UNIVERSIDADE FEEVALE

CESAR SCUR

UMA REVISÃO DAS IMPLEMENTAÇÕES DO *FRONT CONTROLLER* E UMA PROPOSTA DE AJUSTE

Novo Hamburgo
2012

CESAR SCUR

UMA REVISÃO DAS IMPLEMENTAÇÕES DO *FRONT CONTROLLER* E UMA PROPOSTA DE AJUSTE

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Ricardo Ferreira de Oliveira

Novo Hamburgo
2012

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização deste trabalho de conclusão, em especial:

Aos amigos e às pessoas que convivem comigo diariamente, minha gratidão, pelo apoio emocional - nos períodos mais difíceis do trabalho.

Enfim, a minha namorada, corresponsável por este trabalho, que me apoiou incondicionalmente.

RESUMO

Testes ajudam o desenvolvedor a tornar evidente problemas de design. Os testes unitários são notórios por seu apoio como metodologia para o *design* do software dentro da metodologia do *test driven development*. Seguindo esses conceitos é muito importante ter o máximo de cobertura de testes. Contudo, nem todo pedaço de software é facilmente coberto com testes unitários. O *pattern front controller* é um exemplo desta circunstância. Testar unitariamente o *controller* não é um processo simples, e normalmente é abordado por teste de integração. Propor uma alternativa que simplifica a implementação de testes unitários para o *controller* é o intuito deste trabalho. Para tanto, será realizada uma implementação adaptada do *pattern front controller*. O sucesso dessa implementação será medida através de um cenário onde testes unitários serão implementados para o protótipo e comparados aos testes de integração.

Palavras-chave: Front Controller. Unit Test. Framework.

ABSTRACT

The test helps the developer to make design problems evident. The Unit tests are notable by your support as a software design methodology as part of the test driven development methodology. Following these concepts is very important to have the maximum test code coverage. However, not all piece of software is easily covered with unit tests. The front controller pattern is an example of this circumstance. Unit test controllers is not an easy task and normally is done by integration tests. Propose an alternative that simplifies the implementation of unit tests for the controller is the intent of this paper. Therefore, will be done an adapted implementation of the front controller. The success of this implementation will be measured on a scenario where unit tests will be implemented for the prototype and then compared to the integration tests.

Key words: Front Controller. Unit Test. Framework.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

MVC	Model View Controller.
TDD	Test Driven Development.
ORM	Object Relational Mapper.
BDD	Behavior Driven Development.
URL	Endereço (*Uniform Resource Locator*).
IP	Endereço de rede.
HTTP	Protocolo de Tranferencia de Hypertexto.
ERB	Um sistema de templating para ruby.
CRUD	*Create, Read, Update e Delete* – Sistema simples que implementa as tarefaz mais basicas.
ID	Identificador.
ZF	Zend Framework.
LLOC	Linha de Código Lógica.

SUMÁRIO

INTRODUÇÃO

Em 1989, Kent Beck, publicou seu artigo sobre um *framework* de testes de software automáticos *Simple Smalltalk Testing: With Patterns*. Este artigo definia o xUnit, as fundações do teste automático usado até hoje. Este *framework* foi traduzido para Java por Erich Gamma e Kent Beck.

São mais de 20 anos que esse movimento teve início, mas apenas entre os últimos 5 a 10 anos que passou-se a escutar o mercado se movimentando e demandando testes de software. E aos poucos o teste deixou de ser uma ferramenta de apoio, passando a ser uma ferramenta essencial de qualidade, até chegar a uma metodologia de desenvolvimento.

Essa evolução ainda não chegou ao seu fim. Ainda é possível acompanhar e observar os testes entrando mais e mais no escopo de projetos com metodologias como TDD sendo estendidas em BDD.

Os métodos de se executar testes também são bastante variados. Testar uma classe requer um teste unitário, enquanto um teste que valida se o comportamento entre múltiplas classes está ocorrendo apropriadamente é um teste de integração.

As metodologias de aplicação de testes também passam por conceitos como integração contínua onde os códigos de diversos programadores são submetidos aos testes juntos no mesmo ambiente, mostrando que todos, juntos, se comportam como o esperado.

Enfim, o ambiente onde os testes de *software* acontecem é muito rico e está em franca atualização e revisão. Esse ambiente ainda é jovem e evolui rápido a cada dia, com necessidades e abordagens novas para as mesmas soluções antigas. Feathers (2004) acredita inclusive que código sem testes é mau código.

Nesse ambiente, existe um pequeno trecho do *software* onde os testes unitários ainda são pouco empregados. Esse trecho é o *controller* de aplicações MVC. Ele é basicamente coberto por testes de integração. Mesmo testes de integração sendo na pratica suficientes, não são precisos o suficiente para garantir o funcionamento do fonte, e não estão totalmente de acordo com metodologias como o TDD. Que segundo Beck (2010), é muito útil em vencer o medo de problemas complexos, apoiando uma modelagem correta.

O objetivo deste trabalho é justamente atuar na implementação do *front controller*, a fim de simplificar o processo de teste unitário do *controllers*. Adaptando a forma com que o *framework* que implementa o *front controller* interage com a aplicação.

Para tanto este trabalho irá situar o escopo citando a importância do *controller* e o contexto onde o mesmo está inserido. Detalhando a relevância dos design *patterns*, do MVC e do *front controller* nos *frameworks* e aplicações que os usam no capitulo um. Assim como contextualizar a importância dos testes como ferramenta de trabalho para garantir a funcionalidade e como metodologia de trabalho para melhorar o design de código no capitulo dois. E cruzará os dois, temas onde o problema acontece, nos testes de *controller*, no capitulo três.

Por fim, definir os detalhes acerca de como resolver problema de implementação de testes unitários de *controller*. Quais as mais comuns e corriqueiras situações onde os testes devem ser implementados. E finalmente definir um modelo que resolva ou melhore a implementação de testes unitários no *controller* no capitulo quatro.

Para assegurar e medir o sucesso do modelo proposto de solução, um protótipo funcional será implementado com base no modelo e testado dentro das situações comuns onde o mesmo ocorre no capitulo cinco. Sendo assim comparado com implementações regulares de *controller* no capitulo seis.

1 PATTERNS

Design patterns são uma série de soluções conhecidas como verdadeiras, para problemas comuns de arquitetura de software. Cada *design pattern* visa resolver um problema específico de arquitetura. Eles costumam ser apresentados pelos autores na forma de catálogos, onde são listados junto com sua definição e aplicação no mundo real.

Segundo Gamma(1994) *design patterns* são soluções que foram aplicadas por desenvolvedores experientes e se provaram corretas ou eficientes. Essa experiência leva tempo para um iniciante adquirir. O propósito dos padrões é fornecer um repositório que documenta esses padrões para serem aplicados por outros desenvolvedores em seus projetos.

> *"(...) você vai encontrar padrões recorrentes de classes e comunicação de objetos em vários sistemas orientados a objeto. Estes padrões resolvem problemas específicos de design e tornam designs orientados a objeto mais flexíveis, elegantes e ultimamente reusáveis." (GAMMA, 1994, tradução nossa)*

Padrões de projeto tornam o código mais reaproveitável, diminuindo a quantidade de código repetido. Também torna o código mais flexível, o que quer dizer que acrescentar funcionalidades, ou modificá-las torna-se mais fácil.

O grande propósito dos *patterns*, assim como o deste trabalho, é reduzir o custo de manutenção de *software*. O custo de manutenção tende a aumentar com o tempo de vida de um aplicativo. Isso é percebido antes mesmo do programa entrar em produção, em projetos mais longos.

Muito disso tem relação com a complexidade total da aplicação. É praticamente impossível para um programador armazenar mentalmente toda a lógica de uma aplicação a fim de saber exatamente como determinado comportamento acontece em um algoritmo monolítico.

Da mesma forma, a tendência em um algoritmo pouco especialista é que trechos de código se repitam ao longo do programa. Isso leva manutenções a terem de corrigir inúmeros lugares diferentes. *Patterns* são soluções que procuram contornar ao máximo esses problemas, escrevendo algoritmos que resolvem os problemas de maneira mais reaproveitável.

Isso, finalmente, reduz o custo de manutenção que onera a maior parte do *software*. Martin, quando comenta sobre uma situação hipotética, porém plausível, de um projeto que mantém um código ruim:

> *"À medida que a bagunça toma conta, a produtividade do time continua a cair, assintoticamente aproximando-se de zero. À medida que a produtividade diminui, a gestão faz a única coisa que eles podem; eles adicionam mais pessoas no projeto na esperança de aumentar a produtividade. Mas esse pessoal não é versado no design do sistema. Eles não conhecem a diferença entre uma mudança que se adéqua a intenção do design e a mudança que vai contra a intenção do design. Além disso, eles, e todos os outros do time, estão sobre uma pressão horrível para aumentar a produtividade. Então todos eles fazem mais e mais porcarias, levando a produtividade ainda mais para perto do zero." (MARTIN, 2009, tradução nossa)*

Martin acredita que a complexidade do código pode ser vencida de várias maneiras. Não somente com padrões de *design*, mas também, até mesmo a correta nomenclatura de variáveis pode tornar a leitura do código mais eficiente e, portanto mais fácil de entender e dar manutenção.

1.1 Frameworks

A aplicação dos *patterns* em *frameworks* tornou a abrangência dos mesmos muito grande. A maioria – se não todos – os *frameworks* de mercado implementam tantos *patterns* quantos possíveis, para resolver a arquitetura do próprio *framework* ou a arquitetura da aplicação.

Dentro de uma aplicação web, por exemplo, existe uma série de problemas comuns que todas as aplicações compartilham. São problemas inerentes da própria aplicação ou da plataforma. Não são necessariamente problemas, mas sim características.

1.2 MVC

O *Model-View-Control* (MVC) modificou o desenvolvimento de *software*. Com o intuito de separar lógica de aplicação de lógica de apresentação e lógica de negócio, o MVC é um dos *patterns* que se popularizaram bastante e melhoraram muito a manutenção de aplicações *web*.

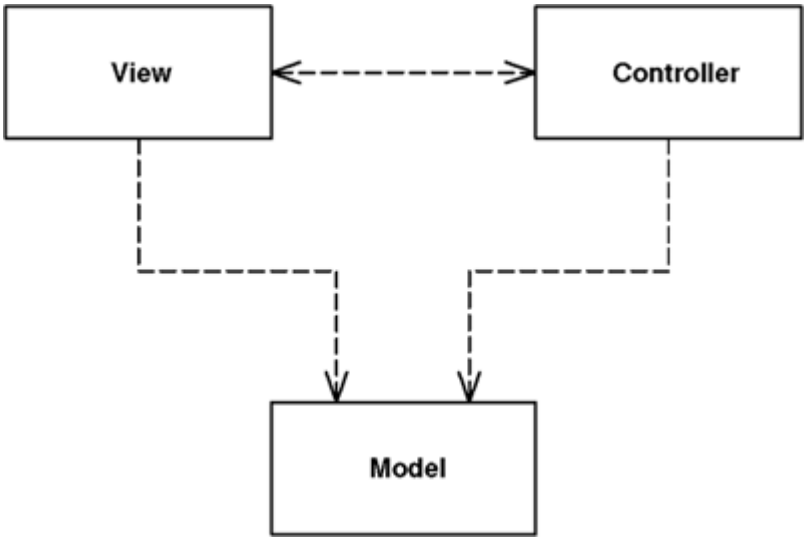


Figura 1.1 - Divide a interação com o usuário nos três papeis distintos.
Fonte: Fowler (2002).

O MVC divide a camada de apresentação (*view*) da lógica de negócio e dados (m*odel*) e usa como cola o *controller*, que recebe a requisição, ou ação do ator da aplicação, conectando o *model* à *view*.

Dividir estas partes da aplicação faz muito sentido, uma vez que pode-se apresentar dados de um *model* de várias maneiras, usando cliente para o exemplo. Pode-se apresentar uma lista de clientes, como em um cadastro para edição ou um *dropbox* quando deve-se preencher um pedido de compra.

Ter múltiplas apresentações possíveis para o mesmo *model* faz muito sentido em uma aplicação. Da mesma forma que tendo o *model* separado do *controller* permite que diga-se que um cliente é listado apenas quando seu status ativo é verdadeiro. Isso quer dizer menos código duplicado.

Muitos *frameworks* implementam não só o modelo MVC, mas também uma série de ferramentas para cada um dos componentes. É bastante comum encontrar *object-relational-mapper* (ORM) para *models*, *template engines* para *views* e *front controllers* quando se trata de uma aplicação *web* para tratar a requisição (*controller*).

1.3 Front Controller

Grande parte dos *frameworks web* de mercado implementam a arquitetura MVC, o que quer dizer que também implementam o *pattern front controller*. O *front controller* é responsável por receber a requisição e disparar o *controller* adequado à requisição.

> *"O Front Controller recebe todas as requisições de um Web site, e é comumente estruturado em duas partes: um Web handler e uma hierarquia de comandos. O Web Handler é o objeto que verdadeiramente recebe as requisições de post e get do servidor Web. Ele pega apenas informação suficiente da URL e da requisição para decidir que tipo de ação iniciar e então delega para um comando dar andamento na ação" (FOWLER, 2002, tradução nossa)*

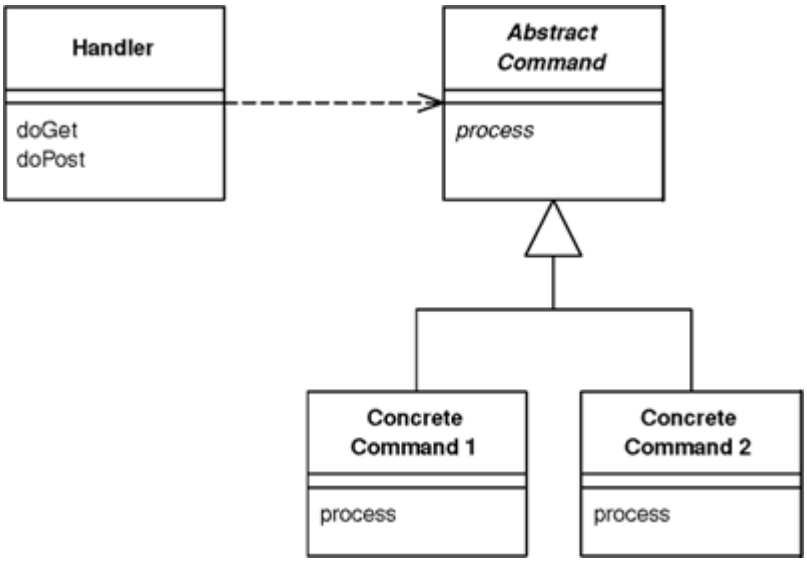


Figura 1.2 - Um controlador que gerencia todos os pedidos de um *site*.
Fonte: Fowler (2002).

O *front controller* é uma parte crucial no uso do MVC. Ele é responsável por toda entrada e roteamento das requisições que chegam à aplicação, endereçando-as ao *controller* adequado à execução. Martin Fowler(2002) fala também em seu livro, de uma abordagem interessante onde o *front controller* é dividido em dois:

> *"Rob Mee me mostrou uma variação interessante do front controller usando um web handler de dois estágios separados em um web handler degenerado e um dispatcher. O web handler degenerado puxa a informação básica dos parâmetros do HTTP e passa-os para o dispatcher de tal maneira que o dispatcher é completamente independente do framework do web Server. Isso torna os testes mais fáceis porque o código de teste pode chamar diretamente o dispatcher sem ter de rodar o web Server". (FOWLER, 2002, tradução nossa)*

A abordagem do *dispatcher* pode ser encontrada em *frameworks* como o *Zend Framework*. Fowler aponta que esse método de implementação do *front controller* tem a vantagem de tornar o processo de testes mais simples por poder iniciar o *dipatcher* sem existir um *request.*

1.3.1 Dispatcher

Na realidade, a implementação do *dispatcher* no mundo real não acontece bem assim. De fato as implementações do *dispatcher*, como a da Zend, requerem um *request* para acontecer. Não um *request* o evento, mas um objeto de *request.*

O *Zend_Controller_Dispatcher_Standard::dispatch()* recebe como parâmetros o *request* e o *response*. Eles são essenciais para indicar qual *controller* e qual *action* o *front controller* deve executar. Contudo, o *request* é passado para o *dispatcher* como um objeto previamente encapsulado no *web handler*, como prevê o modelo conceitual de Fowler (2002).

1.3.2 Controller Action

Controller action é a ponta final do *front controller*. Após receber, encapsular e rotear o *request* e passar a incumbência de executar a ação final para o *dispathcer*, essa ação toma forma de *action*. A *action*, apesar de não ter uma definição clara na literatura, costuma ser implementada com uma classe *Controller Action*. Fowler (2002) apresenta uma implementação de *controller* que estende de *FrontCommand*, apesar disto o nome comum encontrado no mercado é *ControllerAction* (ou similar).

Cada método do *controller* é uma *action*. O *controller action* também costuma ser apoiado pelo *framework* para uma classe abstrata (ou não) contendo as mecânicas relevantes para ligar o *dispatcher* ao *controller*, sob o nome de *ControllerAction*.

No caso do *framework* da Zend, é o próprio *ControllerAction* que executa a parte final de chamar o método indicado para o *request*. Neste estágio, é que ocorre a última interação do *framework* com o *request* diretamente. Apenas deste ponto em diante o programa passa a executar, na linguagem dos desenvolvedores de *framework*, "*userland script*" (*script* do desenvolvedor da aplicação). Ou seja, a *action* do *controller* implementado usando o *ControllerAction.*

```php
class Zend_Controller_Dispatcher_Standard {
    /*...*/
    public function dispatch(/*...*/)
    {
        $className = $this->getControllerFromRequestName();
        $action = $this->getActionFromRequestName();
        $controller = new $className();
        $controller->dispatch($action);
    }
    /*...*/
}

class Zend_Controller_Action {
    /*...*/
    public function dispatch($action)
    {
        $this->$action();
    }
    /*...*/
}

class Userland_Controller extends Zend_Controller_Action {
    public function index()
    {
        /*
         * Userland code
         */
    }
}
```

Figura 1.3 - Pseudo código ilustrando a interação entre *dispatcher* e *action.*
Fonte: O autor.

Estas fronteiras entre o *framework* e o *userland script* são importantes. Pois é nela que o problema deste trabalho começa. *Controllers* são difíceis de serem testados. As técnicas de testes de *controllers* envolvem testes de integração, o que nem sempre é o mais apropriado. Uma solução para esse problema pode estar relacionado com o *controller action*.

2 TESTE

Um aspecto do *front controller* é interessante, essa preocupação com o teste, que motivou a divisão do *web handler* e do *dispatcher*, não se repete quando assunto é o *controller* em si. Testar o *controller* exige testes de integração para que possa ser concluído.

> *"Código sem teste é mau código. Não importa o quão bem escrito esteja; não importa o quão bonito ou orientado ou bem encapsulado. Com testes, podemos mudar o comportamento do código e rapidamente e de maneira verificável. Sem eles, nós realmente não sabemos se nosso código está ficando melhor ou pior." (FEATHERS, 2004, tradução nossa)*

Feathers (2004) expressa de maneira enfática o quanto teste de *software* é importante. O teste de *software* tem sido uma busca constante na qualidade de software, a ponto de mudar a maneira com que o software é escrito.

No entanto, a preocupação dos desenvolvedores quanto ao *design pattern* é que o modelo em si possa ser testado. Isso não contempla o bloco de código que o *front controller* irá disparar. Ou seja, pode-se testar o *front controller* separado do *dispatcher* mas não se pode testar o bloco de código sem envolver o *dispatcher*.

Para testes de *controller*, atualmente usa-se a abordagem de teste de integração. Testes de integração têm como papel testar diversos componentes, e como eles juntos se comportam. Porém, testes de integração têm alguns pontos fracos.

Por não serem precisos a ponto de indicar se o erro ocorre no método, antes ou depois, isso pode levar a falsos positivos. Em um exemplo mais prático, pode-se dizer que por mais que seja visto o carro entrar no lava a jato, entrar seco e sair seco, olhando de fora não há como saber que na verdade ele foi molhado no processo. Quer dizer, o método interno era realmente para usar água?

Para Sanderson (2009), testes de integração são adequados para testar regressões, encontrar *bugs* e testar aceitação. Porém, apenas o teste unitário é capaz de apoiar o *design* do código, como o proposto pelo *Test Driven Development* (TDD).

Ainda segundo Sansderson (2009), o apoio de modelagem que o TDD causa no desenho do software é muito mais relevante do que o teste em si. Este teste torna-se realmente útil quando é feita a refatoração do código, como no proposto pelo desenvolvimento guiado por testes.

Contudo, *controllers* não são fáceis de serem testados unitariamente. O que gera uma preocupação muito grande nos desenvolvedores, em mover todo o código para camadas de serviço ou *models*, mas ainda sim deixando os *controllers* vulneráveis durante uma refatoração.

2.1 Teste de Integração

Nem todo teste automatizado de *software* pode testar o mesmo unitariamente. Alguns trechos de código apresentam uma integração muito grande. Isso acontece quando o método ou classe testada, por exemplo, não tem nenhum retorno e apenas insere os resultados no banco de dados usando um componente de ORM. Nesse caso, a única forma de saber se o resultado do teste é positivo ou negativo é avaliar o conteúdo do banco.

Ou seja, existe uma dependência tão forte entre a classe e o componente de ORM que não é possível avaliar o comportamento da classe testada sem observar o resultado gerado pelo ORM. Mas não apenas nesta circunstância o teste de integração é válido.

Uma das grandes virtudes do teste de integração é indicar se o comportamento de todos os componentes envolvidos em um processo obtém o resultado esperado. Para exemplo, presumi-se uma determinada interface de usuário de um programa. Nessa interface uma série de campos de um formulário simples está preenchida e um determinado botão deve, quando atuado, inserir os campos em um novo registro do banco de dados.

```ruby
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do

    # User avs logs in
    avs = login(:avs)
    # User guest logs in
    guest = login(:guest)

    # Both are now available in different sessions
    assert_equal 'Welcome avs!', avs.flash[:notice]
    assert_equal 'Welcome guest!', guest.flash[:notice]

    # User avs can browse site
    avs.browses_site
    # User guest can browse site as well
    guest.browses_site

    # Continue with other assertions
  end

  private

  module CustomDsl
    def browses_site
      get "/products/all"
      assert_response :success
      assert assigns(:products)
    end
  end

  def login(user)
    open_session do |sess|
      sess.extend(CustomDsl)
      u = users(user)
      sess.https!
      sess.post "/login", :username => u.username, :password => u.passwor
      assert_equal '/welcome', path
      sess.https!(false)
    end
  end
end
```

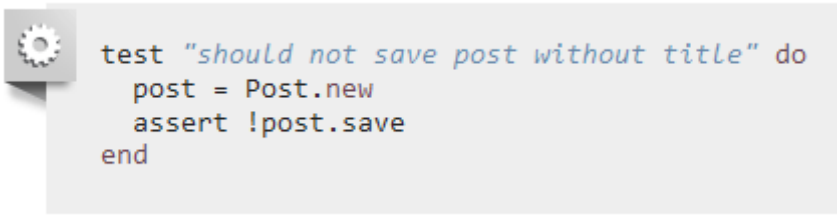
Figura 2.1 - Teste de integração em Ruby on Rails.
Fonte: Ruby on Rails Guide (2012).

Um determinado clique do mouse e os campos da tela são devidamente inseridos no banco. No processo de inserção todas as permissões do usuário logado são avaliadas e caso o mesmo não tenha direito aquela operação um erro deve ser gerado. A situação descrita é bem complexa. Uma série de pontos podem falhar individualmente. Mas a resposta deste teste oferece um panorama geral de que a aplicação, como um todo, faz o que dela é esperado.

Contudo, esse tipo de teste não é preciso o suficiente e está sujeito a mascarar alguns erros. Se no exemplo, em que um programador, ao fazer uma alteração para tornar um determinado campo da interface obrigatório, acidentalmente no processo ele faz também com que a verificação de permissão falhe, ao ser executado o teste de integração o mesmo irá falhar, como o previsto, uma vez que o campo agora é obrigatório. O teste, desta forma, indicará que o programa está rodando como o esperado. Porém isso não é correto.

Sanderson (2009) aponta que o teste de integração é a ferramenta ideal para atingir a meta de gerar um teste de regressão. Onde pode-se verificar que um código que funcionava parou de funcionar.

2.2 Teste Unitário



```
test "should not save post without title" do
  post = Post.new
  assert !post.save
end
```

Figura 2.2 - Teste unitário em Ruby on Rails.
Fonte:Ruby on Rails Guide (2012).

Uma unidade é a menor parte testável de um *software*. Teste unitário é o processo pelo qual uma unidade de *software* é testada para identificar uma única funcionalidade. O teste unitário verifica se uma, e apenas essa, funcionalidade responde como o esperado.

Uma unidade costuma corresponder a uma classe ou um método de uma classe. Cada método representando uma funcionalidade específica do *software*. Normalmente, cada funcionalidade é coberta por mais de um teste. Verificar se ela ocorre como o esperado. Verificar se ela falha como o esperado. Verificar que ela não retorna negativamente como o esperado e assim por diante.

> *"O termo teste unitário tem uma longa história no desenvolvimento de software. Comum à maioria dos testes unitários a idéia de que eles são testes em isolamento de componentes individuais do software. O que são componentes? A definição varia, mas em testes unitários, nós geralmente estamos preocupados com as unidades comportamentais mais atômicas de um sistema. Em um código procedural, as unidades frequentemente são funções. Em código orientado a objeto, as unidades são classes." (FEATHERS, 2004, tradução nossa)*

Martin (2008, tradução nossa) comenta que “Eles são fáceis de escrever e o valor documental deles é maior que o custo de produzi-los” quando comenta sobre o aspecto do teste que é muito relevante. Testes são tão fáceis de ler e tão precisos em explicar que eles acabam se tornando uma ótima ferramenta de documentação por si só.

Sanderson (2009) acredita que a frase popular “qualquer teste é melhor que teste nenhum” não é adequada e que “Um *suite* de testes pode ser um ótimo ativo, ou pode ser um grande fardo que contribui pouco. Isto depende da qualidade destes testes, o que parece ser determinado pelo quão bem os desenvolvedores tem o entendimento das metas e princípios do teste unitário.”

Testes unitários também têm um papel de ferramenta de design de código. O teste unitário compele o desenvolvedor a gerar código testável, o que leva o fonte a ser mais atômico possível o que torna a unidade o mais reaproveitável possível como efeito colateral.

2.3 TDD (Test Driven Development)

O TDD traz uma importância extra para o teste unitário. Nele o teste não tem mais somente o papel de verificar se o código está respondendo como o esperado. O teste passa a ser parte do processo de desenvolvimento. Uma metodologia. Um processo de desenvolvimento de *software* que inicia pelo teste.

Para o TDD desenvolver o teste unitário, significa muito mais que assegurar o código. Significa que o desenvolvedor estará focado em resolver o problema. O programador não precisa e não deve se preocupar com nenhuma questão estética do código ou como essa funcionalidade se encaixa no todo da aplicação. O teste serve muito mais como um apoio psicológico para apoiar o desenvolvedor no foco do problema e resolver o problema.

Por isso o TDD propõe um ciclo de três etapas:
Desenvolver o teste; Rodar o programa e ver o teste falhar;
Desenvolver o programa; Rodar o programa e ver o teste passar;
Refatorar o código;

Uma vez que o código está passando no teste, o programador pode mudar seu foco para a qualidade do programa. Ele pode então, limpar toda a bagunça que foi feita para chegar à solução se preocupando com as questões de *design* de código.

Isso tudo sem se preocupar em quebrar o funcionamento do programa, pois se algum comportamento for mudado o teste irá indicar ao desenvolvedor que a mudança que ele fez afeta o comportamento e não mais apenas o *design* do código.

> *"Imagine que programar é girar uma manivela para puxar um balde com água de um poço. Quando o balde é pequeno, uma manivela sem catraca é ótima. Quando o balde é grande e está cheio de água, você ficara cansado antes do balde ser completamente puxado." (BECK, 2010)*

Beck explica que os dentes da catraca de sua metáfora são como os testes unitários. Eles auxiliam o programador a encarar cada pequena parte do problema de cada vez, sem ter que se preocupar com o todo. Isso o ajuda a contornar o medo de errar.

Ainda segundo Beck (2010) o "Desenvolvimento guiado por teste é uma forma de administrar o medo durante a programação." O medo no sentido legítimo de "esse-é-um-problema-difícil-e-eu-não-consigo-ver-o-fim-a-partir-do-começo" (BECK, 2010). O medo de não conseguir entregar a aplicação no prazo por ela ser muito grande e complexa.

3 TESTE DE CONTROLLER

Controllers não costumam ser foco de testes. Eles são comumente ignorados como fontes a serem testados e o máximo do seu comportamento é transferido para *models* onde ele pode ser testado mais facilmente.

Fatorar o código do *controller* transferindo sua maior parte para o *model* faz sentido se esse código diz respeito ao *model* e trata de regra de negócio. Contudo, uma parte do código do *controller* não faz sentido, nem é viável de ser implementado em um *model*.

Quatro aspectos básicos e recorrentes costumam aparecer em *controllers*, mesmo quando todo seu código já foi fatorado em *models*. São eles, passagem de parâmetros para o *model* a fim de executar dada tarefa, leitura do retorno de uma coleção de dados de um *model* para apresentação na *view*, inferência de mensagens de notificação com base no resultado de alguma operação efetuada pelo *model* e redirecionamentos e encaminhamentos para outro *controller* em dada condição.

Normalmente um *controller* é uma composição desses quatro comportamentos. Estas quatro operações comuns são caracterizadas pelos seguintes exemplos:
Dados do *request* sendo passados para a criação de um novo registro no banco através de um *model*;

```
public function createActiton()
{
    Cadastro::create($this->getRequest()->getParams());
}
```

Figura 3.1 - Parâmetro de *request* sendo consumido.
Fonte: O autor.

Model sendo consultado para exibir uma coleção de dados na *view* para uma listagem (também comumente para edição em um formulário)

```
public function createActiton()
{
    $this->view->cadastros = Cadastro::findAll();
}
```

Figura 3.2 - Formato comum de uma *action* exibição de dados de um *model*.
Fonte: O autor.

Flash messenger [1]sendo disparado para notificação de um parâmetro faltante ou sucesso de um processo;

```
public function createAction()
{
    $params = $this->getRequest()->getParams();
    if(Cadastro::create($params)) {
        FlashMessenger::addMessenge(
                'Cadastro criado');
    } else {
        FlashMessenger::addMessenge(
                'Cadastro com problema');
        FlashMessenger::addMessenges(
                Cadastro::getLastErrors());
    }
}
```

Figura 3.3 - *Flash* sendo disparado para cada evento resultante do cadastro.
Fonte: autor.

Redirecionamentos e encaminhamentos acontecem quando um evento de sucesso conclui um *workflow* retornando para o estado inicial, ou voltando para o ponto anterior quando uma etapa ocorre incorretamente.

```
public function createActiton()
{
    $params = $this->getRequest()->getParams();
    if(!$params) {
        $this->_redirect('/selecao-cliente');
    }
    if(!$errorMsg = Cadastro::create()) {
        $this->_redirect('/listagem');
    }

}
```

Figura 3.4 - Action usando redirecionamentos.
Fonte: autor.

Estes quatro comportamentos comuns de um *controller* serão os balizadores do que a solução de testes deve endereçar. Uma solução satisfatória deve viabilizar testes unitários para cada uma destas situações, incluindo todas juntas.

[1] *Flash Messenger* é um *pattern* usado para acumular em uma fila todas as mensagens geradas pelo programa. As mensagens podem ser geradas em *request*s separados e apresentadas apenas quando existe algum retorno para o usuário de uma vez sem perder mensagens no processo. Endereça principalmente um problema que ocorre com o redirecionamento.

Testes de integração são comuns para testar tais comportamentos. Contudo, dependendo como o teste de integração é executado ele pode, por exemplo, inferir o *router*. Alterações no *router*[2] podem levar o fluxo de execução para outro *controller*, o que resultaria em um falso negativo. A mesma situação poderia levar a falsos positivos.

Finalmente a avaliação do resultado do *flash* por uma leitura da saída de tela pode ser imprevisível dada à internacionalização do programa. O teste esperado na tela poderia estar em outra língua quando um sistema de internacionalização fosse aplicado ao sistema fazendo o teste falhar e indicando um falso negativo.

Ter testes unitários implementados a um *controller* tornaria o desenvolvimento mais ágil, uma vez que o desenvolvedor pode deixar claro o papel do mesmo no processamento da requisição. Hoje esse papel é discutido e bastante evitado pela não cobertura dos testes unitários.

Testes unitários de *controller* também apoiam o modelo de trabalho proposto pelo TDD. Assim, mesmo implementações menores, que não dependem de uma grande arquitetura, poderiam ser cobertas pelo método de trabalho. Implementações pequenas, porém complexas que podem ocorrer mesmo no *controller* ficam descobertas de teste unitário por uma incapacidade prática de aplicar testes ao *controller*.

Para tanto, este trabalho busca adaptar o *front controller* a fim se simplificar o processo de testes unitários do *controller*. Em um modelo similar ao que Fowler descreve que foi feito com *front controller*, ao ser dividido em *web handler* e *dispatcher*, uma adaptação do *dispatcher* poderia eliminar a obrigatoriedade de testes de integração para testar o *controller*, da forma que seria a responsabilidade de um teste unitário.

Dois problemas básicos inviabilizam a implementação de testes unitários para *controllers*. O primeiro deles é que os *frameworks* costumam supor um *request*, o qual o *contoller action* teria acesso internamente. Por isso o método *action* não recebe parâmetros, uma vez que todas as suas informações estão no *reques*t em um atributo interno da classe.

[2] *Router* é um *pattern* implementado junto do *webhandler* que altera o comportamento padrão de qual padrão de *request* vai levar a qual *controller*

O segundo problema aparece quando o método *action* não supõe nenhuma saída. Comumente os *frameworks* usam saídas para a camada *view*, por atributos locais do *controller* ou acessos locais a classe de *view*. Isso implica na ausência de retorno do método.

Estes dois fatores tornam virtualmente inviável testar unitariamente o *controller* sem rodar o *dispatcher* informando um *request*, e virtualmente impossível observar o resultado esperado, sem para isso, olhar para o resultado na camada de apresentação ou resultado em banco de dados como é comum acontecer em testes de integração.

3.1 Os frameworks Analisados

3.1.1 Zend Framework (PHP)

Apesar de contar com a coloração externa o Zend *framework* é construído pela Zend, empresa que desenvolveu o *script engine* do php, *zend engine*. A implementação do Zend *framework* é fortemente baseada nos *patterns* assim como descritos por Fowler (2002). Na própria documentação do *framework* é possível encontrar diversas referências para os *patterns* de Fowler (2002).

Dessa forma o *framework* da Zend implementa o *front controller* de forma muito semelhante à literatura. O que pode ser observado neste *framework* é que o *dispatcher* não chama o método *action* do *controller* diretamente. Em vez disso ele chama um método genérico chamado *dispatch,* contido na classe abstrata de *controller*, o *ControllerAction*. Este *ControllerAction::dispatch()* por sua vez chama a *action* correspondente ao *request*.

Este *framework* também implementa uma camada de rotas que participa na decisão de qual a *action* a ser executada para dado *request.*Antes de disparar a *action* o *dispatcher* verifica qual a *action* que combina com o conjunto de regras definidas no *router*. Caso nenhum regra seja encontrada ele assume a regra padrão. A regra padrão do Zend *framework* de roteamento é `/<controller>/<action>`.

3.1.2 Rails (Ruby)

Diferentemente do anterior Rails é mantido por um grupo de pessoas e não por uma empresa, apesar de ter uma relação intima com 37Signals, já que alguns desenvolvedores trabalham nessa empresa.

Criado por David Heinemeier Hansson em 2003, Rails é responsável pela popularidade da linguagem Ruby. O lema do Rails é priorizar o programador e sua satisfação enquanto mantém a produtividade, permitindo que o mesmo escreva um código bonito por favorecer a convenção sobre a configuração.

Rails usa um módulo externo para compor parte do seu *front controller*. De fato a própria palavra *front* não aparece no *framework*. Rail implementa o *dispatcher* enquanto recebe o *request* do módulo *Rack*, que também é usado para dispor o *response* para o cliente.

Dessa forma o Rails mantém a arquitetura de *router*, *dispatcher* e *controller action*, que na sua implementação chama-se `ApplicationController`, em um formato muito semelhante ao da Zend e por sua vez muito semelhante ao que Fowler (2002) descreve;

3.1.3 Sinatra (Ruby)

Sinatra é outro *framework* para Ruby. Ele foi desenhado e desenvolvido por Blake Mizerany na Califórnia. O foco do Sinatra é bastante diferente dos anteriores. Ele busca ser minimalista na sua própria implementação e dessa forma também simplificar a implementação da aplicação.

Assim como Rails, o Sinatra usa *Rack*, o módulo externo, para gerenciar as requisições, fazendo o papel de *web handler*. Contudo, a implementação do *dispatcher* é ligeiramente diferente.

Primeiramente o Sinatra não usa uma classe *controller* e métodos para *actions*. Ao invés disso, métodos do *framework* são chamados para definir as regras de roteamento, passando-se por parâmetros, o código a ser executado quando dada regra de roteamento é atingida. Isso quer disser que na sua implementação o *dispatcher* vem antes do *router*. Assim como é o próprio *router* que executa o código que está contido em si próprio.

3.2 A entrada de dados do Controller

Zend *Framework* (PHP), Rails (Ruby) e Sinatra (Ruby), nenhum destes *frameworks web* recebem o *request* por parâmetro no método. Todos eles têm acesso a um atributo ou método, local ou global, que contem o objeto de *request*.

Isso quer dizer que nesses *frameworks* não é possível executar um *controller* sem passar pela infraestrutura do *framework.* Não é possível testar um *controller* sem rodar o *dispatcher* do *framework* ou gerar um *request*.

Na realidade não é possível nem instanciar um *controller*. A implementação do Zend *Framwork* recebe como parâmetro as classes de interface do *framework* para o *request* e o *response*. Isso diverge do encontrado no exemplo do livro do Fowler (2002). Em seu livro essas interfaces são esperadas no método *init* da classe *FrontCommand* (neste trabalho chamada de *controller action,* assim como identificada nas implementações analisadas).

Isso tornaria o processo de adaptação para o *controller action* mais simples. Contudo ao identificar essa situação fica evidente que alterações mais profundas, até mesmo no *front controller*, em alguns *frameworks* é inevitável, para permitir um *controller* livre de contexto, livre de *request* e *response*.

A proposta para entrada de dados do *controller* é viabilizar o instanciamento da classe de *controller* da maneira mais pura e assim tornar possível que o método (*action*) desse *controller* seja chamado diretamente. Dessa forma, viabilizando a passagem dos dados do *request* por parâmetro.

3.3 A saída de dados do Controller

Da mesma forma que identificando na entrada de parâmetros, todos os *frameworks* analisados compartilham no modelo de saída de dados do *controller*. Tanto Zend *Framework* (PHP), Rails (Ruby) quanto Sinatra (Ruby) geram seus retornos para o *view,* setando um atributo local. Eles também compartilham o modelo de *forwarding* e *redirecting* e o modelo de *flash.*

Adaptar o retorno de dados para *view* através do retorno do método não seria um grande problema usando um *array* para isso. Contudo, existem mais três tipos de saídas que comumente precisam ser testadas.

A proposta para contornar o *flash*, o *forward* e *redirect* é convencionar posições específicas no *array* para este fim. Sendo assim, o retorno do método deve ter uma chave específica para conter o parâmetro do método *flash* e uma chave específica para conter o parâmetro do método *redirect*. O efeito colateral dessa convenção seria a perda de flexibilidade oferecida nesses dois componentes e a dependência direta do *controller action* para com ambos componentes.

Ter um retorno em *array* tornaria o teste evidente e direto. Para testar se dado *controller* estava gerando certa mensagem usando o *flash* bastaria verificar se o *array* de retorno contém a mensagem esperada na posição "*flash"*. O mesmo modelo funcionaria para o *redirect*.

Para que essa funcionalidade fosse acrescida seria necessário modificar o *controller action* de maneira que o mesmo aguardasse um retorno no formato de um *array*. Esse *array* deve ser avaliado em busca da chave indicativa de um *flash* e de um *redirect*. No caso de encontrar uma dessas chaves o método relacionado deveria ser executado passando como parâmetro o valor contido na chave do *array*.

Para as demais posições do *array* duas abordagens poderiam seguir. Por convenção, todas as demais chaves do *array* poderiam ser consideradas parâmetros a serem passados para *view* e assim serem tratadas. A segunda abordagem seria definir uma chave especifica para o retorno destinado a *view*, assim como no modelo do *flash* e *redirect.*

4 A PROPOSTA

Para este conjunto de situações onde a maioria (quando não todos) os *frameworks* deste trabalho aderem, existirá uma implementação equivalente para endereçar a funcionalidade. Cada uma das funcionalidades apresentadas no capítulo anterior será abordada neste capítulo, com o intuito de apresentar uma proposta individual que irá compor a proposta.

Seguindo o *mantra* do TDD, o primeiro passo é imaginar como seria o código final que gostaríamos de ter. A figura 3.5 ilustra um *controller* com duas *actions*. Uma *action* de listagem de clientes e uma *action* de criação de um novo cliente. Ambas recebem a entrada através do parâmetro do método e retornam um *array* identificado contendo as saídas desejadas para a *action*.

```
public function createAction($params)
{
    $options = array();
    if($errorMsg = Cliente::create($params)) {
        $options['flash'] = 'Cliente criado com sucesso';
        $options['redirect'] = 'index';
    } else {
        $view['flash'] = $errorMsg;
    }

    return $options;
}


public function indexAction($params)
{
    return array('data' => Cliente::findAll());
}
```

Figura 4.1 - Exemplo de implementação proposta.
Fonte. O autor.

4.1 Entrada

Desenvolver um *dispatcher* que forneça os parâmetros do *request* como um parâmetro do *controller*. Esses parâmetros podem ser fornecidos em um *array* ou qualquer outra estrutura adequada para linguagem usada.

Para endereçar a necessidade – funcionalidade – de entrada de dados para o *controller* o método passa a receber um parâmetro que contém as informações do *request* (*post* e *get*). Esse parâmetro será um *array* identificado (*hashmap*) de chave e valor.

O *array* de parâmetros seria composto de todas as entradas definidas no *request* além das entradas padrão que são extraídas para montar a rota, como nome do *controller* e nome da *action*. Esse modelo diminuiria a dependência do *controller* para com a classe de *Request* e de *Response*. Dessa forma possibilitando que um teste criasse uma instância da classe *Controller* e chamasse seu método *Action*, sem necessariamente criar um *Request*, *Response* ou *Dispatcher*.

O *array* deve ser um *array* identificado, permitindo assim a associação do seu conteúdo a um *hash*, assim como normalmente é permitido pelo *router*. A percepção da aplicação em relação à mudança ainda seria positiva no sentido de eliminar a linha de código onde essa informação é reavida do *request*. Processo bem comum e repetitivo que costuma acontecer ao criar um novo objeto a partir de um *post* de um formulário.

Para o exemplo da listagem de cliente o *array* conteria as posições "*controller*" e "*action*" com os respectivos valores "cliente" e "*index*", onde nessa situação hipotética a URL requisitada seria "http://www.exemplo.com/cliente/". Mesmo não existindo um *action* explicito, será assumido o comportamento padrão entre os *framewokrs* de assumir "*index*" para "*action*" na ausência de parâmetro, assim como para "*controller*".

Apesar de simples, já é possível perceber algumas ausências. Nesse modelo, onde o *request* não é um objeto, o mesmo não pode compor uma gama de métodos pertinentes ao *request* como dados do cabeçalho da requisição.

Resolver esse problema não é a prioridade deste documento, mas é possível criar um modelo de *request singleton*. Desta forma o desenvolvedor pode rapidamente lançar mão das informações e métodos comuns a uma requisição.

Como exemplo de uma situação onde isso poderia ser útil, podemos imaginar uma dada página que mostra um conteúdo específico para usuários de uma determinada rede. Sendo assim o programa deve avaliar o IP de onde está partindo a requisição. Essa informação não apareceria no `$request`, mas poderia ser acessada usando a classe de *request* do *framework* para destravar todas essas informações e finalmente executar a condicional. Excetuando-se essa pequena particularidade, a interface é e deve ser simples.

4.2 Saída

Na outra ponta do *controller* existem as saídas. A variabilidade de funções disponíveis torna a saída a parte mais complexa, mas nem por isso esta deve ser mais difícil de ser utilizada. Dentro da abordagem proposta o tratamento da saída exigiria uma convenção de *hashes* específicos do *array* de retorno, significando as operações comuns a serem executadas dentro do *controller*.

Essa convenção poderia ser formada pela associação de cada *hash* ao evento esperado, e seu conteúdo ao parâmetro do evento. Por exemplo, a posição `redirect` do *array* de retorno indicaria que um *redirect* deveria acontecer para o dado parâmetro contido na posição. Dessa forma, para tratar os eventos mais comuns dentro de um *controller* são propostos três *hashes* diferentes.

4.2.1 Flash

O `flash é` responsável por setar a mensagem na fila de mensagens. Esse *hash* poderia aceitar tanto *strings*, arrays quando apenas uma mensagem é definida assim suportando mensagens múltiplas.

O *flash messenger* é um *helper*[3] comum entre os *frameworks*. Ele proporciona o gerenciamento de uma fila de mensagens. Essa fila de mensagens tem a finalidade de notificar o usuário dos eventos da aplicação. Exemplos de uso do *helper* incluem notificação de um campo errado em um formulário, operação incorreta ou sucesso em um processo.

[3] Helpers são conjuntos de código, normalmente plugáveis, que adicionam funcionalidades sem interferir no framework. São usados exclusivamente para programas que não contenham regras de negócio.

Algumas implementações do *flash* possibilitam a informação de um marcador que pode ser usado para sinalizar a apresentação da mensagem. Essa funcionalidade é usada para expressar o tipo da mensagem, entre erro, notificação, informação e sucesso, podendo ainda ser usados outros tipos.

A implementação do *hash* flash do *array* de retorno deverá aceitar dois tipos de dados. O primeiro sendo *string* e o segundo *array*. Caso o primeiro método seja optado a *string* será considerada um flash `warning` e será colocada na pilha como tal.

Para todos os efeitos, uma string como parâmetro deve ter o mesmo efeito que um *array* de uma posição com o *hash* `warning` contendo a mesma *string*. Seguindo a mesma lógica, deverá ser possível adicionar múltiplas posições no *array* para múltiplas mensagens, como demonstra a figura 4.2.

```
1
2   <?php
3
4   //forma string
5   $options['flash'] = 'Erro ao processar';
6   //equivalente
7   $options['flash'] = array('error' => 'Erro ao processar');
8
9   //forma array
10  $options['flash'] = array(
11      'notice' => 'Dados ainda não salvos',
12      'xxx' => '$%%%$',
13      'error' => 'Houve um erro ao inserir',
14  );
15
```

Figura 4.2 - Uso do retorno *flash*
Fonte: autor.

4.2.2 Redirect e Forward

O *redirect* e o *forward* para endereçar as necessidades de redirecionamento de fluxo de aplicação. Ambos aceitariam uma *string* única como parâmetro. Este par de *hashes* deve ser mutuamente exclusivo. Caso ambos sejam setados no mesmo retorno uma exceção deve ser gerada.

Assim como o *flash* este controle de fluxo é um *helper* comum entre os *frameworks*. E apesar da função similar, *redirect* e *forward* são sensivelmente diferentes. Apenas o Zend *Framework* implementa o *forward* diretamente. Nos demais é possível obter a mesma funcionalidade através de chamadas internas dos *frameworks*.

O *redirect* tem a função de interromper a execução do programa gerando uma saída HTTP com cabeçalho de redirecionamento temporário. Essa funcionalidade serve para mover a requisição para o ponto certo em um fluxo de trabalho. Como em uma precedência, para editar um dado, o usuário deve se dirigir a página de seleção de registro para então retornar a página de edição. O *redirect*, em um caso de uso como o citado pode ser acompanhado por um *flash* indicando porque o *redirect* aconteceu.

Diferentemente, o *forward* não finaliza a requisição. Em vez disso ele invoca um novo comando a ser executado na pilha de execução do *dispatcher*. Essa funcionalidade ocorre quando é necessário encadear uma série de eventos sem onerar o cliente com respostas desnecessárias.

Um caso de uso possível para essa funcionalidade é a ausência de permissão para acessar uma página. Após o usuário tentar acessar determinada URL, a qual o mesmo não tem acesso, é apresentado uma página de *login*. Caso o mesmo efetue corretamente o *login*, seus dados serão processados pelo *controller* de autenticação e então este usuário estará apto a ser encaminhado para a página que desejava acessar inicialmente.

Para implementação, ambos os *helpers* usarão uma entrada em *array* de dois parâmetros. Sendo o primeiro a *action* de destino e o segundo o *controller*. Sendo que o segundo parâmetro pode ser omitido. Neste caso o *controller* em execução será utilizado. O *redirect* aceitará um método de retorno alternativo em *string* para expressar páginas fora do domínio da aplicação. Como demonstra a figura 4.3.

No caso de ambos os recursos serem acionados no mesmo *controller*, o `redirect` terá prioridade sobre o `forward`. Sendo assim o *hash* `redirect`, na implementação, deve ser avaliado primeiro, ignorando o resto do *array* e finalizando a execução do fluxo. Para a implementação é sugerido que o *hash* `redirect` seja acompanhado de *return* diretamente após.

```php
<?php

//forma URL
$options['redirect'] = 'cesarscur.com';

//Forma normal redirect
$options['redirect'] = array('index', 'auth');

//Forma normal forward
$options['forward'] = array('index', 'cadastro');

```

Figura 4.3 - Implementação *forward* e *redirect*.
Fonte: autor.

4.2.3 Data ou View

O *data* ou *view* responsável por indicar qual a informação deve ser passada para a camada de visualização, sendo seu parâmetro de qualquer tipo. Seria recomendado que fosse utilizado o tipo *array* para organizar as várias informações a serrem passadas para *view*.

Para resolver a funcionalidade de enviar dados para a *view*. Onde o Sinatra usa a passagem de parâmetro na chamada do renderizador do *template*. Onde o Zend *Framework* utiliza o atributo `view` local. Onde o Rails utiliza o retono do método. O modelo implementado utilizará o *hash* `view` ou `data`.

A escolha redundante de duas palavras para essa finalidade é pensada para resolver uma possível colisão de nomes. Apenas uma das palavras chave será utilizada efetivamente na implementação, dependendo da disponibilidade. O nome *view* será preferido no caso de as duas palavras estarem “vagas”.

```php

<?php

$options['view'] = array(
    'cliente' => $cliente,
    'total' => 3020.65,
    'listaValores' => array(...),
);


//caso a palavra view seja reservada
$options['data'] = array(
    'cliente' => $cliente,
    'total' => 3020.65,
    'listaValores' => array(...),
);

```

Figura 4.4 - Implementação da saída para *view*.
Fonte: autor.

4.3 Entradas e Saídas

Uma vez tendo seguido o mantra do TDD, tendo imaginado a interface que o programa deve ter, tem-se as linhas guias daquilo que deve ser implementado. Um conjunto de funcionalidades inerente a cada tópico de entrada e saída é evidente.

Neste modelo haveria um programa setando um retorno em *array* e indicando na posição pré convencionada a informação necessária para que o retorno seja efetuado. A combinação destes tipos de retorno seria livre, embora alguns possam ser mutuamente exclusivos ou em alguns casos alguns dados poderiam ser ignorados.

Há exemplo dos casos de ambiguidade: `forward` contra `rediret` e `data` contra `view`. Para ambos os casos um regra de "desempate" é prevista. Onde `redirect` tem precedência sobre `forward` e `view` tem precedência sobre `data` pelos motivos e critérios mencionados.

Diferentemente do que acontece quando temos um *redirect* – onde a execução é interrompida imediatamente e o retorno é despachado para o navegador – o retorno só seria despachado após o fim da execução do *controller*. Sendo assim, para obter um

comportamento igual, o marcador *redirect* deve ser seguido de *return*. O que não é uma má prática e é encorajado para implementações usando este modelo.

4.3.1 Possíveis extensões

Estendendo ainda mais essa ideia, os *hashes* poderiam ser associados dinamicamente. Serem configurados e permitindo a adição dinâmica de novos eventos a serem disparados por novos *hashes* executando métodos ou classes de *callback.* Seria possível trazer para o mundo do usuário a possibilidade de setar os *hashes* usados no *controller*.

Seria possível prever um nível de extensão para as saídas deste modelo da seguinte forma: Uma interface de pilha da classe que chama o *controller* seria capaz de receber pares de nomes (*string*) e classes. Uma vez a pilha de *hashes* formada pelo usuário o programa avaliaria o retorno do comando procurando um dos *hashes* da pilha. No caso de encontro do valor acordado no retorno seria passado a classe associada ao *hashe*, possibilitando uma flexibilidade muito grande de funções. Entretanto, este não será o modelo implementado inicialmente.

5 PROTÓTIPO

O modelo proposto no capítulo anterior visa resolver a capacidade de se testar unitariamente um *controller*. Este objetivo é abordado gerando-se saídas e entradas desacopladas do resto do *framework*. O resultado esperado da implementação desse modelo é obter um ambiente onde um *controller* possa ser implementado e testado. Ou seja, um processo de desenvolvimento mais limpo e ágil, que por consequência são os efeito esperados de tal implementação.

Durante as fases de análise de código dos *frameworks*, era possível visualizar os pontos de contato onde a alteração deveria ser efetuada para obter o comportamento descrito na proposta. Seguindo também o *design front controller* era presumível que esse ponto de contato fosse o *dispatcher*.

O *dispatcher* é o responsável pela interação entre o comando e a requisição. Contudo a tentativa inicial de implementação seguindo o modelo se mostrou muito mais complicada. Complicada pelo mesmo motivo que motivou este trabalho. Pela complexidade e acoplamento entre *request*, *front controller*, *dispatcher* e *controller action*.

No Zend *framework* a interdependência entre esses componentes é bastante grande. A requisição é passada para o *controller action* e não para o comando. E este evento ocorre dentro do *dispatcher*. A requisição é gerada dentro do *front controller* que o passa ao *dispatcher*.

Apesar deste nível de acoplamento intrínseco, não é possível considerar isto como uma falha de design, ou falta de flexibilidade do programa. Ele simplesmente não foi pensado para tal uso. Esta ordem de eventos faz bastante sentido quando pondera-se sobre as atribuições do *front controller* como um todo.

Infelizmente este contexto torna o Zend *framwork*, e talvez os demais *frameworks*, maus candidatos à implementação da proposta. A complexidade desta implementação para fins de averiguação da verdade de que tal modelo melhora o desenvolvimento é não pratico.

Contudo, uma solução, em caráter de protótipo pode ser estipulada construindo-se o modelo uma camada acima do comando. Nesta implementação funcional, o design do modelo

estaria posicionado erroneamente. Porém para o comando executado poderia ser imperceptível para o *framework*.

Em outras palavras, uma camada, construída no comando, poderia chamar um pseudo comando. Essa camada serviria de interface, entre o que é esperado que o modelo seja capaz de entregar e o *framework* funcional.

5.1 Uma segunda proposta

A partir de então uma nova proposta deve ser elaborada. Uma proposta que visa implementar tudo aquilo que é apresentado na proposta inicial . Mas, além disso, contornar a complexidade da proposta inicial onde uma grande parte do *framework* usado como base teria de ser modificada devido ao acoplamento.

Não sendo esta a proposta deste trabalho a abordagem mais coerente, portanto, é propor uma alternativa como a introdução deste capítulo sugere. Uma camada intermediaria implementada com o único intuito de interfacear as entradas do *framework* com as entradas do protótipo.

Para isso essa interface deve ser transparente tanto para o *framework* quanto para o protótipo, além de manter todas as características da proposta inicial. Assim como mostra a figura 5.1, uma interface implementada dentro do *controller* (comando) que recebe os parâmetros no formato objeto *request* então chama um método global passando os parâmetros como o indicado na proposta.

```
10
11   get '/user/*' do
12     options = indexAction(params)
13
```

Figura 5.1 - Entrada da interface
Fonte: autor.

Para implementação deste modelo, diferentemente do que planejado inicialmente, a escolha mais acertada parece ser Ruby usando Sinatra. Mesmo que a linguagem a qual o autor tem maior domínio seja outra e o *framework* pelo qual havia maior inclinação inicialmente

fosse Zend, a simplicidade do Sinitra compensa essas vantagens quando não se é necessário alterar a funcionalidade interna do *framework.*

Novamente na figura 5.1 demonstra um exemplo de como essa interface poderia funcionar, chamando um método global passando a entrada como é que entrada seja. No caso do Sinatra a entrada já é um *array*, o que reforça a afirmação de que ele é mais simples uma vez que o conhecimento das internas do *framework* é irrelevante na construção desta inferface.

Neste *framework* a variável `params` existe em todo bloco de código passado como parâmetro para o método *get*. Sendo assim params poderia ser passado diretamente para o *controller*. Além disso, o método `indexAction`, ou como quer que ele chamame-se, pode naturalmente retornar os parâmetros, como previstos na proposta inicial.

5.2 A implementação

Seguindo estas ideias, é possível montar um protótipo. A figura 5.2 mostra como o protótipo de interface pode ser implementado, tratando alguns aspectos previamente ponderados. Sendo este um protótipo então implementado usando o *framework* Sintara, as primeiras linhas do programa são a declaração do *framework.*

Esta implementação já segue a ideia de que o *flash* deve ser o primeiro a ser repassado para o *helper*, uma vez que ele é aplicado mesmo se houver um *redirect*. Para o *flash* uma biblioteca externa do Sinatra foi utilizada, como é possível perceber pela declaração na figrua 5.2.

A terceira etapa da interface é identificar se existe um *redirect*, excuta-lo e interromper a execução dos demais comandos, que agora são irrelevantes. Seguido então da chamada *forward*, que assim como o *redirect*, caso invocado interrompe as execuções subsequentes.

Todas estas etapas são inferidas com base no retorno da chamada do método global, que, no protótipo, representa o conjunto de comandos do *controller*. Por fim é chamado o *template render*, componente responsável por gerar a *view* do programa. Na implementação foi usado um componente tradicional do Ruby conhecido por ERB, ao qual foram passados os dados contidos no retorno sobre a *hash* `view`, como sugeria a proposta.

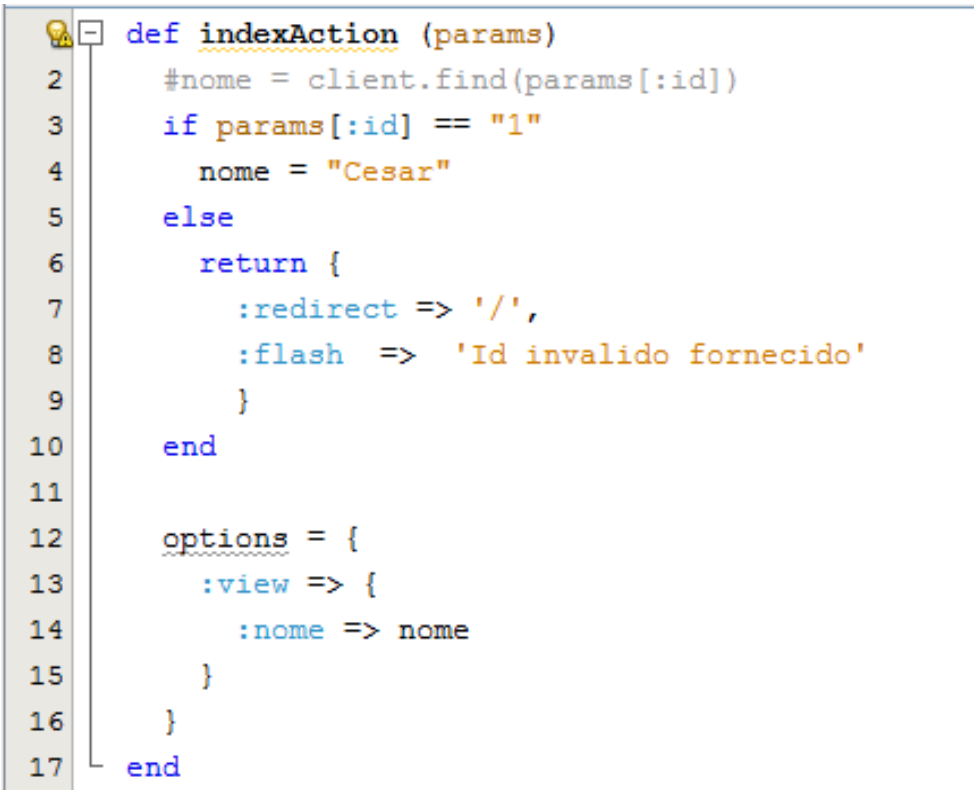
```
1   require 'rubygems'
2   require 'sinatra'
3   require 'sinatra/reloader'
4   require 'sinatra/flash'
5
6   enable :sessions
7
8   get '/user/*' do
9     options = userAction(params)
10
11    if options[:flash]
12      flash[:notice] = options[:flash]
13    end
14
15    if options[:redirect]
16      redirect options[:redirect]
17    end
18
19    if options[:forward]
20      forward options[:forward]
21    end
22
23    erb :view, :locals => options[:view]
24  end
25
```

Figura 5.2 - Protótipo de interface
Fonte: autor.

Na estrutura exibida na figura 5.2 as linhas um a quatro (1-4) são responsáveis pela declaração do *framework*. A linha seis (6) é a configuração de sessão, usada para demonstrar o uso de recursos internos do *framework* não afetados pelo programa. A linha oito (8) é a declaração da rota e o bloco de código que deve ser executado quando a rota é acertada.

O método `get` é interno do *framework* e não do Ruby. Seu propósito é registrar uma rota e um bloco de código. Esse registro de rota no Sinatra é composto e não separado como visto nos demais *frameworks*. A partir deste registro o *front controller* executa o bloco correspondente a requisição.

Com essa montagem de interface é possível observar uma aplicação como mostra a figura 5.3. Sendo este um *controller* plausível e passível de ser encontrado em uma aplicação real, pode-se prosseguir para a modelagem dos casos de uso onde o protótipo será testado.



```
def indexAction (params)
  #nome = client.find(params[:id])
  if params[:id] == "1"
    nome = "Cesar"
  else
    return {
      :redirect => '/',
      :flash  =>  'Id invalido fornecido'
      }
  end

  options = {
    :view => {
      :nome => nome
    }
  }
end
```

Figura 5.3 - Exemplo de *controller* usando o protótipo
Fonte: autor.

Este *controller* pode facilmente ser testado com o teste unitário apresentado na figura 5.4. Um *array* é inicializado com os dados que dada entrada deve retornar. Assim caso a entrada seja a correta o retorno deve ser o mesmo e a asserção retorna positiva, do contrário, caso a entrada seja incorreta o resultado esperado deve ser outro, então a asserção negada retorna verdadeiro mais uma vez.

```
3
4   options = {
5     :view => {
6       :nome => "Cesar"
7     }
8   }
9
10
11  result = indexAction({:id => "1"})
12  puts "id1 = Cesar"
13  puts (result.eql? options)
14
15
16  result = indexAction({:id => "2"})
17  puts "id2 != Cesar"
18  puts (!result.eql?  options)
19
```

Figura 5.4 - Exemplo de teste de *controller*
Fonte: autor.

5.3 Casos de uso

Como metodologia de comprovação de que o protótipo melhora a implementação e torna o teste unitário viável e simples será usado um caso de uso para cada operação primária de um sistema CRUD. Serão criados casos de uso para criação de um registro, a listagem dos registros de uma tabela, a alteração de um registro e a exclusão de um registro da mesma tabela.

Seguindo, mais uma vez o mantra do TDD, para cada *controller* do CRUD, foi, antes, implementado um teste unitário que acerte este *controller*. Seguido do *controller* em si, então, novamente um novo teste até que todos os quatro *controllers* foram implementados.

Além disto, o modelo CRUD pressupõem um acesso a base de dados. Mas sendo um CRUD genérico e hipotético esta funcionalidade foi indicada nos programas, mas comentada. Apesar de ficar claro como o programa usaria o acesso ao banco para realizar uma requisição, dados estáticos são usados no lugar.

Uma estratégia similar a encontrada em alguns mapeadores objeto-relacional, foi usada. Essa estratégia é conhecida como *fixtures.* Onde dados estáticos são usados para simular uma aplicação e rodar os testes sem a necessidade de dados quentes de produção.

Para todos os fins esta medida não deve influenciar nos resultados. Sua aplicação é com intuito de simplificar e agilizar o processo de desenvolvimento. Além disso, o resultado é totalmente funcional. Uma aplicação funcional, porém estática.

5.3.1 Create

Como detalhado anteriormente, para cada implementação, um teste foi desenvolvido, seguido do *controller*. Além disso, nenhum dado é efetivamente criado neste *controller*. Apesar de que as chamadas de função apropriadas para tal estão claramente descritas e comentadas.

A figura 5.6 demonstra como um teste que testa um caso de uso, onde se espera dois tipos de resposta para dois tipos de entradas. Ou de outra forma, uma entrada correta e uma entrada de erro. O teste considera uma entrada correta quando ambos os campos da requisição estão preenchidos com valores diferentes de vazio.

Sendo os valores usados no teste “Cesar” para o parâmetro “nome” e “26” para o parâmetro “idade”. Para entradas preenchidas o retorno esperado um *flash* indicando o sucesso na inserção e um *redirect* para a página de listagem do conteúdo.

A mensagem de sucesso do flash usada como teste é “Registro inserido com sucesso!”. A *string* de saída deve ser exatamente esta para que o programa seja válido. E o parâmetro de retorno *redirect* contém “*/read*”. Para a entrada invalida, com os campos vazios, o retorno esperado é um *flash* com a *string* “Os campos nome e idade são obrigatórios!”.

```
def createAction (params)
  if params[:nome] != "" and params[:idade] != ""
    #inserção na base de dados
    options = {
      :flash => 'Registro incsrido com sucesso!',
      :redirect => '/read'
    }
  else
    options = {
      :flash  =>  'Os campos nome e idade são obrigatórios!'
    }
  end

  options
end
```

Figura 5.5 - *Controller* de *create*
Fonte: Autor.

Seguindo o processo de implementação e tendo o teste implementado, o *controller* que implementa as mesmas funcionalidades é o apresentado na figura 5.5. Neste *controller* tem-se a condicional que valida a o preenchimento dos campos de entrada. Os parâmetros do *request* devem ser diferentes de *string* vazio.

Caso esse critério seja atendido uma chamada ao *model* responsável por inserir o dado no banco seria chamado como apresentado na linha 22 em uma chamada comentada. Esta chamada não é pertinente para o teste, mas o comentário foi colocado para ilustrar onde estaria essa chamada em um programa real.

Seguindo na condição verdadeira existe a atribuição dos valores a variável *options* que recebe os valores de *flash* e *redirect* como os apontados no teste, quando ocorre uma inserção de sucesso. Já na condicional negativa, tem-se a atribuição da mesma variável *options* com os dados pertinentes e já definidos no teste para o caso de retorno para uma entrada invalida.

```
1   requre 'controller'
2
3   options = {
4     :flash => 'Registro incsrido com sucesso!',
5     :redirect => '/read'
6   }
7
8   puts "Registro criado = true"
9   result = createAction({:nome => "Cesar", :idade => 26})
10  puts (result.eql? options)
11
12
13  options = {
14    :flash => 'Os campos nome e idade são obrigatórios!',
15  }
16
17  puts "Campo não preenchido = true"
18  result = createAction({:nome => '', :idade => ''})
19  puts (!result.eql?  options)
20
```

Figura 5.6 - Teste de *create*
Fonte: autor.

5.3.2 Read

Para a próxima etapa no sistema simples proposto, CRUD, tem-se a listagem dos valores da base de dados. Nesse caso a figura 5.8 ilustra o teste implementado para verificar se o retorno do *controller* condiz com o esperado.

Foi utilizado um valor fixo de dados esperado para o teste. Similarmente ao que é utilizado no sistema de *fixtures* para teste de banco, porém sem ter efetivamente um banco rodando na aplicação de testes.

Os dados esperados pelo retorno do *controller*, são de um registro contendo os valores “Cesar” para o campo nome, “26” para o campo “idade” e “1” para o campo “id”. Estes valores foram colocados dentro do marcador *view*, sendo eles os dados enviados para a *view* para serem exibidos em uma lista de uma registros.

```
def readAction (params)
  #pessoa.find_all()
  {
    :view => {
      :nome => 'Cesar',
      :idade => 26,
      :id => 1
    }
  }
end
```

Figura 5.7 - *Controller* de *read*
Fonte: autor.

O teste valida a saída apresentada na figura 5.7. O código apresentado nesta figura ilustra a situação onde um *model* é acessado, reavendo todos os registros, ou um único. Para fins de simplificação o *model* retornaria um único registro.

Neste caso o registro único é colocado diretamente no *array* de retorno. Ou seja, sem a estrutura de registros. Ou sem uma estrutura de *array* em volta dos dados para gerenciar múltiplos registros.

Como é costume no Ruby, não é usada a palavra *return* uma vez que a linguagem automaticamente retorna o resultado da última expressão de uma função. Neste caso sendo o próprio *array*.

```
1  require 'controller'
2
3  options = {
4    :view => {
5      :nome => 'Cesar',
6      :idade => 26,
7      :id => 1
8    }
9  }
10
11 puts "Lista = true"
12 result = readAction()
13 puts (result.eql? options)
```

Figura 5.8 - Teste de *read*
Fonte: autor.

5.3.3 Updade

Seguindo para o passo seguinte na implementação tem-se o *update*. Para este processo do programa a figura 5.10 ilustra uma implementação de teste onde são esperados as seguintes mecânicas:
Alteração com sucesso
Dados de entrada incorretos
ID de referencia invalido

Para cada situação um tratamento especifico é esperado pelo teste portanto: Alterações com sucesso são esperadas quando um dado de entrada é completo, tendo os campos da requisição nome e idade preenchidos com dados. Além disso, é esperado que um ID válido seja fornecido.

Para os dados incorretos, não preenchidos de nome e idade a mensagem do *flash messenger* esperada é de que os campos são obrigatórios. Além disso, nenhum redirecionamento é esperado, uma vez que a página de cadastro ainda deve ser apresentada.

Por fim para uma entrada com ID inválido, o resultado esperado e validado pelo teste é de um redirecionamento para listagem, onde o usuário poderia seleciona um ID correto. Isto agregado de uma mensagem indicando o erro da seleção. Esta situação somente ocorreria por uma falha de sistema ou uso de uma URL incorreta por parte do usuário.

```
def updateAction (params)
  #pessoa.find(params[:id])
  if params[:id] != 1
    options = {
      :flash => 'O id fornecido não é valido',
      :redirect => '/read'
    }
  elseif params[:nome] != "" and params[:idade] != ""
    options = {
      :flash => 'Os campos nome e idade são obrigatórios!',
    }
  else
    #pessoa.update params
    options = {
      :flash => 'Registro atualizado com sucesso!',
      :redirect => '/read'
    }
  end

  options
end
```

Figura 5.9 - *Controller* de *update*
Fonte: autor.

Para este teste tem-se a implementação apresentada na figura 5.9. Neste código, primeiramente é checado para a validade do ID. Em um código normal essa verificação seria feita contra a base de dados. Porém para os efeitos deste trabalho ele é testado contra o valor "1" fixo. Caso este valor seja diferente de "1" então a mensagem de que este é um ID inválido como entrada é retornado.

Subsequentemente, o teste do preenchimento dos dados de entrada é feito. E para um entrada positivo há o redirecionamento para a listagem em conjunto com a mensagem de que o registro foi devidamente inserido. Do contrário, uma mensagem de que os dados estão incorretos e o não redirecionamento da página são fornecidos como retorno.

Ainda para o caso dos dados estarem corretos. É neste momento que os mesmos são atualizados na base. Contudo, nesta aplicação não existe uma base de dados. Portanto temos apenas o pseudocódigo para o evento.

```
1   requre 'controller'
2
3   options = {
4     :flash => 'Registro atualizado com sucesso!',
5     :redirect => '/read'
6   }
7
8   puts "Registro atualizado = true"
9   result = updateAction({:nome => "Cesar", :idade => 31, :id => 1})
10  puts (result.eql? options)
11
12
13  options = {
14    :flash => 'Os campos nome e idade são obrigatórios!',
15  }
16
17  puts "Campo não preenchido = true"
18  result = updateAction({:id => 1, :idade => ''})
19  puts (!result.eql?  options)
20
21
22  options = {
23    :flash => 'O id fornecido não é valido',
24    :redirect => '/read'
25  }
26
27  puts "Id invalodo = true"
28  result = updateAction({:id => 999})
29  puts (!result.eql?  options)
```

Figura 5.10 - Teste de *update*
Fonte: autor.

5.3.4 Delete

Finalmente para o *delete*. A figura 5.12 ilustra o teste para um *controller* que executará a deleção de um registro. Assim como o processo de alteração é fornecido um ID ao qual quer-se deletar. A diferença entre a alteração reside em não haver outros dados de entrada, apenas a chave de acesso ao registro.

Portanto, das saídas esperadas no *update*, uma delas não está presente. Não existe a validação dos campos para preenchimento, ou qualquer outra filtragem em geral. Restando assim uma validação a ser feita.

Validar o fornecimento correto do ID contra a base de dados. Novamente não existindo uma base de dados, foram utilizados dados fixos. Segundo o teste é esperado um ID

igual a “1” para resultar em um sucesso na exclusão, já que o registro existiria na base de dados.

Desta forma o teste espera que para uma entrada como ID igual a “1” uma saída de redirecionamento para a listagem e uma mensagem de sucesso na exclusão são esperados. Do contrário, caso um valor de ID sendo diferente de “1” seja fornecido, o retorno esperado é de invalidade do ID. Redirecionando também para a listagem de onde deveria partir a requisição adequada de deleção.

Assim como no *update*, a situação onde a mensagem de ID inválido é esperada é não convencional. Não deve ocorrer pela operação natural do programa. Esta situação surge com algum problema no programa, ou mais comumente quando o usuário manipula o endereço.

```
def deleteAction (params)
  #pessoa.find(params[:id])
  if params[:id] != '1'
    options = {
      :flash => 'O id fornecido não é valido',
      :redirect => '/read'
    }
  else
    #pessoa.delete params[:id]
    options = {
      :flash => 'Registro removido com sucesso!',
      :redirect => '/read'
    }
  end

  options
end
```

Figura 5.11 - *Controller* de delete
Fonte: autor.

Assim como nos *controllers* anteriores os acessos a banco estão expressos comentados na figura 5.11. Neste caso o programa inicia reavendo os dados do banco, então achando se este dado está disponivel. Contudo, foi implementado para comparar com o valor estático “1” pelos mesmos motivos antes descritos.

Sendo assim, para o caso de uma entrada de ID diferente “1” o *controller* retorna uma mensagem de “O id fornecido não é valido!” e redireciona o usuário de volta para a página de listagem.

Caso contrário, sendo o ID válido (igual a “1” neste caso) o registro é hipotéticamente removido do banco, como mostra a linha 81 do programa. Seguido da saída contendo uma mensagem de sucesso da operação e um redirecionamento também para página de listagem.

```
1   require 'controller'
2
3   options = {
4     :flash => 'Registro removido com sucesso!',
5     :redirect => '/read'
6   }
7
8   puts "Registro removido = true"
9   result = deleteAction({:id => '1'})
10  puts (result.eql? options)
11
12
13  options = {
14    :flash => 'O id fornecido não é valido',
15    :redirect => '/read'
16  }
17
18  puts "Id invalodo = true"
19  result = deleteAction({:id => 999})
20  puts (result.eql?  options)
21
```

Figura 5.12 - Teste de delete
Fonte: autor.

6 ANALISE DOS RESULTADOS

Da maneira ressaltada no capítulo dois e citada por autores como Feathers, teste de *software* é imprescindível para um bom código. Sem teste não se tem qualidade e sem teste não se tem código limpo. Teste é o primeiro passo na metodologia de Kent Back e deve fazer parte do ciclo de desenvolvimento para gerar confiança para o desenvolvedor. Não testar o *controller* dentro deste ciclo é uma lacuna que pode ser preenchida.

O *controller* desempenha um papel central na execução de uma requisição. Ele é responsável por endereçar os dados de entrada e destiná-los aos programas responsáveis por processar estes dados. Não somente isso, mas também obter os resultados e coordenar as respostas a interface de usuário ou programa.

Este papel é fundamental para a boa execução de um sistema e não deve, jamais, ser subestimado em importância. Para tanto, testes unitários garantindo o bom *desing* destes são de grande importância. Não somente garantir o *desing*, mas a funcionalidade percebida diretamente pelo usuário final.

Essa é a importância que este capítulo traz. A convergência desses aspectos e a avaliação dos resultados torna desse capitulo o ápice deste trabalho. Momento em que os resultados produzidos pela execução dos programas e testes são analisados.

Dessa forma, tem-se o momento de rodar os testes implementados no capítulo anterior a avaliar os resultados dos testes e analisar cada teste implementado individualmente quanto ao atingimento das metas.

Além de rodar cada teste, este capítulo se encarregara de validar a funcionalidade deste programa quanto a sua funcionalidade como programa. Ou seja, demonstrar que a interface não interrompe o funcionamento natural do *framework*.

Para a avaliação de cada teste e programa este capítulo seguirá a mesma sequência do capítulo anterior. Cada etapa do CRUD será dividida em programa e teste. Onde respectivamente será avaliada a funcionalidade do programa e os resultados dos testes. Subsequentemente cada teste será avaliado quanto ao atendimento das metas.

6.1 Create

6.1.1 Programa

Comparando os resultados apresentados pela interface (navegador) com o código construído para a *action* de criação é evidente que as funcionalidades foram mantidas intactas. Foi possível implementar, rodar e ver o resultado coerente de uma aplicação, de um formulário de entrada de dados genérico.

Assim como foi possível observar os comportamentos de *feedback* esperado de um formulário de entrada exibindo as saídas pertinentes ao sucesso e a falha na entrada dos dados.

É considerável que os resultados com relação ao programa, que podem ser vistos nas figuras 6.1 e 6.2 sejam os mesmos esperados, em termos de capacidades, ao *framework* antes do incremento da camada proposta.

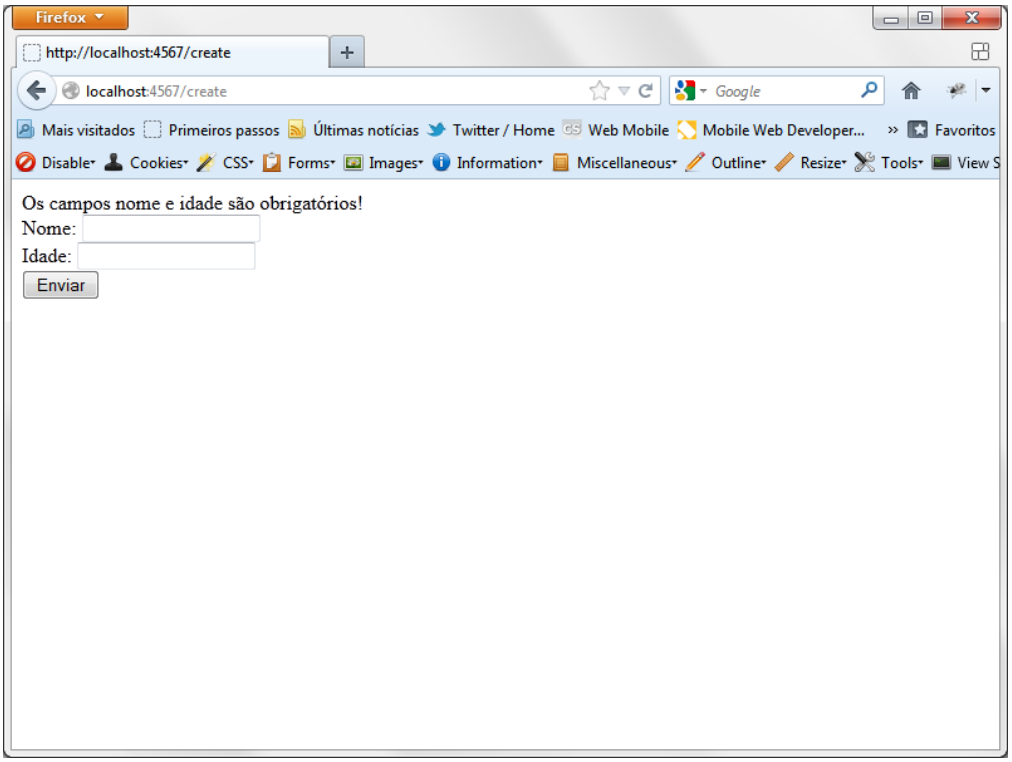


Figura 6.1- Falha na entrada de dados
Fonte: autor.

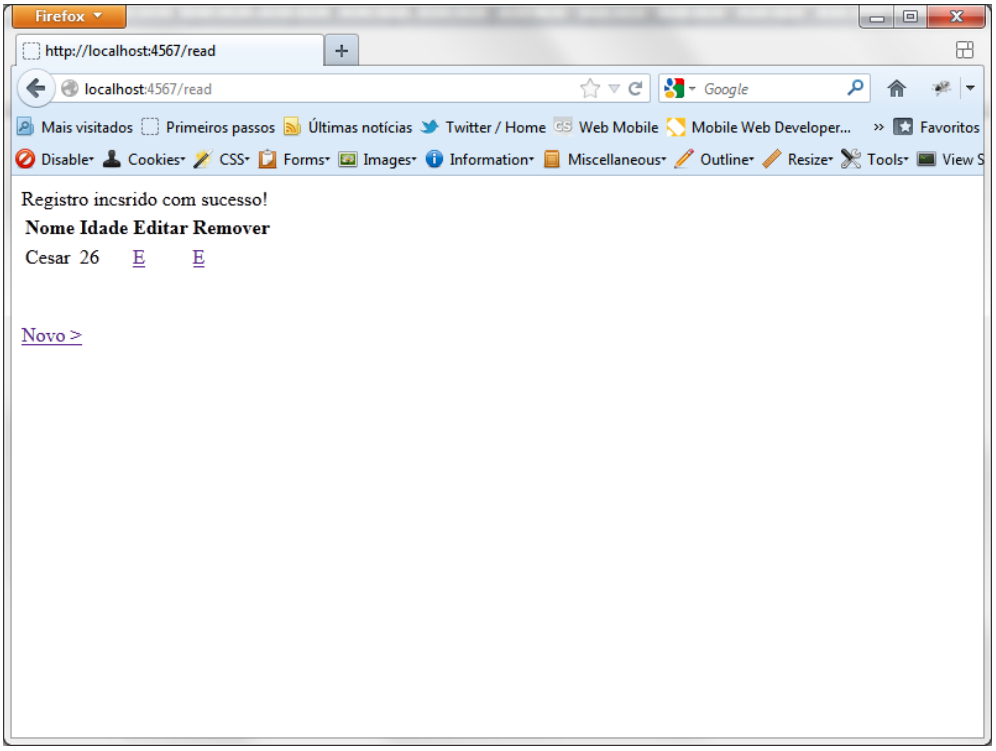


Figura 6.2 - Sucesso na entrada de dados no formulário
Fonte: autor.

6.1.2 Teste

Rodado o teste desenvolvido, pode-se observar o resultado esperado de sucesso na asserção dos mesmos. Ambos os teste desenvolvidos passaram ao executar o mesmo programa usado para apresentar os resultados no navegador. Os testes efetivamente rodaram sem a necessidade da execução de nenhuma outra parte do *framework* e obtiveram o mesmo resultado esperado por um teste unitário, como mostram os resultados apresentados na figura 6.3.

Com relação aos códigos de teste implementados para testar unitariamente a *action* de criação do programa, foi possível avaliar que sua construção foi muito mais curta, menos código foi produzido para chegar ao resultado. Sendo também mais eficaz com relação a um teste integrado, não necessitando a instanciação de nenhuma classe não relacionada com o processo.

Também foi possível avaliar que a implementação do teste manteve o desenvolvedor muito mais focado no resultado esperado pelo *controller* diretamente. Uma vez que não houve distrações com relação ao *framework*. Caso estes problemas aparecessem, brevemente seriam

anotados para serem tratados em um momento posterior, uma vez que não pertenciam a este teste.

Sendo assim, para o teste implementado para avaliar as funcionalidades de inserção com sucesso e falha no preenchimento correto dos dados, os testes foram efetivamente melhores atingindo a meta deste trabalho.

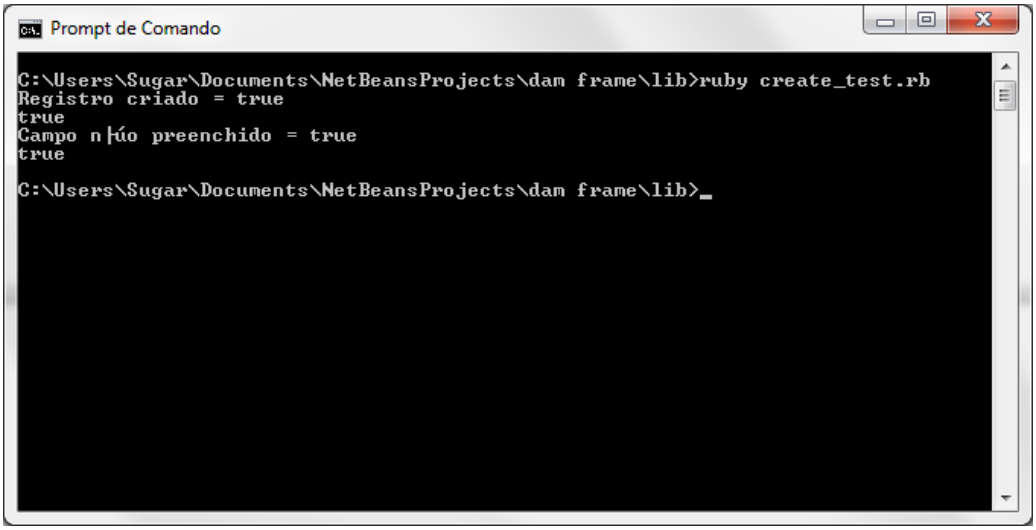


Figura 6.3 - Resultados teste *create*
Fonte: autor.

6.2 Read

6.2.1 Programa

A leitura do *array*, que aqui representa o acesso ao banco e a subsequente apresentação dos dados em tela, acontece normalmente como em qualquer programa. Na figura 6.4 pode-se observa este resultado.

Os dados lidos do *array* foram iterados e listados em um formato de tabela para a apresentação. Cada linha da tabela contém as colunas de dados e os *links* de edição e remoção para o respectivo registro. Bem como um *link* extra para a criação de um novo registro abaixo da tabela de apresentação dos registros.

Este programa consegue apresentar um resultado compatível a qualquer programa implementado sem a camada intermediaria, o que leva ao resultado de sucesso neste aspecto da leitura.

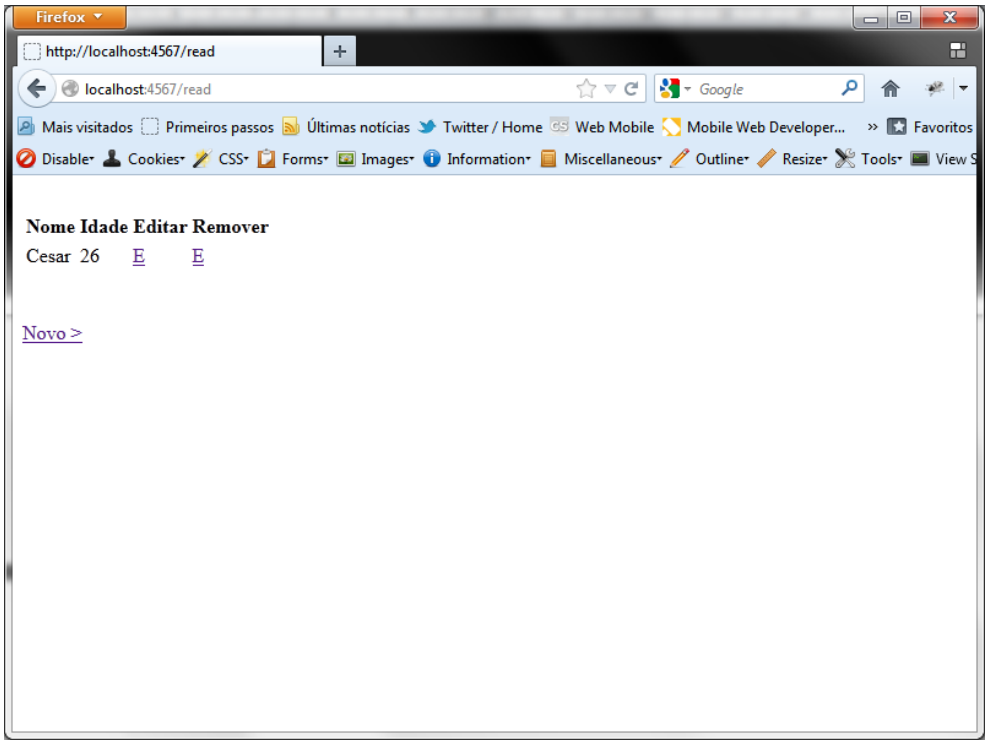


Figura 6.4 - programa de leitura
Fonte: autor.

6.2.2 Teste

Para o teste da leitura é comparada a lista de dados no formato *array* a um *array* igual. Na figura 5.6 pode-se ver este teste passando com sucesso. Isso indica que tanto o *array* de dados (aqui usado para representar um resultado de banco) quanto o *array* de comparação são idênticos.

O sucesso deste teste confirma o que já foi constatado na interface do programa. Que o programa implementado sobre a camada atende o requisito de apresentação. Mas não somente isso ele comprova que o resultado pode ser testado unitariamente.

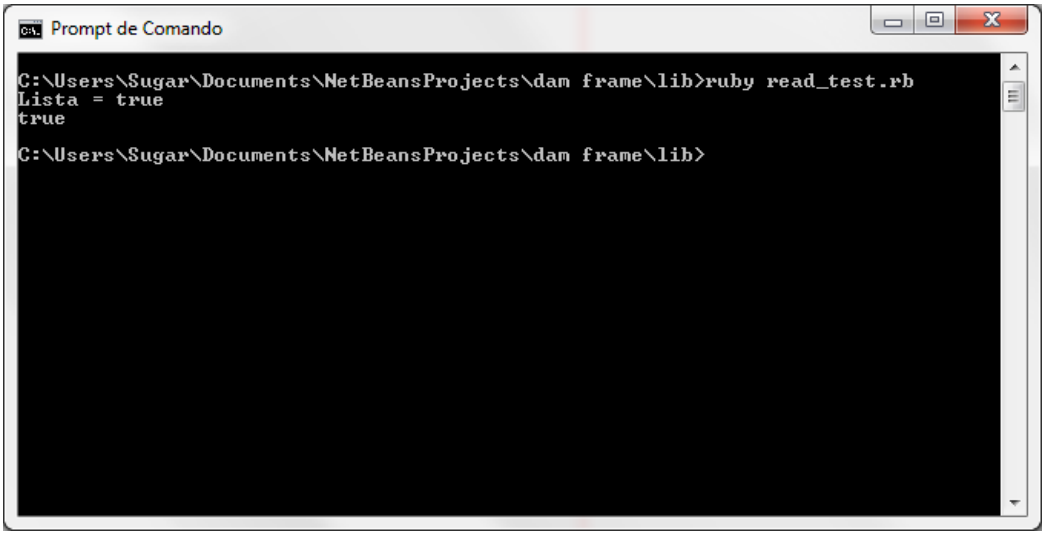


Figura 6.5 - Teste de Leitura
Fonte: autor.

6.3 Update

6.3.1 Programa

A atualização de dados é o componente do programa que apresenta o maior número de testes. Ele incorpora os testes usados na criação de um registro, bem como os de exclusão, pela necessidade de se fornecer um ID.

O programa implementado contém a funcionalidade de um ID fornecido ser inválido e retornar apresentando o resultado visto na figura 6.6. Nesta interface pode ser visto a rotina de validação de ID apresentando a resposta para um ID incorreto.

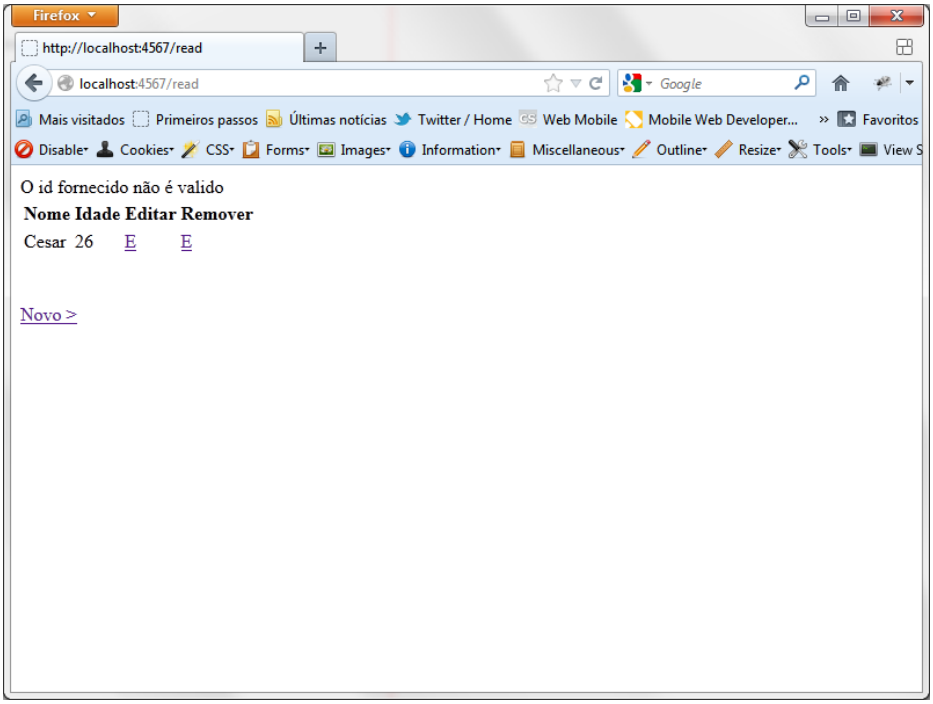


Figura 6.6 - ID inválido na Atualização
Fonte: autor.

Para a funcionalidade seguinte, onde os campos de entrada são validados (partindo do presuposto que um ID valido foi fornecido) tem-se a figura 6.7. Nessa figura é apresentado a mensagem gerada pela rotina de validação dos campos. Sendo im dos campos não preenchidos o retorno apresenta a mensagem indicativa.

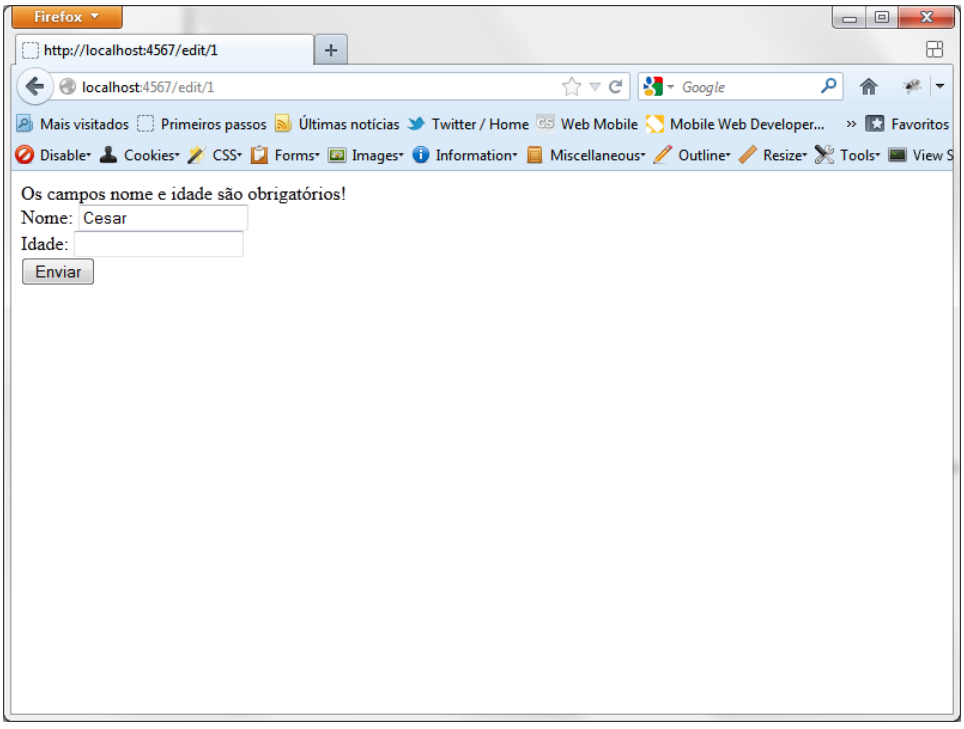


Figura 6.7 - Dados fornecidos inválidos
Fonte: autor.

Finalmente a funcionalidade concluinte desta etapa, o sucesso na atualização dos dados. Sendo esat uma entrada que contem um ID valido e possue ambos os campos preenchidos a rotina retorna como resposta a mensagem de sucesso na alteração.

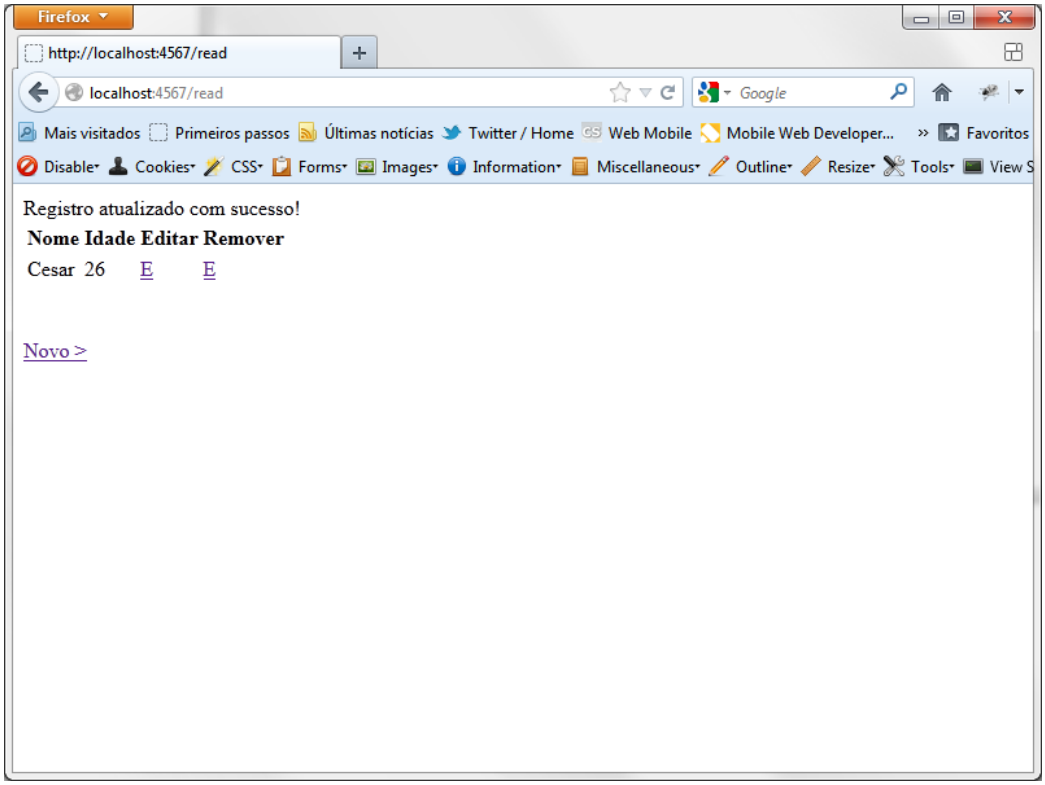


Figura 6.8 - Registro atualizado com sucesso.
Fonte: autor.

6.3.2 Teste

Para os testes das funcionalidades do *uptade*, ID invalido, campos não preenchidos e alteração com sucesso, pode-se ver na figura 6.9 as asserções sendo validadas retornando *true*. Este teste mostra que o programa está corretamente implementado. Sendo assim efetivo com a proposta. Este teste roda e demonstra que as funcionalidades estão presentes e implementadas corretamente mesmo não sendo implementado utilizando-se de toda a estrutura do framework (*request* e *response*).

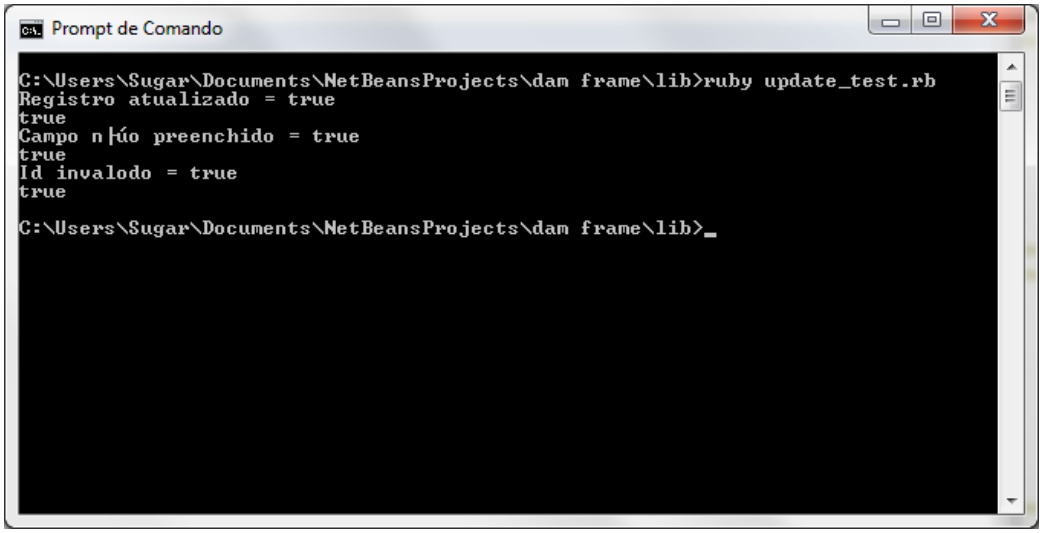


Figura 6.9 - Teste da funcionalidades do update.
Fonte: autor.

6.4 Delete

6.4.1 Programa

Enfim o ultimo programa a ser avaliado, o *delete*. Na figura 6.10 é possível observa o retorno de ID invalido, muito similar ao retorno utilizado no *update*. De fato esta é uma rotina idêntica, e como tal apresenta uma mensagem de invalidade quando um ID fornecido é diferente do esperado.

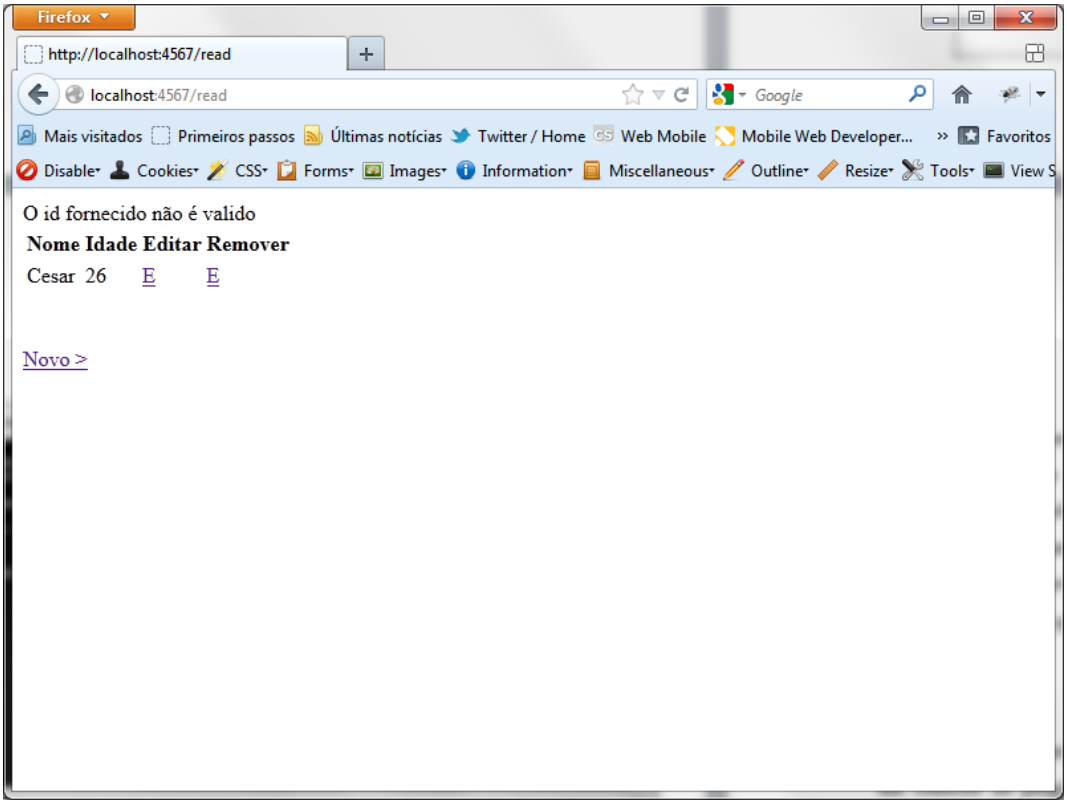


Figura 6.10 Id invalido na deleção.
Fonte: autor.

Da mesma forma, a funcionalidade seguinte, a remoção bem sucedida de um registro, é similar a alteração bem sucedida. Para que tal evento ocorra apenas a rotina de checagem de ID deve falhar. Sendo assim a figura 6.11 ilustra um ID correto fornecido e subsequentemente a mensagem de remoção sendo apresentada no cliente.

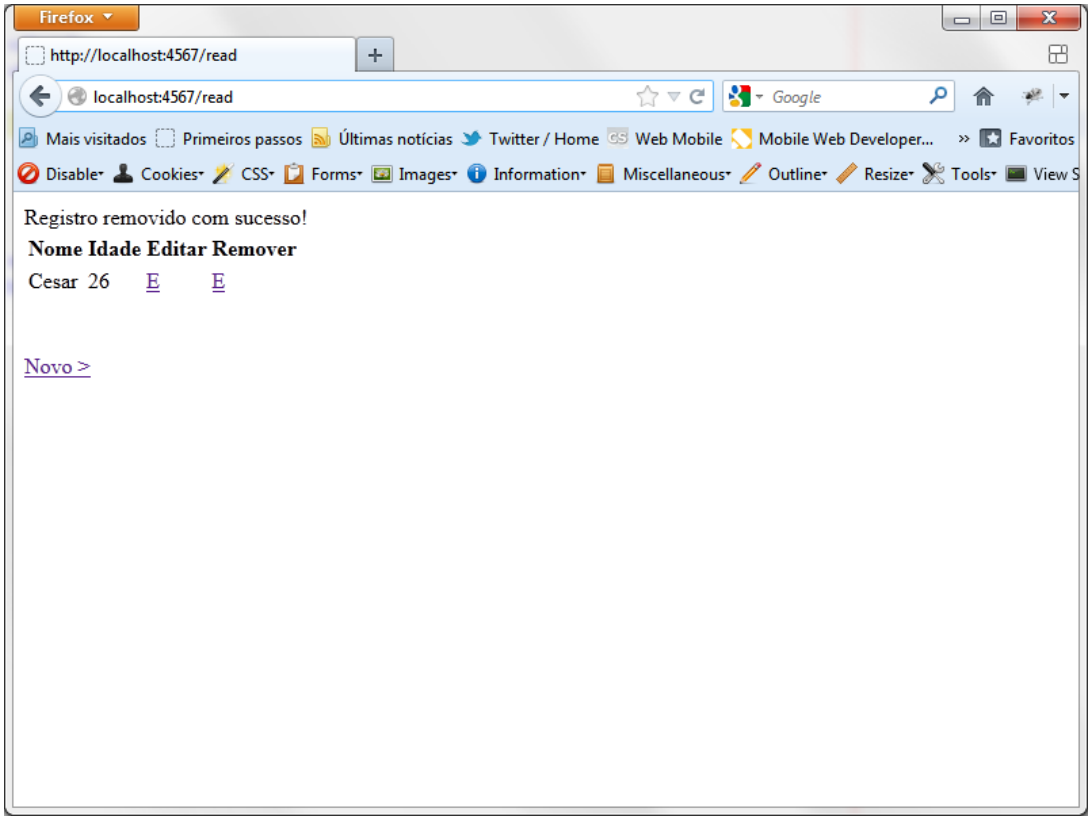


Figura 6.11 Registro removido com sucesso.
Fonte: autor.

6.4.2 Teste

Para os testes das funcionalidades do *delete*, ID invalido e deleção com sucesso, pode-se ver na figura 6.12 as asserções sendo validadas retornando *true*. Este teste mostra que o programa está corretamente implementado. Sendo assim efetivo com a proposta. Este teste roda e demonstra que as funcionalidades estão presentes e implementadas corretamente mesmo não sendo implementado utilizando-se de toda a estrutura do framework (*request* e *response*).

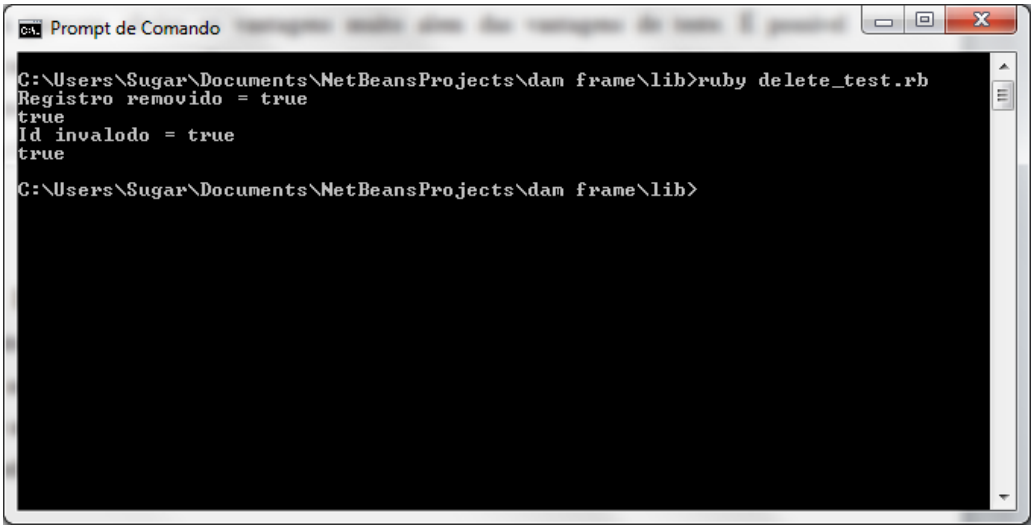


Figura 6.12 Teste das funcionalidades da deleção.
Fonte: autor.

6.5 Outros aspectos do resultado

Enquanto as comparações anteriores demonstram a efetividade, o sucesso na implementação, bem como atingimento da meta de criar uma variação de *framework* que seja capaz de produzir *controllers* facilmente testáveis, existem também alguns efeitos colaterais esperados a serem analisados. Na sequência deste, serão apresentadas comparações feitas entre as *actions* desenvolvidas no novo *framework* modificado (aqui chamado de sinatra+) e uma implementação equivalente usando Zend Framework.

A escolha deste segundo para a comparação é arbitrária uma vez que o autor domina este mais profundamente. Da mesma forma que é mais suscetível a tornar evidentes algumas particularidades onde os ganhos podem ser mais acentuados. No entanto é importante desprezar nessa comparação as pequenas diferenças entre as linguagens.

Embora as linguagens apresentem particularidades que podem favorecer uma em detrimento da outra, é importante observar que um ZF+ destoaria em pouco do sinatra+. Tendo estas nuances em mente pode-se apresentar as tabelas abaixo para comparação dos resultados.

Tabela 6.1 - *Create*

	Sintra +	Zend FW	Redução(%)
Caracteres	302	458	34%
Chamadas	0	8	100%
Linhas	14	14	0%
Atribuições	1	2	50%

Fonte: Autor.

Tabela 6.2 - *Read*

	Sintra +	Zend FW	Redução(%)
Caracteres	124	172	28%
Chamadas	0	0	0%
Linhas	10	10	0%
Atribuições	0	2	100%

Fonte: Autor.

Tabela 6.3 - *Update*

	Sintra +	Zend FW	Redução(%)
Caracteres	443	746	41%
Chamadas	0	12	100%
Linhas	20	20	0%
Atribuições	3	1	-200%

Fonte: Autor.

Tabela 6.4 - Delete

	Sintra +	Zend FW	Redução(%)
Caracteres	315	572	45%
Chamadas	0	10	100%
Linhas	16	17	6%
Atribuições	2	1	-100%

Fonte: Autor.

Em uma comparação das *actions create*, houve uma redução de 34% na quantidade de caracteres. O que significa um programa mais curto para escrever e mais simples de entender. Uma redução drástica na quantidade de chamadas de métodos de 100% o que resulta em um programa mais simples de entender e mais rápido. O números de linhas de código muito embora sejam iguais aos números de linhas de código lógicas (LLoC) do sinatra+ é menor. Linhas como o fechamento do bloco de *array* são linha lógicas pertencentes a abertura e poderiam ser colocadas na mesma linha da abertura. Quanto as atribuições houve uma redução de 50% o que resulta e melhor aproveitamento de memória.

As comparações mostradas nas tabelas 6.2, 6.3 e 6.4 mostram um panorama muito próximo do detalhado na primeira, onde alguns pontos mostram maior ou menor eficiência na redução de caracteres, chamadas e LLoC. Contudo, ficou evidente um aumento na quantidade de atribuições. Mais atribuições são usadas para resolver a mecânica de retornar o *array* correto. Porém essas atribuições são condicionais, ou seja, são mutuamente exclusivas. O que efetivamente em um programa rodando significa apenas uma atribuição em *runtime* onde encontramos três no programa.

Além disto, as atribuições são de *arrays* menos exigentes em termos de memória se comparados a quantidade de memória alocada dentro das chamadas de métodos onde são usadas atribuições de objeto.

CONCLUSÃO

Até o presente estágio do trabalho não foi possível identificar porque a abordagem proposta não é um denominador comum entre os *frameworks*. Analisando a proposta, parecem óbvias as vantagens. Poder testar um *controller* unitariamente é um ganho enorme. Embora, nenhum dos frameworks pesquisados implementa dessa forma. Felizmente este trabalho abre um precedente para que isso ocorra.

Uma vez que de fato, é possível implementar desta forma. Pode-se, utilizando a abordagem de se implementar uma camada intermediaria criando uma interface simples de comunicação do *controller*, simplificar o *request* e o *reponse* em *arrays*, criando um *controller* facilmente testável unitariamente

Muito além das vantagens de se testar unitariamente o *controller*, foi possível identificar outros ganhos. Que se comprovaram quando executados os testes no capitulo seis e ficam claros nas analises comparativas da seção 6.5. Talvez estes tão interessantes quando a viabilidade de testar unitariamente o *controller*. Tornar o desenvolvimento mais ágil é um efeito colateral que qualquer solução sonharia em ter.

Os medos que segundo Beck (2010) podem ser vencidos utilizando o TDD agora podem ser aplicados a *controllers*. Isto é comprovado pelos teste aplicados a cada um dos casos de uso propostos. Todos obtiveram sucesso em implementar um programa funcional capaz de ser testado unitariamente e assim contemplar o mantra do TTD.

E como se não bastasse, uma analise mais detalhada do *controller* resultante apresentada na seção 6.5 mostrou quanto ganho em performance, simplicidade em codificação e entendimento pode-se ter. Segundo Martin (2008), estes dois pontos, são grandes ganhos. Não somente um código bem escrito e fácil de entender ajuda a mitigar a entropia de um sistema, mas também o teste pode contribuir em muito para isso. Neste sentido este é um ganho duplo para a longevidade do programa.

Para trabalhos futuros é possível elencar dês de uma serie de funcionalidades que poderiam se agregadas a uma implementação e disponibilização no mundo real desta customização. Algumas das possibilidades são elencadas abaixo.

É interessante observar também que, com base na proposta, já é possível vislumbrar um controle de *plugins* necessários para endereçar problemas de arquitetura com relação aos retornos desejados do *controller*. Em uma implementação simplista haveria um grau de acoplamento grande, onde cada recurso disposto no retorno do *controller* que necessitaria estar atrelado ao *controllerAction*, ou ainda ao *dispatcher*.

Seria excelente poder contar com um sistema de distribuição de *plugins*, no qual o usuário pudesse escolher quais os *plugins* estariam disponíveis no retorno do seu *controller*, assim como estender mais *plugins*. Contudo, este processo pagaria o preço de tornar a implementação e o entendimento do *framework* mais complexo.

Ainda assim, a agilidade provida tornando a implementação do *controller* mais simples seria um ganho formidável que, provavelmente compensaria este custo. Contudo, o problema principal, os testes unitários, ainda estariam sendo endereçados. O programa de testes não teria a necessidade de inicializar o sistema de distribuição dos *plugins*, muito menos executá-los. A simples avaliação dos parâmetros a serem passados a diante para os *plugins* seria suficiente para verificação do teste.

Outro ponto interessante são os retornos do *controller* para o *flash*. Estes poderiam ser um ID. Esse ID corresponderia a uma *string* em alguma base de dados ou arquivo de configuração. Desta forma seria mais fácil testa um código sem problemas de falha de digitação. Ou seja, no lugar da string ser passada no atributo *flash* do retorno, um número seria retornado. Esse número único por mensagem representaria uma string única. Alem disso para efetuar o teste esse numero não precisaria ser convertido em string, uma comparação número-número seria muito mais rápida. Esta tática ainda poderia ser usada em conjunto com a internacionalização do programa sem perdas.

Esta é uma importante melhoria, pois foi possível observar que em alguns testes a *string* estava escrita errada, apesar de o programa funcionar perfeitamente a *string* poderia causar um erro no teste. O ID reduziria a chance deste erro acontecer.

Também como foi destacado ao longo do documento, reaver dados adicionais do *request* pode ser um problema, já que essa arquitetura não prevê esta funcionalidade. Contudo, é possível criar um modelo de *request singleton*. Desta forma o desenvolvedor pode rapidamente lançar mão das informações e métodos comuns a uma requisição.

Finalmente a implementação deste protótipo no cerne do framework e a distribuição do mesmo é um trabalho inteiro a ser estudado. Comparar estes resultados com resultados em campo poderiam reforçar ou desprovar estas conclusões. Este sem duvida é o próximo passo ideal para este trabalho. Subsequentemente a este a implementação em outras linguagens e outros *frameworks* seria o caminho lógico.

REFERÊNCIAS BIBLIOGRÁFICAS

BECK, Kent. TDD: **Teste Guiado por Teste**.1º ed. Porto Alegre: Bookman Companhia Ed, 2010. 240p.

FEATHERS, Michael C. **Working Effectively with Legacy Code**. 1º Ed. New Jersey: Prentice Hall, 2004

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. 1º Ed. New Jersey: Prentice Hall, 2002

GAMMA, Erich; HELM, Richard; JHONSON, Robert; VLISSIDES, Jhon. **Design Patterns**. 1º Ed. Addison Wesley, 1994

MARTIN, Robert C**. Clean Code**. 1º Ed. New Jersey: Prentice Hall, 2008

SANDERSON, Steven. 2009. **Writing Great Unit Tests:** Best and Worst Practices. Disponível em: <http://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/> Acesso em 08 setembro 2011.

Ruby on Rails, **Ruby on Rails Guide.** Disponível em: <http://guides.rubyonrails.org>. Acesso em 08 setembro 2011.

Zend Framework, **Documentação Zend Framework.** Disponível em: <http://framework.zend.com/manual/1.12/en/manual.html>. Acesso em 08 setembro 2011.

Sinatra, **Sinatra Documentation.** Disponível em: <http://www.sinatrarb.com/documentation>. Acesso em 08 setembro 2011.