DESENVOLVIMENTO ORIENTADO A TESTES EM SISTEMAS WEB

Chrystian Raony Perazzoli
Lilian Jeannette Meyer Riveros
Paulo Roberto Perazzolli

Resumo

Testes utilizados de forma eficaz garantem a qualidade de aplicações sejam desktop ou webs. Para apoiar o uso de testes em sistemas webs, podem-se aplicar diversas técnicas que podem auxiliar no desenvolvimento. TDD é uma técnica muito eficaz, mas pouco utilizada, muitas vezes pela falta de conhecimento e ciência das tecnologias existentes. Assim este artigo tem o objetivo de comparar a forma tradicional de testes aplicados com o desenvolvimento orientado a testes, ainda mostrando exemplo e os benefícios alcançados com sua utilização.

1 INTRODUÇÃO

Ao estudar Engenharia de Software alguns dos principais objetivos são, a redução de custos ao se criar um software, na qualidade de software e a facilidade de manutenção. Para alcançar esses fatores o software deve ser bem projetado e testado, assim terá uma redução no custo com tempo de manutenção posteriormente aplicado e garantirá a qualidade de software.

Hoje com o crescimento exponencial dos processadores de computadores, tornou-se possível o uso de testes automatizados em softwares de qualquer escala. Deixando de lado a necessidade de testes manuais repetitivos e a necessidade de uma equipe de testes que utilizando o software saem em busca de erros e bugs. Trazendo uma redução de custos no desenvolvimento. Contudo os testes automatizados devem ser aplicados de forma correta para que surja o efeito desejado, através do Test-Driven Development (TDD) esses objetivos podem ser alcançados.

2 **DESENVOLVIMENTO**

2.1 Testes em Modelos Prescritivos Tradicionais

Em 1970 Winston Royce publicou sobre o modelo em cascata [2] e o modelo original descrevia a seguinte sequência de passos para a construção de um software: Licitação de requisitos, Projeto, Construção (implementação ou codificação), Integração, Teste e depuração, Instalação e Manutenção de software [3].

Atualmente o modelo em cascata encontra-se com os seguintes passos (Figura 1):

Ao analisar os passos do modelo original com o atual se percebe que os testes são feitos posteriormente à codificação do software. O mesmo podemos perceber no Modelo Espiral que foi sugerido como um substituto ao Modelo em Cascata.

O maior problema ao fazer a codificação anteriormente a criação dos testes, é que apenas são criados os testes para satisfazer o resultado de um método ou função de um software.

Enquanto deve-se fazer com que o código satisfaça o requisito de um teste e somente com o código necessário para tal, deixando um código limpo, sendo este um dos princípios do TDD.

2.2 TDD, XP, Aliança Ágil

A primeira vez que foi falado sobre o conceito "Test, then code" (Teste, então codifique) foi em 1987[4] em uma Lightning Talk (Palestra Rápida) e após um período grande de tempo, em 1996 Kent Beck, Ron Jeffries and Howard G. Cunningham[2]. Iniciaram a primeira versão do projeto XP (Extreme Programming ou Programação Extrema), onde descrevem processos e metodologias para a criação de software de forma rápida e suscetível à alterações de requisitos em qualquer etapa do projeto, focando também na qualidade de código gerado. Posteriormente em 2001 Kent Beck juntamente com mais 16 outros desenvolvedores, produtores e consultores

de software conhecido como aliança ágil, assinaram o Manifesto para o Desenvolvimento Ágil de Software[5].

Dois anos depois Beck publicou o livro chamado "Test Driven Development"[2] (Desenvolvimento Orientado à Testes), onde "ressuscitou" o conceito "Test, then code". Que posteriormente, foi agregado à metodologia ágil XP.

2.2.1 TDD

Chaplin em 2003, [6] escreveu um artigo discutindo sobre o TDD criado por Beck e em sua introdução, ele elencou alguns fatores que poderiam afetar positivamente na criação de um software quando utilizado TDD, sendo estes:

- Sem mais bugs repetidos.
- Todo teste de regressão de seu sistema demora menos que um minuto.
- Você gasta maior parte do seu tempo desenvolvendo novas funcionalidades, invés de ficar debugando (verificando) código existente.
- Tempo de debug é reduzido pelo menos em um fator de 10.
- Testes são somente testados manualmente uma vez.

Ainda em seu artigo ele fala como devem ser feitas as etapas para a prática do TDD de forma correta, sendo estas abaixo:

- Escolha um caso de teste.
- Escreva o teste. O teste não irá compilar, já que o código ainda não está escrito.
- Escreva o esboço do código. O teste agora irá compilar, mas você irá ter uma luz vermelha por que ele deve falhar.
- Escreva o código até passar o teste.

Essas etapas foram descritas para o exemplo que ele utilizou utilizando JUnit em Java, para aplicações web que não há necessidade de

compilação de código, como PHP, Ruby, Python. As etapas ficam melhor descritas da seguinte maneira:

- Escolha um caso de teste.
- Escreva o teste. O teste deverá falhar, já que o código ainda não está escrito.
- Escreva o código até passar o teste.
- Refatore ou refaça o código. (Melhoria de código)

Dessa forma, o código terá uma qualidade maior e não ocorrerá os mesmos problemas futuramente, já que todo código é feito em cima de uma base sólida de testes. Ou seja, todo código somente existirá se um teste se estiver cobrindo seu esboço.

2.2.1.1 Testes em Aplicações Web

Pressman (2010) fala que, antes de iniciar a codificação de qualquer aplicação seguindo a metodologia TDD, é necessário que se faça a criação de testes unitários criando um arcabouço para que os testes sejam automatizados e consequentemente os testes possam ser executados de forma fácil e rapidamente. Encorajando uma estratégia de regressão, ou seja, quando o código é modificado, pode-se rodar os testes novamente para garantir que os testes continuem passando e quando falharem significa que suas alterações afetaram outras partes da aplicação. Garantindo a qualidade de software.

Porém quando estamos pensando em TDD para aplicações Web, testes unitários são uma pequena parte que precisa ser testado. Pressman (2010), ainda refere-se a vários testes como teste de conteúdo, teste de interface visual, teste de navegação, teste de componente, teste de configuração, teste de segurança e teste de desempenho, porém precisasse elencar os testes que podem e devem ser aplicados em uma aplicação web utilizando TDD.

Os Testes de Conteúdo e Interface Visual, servem para verificar a parte estética da aplicação juntamente com a navegação criada para o usuário, como menus e links, porém essa parte após desenvolvida e passada pelo teste de aceitação do usuário, dificilmente irá ser alterada, dessa forma esses não são testes candidatos a serem feitos no conceito "Test, then code".

Os testes de Navegação também são testes de aceitação feitos para validar a navegação na aplicação de uma forma correta, porém estes testes hoje podem ser feitos automatizados, utilizando Drivers que emulam um navegador e preenchem os campos simulando a ação de um usuário real na aplicação.

Já nos Testes de Componente, Pressman (2010) cita que uma página web por encapsular conteúdo, ligações de navegação e elementos de processamento como formulários e scripts, pode ser considerada um componente funcional, dessa forma testes funcionais são necessários e quando uma aplicação web têm diferentes fluxos de dados em diferentes páginas web, testes de integração se faz necessário, além dos testes unitários para verificar a integridade das informações trafegadas pelo sistema.

Ainda os Testes de Segurança também podem ser adicionados aos testes de integração, já que se houver uma tentativa de violação o sistema precisa se comportar corretamente.

E os últimos dois testes de desempenho e configuração, são processos que não demandam testes automatizados, seguindo o conceito TDD.

Dessa forma os testes necessários a serem aplicados para que uma aplicação web possa ser desenvolvida orientada a testes e que contemple os testes que Pressman descreve como essenciais, são: Testes Unitários, Testes Funcionais e Testes de Integração.

2.2.1.2 A prática do TDD no desenvolvimento de Sistemas Web

O processo de criação de software tradicionais seguindo o modelo cascata, espiral ou incremental é necessário fazer todo o levantamento de

requisitos, casos de uso e as especificações do caso de uso, porém enquanto um ciclo de desenvolvimento não é finalizado mudanças na definição do software não são bem vindas. E como TDD é uma das atividades da metodologia ágil XP, mudanças são aceitadas em qualquer etapa do projeto, já que o software está todo testado os efeitos colaterais são facilmente encontrados e resolvidos.

Uma das falhas de um software tradicional é que todos os testes são passados dos analistas/engenheiros para os programadores, mas somente são aplicados quando a funcionalidade já está pronta, como uma forma de validar o código. Dessa forma o código feito pode não ter sido feito da melhor maneira e quando atendido ao requisito e o teste ter passado é considerado suficiente.

Porém quando construído no conceito TDD, a primeira etapa antes de começar codificar a funcionalidade é construir o teste e fazê-lo falhar, a segunda etapa é construir a funcionalidade para satisfazer o teste, após o teste passar, retorna-se ao código para refatoração para melhorar a qualidade do código. Normalmente um teste é o atendimento de um requisito da aplicação (caso de uso).

2.2.1.3 Exemplo do TDD utilizando Ruby on Rails

Para mostrar os diferentes testes que precisam ser abordados em uma aplicação web, será utilizado o framework MVC chamado Ruby on Rails(RoR). Que foi desenvolvido utilizando a linguagem de programação ruby. A aplicação exemplo será um CRUD (Create, Read, Update, Delete), de uma entidade "Aluno" que terá dois atributos: nome e idade.

Nas pastas em que estão expandidas tem respectivamente, onde os códigos da aplicação (app) ficam armazenados, sendo eles separados por diretórios que definem o padrão MVC utilizado no framework e na pasta test, temos os diretórios com os testes padrões utilizados pelo Ruby on Rails.

Na documentação de testes do RoR[8], a prática dos testes unitários é criando os testes que ficam dentro da pasta de fixtures e models.

- Fixtures: "É uma palavra bonita para dados de amostra"[8], que servem para popular o banco de dados antes dos testes serem executados, ou seja, é possível simular um banco de dados populado antes de se executar a suíte de testes.
- Model: É onde os dados são tratados, validados e inserido no banco de dados.

Dessa forma os testes unitários se encaixam especificamente no model da aplicação. Tornando os dados inseridos no banco de dados íntegros e coesos.

Ainda os testes funcionais, são considerados os testes de controllers, mailers e helpers, pois:

- Helpers: são responsáveis por fornecer métodos e auxíliar os controllers, como uma forma de quebrar uma tarefa complexa em métodos menores.
- Mailers: responsáveis pelo envio dos e-mails da aplicação.
- Controllers: responsável por tratar as requisições e retornadas para o usuário com informações do banco de dados ou somente com as views (HTML).

Para os testes de integração há um diretório específico:

- Integration: que é utilizado para tratar dos testes de fluxo que podem haver na aplicação, no nosso caso por exemplo, ao iniciar o cadastro de um aluno precisamos garantir que o primeiro passo é mostrar a página para gravar o nome do aluno, o segundo passo é a página para inserção da idade do aluno e posteriormente finalizar o cadastro, persistir os dados no banco de dados e retornar uma mensagem de feedback ao usuário.

Conhecendo melhor a estrutura e a suíte de testes do RoR, precisasse configurar o ambiente para que haja conexão com o banco de dados e alguns passos iniciais, estes podem ser seguidos da documentação do RoR[9].

Sabendo que é necessário fazer um sistema que faça as operações para manter um aluno, é preciso criar os testes unitários, ou seja, a fixture e o teste do model, para o aluno.

A fixture utiliza o tipo de arquivo YAML que é utilizado para definir uma estrutura de dados, parecido com o formato XML, mas de uma forma mais limpa. Dessa forma o RoR utilizará essa fixture para inserir pelo menos 2 registros no banco de dados: "joao" e "maria".

Dessa forma criamos o esboço para os testes do model aluno, porém ainda não há testes sendo verificados, podemos notar na figura 2, onde é executado a suíte de testes e temos algumas informações iniciais.

No resultado da execução dos testes, abaixo da linha "# Running tests:" mostrará todos os testes, simbolizando-os da seguinte maneira:

- .(ponto) - Para testes que passarem
- E - Para testes que houverem erros(errors), isso significa códigos com erros
- F - Para testes que falharam(failures), ou seja, não gerou erro mas não satisfez o teste.
- S - Para quando um teste é ignorado(skips), como uma forma de não precisar excluir um teste, apenas utilizar uma flag para resolver posteriormente e não atrapalhar o restante da suíte de testes.

Na última linha ainda é exibido o total dos testes, juntamente com a quantia de verificações(assertions), os testes falhos, com erros e ignorados.

Para garantir a consistência das informações no banco de dados, vamos garantir que somente um aluno seja inserido se o nome e idade estiverem preenchidos. Dessa forma iremos fazer dois testes, o primeiro para verificar se o sistema não está deixando o inserir tuplas vazias no banco de dados. Apenas para que um teste unitário de um model funcione corretamente precisasse do model correspondente, que será criado dentro da pasta app/models.

Agora pode iniciar a criação dos testes antes mesmo de criar códigos para o model aluno. (Os nomes dos testes estão em português para melhor exibição). Ficando da seguinte maneira:

Na Figura 3, linha 9. A validação ocorre da seguinte forma, quando executado o método save o model chama alguns callbacks de validação se houver, caso contrário insere no banco de dados, como não foi informado o nome, nem idade para a instância aluno e ainda não há validação, o model insere no banco de dados e como a o sinal de negação(!) do retorno, fazendo com que o teste falhe, gerando o seguinte resultado ao executar a suíte de testes:

Dessa forma cumpre-se o primeiro passo do TDD, criar um teste e fazê-lo falhar, o segundo passo é fazer as correções criando o código necessário para fazer funcionar o código corretamente, ficando da seguinte maneira o model Aluno:

Os callbacks validates são utilizados para validar a presença dos valores nos atributos do objeto, antes de serem encaminhados para o ActiveRecord, responsável pela criação das consultas SQLs e execução das mesmas no banco de dados. Dessa forma agora nosso teste irá passar, pois quando houver a validação não deixará inserir um aluno sem os dados definidos.

Após o teste ter passado, a metodologia TDD sugere que retorne-se ao código para executar melhorias de códigos com o objetivo de aumentar a qualidade do código escrito, como o código desse exemplo é básico e com poucas funções internas não há essa necessidade.

Com os testes unitários feitos até agora garante que o modelo somente aceitará inserção de tuplas no banco de dados que houverem dados preenchidos, gerando um banco de dados consistente.

Após concluído os testes unitários da entidade, precisasse fazer o mesmo para os testes funcionais relacionados com o controller da entidade aluno.

Como o RoR utiliza o padrão RESTFUL para as URLs não há somente os métodos HTTP (HyperText Transfer Protocol) GET e POST como as aplicações tradicionais, além destes há, os métodos: PUT/PATCH e DELETE. Para tratar de forma diferenciada os tipos de requisições feitas pelo usuário, exemplo:

- Usuário envia formulário para cadastro (POST)

- Usuário envia solicitação para deletar (DELETE)
- Usuário envia formulário para atualizar (PUT/PATCH)
- Usuário envia solicitação para visualização (GET)

Assim pode-se criar cada teste individual somente para buscar um funcionário e assim por diante, até termos um resultado que contemple os testes em todos os tipos de requisições em um controller, como no nosso caso.

Assim para cada teste criado no teste funcional, cria-se seu respectivo código para validar e criar a funcionalidade.

Assim se garante todos os passos de um CRUD em uma aplicação web, antes mesmo da aplicação ter sido executada e testada no navegador, pois estamos garantindo que os controllers e models se comportem da maneira que esperamos, através dos testes automatizados.

Os testes de integração nessa aplicação não se fez necessário, já que não houve tráfego de informações em diferentes controllers da aplicação pois há somente um controller.

Após a lógica do controller e model prontas, o próximo passo é a criação da parte visual da aplicação, ou seja, os HTMLs com os formulários, para a aplicação sendo esses a parte visual um teste de aceitação feito pelo cliente e os testes de navegação na aplicação podem ser feitos no RoR com uma biblioteca terceira chamada Capybara, que fornece uma estrutura para testar a utilização da aplicação de forma automática, como o preenchimento de dados em formulários e ações nos botões.

No teste de aceitação utilizando capybara, o teste visita a página de criar um novo aluno, através da funcionalidade de XPATH, uma forma de encontrar elementos dentro de um DOM (Data Object Model), é obtido o formulário e preenchido os campos através de seus atributos name, e no final é validado o teste, se houve realmente a criação do aluno é redirecionado para outra página contendo a mensagem que o aluno foi criado com sucesso.

Dessa forma, se em algum momento algum código da estrutura de um formulário ou botão for alterado, afetará os testes e quando rodado a suíte de testes, mostrará todos os lugares que foram afetados. Assim o tempo de debug é reduzido significativamente e torna um código de alta qualidade.

3 CONCLUSÃO

O tempo gasto na construção de testes e a curva de aprendizado para uma aplicação correta utilizando TDD, é de certa forma um pouco complexa inicialmente, mas os benefícios trazidos com sua utilização é nítida, já que é possível criar todos os testes que uma aplicação web necessita, como Teste de Aceitação, Teste Funcional, Teste de Integração e Teste Unitário. Ao utilizar TDD, o desenvolvedor consegue ter uma noção real do fluxo de dados e os requisitos da aplicação, somente analisando os testes escritos. Ainda todo o código da aplicação somente existirá se houver um teste cobrindo seu esboço, dessa forma não haverá uma aplicação parcialmente com testes e sim toda ela. Garantindo a qualidade do software que é buscada ao aplicar conceitos da Engenharia de Software.

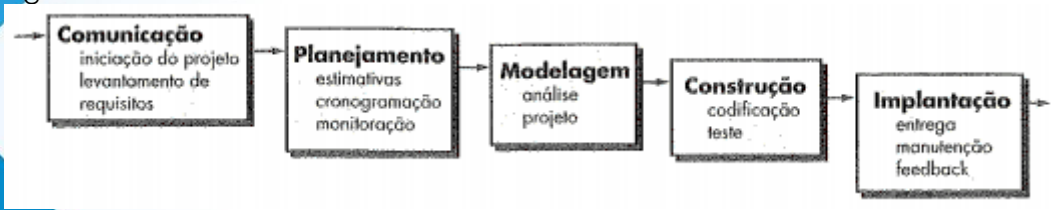
REFERÊNCIAS

http://www.testingreferences.com/testinghistory.php [2]
http://pt.wikipedia.org/wiki/Modelo_em_cascata [3]
http://repository.lib.ncsu.edu/ir/bitstream/1840.16/2431/1/etd.pdf [4] p. 23
Gelperin, D. and W. Hetzel. Software Quality Engineering. in Fourth International Conference on Software Testing, Washington D.C. June 1987 [5]
Pressman, R. (2010). Engenharia de software, 6ª edição. [6]
Chaplin, D. (2003). Test Driven Development - http://www.byte-vision.com/TestDrivenDevelopmentArticle.aspx [7]
http://guides.rubyonrails.org/testing.html [8]
http://guides.rubyonrails.org/getting_started.html [9]
https://github.com/jnicklas/capybara [10]

Sobre o(s) autor(es)
Chrystian Raony Perazzoli, Pós Graduado do curso de Gestão da Tecnologia da Informação, chrystian.raony@gmail.com
Lilian J. Meyer Riveros, Mestre em Ciência Da Computação. Professora titular da Unoesc Campus. lilian.riveros@unoesc.edu.br.

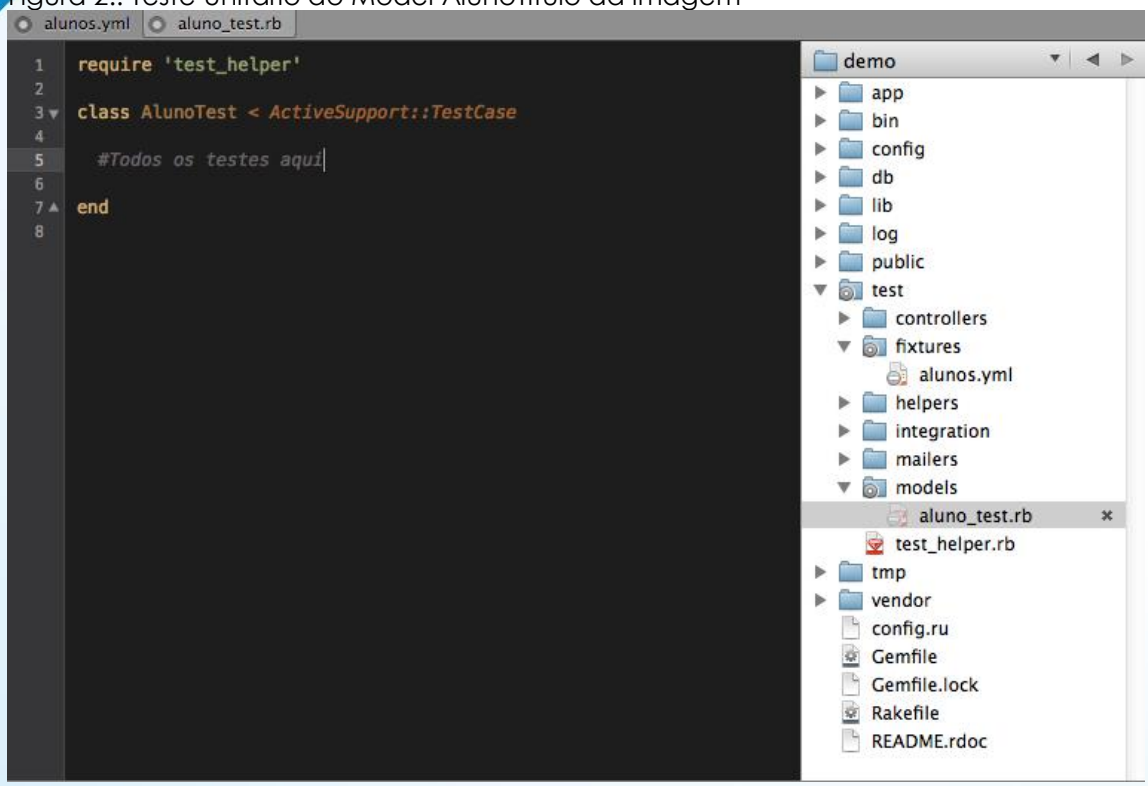
Paulo Roberto Perazzolli, Especialista em Redes e Segurança de Sistemas, Professor titular da Unoesc Videira, perazzolli@gmail.com

Figura 1: Modelo em Cascata



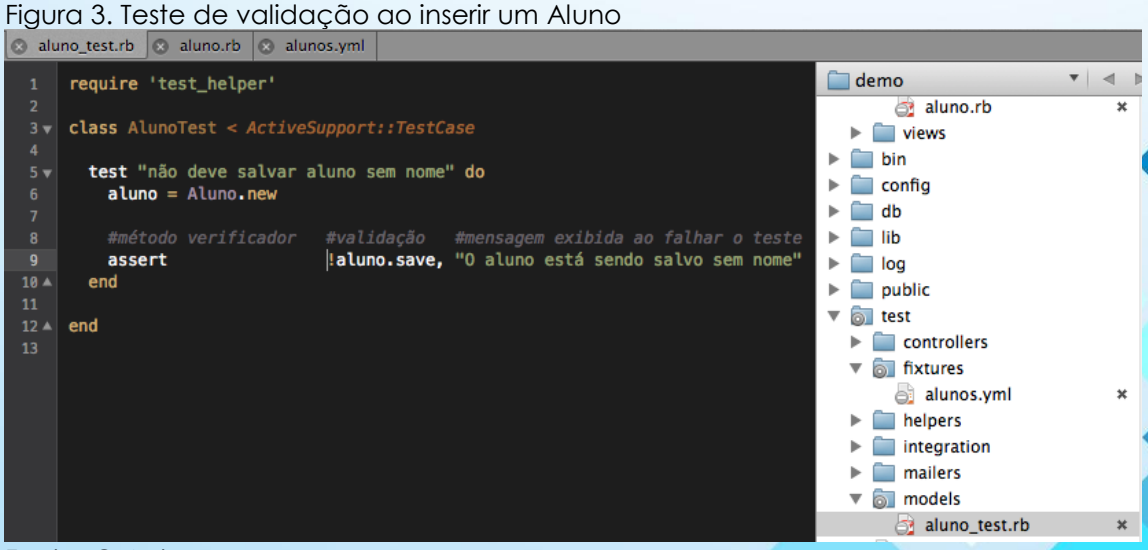
Fonte: Pressman (2010)

Figura 2:. Teste Unitário do Model AlunoTítulo da imagem



Fonte: O Autor

Figura 3. Teste de validação ao inserir um Aluno



Fonte: O Autor

Fonte: Fonte da imagem

Título da imagem

Fonte: Fonte da imagem

Fonte: