AVALIAÇÃO DA FERRAMENTA MINITEST NO DESENVOLVIMENTO GUIADO POR TESTES DO FRAMEWORK RUBY ON RAILS

Bruno Cezar Scopel Sarcinelli
Graduado em Análise e Desenvolvimento de Sistemas

RESUMO. Este artigo descreve o desenvolvimento de um sistema utilizando o modelo de desenvolvimento guiado por testes (TDD - *Test-Driven Development*) com as ferramentas do framework Ruby on Rails. O objetivo é demonstrar de maneira prática a aplicação dos conceitos de TDD nesse framework através da implementação de testes de modelo, funcionais e de visão. Este artigo pretende ser uma fonte esclarecida de consulta para iniciantes e profissionais já experientes que pretendem desenvolver utilizando o TDD no Ruby on Rails.

Palavra-chave: Ruby on Rails. TDD. Desenvolvimento ágil guiado por testes. Testes de Software. MVC.

INTRODUÇÃO

A atividade de teste de software tem como objetivo encontrar defeitos inseridos no decorrer do processo de desenvolvimento, constituindo um elemento crítico da garantia de qualidade de software, pois representa a revisão da especificação, projeto e geração de código (SOUZA e GASPAROTTO, 2013; PRESSMAN, 2002).

O teste é uma atividade realizada para avaliação da qualidade do produto, efetuando sua melhoria através da identificação de defeitos e problemas (SWEBOK, 2004).

O desenvolvimento guiado por testes (TDD - Test Driven Development) é uma técnica de desenvolvimento de software baseada em ciclos curtos de repetições, onde primeiramente o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade, produz um código que possa ser validado pelo teste e, logo após, o refatora para um código sob padrões aceitáveis (BECK,2010). O princípio básico do TDD é incluir a atividade de teste de software no decorrer do processo de desenvolvimento, fornecendo feedback constante sobre o código que está sendo produzido.

O Ruby on Rails (RoR) é um framework de desenvolvimento ágil de software web criado em 2003 que permite o desenvolvimento de software na linguagem Ruby, utilizando a arquitetura MVC (BETTER EXPLAINED, 2017). O RoR foi adotado como plataforma de desenvolvimento por aplicações como Twitter (TECHTUDO, 2017), GitHub (GITHUB, 2017) e Basecamp (BASECAMP, 2017) e, por possuir diversas características que auxiliam e facilitam o desenvolvimento rápido de software, também é adotado por aplicações de pequeno e médio porte.

Segundo Ruby-doc (2017), o RoR foi projetado para dar suporte nativo ao TDD. Apesar disso, a maioria dos materiais que abordam TDD no RoR enfatizam a parte técnica do uso das ferramentas, mas não fornecem um embasamento sólido de como utilizar o TDD de maneira sistemática com as ferramentas do framework, dificultando o aprendizado correto da técnica nesse ambiente.

Os objetivos deste artigo são desenvolver um sistema baseado em demandas reais utilizando os conceitos de TDD em um ambiente RoR e, de maneira gradativa; avaliar as ferramentas de TDD disponíveis nesse ambiente, expondo suas vantagens e desvantagens, com o objetivo de fornecer um embasamento teórico e prático para desenvolvedores que tenham interesse de aplicar TDD nesse ambiente.

Este artigo está organizado da seguin-

te maneira: a Seção 1 apresenta os conceitos de desenvolvimento guiado por testes. A Seção 2 define o framework RoR e seus conceitos básicos. A Seção 3 apresenta a implementação e os resultados deste trabalho, culminando na conclusão.

1 DESENVOLVIMENTO GUIADO POR TESTES

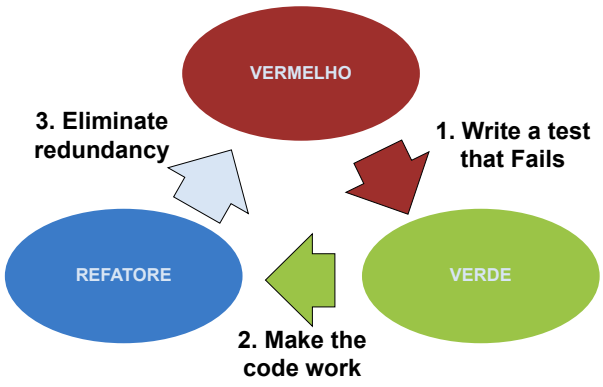
O desenvolvimento guiado por testes é uma metodologia que consiste em pequenos ciclos de desenvolvimento baseados em testes. O ciclo geral do TDD está apresentado na Figura 1 e pode ser descrito da seguinte maneira:

1) pense o que um determinado código do seu software deve implementar, descreva o contexto e defina quais as verificações por realizar. Escreva um teste para verificar a corretude desse código. Como, inicialmente, esse código não está implementado, o teste acusará que o código não foi implementado corretamente;

2) escreva o mínimo de código possível para que o teste criado, anteriormente, passe. Nesse momento, o importante é criar um código que seja aprovado pelo teste, mesmo que ele não esteja na sua forma mais completa;

3) refatore. Uma vez que o teste aprovou o código da funcionalidade, verifique o que pode ser melhorado, nesse código, sem que o teste deixe de aprová-lo.

FIGURA 1 - Ciclo Básico do TDD (BECK, 2010).



Com o TDD é possível refletir sobre a modelagem antes de escrever o código funcional, fazendo com que o sistema seja desenvolvido através de pequenos passos, até chegar a sua totalidade (SANCHEZ, 2006). Além disso, o TDD visa um código limpo, confiável, que atenda aos requisitos de forma satisfatória e cujos testes facilitem a manutenção (BAUMEISTER e WIRSING, 2017).

Alguns motivos para a adoção do TDD são: a leitura dos testes auxilia no entendimento do sistema; código desnecessário não é desenvolvido, pois só são implementados os códigos suficientes para os testes funcionarem e, consequentemente, o sistema funcionar; não existe código sem teste; os testes permitem uma refatoração segura do código, pois garantem que as mudanças não alteram o funcionamento do sistema (KAUFMANN e JANZEN, 2003).

Causevic et al. (2012) mostraram, através de experiências práticas, as vantagens que o TDD trouxe na qualidade do código a longo prazo, em conjunto com práticas de desenvolvimentos ágeis. Através de um comparativo entre o desenvolvimento convencional e o TDD, chegaram a conclusão que o TDD revela muito mais requisitos não especificados essenciais que o método convencional.

Gupta et al. (2007) demonstraram a eficácia do TDD no desenvolvimento de software de grande porte, realizando também um comparativo com o desenvolvimento sem testes. Para avaliar o processo de desenvolvimento, foram utilizadas algumas métricas como a produtividade do desenvolvedor, a qualidade do código desenvolvido e esforços gerais no desenvolvimento (manutenção, concepção, desenvolvimento e testes). Os autores concluiram que o TDD ajuda a minimizar os esforços no desenvolvimento, melhora a produtividade e a qualidade do código dos softwares, além de permitir um entendimento mais claro dos requisitos.

Como toda metodologia de desenvolvimento, há dificuldades na sua implantação.

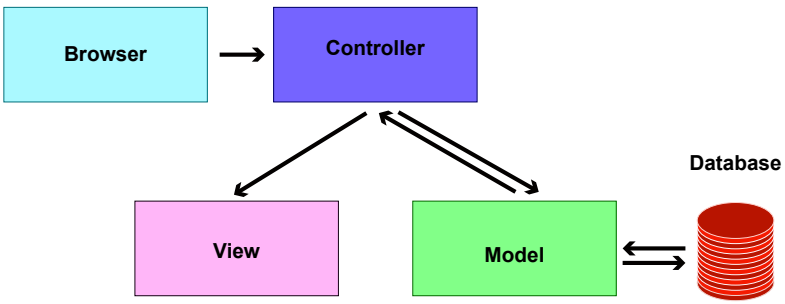
No TDD, a principal delas é a mudança cultural pela qual a equipe de desenvolvimento e a empresa precisam passar, pois o tempo gasto para a definição e o desenvolvimento dos testes precisa ser encarado pela empresa como investimento, e os desenvolvedores têm de mudar sua mentalidade na hora do desenvolvimento, dando a devida importância aos testes (ANDRADE, 2011).

2 RUBY ON RAILS (ROR)

O framework Ruby on Rails (RoR) foi desenvolvido pensando na praticidade e facilidade no desenvolvimento de aplicações Web. O framework surgiu em 2004, foi criado por David Hanson e utiliza a linguagem Ruby, que é uma linguagem orientada a objetos, interpretada, de tipagem forte e dinâmica (SOFTWARE LIVRE BRASIL, 2017).

O RoR segue o padrão de projeto MVC (*Model-View-Controller*), ilustrado na Figura 2. O MVC divide o código da aplicação em três camadas: a camada de modelos, responsável pela comunicação entre a aplicação e a base de dados; a camada de controladores, responsável por atender as requisições e preparar os dados que serão exibidos para o usuário e a camada de visões, que recebe os dados preparados pela camada de controladores e realiza a interação com o usuário (BETTER EXPLAINED, 2017).

FIGURA 2 - Padrão de projeto MVC (THOMAS, 2008)



É possível aplicar testes que cobrem todas as áreas de uma aplicação RoR, desde a entrada de dados, requisições, respostas das controladoras, visões e fluxo da aplicação (GUNDERLOY, 2017). O RoR facilita a escrita de testes pois, a medida que a aplicação é desenvolvida, arquivos de teste básicos são gerados, cabendo ao desenvolvedor estendê-los para atender às demandas da aplicação (SEA, 2009).

Segundo Thomas (2008), os possíveis testes em uma aplicação Rails são:

- teste unitário: executados continuamente durante o ciclo de desenvolvimento com o objetivo de avaliar separadamente pequenos trechos de código (unidades) de um sistema, procurando por erros de lógica e de implementação e verificando se o comportamento de classes e funções é o esperado (BARBOSA et al, 2000). Permite detectar falhas de lógica, comportamentais e de digitação, além de auxiliar na refatoração de código;
- teste funcional: avalia as requisições e respostas das controladoras. Por exemplo, avalia os métodos de um controlador com o objetivo de analisar se as requisições são bem sucedidas, se o usuário está sendo redirecionado para a página correta, se os objetos necessários para a renderização das visões foram criados corretamente, dentre outras ações. Também permite validar as respostas fornecidas pelas visões (RAILS GUIDE, 2017).
- teste de integração: analisa o fluxo da aplicação, avaliando a interação entre modelos e controladores (RAILS GUIDES TESTING, 2017).

Rappin et al. (2011) abordam o desenvolvimento guiado por testes utilizando RoR, fazendo uma correlação entre o desenvolvimento convencional e o desenvolvimento usando TDD. Entretanto, o livro não expõe as facilidades e dificuldades da técnica no RoR, focando apenas em exemplos técnicos do uso das ferramentas.

Corbucci e Aniche (2014) abordam o

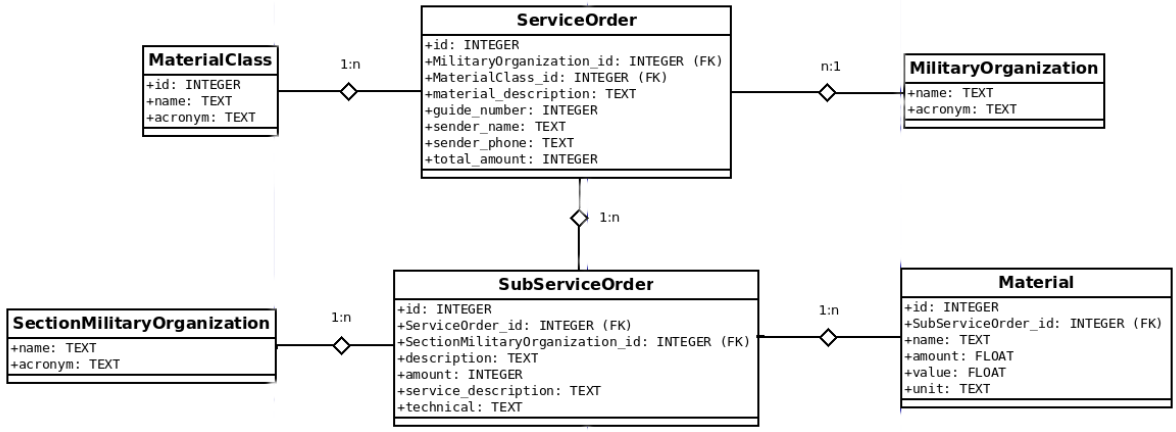
TDD através de exemplos, desde os testes mais simples até testes mais complexos. Entretanto, o livro recai no mesmo problema de outros materiais: não expõe as experiências, dificuldades e facilidades da aplicação desse método de desenvolvimento no RoR.

3 IMPLEMENTAÇÃO E RESULTADOS

Para demonstrar os conceitos de TDD na prática e avaliar as gems de TDD do RoR (AKITA ON RAILS, 2017), foi implementado um sistema de ordens de serviço para atender as necessidades de uma determinada empresa (fictícia), cujo processo de abertura e controle das ordens de serviço ainda são realizados manualmente.

O diagrama entidade-relacionamento mostrado na Figura 3 foi utilizado como ponto de partida para os primeiros testes do sistema.

FIGURA 3 - Diagrama entidade-relacionamento



O RoR permite o reaproveitamento de componentes utilizando gems (AKITA ON RAILS, 2017), que são pacotes com código Ruby gerenciados por uma aplicação do sistema operacional, chamada rubygems (AKITA ON RAILS, 2017). Utilizando gems é possível reutilizar componentes prontos que auxiliam em várias partes de uma aplicação, como por exemplo na autenticação de usuários, na renderização de imagens, e também no desenvolvimento de testes (RAILSGUIDE,2017).

Existem muitas ferramentas de testes para RoR, para comparar e exemplificar a diferença da ferramenta Minitest com outra, escolheu-se a ferramenta RSPEC. A Minitest é a biblioteca nativa do RoR para a escrita de testes de modelo, de controlador e de integração. Com a Minitest é possível testar código Ruby, utilizando uma grande variedade de asserções (RAILSGUIDE,2017).

Segundo Relish (2017) a RSpec é uma ferramenta de teste baseada na metodologia BDD (*Behaviour-Driven Development*), que permite especificar o comportamento desejado do código Ruby utilizando uma DSL (*Domain-Specific Language*) simples e expressiva, facilitando a leitura e o entendimento do código de teste. A RSPec também pode ser usada para especificar testes de modelo, controlador e integração.

O ponto de partida no desenvolvimento da nossa aplicação foi o mapeamento do diagrama entidade-relacionamento na camada de modelos do RoR. Para isso, os primeiros testes de modelo foram elaborados utilizando as gems Minitest, segundo o RAILSGUIDE (2017) e Rspec (RELISH, 2017).

3.1 TESTES DE MODELO

Com o objetivo de entender as diferenças entre a Minitest e a RSPec, os primeiros testes de modelo foram elaborados, utilizando as duas gems, e criados para atender o seguinte requisito da entidade ServiceOrder, que modela uma ordem de serviço no sistema: uma ordem de serviço não pode ter uma descrição de material vazia.

Seguindo a abordagem TDD, o primeiro passo é criar um teste que analise se é possível salvar uma ordem de serviço com um nome vazio. Caso seja possível, o teste deve acusar um erro. O código a seguir mostra o código desse teste escrito utilizando Minitest. Por questões de simplicidade, a entidade Service-Order passará a ser chamada de So a partir de agora.

```
1. Class Sotest < ActiveSupport:: TestCase
2.  fixtures :sos
3. test "service order should have material description"
4.   service_order = sos(:material_description_nil
5.   assert !service_order.save,
6.   "service order saved without material description"
7.  end
8. end
```

A linha 1 declara a classe de teste do modelo So, que sempre deve estender da classe base de teste de modelos do Minitest (ActiveSupport::TestCase). Por convenção, todo nome de classe de teste escrita com Minitest deve terminar com a palavra Test, e todos os métodos que realizam testes devem começar com a palavra-chave test. Essa é a convenção adotada pelo RoR para que, quando os testes forem executados, ele possa identificar as classes e métodos que realizam testes e invocá-los.

Na maioria dos testes, há necessidade de existir uma base de testes preenchida com informações de interesse. Antes da execução de qualquer teste, o RoR carrega uma base de testes e a deixa disponível para qualquer teste em execução acessá-la. A linha 2 informa que essa classe de teste utilizará as *fixtures* de uma So. *Fixtures* são conjuntos de dados pré-definidos em um arquivo de texto YML [28], que são automaticamente carregados para a base de dados de teste antes de qualquer teste ser executado. Com *fixtures* é possível acessar linhas específicas de uma tabela utilizando rótulos, tornando-a uma ferramenta bastante atrativa para os primeiros testes de modelo.

A linha 3 declara o método de teste. A linha 4 acessa a *fixture* material_description_nil, recuperando da tabela So uma linha com o campo material_description nulo, e armazenando uma referência para essa linha na variável service_order. As linhas 5 e 6 realizam uma asserção, que é um comando que avalia um objeto de acordo com um resultado esperado. Nesse caso, espera-se que não seja possível salvar a referência service order, uma vez que ela não possui uma descrição. O método save pode ser invocado a partir da referência service_order, pois o RoR segue o padrão de projeto Active-Record (RAILS GUIDES, 2017).

O primeiro erro acusado pelo teste foi na linha 2, pois o arquivo YML com as *fixtures* não foi criado. Após a criação do arquivo de *fixtures*, o próximo erro acusado foi a ausência de uma tabela So na base de testes, pois a primeira ação do RoR ao detectar um arquivo de *fixtures* é carregá-lo na base de testes. Para solucionar esse erro, dois arquivos foram criados: um arquivo de modelo So vazio, que representará uma ordem de serviço na camada de modelos do sistema, e um arquivo de migração, responsável por criar a tabela So no banco de dados.

O próximo erro acusado foi a ausência de uma *fixture* chamada material_description_nil, uma vez que o arquivo de *fixtures* foi criado mas não foi preenchido. Após a inserção da linha material_description_nil no arquivo de *fixtures*, o próximo erro acusado foi a ausência da coluna material_description na tabela de ordens de serviço. Para corrigir esse erro, um novo arquivo de migração foi criado.

Após as alterações mencionadas anteriormente, o teste parou de informar a presença de erros e indicou uma falha: as linhas 5 e 6 acusaram que foi possível salvar uma ordem de serviço sem o campo material_description. Para corrigir essa falha, foi inserida uma validação de presença do campo material_description no arquivo de modelo So. Após essa validação, o teste foi executado sem erros e falhas.

Os pequenos ciclos de desenvolvimento utilizados para atender o requisito apresentado, também chamados de baby steps,

auxiliam no desenvolvimento gradativo de uma aplicação, levando à criação apenas de arquivos, métodos e atributos essenciais para atendê-lo, contribuindo com a limpeza de código.

Com o intuito de avaliar as diferenças entre o Minitest e o RSPec, o teste implementado anteriormente foi transcrito para o RSPec, e está apresentado no algoritmo abaixo.

```
1. RSpec.describe So, :type => :model do
2.  fixtures :sos
3.  it "must have material description"do
4.   service_order =
5.   sos(:material_description_nil).should_not be_valid
6.  end
7. end
```

As linhas 1 e 2 são semelhantes às linhas do código de teste com Minitest, informando que testes RSpec de modelo serão escritos para a entidade So e declarando as *fixtures*, respectivamente. Já as linhas 3 e 4 demonstram a principal diferença entre as duas gems: no RSPec, a descrição dos testes é feita de maneira comportamental, utilizando uma notação que se assemelha a uma descrição textual de um teste, auxiliando no entendimento do código.

Apesar do RSPec possuir uma linguagem mais informal para descrição dos testes, as duas gems analisadas atenderiam satisfatoriamente os testes desenvolvidos neste trabalho. Pelo fato de ser a gem padrão do framework, optou-se por utilizar o Minitest nos demais testes de modelos, funcionais e de visão.

Todas as entidades da aplicação foram mapeadas na camada de modelos do framework utilizando TDD com a gem Minitest, inclusive com restrições referentes à presença ou ausência de atributos, formato, tamanho mínimo, tamanho máximo e relacionamento entre as entidades, gerando uma camada de modelos coerente com as restrições impostas na modelagem de dados.

3.2 TESTES FUNCIONAIS

Testes funcionais foram criados para avaliar as requisições e respostas de todas as classes controladoras da aplicação. Para exemplificar como esses testes foram elaborados, serão apresentados os testes funcionais da controladora Material, responsável por gerenciar o acesso à entidade Material.

Requisições realizadas para um controlador RoR são feitas utilizando um dos seguintes métodos HTTP: GET, POST, PATCH, PUT ou DELETE. Dependendo do método utilizado na requisição e dos parâmetros passados, o RoR faz um roteamento automático da requisição para um método de um controlador, que será responsável por realizar todo o processamento necessário para respondê-la.

Os seguintes métodos são necessários na controladora Material:

- new: invocado através de uma requisição HTTP GET. Prepara os dados necessários para a criação de um Material e redireciona para uma visão;
- create: invocado através de uma requisição HTTP POST. Recebe um novo Material proveniente de uma visão, salva-o no banco de dados e redireciona para uma visão;
- destroy: invocado através de uma requisição HTTP DELETE. Recebe uma identificação de um Material já existente no banco de dados, remove-o e redireciona para uma visão.

Uma das restrições da modelagem de dados do sistema é que um Material só pode existir se estiver associado a uma SubServiceOrder e a uma ServiceOrder. Portanto, qualquer requisição para a controladora Material deve passar como parâmetro a identificação da SubServiceOrder e da ServiceOrder. Conforme o Rails Guides (2017), esse conceito pode ser implementado no RoR utilizando o conceito de rotas aninhadas, que força todas as requisições à controladora Material a passarem identificadores da SubServiceOrder e da ServiceOrder. Caso esses parâmetros não sejam passados, a controladora Material não

recebe a requisição.

A Tabela I mostra todas as possíveis requisições à controladora Material usando rotas aninhadas.

TABELA 1 - ROTAS ANINHADAS DA CONTROLADORA MATERIAL

Method	Routes	Controller#Action
GET	/sos/:so_id/sub_service_orders /:sub_service_order_id/materials/new	materials#new
POST	/sos/:so_id/sub_service_orders /:sub_service_order_id/materials	materials#create
DELETE	/sos/:so_id/sub_service_orders /:sub_service_order_id/materials/:material_id	materials#destroy

Para demonstrar como os testes das controladoras foram elaborados e como eles lidaram com os diferentes tipos de requisições e rotas, o próximo algoritmo descreverá três testes do controlador Material, um teste para cada tipo de requisição aceita. A controladora Material foi escolhida, pois as rotas de acesso aninhadas tornam seus testes mais complexos que os testes de outras controladoras.

```
1.  class MaterialsControllerTest < ActionController::TestCase
2.  def setup
3.   create (:material)
4.   @so= So.first
5.   @sub_service_order = SubServiceOrder.first
6.   @material = Material.first
7.  end
```

A linha 1 declara a classe de teste do controlador Material, que sempre deve estender da classe base de teste de controladoras do Minitest (ActionController::TestCase), para os testes funcionais.

A linha 2 define um setup para a entidade material. Setup é um conceito RoR que permite executar um bloco de código antes do início de cada teste (RAILSGUIDE,2017).

Uma das maneiras de utilizar fixtures nos testes é criar tuplas que rompem as restrições do banco de dados e utilizá-las nos testes. Entretanto, conforme os modelos da aplicação se tornam coerentes e testados, as tuplas inconsistentes das *fixtures* geram erros indesejados, exigindo revisão e manutenção constantes. Por essa razão, *fixtures* foram substituídas por factories em todos os testes.

Factories são modelos de registros que podem ser usados para popular uma base de testes, criando assim uma base de testes mais consistente e sem repetições. FactoryGirl é uma gem utilizada como fábrica de instâncias, mais versátil que as fixtures e cuja lógica fica isolada da implementação específica dos testes (THOUGHTBOT, 2017).

Muitos testes funcionais necessitam de uma base de testes para funcionarem. Na linha 4, a base de testes foi populada seguindo o modelo de uma factory, utilizando o método create. Esse método salva uma instância de Material na base de dados de teste para que possa ser utilizado em todos os testes.

Na linha 5, a variável global @so recebe o primeiro objeto encontrado na base de dados da entidade So, utilizando do método first da ActiveRecord. A partir dessa definição, qualquer teste da classe Material pode acessar essa variável e utilizar-se de seus atributos. O mesmo acontece nas linhas 6 e 7 com as entidades SubServiceOrder e Material.

```
10. test "action new should create an instance variable" do
11.   get :new, so_id: @so,
12.   sub_service_order_id: @sub_service_order
13.   assert_not_nil assigns (:material),
14.   "action new does not create an instance variable of
15.   type material"
16.  end
17.  test "action create redirects to action show after
18.   creating a well-formed material based on the form" do
19.   post :create, so_id: @so,
20.   sub_service_order_id: @sub_service_order,
21.   material: {name: "Name", amount: 10.0,
22.   value: 10.0, unit: "Unit",
23.   sub_service_order_id: @sub_service_order}
24.   asssert_redirected_to
```

```
25.    so_sub_service_order_paht(@so,
26.    @sub_service_order)
27.  end
28.  test "after calling the destroy action passing
29.  an invalid id, you must be redirected to the
30.  action show of the sub service order"do
31.    delete :destroy, so_id: @so,
32.    sub_service_order_id: @sub_service_order,
33.    id: -1
34.    assert_redirected_to
35.    so_sub_service_roder_paht (@so,
36.    @sub_service_order)
37.  end
38. end
```

O teste presente entre as linhas 10 e 16 analisa se, após uma requisição GET para a action new, o controlador instanciou corretamente uma variável material, que será utilizada na visão para a criação de um novo Material. Note que, devido ao aninhamento de rotas, foi necessário passar por parâmetro os identificadores da So e da SubServiceOrder.

O teste presente entre as linhas 17 e 27 verifica se a *action create* redirecionará para o local correto após ter recebido um Material preenchido do formulário, através de uma requisição POST. Já o teste presente entre as linhas 28 e 38 analisam se a *action destroy* se comporta corretamente caso o id da Material a ser removida for inválido.

Utilizando os recursos do Minitest, foi possível desenvolver utilizando TDD todos os métodos de todas as controladoras, testando de maneira exaustiva os dados por eles preparados, o processamento realizado e os seus redirecionamentos, considerando parâmetros válidos e inválidos.

3.3 TESTES DE VISÃO

Testes de visão foram criados com o objetivo de verificar se a interface com o usuário está de acordo com o que foi especificado. Ou seja, com esses testes é possível verificar se as *tags* html necessárias estão presentes na tela e se as informações nelas contidas são exatamente as informações que deveriam estar sendo exibidas.

Conforme o RailsGuide (2017), o *assert select* é um parser RoR que analisa o conteúdo HTML retornado por uma requisição feita a um controlador, e é uma poderosa ferramenta na validação de visões.

Nessa seção, apresentaremos alguns requisitos e restrições que devem ser atendidos pelas visões do controlador ServiceOrder, demonstrando como testes de visão podem ser implementados em RoR.

Como sistema elaborado é de um controle de ordens de serviço, a primeira visão apresentada deve conter uma tabela com todas as ordens de serviço já criadas. Essa visão é retornada pela *action index* do controlador ServiceOrder.

O primeiro requisito a ser validado foi que a visão *index* deve conter exatamente uma tag h2 com o conteúdo SERVICE ORDERS. O teste abaixo valida esse requisito.

```
1. test "index.html.erb must have a h2"do
2.  get :index
3.  assert_select 'h2', 1
4. end
5. test "index.html.erb h2 must have header SERVICE
6.  ORDERS" do
7.  get :index
8.  assert_select 'h2'do
9.   assert_select 'b', {count: 1,
10.   text: "SERVICE ORDERS"},
11.   "There is no header for service orders"
12.  end
13. end
```

O teste das linhas 1 até 4 está cobrando a presença de exatamente uma tag h2. Para isso, faz uma requisição GET para a *action index*, requisitando à controladora que redirecione para a visão *index*, como se fosse um usuário acessando via navegador a action index da ServiceOrder. O html é retornado e pode ser analisado pelo comando assert_select. O assert_select verifica se, nesse html retornado, possui exatamente uma tag do tipo h2. No teste das linhas 5 até 13, após selecionar a única tag existente do tipo H2, o assert_select analisa se dentro da tag h2 há exatamente uma tag b com o conteúdo dizeres SERVICE ORDERS.

Ambos os testes falham, pois a *view*

index encontra-se vazia. Para os testes passarem basta adicionar o código descrito abaixo, garantindo a existência de uma tag h2 e um cabeçalho para a *index* da ServiceOrder.

```
1.  test "index.html.erb must have a table"do
2.   get :index
3.   assert_select 'table', 1
4.  end
5.  test "index.html.erb must contain exactly one table
6.   with id sos" do
7.   get :index
8.   assert_select 'table'do
9.    assert_select "[id=?]", "sos"
10.  end
11. end
12. test "field material description of SOs is being
13.  displayed on table" do
14.  get :index
15.  @sos = So.all
16.  assert_select "table"do
17.   assert_select "[id=?]", "sos"do
18.    assert_select "tr" do
19.     @sos.each do |so|
20.      assert_select "td", {text:
21.       so.material_description} do
22.        assert_select "[id=?]",
23.        "so_material_descrption_"
24.        +so.id.to_s, {count: 1}
25.      end
26.     end
27.    end
28.   end
29.  end
30. end
```

```
<h2><b>SERVICE ORDERS</b></h2>
```

O próximo teste foi criado para validar a presença de uma tabela que lista as ordens de serviço na *index*. Essa tabela deve ter um *id* específico e, inicialmente, exibir pelo menos o campo material_description de todas as ordens de serviço presentes no banco. O código seguinte implementa esse teste.

Nas linhas 12 até 30, o teste fez uma requisição get para a controladora, recebendo de volta o conteúdo html da *index*. A variável global @sos recebeu todas as instâncias de So que estão na base de dados e, logo após, analisou se, para cada So presente no banco, há um campo td exibindo o atributo material_description.

Utilizando TDD, foi possível construir de maneira gradativa a tela inicial do sistema, validando todos os campos da tabela e todos os links que devem estar presentes. A Figura 4 mostra essa *interface*.

FIGURA 4 - *Interface INDEX* da ordem de serviço

SERVICES ORDERS

MATEIAL DESCRIPTION	GUIDE NUMBER	OPTIONS		
Viatura Blindada	2014110001	Show	Edit	Destroy
Motor de Embarcação	2014110002	Show	Edit	Destroy

Create new service order
Manage material classes
Manage military organizations
Manage section military organizations

Todas as demais telas do sistema, que envolvem criação, remoção, edição e atualização de várias entidades do sistema, incluindo entidades que possuem relacionamentos, foram criadas por meio de testes. As telas ficaram simplificadas, mas exibiram de maneira funcional todas as informações exigidas pelo cliente, buscando minimizar os problemas durante a utilização do sistema.

Testes de visão são muito importantes, pois, são através das visões que os usuários interagem com o sistema. Caso não sejam bem testados, erros graves podem acontecer durante a execução do sistema, podendo causar muito prejuizo a quem o utiliza.

É importante salientar que, apesar do layout de todas as visões estarem simplificados, os testes não levam em consideração os detalhes de estilo da aplicação. Isso significa que, caso um desenvolvedor deseje tornar a visão mais bem elaborada, os testes continuarão passando, desde que os dados sejam exibidos corretamente.

3.4 OUTRAS FERRAMENTAS PARA TESTE

Há outras gems RoR que podem ser usadas em um ou mais tipos de testes. Nesta seção duas delas serão descritas: o Cucumber e o Capybara.

Segundo o site oficial da ferramenta, o Cucumber é uma gem que cria um novo ambiente no projeto e permite a escrita de testes

de aceitação em uma linguagem muito próxima da natural. Já o Capybara, segundo Jnicklas (2012), é uma gem que ajuda a testar aplicações web, simulando como um usuário real interagiria com a aplicação.

O Cucumber foi desenvolvido para um padrão de desenvolvimento diferente do abordado neste artigo, chamado BDD (*Behavior Driven Development*). Seus testes são estabelecidos por cenários de teste onde é descrita uma ação e como o sistema deve se comportar. Um exemplo para nosso sistema de ordem de serviço é preencher todos os campos do formulário para uma nova ServiceOrder e clicar em salvar. O teste deve garantir que o usuário não ficou retido na mesma visão. O código abaixo descreve esse teste escrito com Cucumber.

```
1. Funcionalidade: Preencher o formulário da SO
2. Cenário:
3.    Deve preencher todos os campos do formulário e
salvar
4.   com sucesso
5.   Dado que eu estou na págino do formulário da SO
6.   Quando eu preencher todos  os campos
7.   E clicar em "Create SO"
8.   Então deve redirecionar para a action show da SO
```

A linha 1 descreve a funcionalidade do teste. As linhas 2 a 8 descrevem o cenário que deve ser realizado. Uma vez descrito o cenário no Cucumber, com o auxílio da *gem* Capybara, os campos do formulário serão preenchidos, como se fosse um usuário interagindo com a interface. Segue abaixo o código do Capybara para preenchimento do formulário da So.

```
1.   Dado/^que eu estou na página do formulário$/ do
2.     visit new_so_path
3.   end
4.   Quando /êu preencher todos os campos$/ do
5.     fill_in "material_description", :with=>"Blindado"
6.     fill_in "guide_number", :with=> "2014120001"
7.     fill_in "sender_name", :with=> "CB CICLANO"
8.     fill_in "sender_phone", :with=> "(67)1234-5678"
9.     fill_in "total_amount", :with=> "10"
10.    page.select "1", :from =>'material_class_id'
11.   page.select "2", :from =>'military_organization_id'
12.  end
13.  E /^clicar em "(.*?)"$/ do  |so_submit|
14.    find_button (so_submit).click
15.    save_and_open_page
16.  end
17.  Então /^deve redirecionar$/ do
18.    visit so_path(@so)
19.  end
```

Nas linhas 1 até 3, é realizada a requisição de um novo formulário para criação de uma nova So. Nas linhas 4 até 12, são realizados os preenchimentos dos campos, o código fill_in é responsável por preencher os fields do formulário. O page.select, nas linhas 10 e 11, realizam a seleção de um item do *select*.

As linhas de 13 até 16 simulam um clique no botão so_submit, salvando como o método save_and_open_page. As linhas 17 até 19 redirecionam para a *action show* da So.

CONCLUSÃO

O TDD é uma área muito ampla e relativamente recente, entretanto poucos materiais abordam por completo essa forma de desenvolvimento no ambiente RoR. Este artigo abordou o uso de desenvolvimento guiado por testes no RoR através da implementação de um sistema de ordem de serviço, mostrando como os testes unitários, funcionais e de visão podem ser feitos. Durante o desenvolvimento, foram citadas algumas dificuldades e facilidades dessa metodologia, com intuito de auxiliar aqueles que desejam utilizar essa forma de desenvolvimento no RoR.

O uso do TDD na implementação do sistema de ordens de serviço permitiu com que o sistema fosse desenvolvido de maneira gradativa, atendendo apenas os requisitos essenciais para o cliente. Por conta da mudança e adaptação com o novo paradigma, houve um atraso no desenvolvimento, mas que foi compensado pelo fato da aplicação conter um código bem testado, que facilitará futuras manutenções.

Outra vantagem dessa forma de desenvolvimento é a refatoração e a depuração de erros do código. Refatorar o código garante que ao final do desenvolvimento tem-se um código mais limpo, somente com o que interessa para que o sistema funcione, acabando com inserção de códigos desnecessários que po-

luem e dificultam a depuração.

Com os testes de modelos, funcionais e de visão o sistema será capaz de detectar alterações indevidas em qualquer uma das três camadas. Isso mostra que o TDD é muito importante em ambientes onde há alta rotatividade de desenvolvedores, pois evitará que erros cometidos por desenvolvedores em fase de adaptação com o sistema, auxiliando na estabilidade da aplicação.

O framework RoR possui ferramentas suficientes para o desenvolvimento TDD, e sua ferramenta padrão (Minitest) é suficiente para a realização de testes completos de modelo, funcionais e de visão completos. Dentre os três tipos de testes, o que mais houve dificuldade foi o teste de visão, pois a documentação da principal função utilizada (assert_select) é confusa e pouco explicativa.

Manter uma base de testes consistentes é de extrema importância para que os testes tenham efeito. Em diversos momentos, no decorrer do desenvolvimento, testes estavam aparentemente funcionando, quando na verdade nem estavam sendo executados por conta de um preenchimento equivocado da base de testes. O uso de *Factories* se mostrou mais promissor que o uso de *Fixtures*, pois facilita a criação e manutenção da base de testes.

Futuramente, pretende-se melhorar o sistema de ordem de serviço, aplicando as demais ferramentas de TDD do RoR, como os testes de integração do Minitest, e também aplicar o conceito de BDD utilizando as gems RSpec, Cucumber e Capybara.

Alguns artigos sobre TDD mencionam possíveis métricas para se avaliar, a longo prazo, os impactos do TDD. Ainda em um escopo futuro pretende-se avaliar os impactos causados por conta da implementacão desse sistema, auxiliando também na criação de novas métricas para análise da eficácia dos testes de software.

REFERÊNCIAS

AkitaonRails. Entendendo RubyGems. Disponivel em: <http://www.akitaonrails.com/2009/02/02/entendendo--rubygems#.VI4YMCvF91Y >. Acesso em: 05 setembro 2017.

AkitaonRails. Entendendo RubyGems. Disponível em <http://www.akitaonrails.com/2009/02/02/entendendo--rubygems#.VIdEsq2BvFY>. Acesso em: 09 setembro 2017.

ApiRails. Disponivel em: <http://guides.rubyonrails.org/testing.html >. Acesso em: 05 setembro 2017.

ANDRADE, Bruno Eustáquio. et al. TDD-Test Driven Development. 2011 Disponivel em: <http://pt.scribd.com/doc/53948144/Texto-Explicacao-TDD >. Acesso em: 20 outubro 2017.

Barbosa, E.F; Maldonado, J.C; Vincenzi, A.M.R.; Delamaro, M.E;Souza, S. R. S; Jino, M. **Introdução ao Teste de Software**. ICMC-USP-São Carlos, 2000.

Basecamp. The best parts of Basecamp have been turned into open-source projects. Disponível em <https://basecamp.com/open-source>. Acesso em: 09 outubro 2017.

Baumeister, H. and Wirsing, W., **Applying Test--First Programming and Iterative Development in Building an E-Business Application**. Disponível em http://citeseerx.ist.psu.edu/viewdoc/download?-doi=10.1.1.19.4707&rep=rep1&type=pdf. Acesso em: 18 outubro de 2017.

Beck, Kent, TDD - Desenvolvimento guiado por testes. Bookman, 2010.

BetterExplained. Intermediate Rails: Understanding Models, Views and Controllers. Disponível em <http://betterexplained.com/articles/intermediate-rails-understanding-models-views-and-controllers/>. Acesso em: 09 setembro 2017.

Causevic, A.; Punnecatt, S.; Sundmark, D., **Quality of Testing in Test Driven Development**, IEEE: 2012 Eighth International Conference on the Quality of Information and Communications Technology.

Corbucci, H e Aniche, M, **Test Driven Development:** Teste e design no mundo real com Ruby. Casa do Código, 2014, 207 p.

Cukes. Cucumber. Disponível em <https://cucumber.io/>. Acesso em: 19 outubro 2017.

Github. GitHub developer. Disponível em <https://developer.github.com/>. Acesso em: 09 setembro 2017.

GUNDERLOY, M.; NAIK, P.; NORIA, X. **Guia do Rails** - Um Guiapara Testar Aplicações Rails , 2011. Disponivel em: <http://guias.rubyonrails.com.br/testing.html >. Acesso em: 10 outubro 2017.

Gupta, A. and Jalote, P., **An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development** , IEEE: 2007 First International Symposium on Empirical Software Engineering and Measurement.

Jnicklas. Test your app with Capybara. Disponível em <http://tutorials.jumpstartlab.com/topics/capybara/capybara_with_rack_test.html>. Acesso em: 19 outubro 2017.

Kaufmann, R. and Janzen, D., **Implications of test-driven development:** a pilot study , OOPSLA '03 Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2003.

Pressman, R. S. **Engenharia de Software**. 5 ed. Rio de Janeiro: Mc Graw Hill, 2002, 843 p.

RailsGuides. Disponivel em: <http://guides.rubyonrails.org/routing.html>. Acesso em: 05 outubro 2017.

_____. Disponivel em: <http://guides.rubyonrails.org/ >. Acesso em: 11 outubro 2017.

Rails Guides. Active Record Basics Disponivel em: <http://edgeguides.rubyonrails.org/active_record_basics.html>. Acesso em: 05 setembro 2017.

RailsGuidesTesting. Disponivel em: <http://guides.rubyonrails.org/testing.html>. Acesso em: 11 outubro 2017.

Rappin, Noel, Rails test prescriptions - Keep Your Application Healthy, 1 ed. Pragmatic Bookshelf, 2011.

Relish. RSpec. Disponível em <https://relishapp.com/rspec>. Acesso em: 02 setembro 2017.

Ruby-Doc. Minitest. Disponível em <http://ruby-doc.org/stdlib-2.0/libdoc/minitest/rdoc/MiniTest.html>. Acesso em: 12 setembro 2017.

SANCHEZ, I., **Introdução do Desenvolvimento voltado a Testes (TDD)** ¡ Coding Dojo Floripa. Coding Dojo Floripa , 2006. Disponivel em: <http://dojofloripa.wordpress.com/2006/11/07/introducao-ao-desenvolvimento-orientado-a-testes>. Acesso em: 11 novembro 2014.

SEA, T., Minicurso de TestesOnRails. Slideshare, 10 Agosto 2009 .Disponivel em: <https://pt.slideshare.net/seatecnologia/minicurso-de-testesonrails>. Acesso em: 11 outubro 2017.

Software Livre Brasil. Disponível em <http://softwarelivre.org/ruby-on-rails>. Acesso em: 12 outubro 2017.

Souza, K. and Gasparotto, A., **A importância da atividade de teste no desenvolvimento de software**, XXXIII Encontro nacional de engenharia de produção, 11 outubro 2013 . Disponível em:<http://www.abepro.org.br/biblioteca/enegep2013_TN_STO_177_007_23030.pdf>. Acesso em: 18 outubro 2017.

Swebok 2004,Guide for the software engineering body of knowledge, 2004 version, IEEE computer society, California, EUA.

Techtudo. Guia do Twitter: descubra como fazer tudo com dicas e tutoriais. Disponível em <http://www.techtudo.com.br/dicas-e-tutoriais/noticia/2011/03/twitter-guia-completo.html>. Acesso em: 09 setembro 2017.

Thomas, D. H., **Desenvolvimento Web Ágil com Rails**. Porto Alegre: Editora Bookman, 2008.

Thoughtbot, Inc. "Factory Girl". Disponível em <https://github.com/thoughtbot/factory_girl_rails>. Acesso em: 20 outubro 2017.

O Autor é Bacharel em Análises e Desenvolvimento de Sistemas pela UFMS, Instrutor dos Cursos IT Essencial, CCNA 1, CCNA 2 da Academia Cisco e possui o curso de Guerra Cibernética para sargentos e pode ser contactado por intermédio do email scopel.bruno@eb.mil.br.