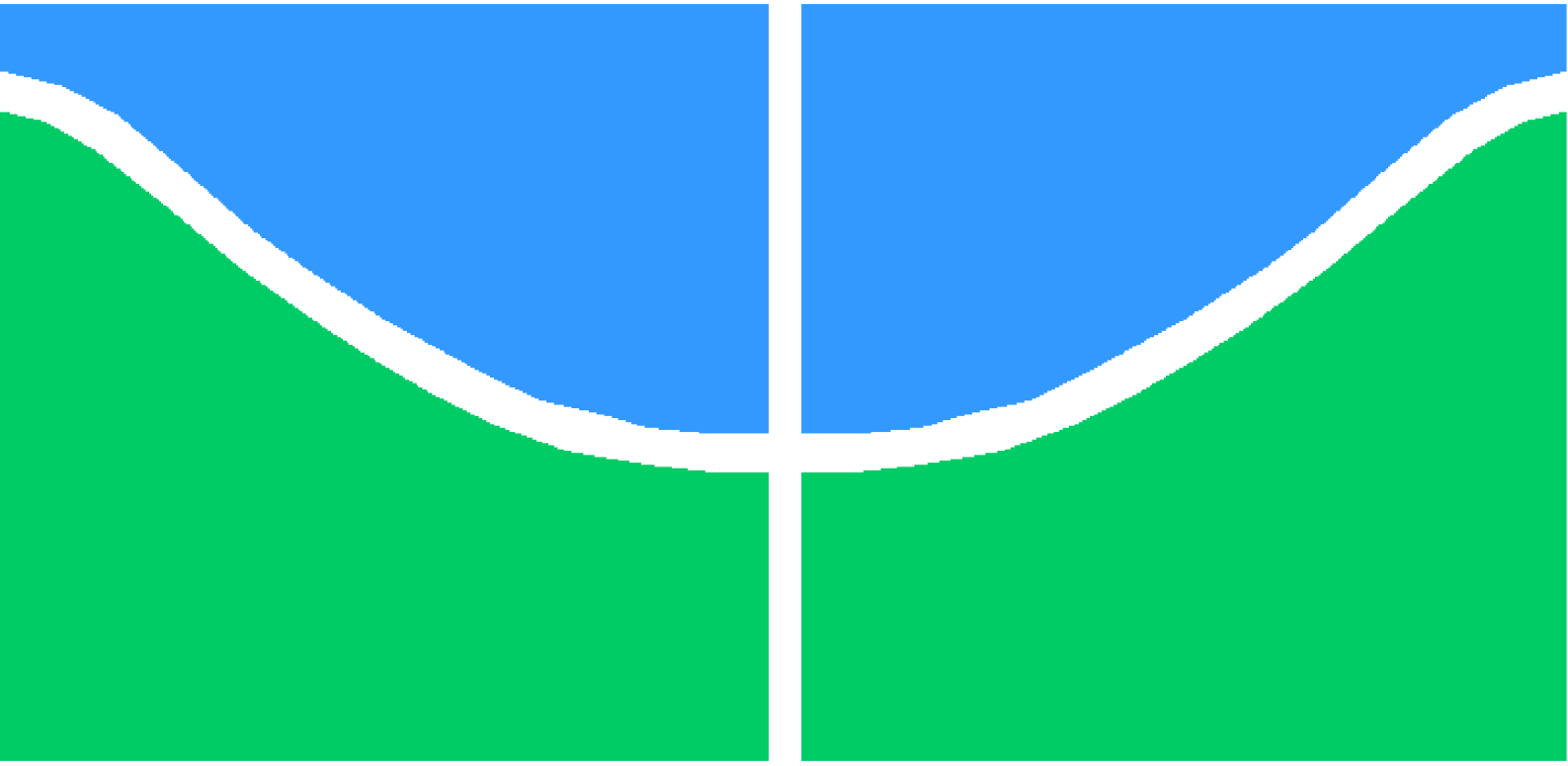

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Técnicas de testes automatizados em processos de desenvolvimento de software empírico: um estudo de caso do projeto Noosfero

Autor: Rodrigo Medeiros Soares da Silva

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2014

Rodrigo Medeiros Soares da Silva

Técnicas de testes automatizados em processos de desenvolvimento de software empírico: um estudo de caso do projeto Noosfero

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2014

Rodrigo Medeiros Soares da Silva

Técnicas de testes automatizados em processos de desenvolvimento de software empírico: um estudo de caso do projeto Noosfero / Rodrigo Medeiros Soares da Silva. – Brasília, DF, 2014-

36 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Testes. 2. Software livre. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Técnicas de testes automatizados em processos de desenvolvimento de software empírico: um estudo de caso do projeto Noosfero

CDU 02:141:005.6

Rodrigo Medeiros Soares da Silva

Técnicas de testes automatizados em processos de desenvolvimento de software empírico: um estudo de caso do projeto Noosfero

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, Dezembro de 2014:

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Prof..
Convidado 1

Prof.
Convidado 2

Brasília, DF
2014

Resumo

Abstract

Lista de ilustrações

Lista de abreviaturas e siglas

BDD	Behavior Driven Development
FGA	Faculdade UnB Gama
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICC	Instituto Central de Ciências
LAPPIS	Laboratório de Produção, Pesquisa e Inovação em Software
LDAP	Lightweight Directory Access Protocol
MES	Manutenção e Evolução de Software
MVC	Model-View-Controller
ProIC	Projeto de Iniciação Científica
PuSH	PubSubHubbub
Rails	Ruby on Rails
SMT	Tecnologias de Mídia Social
SSL	Secure Socket Layer
TCC	Trabalho de Conclusão de Curso
TDD	Test Driven Develepment
UnB	Universidade de Brasília
USP	Universidade de São Paulo
W3C	World Wide Web Consortium
XP	Extreme Programming

Sumário

1 Introdução

1.1 Objetivos

1.1.1 Objetivos Gerais

1.1.2 Específicos

1.2 Organização do Trabalho

2 Testes

Testar é uma pratica intrínseca ao desenvolvimento e é antiga a necessidade de cirar programas para testar cenários específicos (EVERETT et al., 2007). Testes de serão abordados neste cápítulo, iniciando com uma visão geral de testes automatizados e conhecendo algumas práticas e padrões utilizados para desenvolver os mesmos. A automação de testes é uma prática ágil, eficaz e de baixo custo para melhorar a qualidade dos sistemas de software.

No entanto utilizar testes automatizados como uma premissa básica do desenvolvimento é um fenômeno relativamente recente,] com início em meados da década de 1990 (POTEL; COTTER, 1995). Além do fato de ser uma técnica bastante utilizada pelas metodologias ágeis de desenvolvimento.

2.1 Testes Automatizados

Testes automatizados é a prática de tornar os testes de software independentes da intervenção humana, criando scripts ou programas simples de computador que exercitam o sistema em teste, capturam os efeitos colaterais e fazem verificações, tudo automaticamente e dinamicamente (MESZAROS; WESLEY, 2007).

Os testes automatizados afetam diretamente a qualidade dos sistemas de software, portanto agregam valor ao produto final, mesmo que os artefatos adicionais produzidos não sejam visíveis para os usuários finais do sistemas. Estes testes podem ser divididos em diversos tipo, o que facilita a manutenção dos mesmos, coleta de métricas.

1. **Testes de unidade:** teste de correção responsável por testar os menores trechos de código de um sistema que possui um comportamento definido e nomeado. Normalmente, ele é associado a funções para linguagens procedimentais e métodos em linguagens orientadas a objetos.
2. **Testes funcionais:**
3. **Testes de integração:**denominação ampla que representa a busca de erros de relacionamento entre quaisquer módulo de um software, incluindo desde a integração de pequenas unidades até a integração de bibliotecas das quais um sistema depende, servidores e gerenciadores de banco de dados.
4. **Testes de interface de usuário** testes que verificam a correção por meio da simulação de eventos de usuário, a partir destes eventos, são feitas verificações na interface e em outras camadas.

5. **Testes de leiaute:** testes que buscam avaliar a beleza da interface e verificar a presença de erros após a renderização, dificeis de indentificar com testes comuns de interface

6. **Testes de aceitação:** são testes de correção e validação, idealmente especificados por clientes ou usuários finais do sistema para verificar se um modulo funciona como foi especificado (MARTIN, 2005). Testes de aceitação devem utilizar linguagem proxima da natural para evitar problemas de interpretação e de ambiguidades (MUGRIDGE; DCUNNINGHAM, 2005).

7. **Testes de desempenho:** testes que executam trechos do sistema e armazenam os tempos de duração obtidos, que ajudam a identificar gargalos que precisam de otimização para diminuir o tempo de resposta para o usuario (LIU, 2009).

8. **Testes de carga:** teste que exercita o sistema sobre condições de uso intenso para avaliar se a infraestrutura é adequada para a expectativa de uso do sistema (AVRITZE; WEYUKER, 1994).

9. **Testes de estresse:** teste que visa descobrir os limites do uso da infraestrutura, isto é , qual a quantidade máxima de usuários e requisições que o sistema consegue antender corretamente e em um tempo aceitável.

10. **Testes de longevidade:** teste que tem por objetivo encontrar erros somente visiveis com um

11. longo tempo de execuçao do sistema, erros que podem ser de cache, replicação, execução de serviços agendados, vazamento de memória.

12. **Testes de segurança:** os testes de segurança servem para verificar se os dados ou funcionalidades confidenciais de um sistema estão protegidos de fraude ou de usuários não autorizados. A segurança de um sofware pode envolver aspectos de confidenciabilidade, integridade, autenticação, autorização, privacidade (WHITTAKER, 2006).

2.2 Técnicas de Desenvolvimento de testes automatizados

Automação de testes é uma técnica voltada principalmente para a melhoria de qualidade dos sistemas de software. No processo de desenvolvimento de software é fundamental controlar o custo do processo de testes, para isso baterias de testes automatizados devem ser bem definidas e implementadas. Assim é importante conhecer boas práticas e técnicas de de desenvolvimento de testes automatizados. Existem várias técnicas de desenvolvimento de software com testes que influenciam diretamente na qualidade do sistema.

Estas técnicas geralmente possuem um processo de atividades pequeno e simples, como TDD e BDD.

2.2.1 TDD - Test Driven Development

Desenvolvimento dirigido por testes, também conhecido como TDD *(Test-Driven Develepment)* é uma técnica de desenvolvimento de software que se dá pela repetição dosciplinada de um ciclo curto de passos de implementação de testes e do sistema (KOSKELA, 2007). O ciclo de TDD é definido pelos seguintes passos:

1. Implementar um caso de teste;
2. Implementar um trecho do código suficiente para o novo caso de teste ter sucesso de tal modo que não quebre os testes previamente escritos;
3. Se necessário, refatorar o código produzido para que ele fique mais organizado;

A técnica de desenvolvimento dirigido por testes foi definida por Kent Beck em seu livro *Test-Driven Development: By Example* (BECK, 2002). Os passos estão representados na figura abaixo:

Figura 1 – Ciclo de atividades TDD (BECK, 2002)

2.2.1.1 Benefícios do TDD

Uma boa prática do TDD é a bateria de testes, que ajuda o desenvolvedor a evitar erros de regreção, quando o desenvolvimento de uma nova funcionalidade quebra uma já

existente. TDD também tende a contribuir com uma alta cobertura de código, uma fez que o desenvolvedor precisa escrever o testes antes da funcionalidade, possibilitando a criação de um código mais preciso, coeso e menos acoplado. Para Massol em *JUnit in Action*, "o objetivo de TDD é 'código claro que funciona' (MASSOL; HUSTED, 2003). TDD propõe o desenvolvimento sempre em pequenos passo, deve-se escrever testes sempre para uma menor funcionalidade possível, escrever o código mais simples que faça o teste passar e fazer sempre apenas uma refatoração por vez (BECK, 2002). Assim o desenvolvedor se detém a criar soluções simples, sempre acompanhado de um constante feedback dos testes. O ciclo curto de passos definidos por TDD cria uma dependência forte entre codificação e testes, o que favorece e facilita a criação de sistemas com alta testabilidade (BERNARDO, 2011). Índices altos de cobertura de código e testabilidade não garantem necessariamente qualidade do sistema, mas são métricas bem vistas para sistemas bem desenvolvidos Além de uma técnica de testes automatizados, TDD também é uma prática de desenvolvimento de software, pois muda a natureza do processo de codificação, auxiliando na produção de um código funcional e limpo.

2.2.2 BDD - Behavior Driven Development

Desenvolvimento dirido por comportamento *(BDD - Behavior Driven Development)* é uma prática que recomenda o mesmo ciclo de desenvolvimento de TDD, contudo, induzindo a utilização de uma linguagem ubíqua entre cliente e equipe de desenvolvimento (BERNARDO, 2011), subistituindo termos como assert, *assert, test case, test suite* por termos mais comuns ao cliente, como *should, context, specification* O BDD é um processo de desenvolvimento de software baseado no TDD. Embora seja principalmente principalmente uma ideia de como um processo de desenvolvimento de software deve ser gerenciado. a prática do BDD assume a utilização de ferramentas como suporte para o desenvolvimento de software (HARING, 2011). O BDD utiliza essas ferramentas para que os testes tenham como ponto de partida o compartemento dos objetos. O BDD coloca em foco o comportamento em vez da estrutura e faz isso em todos os níveis de desenvolvimento. Uma vez que nós reconhecemos isso, ele muda a forma como pensamos sobre desenvolvimento para fora do código e começamos a pensar mais sobre as intereações sobre pessoas e sistemas, ou entre os objetos, do que sobre a estrutura do objeto (CHELIMSKKY et al., 2010). Para que o comportamento seja analisado, é necessário entender o ponto de vista do cliente, entendendo o comportamento que o sistema deve ter a partir da visão do cliente.

2.2.2.1 Príncipios do BDD

De acordo com Chelimky, estes são os três príncipios do BDD:

1. **O suficiente é suficiente:** parte da ideia de gerenciar o esfoço no planejamento

inicial do sistema, para não fazer menos nem mais do que o necessário para começar, o que se aplica também ao processo de automação.

2. **Agregar valor as partes interessadas:** Se você está fazendo algo que não agrega valor ou que não aumenta a capacidade de agregar valor, pare e faça outra coisa em seu lugar.

3. **Tudo é comportamento:** Do código à aplicação, pode-se usar o mesmo pensamento e as mesmas construções linguísticas para descrever o comportamento, em qualquer nível de granularidade.

O comportamento do sistema é descrito em historias de usuário, histórias que são escritas com a participação tanto de clientes como desemvolvedores do sistema. Assim cada hisória de usuário deve seguir, de certa forma, a seguinte estrutura: **Título:** do que se trata a história. **Narrativa:** deve identificar as partes interessadas para essa história, as funcionalidades requeridas, e o benefício deste comportamento. A narrativa de uma história pode ter vários formatos. **Critério de aceitação:** descrição de cada caso específico da narrativa, iniciado pela especificação da condição inicial do cenário, seguidos pelos estados que ativam este cenário, finalizado pelos estados esperados ao final do cenário.

2.2.3 Considerações Finais

Testes automatizados devem ser desenvolvidos com prioridade, buscando um rápido *feedback*, contribuindo assim com a melhoria do sistema. Para isso é necessário que os cenários de testes estejam bem definidos junto à equipe.

3 Métodos de Desenvolvimento Empírico

O Empirismo baseia-se na aquisição de sabedoria através da percepção do mundo externo, ou então do exame da atividade da nossa mente, que abstrai a realidade que nosé exterior e as modifica internamente (CHAUí, 2003). Mecanismos do Controle de Processo Empírico, onde ciclos de feedback constituem o núcleo da técnica de gerenciamento que são usadas em oposição ao tradicional gerenciamento de comando e controle.É uma forma de planejar e gerenciar projetos trazendo a autoridade da tomada de decisão a níveis de propriedade de operação e certeza (SCHWABER, 2004). Neste capítulo será abordado algumas ideias e práticas de processo de desenvolvimento de software que utilizam métodos ágeis, processos amplamente utilizados no desenvolvimentode software livre, outro assunto também abordado no decorrer deste texto.

3.1 Software Livre

Software livre é uma filosofia que trata programas de computadores como fontes de conhecimento que devem ser compartilhados entre a comunidade, evoluindo assim o desenvolvimento do pensamento no que se diz respeito a desenvolvimento de software. De acordo com Richard M. Stallman, ativista fundador do movimento software livre, um software deve seguir quatro princípios:

1. Liberdade de execução para qualquer uso;
2. Liberdade de estudar o funcionamento de um software e de adaptá-lo às suas necessidades
3. Liberdade de redistribuir cópias;
4. Liberdade de melhorar o software e de tornar as modificações públicas de modo que a comunidade se beneficie da melhoria (STALLMAN, 2001).

Um software é considerado software livre se segue estes quatro princípios, portanto o usuário deve poder utilizar, estudar e modificar o software como ele bem entender, não significa que o software é necessariamente de graça, mas a partir do momento em que se obtém posse de um programa um usuário pode modificar e redistribuir o mesmo programa. Para Stallman, a partir do momento em que os custos de desenvolvimento de um software são pagos, não há motivos para restrição de acesso, pois a disseminação de conhecimento é muito mais benéfica do que potenciais lucros para o produtor.

3.1.1 GNU e GNU GPL

Umas das grandes conquistas de Stallman e da Fre Software Foudation (FSF), principal organização dedicada a produção e à divulgação do software livre, foram o projeto GNU e a licença de software General Public License (GPL). O projeto GNU consistiu em desenvolver um sistema operacional baseado no sistema Unix, porém livre de código proprietário, proporcionando aos usuários do Unix um sistema totalmente compatível com o Unix, com seu código disponível para todos e a liberdade de buscar suporte e personalizações da forma que quisessem. om o GNU, também foi desenvolvida a GPL, licença que dá amparo legal e formaliza a ideologia de software livre, amplamente utilizada pelos software livres. No final do desenvolvimento do GNU, o finlandês Linus Torvalds iniciou o desenvolvimento de um núcleo de sistema operacional também baseado no Unix e deu o nome do núcleo de Linux, disponibilizando-o pela licença GNU GPL. Assim foi promovida a integração entre GNU e Linux, criando assim o GNU/Linux, amplamente utilizado até os dias de hoje.

3.2 Métodos ágeis

A utilização de métodos ágeis no desenvolvimento de software tem como caracteristicas intrínsecas a flexibilidade e rapidez nas respostas a mudanças. A agilidade, para uma organização de desenvolvimento de software, é a habilidade deadotar e reagir rapidamente e apropriadamente a mudanças no seu ambiente e por exigênciasimpostas pelos clientes (NERUR; MAHAPATRA; MANGALARAJ, 2005). Os métodos ágeis compartilham valores como comunicação, feedback constante, colaboração com o cliente e constante adaptação são baseados no manifesto ágil. Os quatro princípios básicos do manifesto ágil mostra o que se espera de qualquer método de desenvolvimento desta categoria:

1. Indivíduos e interações sobre processos e ferramentas;
2. Software funcionando sobre documentação extensiva;
3. Colaboração com o cliente sobre negocioação de contrato;
4. Responder as mudanças sobre seguir um planejamento;

Em projetos ágeis o cliente é mais ativo durante o processo de desenvolvimento, determinandoem conjunto com a equipe de desenvolvimento o que será desenvolvido, além de participarda validação. Os projetos ágeis também buscam estabelecer um tempo determinado e curto para entregade novas releases do sistema, com o objetivo de trazer mais satisfação ao cliente. A partir destes curtos ciclos, que são as interações, a equipe de desenvolvimento devese preocupar mais com a evolução dos requisitos, que pode gerar mudanças, porém mantém o projetoatualizados e diminui o riscos de grandes mudanças

a medida que o projeto chega ao final. Da perspectiva do produto, métodos ágeis são mais adequados quando os requisitos estão emergindo e mudando rapidamente, embora não exista um consenso completo neste ponto. De uma perspectiva organizacional, a aplicabilidade pode ser expressa examinando três dimensões chaves da organização: cultura, pessoal e comunicação. Em relação a estas áreas inúmeros fatores chave do sucesso podem ser identificados (COHEN; LINDVALL; COSTA, 2004).

3.2.1 Programação Extrema - XP

Um método ágil conhecido como Programação extrema, (Extreme Programming - XP) se tornou-se bastante popular por utilizar práticas focadas em codificação, como programação pareada, integração contínua e desenvolvimento dirigido por testes. O objetivo principal do XP é a excelência no desenvolvimento de software, visando um baixo custo, poucos defeitos, alta produtividade e alto retorno de investimento (SATO, 2007). O XP conta com algumas práticas de desenvolvimento para dar suporte à busca pelos objetivos citados, essas práticas são: refatoração, integração contínua, testes automatizados, código coletivo e programação em pares.

4 Estudo de Caso: Noosfero

Durante o quarto capítulo deste trabalho de conclusão de curso, discutiremos sobre o ambiente de trabalho e a plataforma de desenvolvimento, chamada Noosfero, assim como os testes e as funcionalidades desenvolvidas dentro do processo de colaboração com esta plataforma. Noosfero é uma plataforma desenvolvida pela Colivre (Cooperativa de Tecnologias Livre), possui licença AGPL e é utilizado, principalmente em universidades públicas, para desenvolvimento de redes colaborativas. O Noosfero foi desenvolvido na linguagem de programação Ruby, versão 1.8.7, e utiliza o framework Model-View-Controller (MVC) para aplicações web Ruby on Rails, versão 2.3.5. A escolha destas tecnologias, por parte dos criadores do Noosfero foi baseada fato de que o Ruby possui sintaxe simples, elegante e de fácil leitura, o que aumenta a manutenibilidade do sistema, uma característica importante num projeto de software livre que visa atrair desenvolvedores externos (MEIRELLES, 2013).

4.1 Desenvolvimento no processo de colaboração ao Noosfero

Por tratar de um software livre, a plataforma Noosfero possui uma grande quantidade de colaboradores, formado por equipes de desenvolvimento como na Universidade de Brasília, ou por desenvolvedores independentes. Assim, para que haja sucesso na colaboração, são feitas exigências durante o desenvolvimento, assim como testes bem definidos para aprovar novas funcionalidade. De acordo com os contribuidores do noosfero, a plataforma livre possui as seguintes caracteristicas:

1. Rede Social (três entidades: pessoas, comunidades e organizações);
2. CMS: pastas, artigos, RSS, upload e publicação de imagens e arquivos;
3. Blog com sistema de notificação de comentários;
4. Galeria de imagens;
5. e-Portfolios (individuais e de grupos);
6. Compartilhar Interesses;
7. Fóruns / Discussão Temática;
8. Agenda de eventos compartilhadas;
9. Vitrine Digital para exposição de produtos e serviços e carrinho de compras;

10. Acompanhamento de atividades de usuários e grupos;

11. Mensagens assíncronas e Web Chat.

Além da utilização da linguagem de programação Ruby, o desenvolvimento da plataforma Noosfero também se baseia em JavaScript, CSS, dentre outras linguagens, como pode-se ver na figura abaixo:

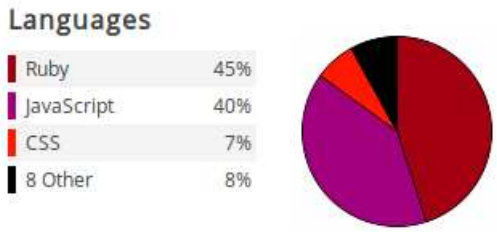

Figura 2 – Desenvolvimento Noosfero

4.1.1 Ciclo de desenvolvimento do Noosfero

O desenvolvimento para a plataforma Noosfero é realizado em ciclos e, pela Colivre, possui as seguintes fases: desenvolvimento, release 1, release 2 e release final.

4.1.1.1 Fase de Desenvolvimento

Antes da fase de desenvolvimento for iniciada é necessário que seja feita a documentação do que será desenvolvido. Os desenvolvedores responsáveis criam um Action Item (Item de ação), contendo a historia de usuário ou os requisitos do novo recurso. Um action item pode ser descrito como uma feature (novo recurso) ou como um bug (defeito) encontrado, como demonstrado nas figuras abaixo:

Além da documentação, é necessário o envio de um email a comunidade descrevendo o que será desenvolvido. A partir deste email, a comunidade irá avaliar a funcionalidade descrita e decidir se é um funcionalidade importante para se incorporar ou não. Este processo de avaliação tem a duração de uma semana, geralmente. Com a aprovação da comunidade são feitas as revisões do ciclo de desenvolvimento, que deve seguir os seguintes passos:

1. Criar um 'merge request' juntamente com o código;

2. Código é revisado pela comunidade;

3. Código é bom?

   Se sim:

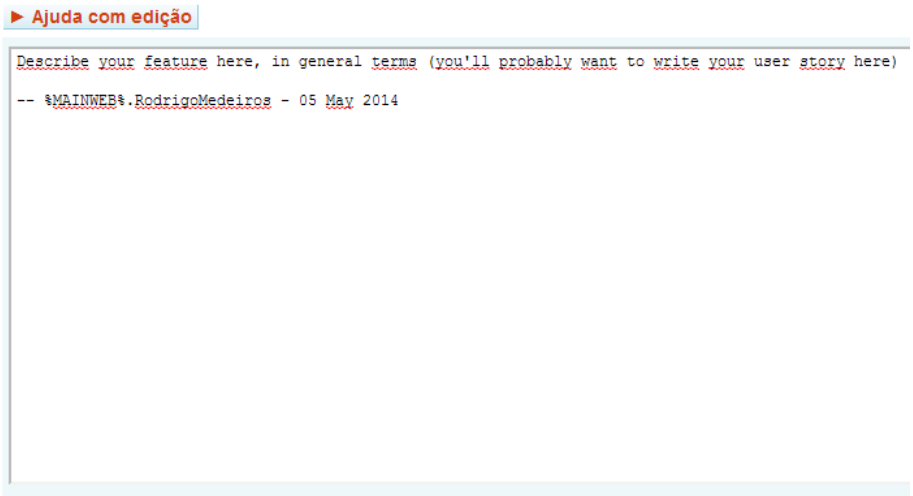

Figura 3 – Descrição de feature - Noosfero

Figura 4 – Descrição de bug - Noosfero

Vá para o passo 4;

Se não:

'Merge request' é rejeitado e as razões por tal são comentadas e respondidas.

Desenvolvedores revisam o que está errado reabre atualiza o merge request;

4. Código é incluido no código principal.

4.1.1.2 Fase de Release 1

A entrega da release consiste na instalacao da nova versao do codigo no repositorio de testes, assim como no código principal quando não houver mais defeitos, ou todos os defeitos encontrados forem tratados devidamente. Porém se algum erro crítico for encontrado e não for tratado a tempo da release final, o lançamento pode ser adiado para a próxima release.

4.1.1.3 Fase de Release 2

A release 2 não é obrigatória, so ocorre se houver muitas mudanças na release 1 e requerer novos testes. Vale lembrar que os procedimentos realizados para aprovar a release 2 são os mesmos da release 1

4.1.1.4 Release Final

A versão final do código é lançada após todos os testes serem aprovados nas releases 1 e 2, assim o código pode ser atualizado para o código principal do Noosfero, encerrando o ciclo de desenvolvimento.

4.2 Testes no processo de colaboração ao Noosfero

Os responsáveis por manter o Noosfero, pessoas que aprovam as solicitações de alteração na plataforma, exigem que os plugins passem por uma bateria de testes para que esses plugins tenham capacidade de ser incorporados à uma versão do Noosfero, prevenindo assim uma possível inserção de código defeituoso que possa afetar o núcleo da plataforma até outros plugins, além de manter a qualidade de código no padrão desejado. Alguns testes automatizados são realizados no noosfero para validação de novos recursos de software, dentre eles estão: testes funcionais, testes de aceitação e testes unitários, testes estes que são executados na própria plataforma do noosfero.

4.2.1 Testes de aceitação - Cucumber

Os testes de aceitação, que possuem uma visão mais voltada para o usuário, fazem parte de uma fase do processo de teste em que um teste de caixa-preta é realizado num sistema antes de sua disponibilização. Para isso utilizamos o Cucumber, uma ferramenta que pode executar documentação de funcionalidades escrita em texto puro. Com base nesta especificações, o Cucumber executa testes. O cucumber proporciona uma melhor comunicação entre equipe de desenvolvimento e cliente, por utilizar uma linguagem em texto puro. No cucumber, uma feature (novo recurso de software) é um requisito de alto nível expressado da pespectiva de uma pessoa e possui uma estrutura similar as historias de usuario do XP (CHELIMSKKY et al., 2010). Essa estrutura é proposta da seguinte forma:

1. **Titulo:** Palavra-chave 'feature' e um titulo curto que representa o objetivo da feature.

2. **Narrativa:** um texto curto que demonstre os cenários de execução, exatamente como a narrativa de uma história de usuário:

3. **Cenarios:** são a parte concreta de como queremos que o software se comporte (CHELIMSKKY e 2010), e sendo a parte essencial do teste realizado no cucumber. Após a palavra-chave 'scenario' define-se o nome do cenário em questão:

4. **Passos:** Cada cenario possui uma serie de passos que demontram o seu comportamento, que são linhas simples iniciadas com as seguintes palavras-chaves: Given, When, Then, And, But.

5. **Given:** Indica uma condição inicial para que o cenario seja executado, trata-se das pré-condições do cenário.

6. **When:**Indica o evento do cenário

7. **Then:** Indica o que é esperado após o evento ocorrer.

4.2.2 Testes Funcionais

Os testes funcionais são escritos da seguinte forma: Setup: indica as condições inciais dos testes, setando variáveis de ambiente e de configuração por exemplo.

Titulo: titulo do teste iniciado com a palavra 'should' e finalizado com 'do' Passos: código que define o comportamento do teste Verificação: Assertiva que verifica se a ação foi realizada como esperada.

4.3 Funcionalidades desenvolvidas

O processo de desenvolvimento de software, assim como o desenvolvimento de testes deste trabalho de conclusão de curso teve seu enfoque na rede colaborativa baseada no noosfero desenvolvida para a Universidade de Brasília (UnB), trata-se na rede chamada de Comunidade UnB, que se encontra em produção, sendo utilizada por aproximadamente 160 usuários.

5 Considerações finais

5.1 Resultados

5.2 Propostas futuras

Referências

AVRITZE, A.; WEYUKER, E. J. Generating test suites for software load testing in international symposium on software testing and analysis (issta). 1994. Disponível em: <http://dl.acm.org/citation.cfm?id=186258.186507>. Citado na página 18.

BECK, K. *Test-Driven Development by Example*. [S.l.]: Addison-Wesley Prefessional, 2002. Citado 3 vezes nas páginas 9, 19 e 20.

BERNARDO, P. C. *Padrões de testes automatizados*. Dissertação (Mestrado) — Instituto de Matemática e Estatística – Universidade de São Paulo, 2011. Disponível em: <http://www.teses.usp.br/teses/disponiveis/45/45134/tde-02042012-120707/>. Citado na página 20.

CHAUí, M. *Convite a filosofia*. [S.l.: s.n.], 2003. Citado na página 23.

CHELIMSKKY, D. et al. *The RSpeec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. [S.l.: s.n.], 2010. Citado 3 vezes nas páginas 20, 30 e 31.

COHEN, D.; LINDVALL, M.; COSTA, P. *An introduction to agile methods. In Advances in Computers*. [S.l.: s.n.], 2004. Citado na página 25.

EVERETT, G. D. et al. *Software Testing*. [S.l.: s.n.], 2007. Citado na página 17.

HARING, R. Behavior driven development: Beter dan test driven development. Java Magazine, 2011. Citado na página 20.

KOSKELA, L. *Test Driven: Pratical TDD and Acceptance TDD for Java Developers*. [S.l.]: Manning Publications, 2007. Citado na página 19.

LIU, H. H. *Software Performance and Scalability: A Quantitative Approach (Quantitative Software Engineering Series)*. [S.l.: s.n.], 2009. Citado na página 18.

MARTIN, R. C. The test bus imperative: Architectures that support automated acceptance testing. 2005. Disponível em: <http://www.martinfowler.com/ieeeSoftware/testBus.pdf>. Citado na página 18.

MASSOL, V.; HUSTED, T. *JUnit in Action*. [S.l.]: Manning Publications, 2003. Citado na página 20.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese (Doutorado) — Instituto de Matemática e Estátistica – Universidade de São Paulo (IME/USP), 2013. Citado na página 27.

MESZAROS, G.; WESLEY, A. *XUnit Test Patterns: Refactoring Test Code*. [S.l.: s.n.], 2007. Citado na página 17.

MUGRIDGE, R.; DCUNNINGHAM, W. *Fit for Developing Software: Framework for Integrated Tests*. [S.l.: s.n.], 2005. Citado na página 18.

NERUR, S.; MAHAPATRA, R.; MANGALARAJ, G. *Challenges of migrating to agile methodologies*. [S.l.: s.n.], 2005. Citado na página 24.

POTEL, M.; COTTER, S. *Inside Taligent Technology*. [S.l.]: Taligent Press, 1995. Citado na página 17.

SATO, D. T. *Uso eficaz de métricas em métodos Ágeis de desenvolvimento de software*. 2007. Citado na página 25.

SCHWABER, K. *Agile Project Management with Scrum*. [S.l.]: Microssoft Press, 2004. Citado na página 23.

STALLMAN, R. M. *Free Software: Freedom and Cooperation*. 2001. Disponível em: <http://www.gnu.org/events/rms-nyu-2001-transcript.txt>. Citado na página 23.

WHITTAKER, M. A. J. A. *How to break Web software: functional and security testing of Web applications and Web services*. [S.l.: s.n.], 2006. Citado na página 18.