Avaliação da Qualidade de Código em *Ruby on Rails* com *Code Climate*

Guilherme Côrtes Silva, Wander Antunes Gaspar Valente

Centro de Ensino Superior de Juiz de Fora – Juiz de Fora – MG – Brasil

guilhermecortes@gmail.com, wandergaspar@pucminas.cesjf.br

Resumo. *O presente trabalho discorre sobre ferramentas de análise estática para a linguagem Ruby on Rails. Inicialmente são apresentados os fundamentos da análise estática de código fonte e descritas as características das ferramentas avaliadas. Em seguida, são introduzidos os métodos utilizados para gerar as notas de avaliação e é apresentado um estudo de caso que emprega a ferramenta Code Climate em uma aplicação desenvolvida em Ruby on Rails. O estudo de caso consiste em obter métricas a partir da ferramenta após o término do desenvolvimento do software, proceder às alterações conforme recomendações da ferramenta e submeter a aplicação a uma posterior análise para verificar as melhorias na qualidade do código fonte considerando-se os refinamentos implementados. Como conclusão do trabalho, são relacionados os resultados alcançados referentes à melhoria da qualidade do software desenvolvido em Ruby on Rails quando se utiliza a ferramenta de análise estática Code Climate.*

Abstract. *This paper discusses static analysis tools for Ruby on Rails language. Initially are presented the elements of static analysis of source code and described the characteristics of the evaluated tools. Then, the methods used to generate the evaluation grades are introduced and a study case that employs the Code Climate tool in an application developed in Ruby on Rails is presented. This study case consists of obtaining metrics from the tool after the software development ends, proceed with the changes as tool recommendations and submit the application at a later analysis to verify the improvements in the quality of the source code considering the implemented refinements. As a conclusion, the results achieved in the quality of software developed in Ruby on Rails are related when using static analysis tools like Climate Code.*

1. Introdução

Escrever um bom código é de fundamental importância para manter a qualidade e facilitar a manutenção. Segundo Pressman (2011), um código mal escrito é um dos oponentes do mercado de software, sendo responsável por até 45% do tempo que um sistema fica inativo e custa bilhões de dólares com manutenção e redução de produtividade, não incluindo ainda a perda de clientes devido à insatisfação.

Todo sistema está sujeito à manutenção, procedimento que tem início tão logo ocorra a liberação para o usuário final. Ainda de acordo com Pressman (2011), o software estará em constante evolução com o tempo, independente do domínio de aplicação, complexidade ou tamanho.

Segundo Martin (2008), todo desenvolvedor já teve o trabalho impactado devido a algum código mal escrito. O código desorganizado pode se tornar problemático para a equipe, aumentando a complexidade e diminuindo a produtividade. Cada alteração feita no sistema causa impacto em diversas partes da aplicação.

Em função dessas considerações, é necessário se preocupar com a qualidade do código, para reduzir o tempo gasto com a manutenção do software e a redução de custos envolvidos, além de facilitar o entendimento da aplicação [Martin 2008].

No presente trabalho, uma aplicação desenvolvida em *Ruby on Rails* será submetida a uma ferramenta de análise estática de código denominada *Code Climate* para avaliar a qualidade da escrita. Após o primeiro resultado, o código será alterado para facilitar o entendimento e submetido novamente para análise. Também será apresentada uma ferramenta similar, *PullReview*, e descritas as principais características de cada uma.

2. Ferramentas de Análise Estática

Desde 2000, a utilização de ferramentas para garantir a qualidade do código durante o desenvolvimento e manutenção do software tem aumentado. Existem diversas opções *open source* e comerciais disponíveis no mercado, muitas delas com objetivos variados [Muske e Bokil 2015].

Durante a revisão do código, desenvolvedores tentam melhorar a qualidade do que foi escrito, corrigindo *bugs* ou fazendo com que o software fique mais fácil de ser entendido para uma futura manutenção. Um método que pode auxiliar nesta tarefa são as ferramentas de análise estática [Panichella *et al*. 2015].

Análise estática é o processo de analisar estaticamente e de forma automatizada o código fonte de um programa para detectar possíveis inconsistências, sem que o software precise ser executado [Prähofer *et al*. 2012].

Utiliza-se cada vez mais cedo, no processo de desenvolvimento, uma ferramenta para auxiliar a detecção de possíveis vulnerabilidades, como falhas de segurança que podem ter sido incluídas no processo de desenvolvimento, pois economiza-se tempo com manutenção e custos ao encontrar e corrigir erros nesta etapa de desenvolvimento do que após ser disponibilizado para o cliente. Mesmo com a automatização das ferramentas, ainda é necessário o fator humano na análise, pois os resultados apresentados podem ser considerados falsos positivos. Um alerta é classificado como falso positivo quando o desenvolvedor responsável pela análise julga que o alerta não precisa ser corrigido [Baca et al. 2009].

Ferramentas de análise estática possuem a capacidade de identificar informações a respeito do software sem a necessidade de executá-lo. Isto torna as ferramentas apropriadas para a detecção de problemas no código, que podem auxiliar na inclusão de vulnerabilidades na aplicação. Devido a isto, tais ferramentas se tornam adequadas para o esforço de melhorar a qualidade do código [Ribeiro 2015].

Os Estados Unidos investem um grande volume de recursos em programas com foco na segurança da informação. O *National Institute of Standards and Technology* (NIST) possui um grupo, financiado pelo departamento de segurança nacional, focado em métricas de qualidade de software e avaliação das ferramentas com objetivo de garantir a qualidade [Ribeiro 2015].

Existem diversas opções de bibliotecas que permitem a avaliação do código desenvolvido em *Ruby*. Cada biblioteca possui um foco de análise, como segurança, duplicação ou complexidade, por exemplo, e não possui usabilidade que torne simples a manipulação, muitas vezes sendo necessário a utilização da biblioteca através do terminal. Com isso, foi escolhido a utilização de ferramentas que possuem melhor usabilidade e envolva mais de uma métrica durante a análise do código.

As subseções seguintes apresentam duas ferramentas que analisam o código desenvolvido em *Ruby* e uma breve comparação entre elas.

2.1. PullReview

De acordo com o site oficial [PullReview, 2015], a ferramenta foi criada em 2011 por Christophe Philemotte, com objetivo de automatizar a revisão de código em *Ruby on Rails*.

Ainda segundo o *website*, já foram processados mais de um milhão de *commits*, caracterizados pela gravação no banco de dados de todas as alterações que foram feitas nos arquivos [Chacon et al. 2014]. Além disso, já foram realizadas nove milhões de correções em códigos e identificados cinquenta milhões de problemas. Diariamente, são analisados mais de 2000 projetos.

PullReview possui integração com *GitHub*, *BitBucket* e *GitLab*, que são plataformas para gerenciar o versionamento de código e que possuem *Ruby on Rails* como linguagem para suporte.

Fork consiste em realizar a cópia de um determinado projeto utilizando o *GitHub*, e permite realizar alterações sem afetar o projeto original. Para submeter uma aplicação à análise pela ferramenta em que a origem do projeto é feita através de um *fork*, é necessário primeiramente contribuir com algum código para a aplicação do autor que foi realizado o *fork*.

A ferramenta analisa aspectos do código como estilo, duplicação, complexidade, documentação e segurança, disponibilizando formas de resolver o problema apontado no resultado [PullReview, 2015].

Para realizar a verificação do projeto, deve-se conectar o PullReview com a conta escolhida, dando a permissão necessária, e submeter a aplicação para análise. Um exemplo do retorno da ferramenta após a verificação do projeto pode ser observado na Figura 1.

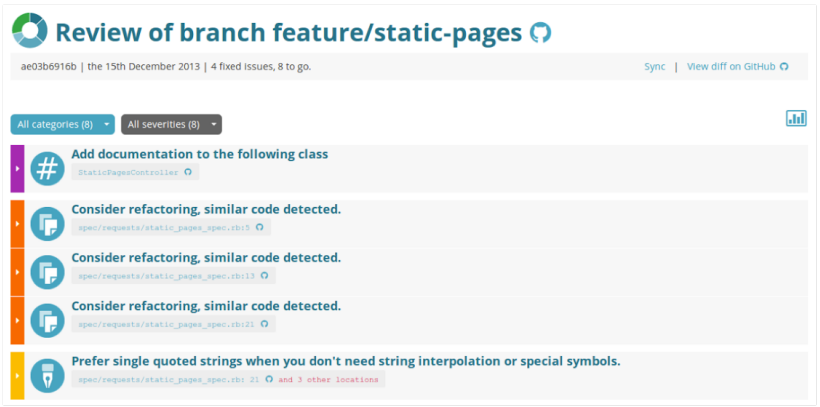


Figura 1. Resultado da análise do projeto

Após realizar o *commit* das alterações sugeridas pela ferramenta, é exibido o que foi corrigido com essa modificação, como observado na Figura 2.

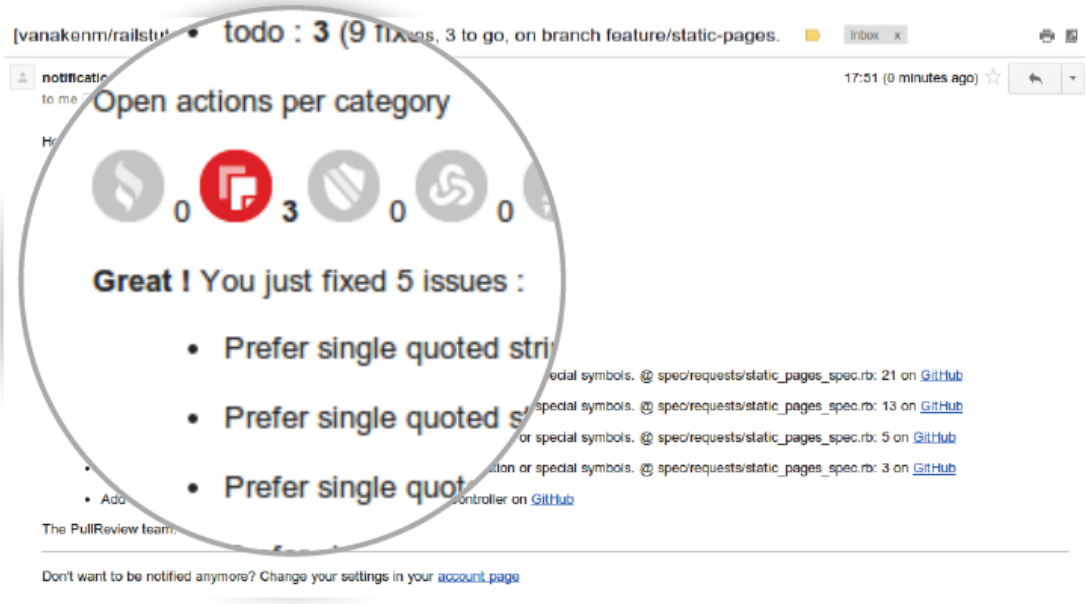


Figura 2. Após alteração do código e nova análise pela ferramenta

As informações sobre a ferramenta foram obtidas exclusivamente através do website oficial. Devido a dificuldade de obter dados, foi realizada uma análise superficial sobre a PullReview, e o foco maior do trabalho foi sobre a Code Climate, que é apresentada na subseção seguinte.

2.2. Code Climate

Code Climate foi desenvolvido em 2011, na cidade de Nova Iorque, e já analisou mais de 40 mil projetos *open source*. Todos os dias, estima-se que a ferramenta auxilia 50 mil desenvolvedores e analisa 700 bilhões de linhas de código, possuindo suporte a diversas linguagens tais como *Ruby*, *JavaScript*, PHP, *Python*, *Go*, *Ember.js*, *CoffeeScript*, CSS, *RubyMotion*, dentre outras [Helmkamp 2015].

Para utilizar a Code Climate deve-se cadastrar o projeto, que pode ser público ou privado. Em seguida, será iniciado o processo de análise, que gerará, no final, uma nota que vai de zero a quatro, como exemplificado na Figura 3 [Akita 2014].

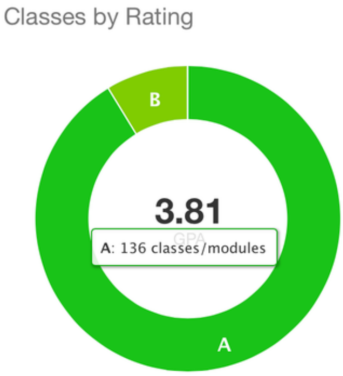


Figura 3. GPA Code Climate

A *Grade Point Average* (GPA) é uma métrica padrão utilizada nos Estados Unidos e em outros países para indicar um nível de performance acadêmica do usuário. É determinado um único número que resume a carreira acadêmica do estudante. Ainda que a GPA seja originalmente voltada para o ambiente acadêmico, pode ser usada em qualquer contexto. A ferramenta baseia-se neste conceito de GPA para atribuir notas aos projetos analisados. Em cada projeto é atribuído uma GPA com base na qualidade do código apresentado, sendo calculado a partir das notas de cada arquivo [Code Climate 2015].

De acordo com a ferramenta [Code Climate 2015], quanto mais próximo de quatro for a GPA, mais adequado estará o código segundo as métricas analisadas. A nota é gerada após analisar todos os arquivos *Ruby* do projeto, fazendo o cálculo de complexidade, duplicação de código, quantidade de linhas de código e número de vezes que o arquivo foi modificado. Também são verificadas questões de segurança, como por exemplo, *SQL Injection*, *Denial of Service*, *Cross Site Scripting*, entre outros [Akita 2014].

Cada arquivo *Ruby* do projeto possui uma nota que varia de A a F, calculada após análise das métricas apresentadas na Figura 4. Cada coluna representa uma informação, onde *Rating* é a nota; *Name* o nome do arquivo; LOC são as linhas de código, excluindo comentários e espaços em branco; *Duplication* é a quantidade de código similar; *Churn* representa o número de vezes que o arquivo foi modificado no *Git*; e *Issues* é o número de locais em que o problema aparece.

Rating	Name	LOC	Duplication	Churn	Issues
F	TicketsController	162	24	8	5
D	Ticket	116	0	9	2
C	NotificationMailer	81	70	3	3
B	Rule	40	0	2	1
A	ApplicationController	50	0	5	1

Figura 4. Notas de arquivos

A ferramenta possui capacidade de verificar 23 itens durante a análise do projeto desenvolvido em *Ruby*. O Code Climate é integrado com diversas plataformas, dentre elas *GitHub*, *BitBucket*, *Slack*, *HipChat* e *Jira* e, de acordo com o *website*, possui mais de 1.000 clientes [Code Climate 2015].

Após cada *commit* realizado, as mudanças no código são verificadas automaticamente e uma mensagem é enviada para a equipe de desenvolvimento com as considerações a respeito das alterações. A mensagem sugere que o código deva ser modificado caso tenha uma nota inferior à avaliação anterior, ou um elogio se a nota foi superior [Code Climate 2015].

A Figura 5, exemplifica o caso de uma nota inferior à anterior.

Figura 5. Nota inferior

Para realizar a análise de um projeto que foi feito um *fork* no GitHub, deve-se adicionar a aplicação, indicando o caminho do GitHub que está o sistema, ao contrário do *PullReview*, que não permite esta análise se não possuir alguma contribuição para o projeto.

3. Métricas Analisadas

Na linguagem *Ruby*, a análise realizada pelo *Code Climate* é divida em três grupos: duplicação, complexidade e segurança. Nas subseções seguintes serão definidos cada um desses conceitos.

3.1. Duplicação

Caso encontre a mesma estrutura de código em locais distintos no software, pode-se afirmar que o programa será mais compreensível após aplicar técnicas de refatoração [Fowler 1999].

A duplicação é o primeiro desafio de um software bem desenvolvido. Representa dificuldade de compreensão e manutenção, além de complexidade desnecessária. Linhas de código parecidas representam casos de duplicação [Robert 2008].

O problema mais simples é quando o mesmo código está presente em mais de um método dentro da própria classe no programa. Pode acontecer de a estrutura ser parecida, mas não a mesma, sendo necessário aplicar técnicas de refatoração para eliminar a inconsistência. Existe outra possibilidade quando a duplicação ocorre em classes distintas, sendo necessário escolher um único local para a chamada do método [Fowler 1999].

Um exemplo pode ser verificado na Figura 6.

```
1  def pedido status
2    if status == "confirma"
3      envia_email_confirmacao_pedido user.email
4    else
5      envia_email_alteracao_status_pedido user.email
6    end
7  end
8  def envia_email_confirmacao_pedido user
9    EnviaEmail.confirma_pedido(user, "Pedido confirmado com sucesso").deliver
10 end
11 def envia_email_alteracao_status_pedido user
12      EnviaEmail.altera_status_pedido(user, "O pedido teve status
atualizado").deliver
13 end
```

Figura 6. Exemplo de Duplicação

Uma ferramenta utilizada no *Code Climate* para verificar a duplicação é o *Flay*. Ele procura por códigos idênticos ou estruturas similares, ignorando diferenças de valores literais, nomes de atributos, espaços em branco e estilo de programação [Code Climate 2015].

3.2. Complexidade

De acordo com Kannavara (2012), a complexidade mede a quantidade de decisões de um programa através do número de caminhos independentes do código fonte. Um método de elevada complexidade implica em alta probabilidade de adicionar *bugs* durante a

manutenção ou adição de novas funcionalidades. O tamanho do método pode torná-lo complexo, dificultando o entendimento e inserindo duplicação no código.

Uma das formas que o *Code Climate* utiliza para definir a complexidade é através da métrica ABC. A métrica é calculada considerando *assignments*, *branches* e *conditionals*. *Assignments* são atribuições de dados a variáveis. *Branches* levam em consideração a chamada de métodos, que aumenta à medida que o código sai do seu fluxo principal para executar código definido em outra parte do sistema, não considera lógicas como if/else. *Conditionals* é qualquer condição lógica [Code Climate 2015].

Uma biblioteca usada pelo *Code Climate* para analisar a complexidade e tamanho do software é o *Flog*. Ele utiliza a métrica ABC, considerando a dificuldade de leitura e manutenção do código analisado [Code Climate 2015].

3.3. Segurança

De acordo com Batista (2007), para garantir a segurança do software é necessário a composição de três propriedades: confidencialidade, integridade e disponibilidade. Disponibilidade é a garantia que a informação estará disponível sempre que precisar. Integridade garante que as informações não foram alteradas. Confidencialidade certifica que à informação só será acessada por pessoas que possuem a devida autorização.

As empresas passaram a se preocupar com a segurança das aplicações nos últimos anos. Isto ocorre devido à preocupação do cliente com a integridade do software, pois devido a utilização dos sistemas na internet, as vulnerabilidades podem ser exploradas. Com o intuito de evitar essa ameaça, as organizações sentiram a necessidade de melhorar a segurança do código para evitá-la [Baca et al. 2009].

Brakeman é uma ferramenta de análise estática *open source* utilizada para identificar problemas de segurança nos projetos *Rails* diretamente no código fonte, diferente de outras ferramentas que analisam a vulnerabilidade [Code Climate 2015].

Para utilizar a ferramenta *Brakeman* em um projeto desenvolvido em *Rails*, sem utilizar a *Code Climate*, deve-se primeiro adicionar a dependência da ferramenta no arquivo de bibliotecas do projeto. Em seguida, é necessário acessar o diretório do projeto através do terminal e executar o comando "brakeman" para iniciar a análise. Após isso, o resultado da verificação é exibido no próprio terminal.

A análise de segurança ainda não está disponível para repositórios gratuitos adicionados na *Code Climate*, a métrica de segurança só é verificada quando adiciona-se um projeto privado. Projetos privados são aqueles em que o código fonte se mantém restrito a uma pessoa e o acesso não é disponibilizado para a visualização de todos. Projetos privados só podem ser adicionados através da versão paga da *Code Climate* [Code Climate 2015].

4. Estudo de Caso

No presente trabalho, uma aplicação desenvolvida em *Ruby on Rails* versão 4.2.0, foi submetida para a análise com a ferramenta *Code Climate, s*endo utilizado o *git* para controle de versão e, por se tratar de um trabalho acadêmico, foi gerada uma *branch* para o desenvolvimento. A análise de outras linguagens não faz parte do escopo do estudo, sendo assim, somente arquivos *Ruby* foram verificados.

A aplicação desenvolvida tem como objetivo controlar as tarefas de uma empresa e optou-se pela utilização de *Ruby on Rails* pois é um *framework open source* que utiliza a arquitetura *Model-View-Controller* (MVC) e, de acordo com Lima (2014), possui como ênfase a velocidade e simplicidade no desenvolvimento de aplicações web.

De acordo com o RedMonk (2015), *Ruby* está entre as cinco linguagens de desenvolvimento mais populares ao lado de C# e C++, como pode ser observado na Tabela 1.

Ranking	Linguagem
1	JavaScript
2	Java
3	PHP
4	Python
5	C#, C++, Ruby

Tabela 1 Linguagens populares

O sistema está disponível no GitHub a partir do link https://github.com/guilhermecortes/HelpDesk e no *Code Climate* através da URL https://codeclimate.com/github/guilhermecortes/HelpDesk. O software foi desenvolvido e submetido para análise com a intenção de avaliar os resultados obtidos. Após a avaliação, alterou-se a aplicação com o objetivo de eliminar os alertas recebidos. Depois de realizar as modificações, a aplicação foi enviada novamente para análise e avaliou-se os novos resultados.

4.1. Primeira Análise

O software submetido à análise da ferramenta obteve um GPA de 2.06, como pode ser observado na Figura 7.

Figura 7. GPA da primeira análise

A ferramenta apontou cinco problemas com complexidade e dezoito de duplicação, como apresentado na Figura 8.

Figura 8. Complexidade e duplicação da primeira análise

Nos itens de duplicação, observou-se que todas as inconsistências apontadas foram para códigos similares, não tendo sido encontradas ocorrências de código idêntico no resultado.

Em relação aos cinco itens de complexidade, três são referentes aos métodos de criação de objetos, *Customers*, *Employees* e *Tickets*. Os outros dois itens pertencem à biblioteca que é utilizada para fazer o controle de usuários no sistema, denominada *Devise*.

Após ordenação em ordem decrescente da classificação dos arquivos, tem-se o resultado de acordo com a Figura 9.

Rating	Name	LOC	Duplication	Churn	Issues
F	Admin::CustomersController	59	180	1	4
F	Admin::EmployeesController	59	180	1	4
F	Admin::TicketsController	67	180	2	4
D	Admin::CategoriesController	56	153	1	3
D	Admin::GruposController	56	153	1	3
D	Admin::WorkTypesController	56	153	1	3
B	RegistrationsController	37	0	1	1
A	PasswordsController	40	0	1	1
A	Ability	11	0	1	0

Figura 9. Ordenação por classificação dos arquivos da primeira análise

Na Figura 10 é apresentado o item de complexidade do método *create*, encontrado no controlador *CustomersController*, que obteve classificação F após a análise realizada pela ferramenta. Na linha 1, é iniciado e definido o nome do método. Em seguida, linha 2, o objeto é instanciado e são atribuídos os parâmetros definidos. Entre as linhas 3 e 13, é definido um bloco que verifica se o objeto foi salvo e determina a mensagem, o redirecionamento e tipo de resposta que é exibido ao usuário.

```
1  def create
2    @customer = Admin::Customer.new(customer_params)
3    respond_to do |format|
4      if @customer.save
5        flash[:info] = 'Cliente criado com sucesso'
6        format.html { redirect_to action: 'index' }
7        format.json { render action: 'index', status: :created, location:
@customer }
8      else
9        flash[:alert] = "Ocorreu um erro ao salvar.
#{@customer.errors.full_messages.to_sentence}"
10       format.html { redirect_to action: 'new' }
11       format.json { render json: @customer.errors, status:
:unprocessable_entity }
12     end
13   end
14 end
```

Figura 10. Método Complexo no Controlador *Customers*

Na Figura 11 é exibido o item de duplicação do método *destroy*, encontrado no controlador *CustomersController*, após a verificação da ferramenta. Na linha 1, é iniciado e definido o nome do método. Entre as linhas 2 e 12, é definido um bloco que verifica se o objeto foi excluído e define a mensagem, o redirecionamento e tipo de resposta que é exibido ao usuário.

```
1  def destroy
2    respond_to do |format|
3      if @customer.destroy
4        flash[:info] = 'Cliente excluído com sucesso'
5        format.html { redirect_to action: 'index' }
6        format.json { head :no_content }
7      else
8        flash[:alert] = "Ocorreu um erro ao excluir.
#{@customer.errors.full_messages.to_sentence}"
9        format.html { redirect_to action: 'index' }
10       format.json { render json: @customer.errors, status:
:unprocessable_entity }
11     end
12   end
13 end
```

Figura 11. Duplicação no Controlador *Customers*

O objetivo é reduzir todos os itens de complexidade e duplicação apresentados e melhorar a classificação dos controladores que obtiveram notas D e F, pois de acordo com a *Code Climate* (2015), arquivos que possuem nota A e B apresentam, respectivamente, performance excelente e boa.

Para solucionar os itens de complexidade e duplicação apresentados nos controladores, foi utilizada a biblioteca *Responders*, que auxilia na redução de código das aplicações *Rails* a partir da versão 4.2.

A biblioteca centraliza em um único local a forma dos controladores responderem a solicitação da aplicação e as mensagens de aviso, que são exibidas após a realização de qualquer ação resultante em alteração de valor no banco de dados.

4.3. Segunda Análise

Após a utilização da biblioteca *Responders* e, consequente, a alteração dos códigos identificados pela ferramenta que apontaram duplicação e complexidade, a aplicação foi submetida a uma nova análise e obteve um GPA de 3.92, como apresentado na Figura 12.

Figura 12. GPA da segunda análise

Foram realizadas alterações em três controladores que obtiveram nota F, *Customers*, *Employees* e *Tickets*, e em outros três que alcançaram nota D, *Categories*, Grupos e *WorkTypes*. Após as modificações nos controladores, os seis arquivos obtiveram a nota A, como pode ser observado na Figura 13.

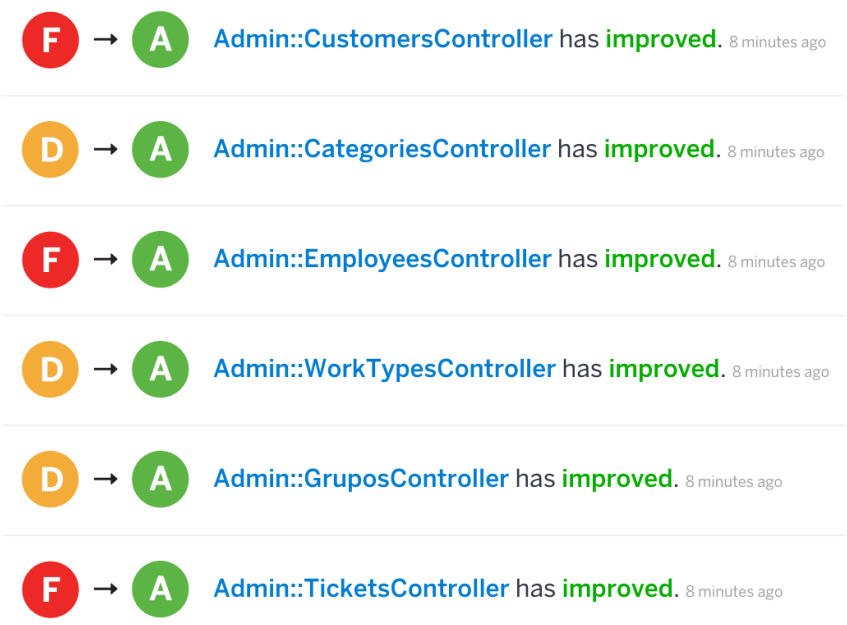


Figura 13. Melhoria das notas dos controladores

A Figura 14 apresenta o método *create* e a Figura 15 exibe o método *destroy* do controlador *Customers* após a alteração realizada. Observa-se que os métodos não possuem mais a responsabilidade de determinar o tipo de resposta dada à aplicação e também não definem mais a mensagem exibida para o usuário. Essas atribuições são definidas, respectivamente, no controlador da aplicação e no arquivo responsável por enviar mensagens ao usuário.

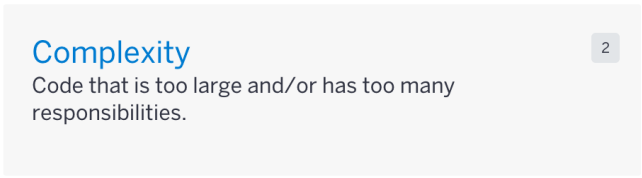
```
1 def create
2  @customer = Admin::Customer.new(customer_params)
3  respond_with @customer, location: @customer.save ? admin_customers_path :
new_admin_customer_path(@customer)
4 end
```

Figura 14. Método *create* do Controlador *Customers* após a alteração

```
1 def destroy
2  respond_with @customer, location: admin_customers_path if @customer.destroy
3 end
```

Figura 15. Método *destroy* do Controlador *Customers* após a alteração

Após as alterações, todos os itens de duplicação e as três inconsistências de complexidade foram resolvidas, conforme apresentado na Figura 16. Permaneceram apenas as



observações referentes aos métodos utilizados pela biblioteca *Devise*.

Figura 16. Complexidade após a alteração

A Figura 17 e a Figura 18 apresentam, respectivamente, a complexidade dos controladores *Registrations* e *Passwords*, ambos pertencentes à biblioteca *Devise*.

Para resolver as inconsistências restantes de complexidade da biblioteca Devise, utilizou-se a extração de método, uma técnica de refatoração.

```
1 def create
2  build_resource(sign_up_params)
3  resource_saved = resource.save
4  yield resource if block_given?
5  if resource_saved
6    if resource.active_for_authentication?
7      set_flash_message :notice, :signed_up if is_flashing_format?
8      flash[:estado] = "Inserido-sucesso"
9      redirect_to admin_usuarios_path
10   else
11     set_flash_message :notice,:"signed_up_but_#{resource.inactive_message}"
if is_flashing_format?
12     expire_data_after_sign_in!
13     respond_with resource, location:
after_inactive_sign_up_path_for(resource)
14   end
15 else
16   clean_up_passwords resource
17   respond_with resource
18 end
19end
```

Figura 17. Método *create* do controlador *Registrations*

```
1 def update
2   self.resource = resource_class.reset_password_by_token(resource_params)
3
4   if resource.errors.empty?
5     resource.unlock_access! if unlockable?(resource)
6     flash_message = resource.active_for_authentication? ? :updated :
:updated_not_active
7     set_flash_message(:notice, flash_message) if is_navigational_format?
8     sign_in(resource_name, resource)
9     redirect_to admin_usuarios_path, :notice => "Passwords has been change"
10  else
11    respond_with_resource
12  end
13end
```

Figura 18. Método *update* do controlador *Passwords*

Extrair método é uma das técnicas de refatoração mais comuns que existem. Entende-se por métodos longos e complexos aqueles que possuem muitas responsabilidades e lógicas complexas, sendo necessário comentários para entender os objetivos que foram definidos. Para solucionar este problema deve-se então criar métodos menores, que utilizem fragmentos do anterior, e que o nome do método explique o objetivo [Fowler 1999].

Posteriormente a extração do método *update* do controlador *Passwords*, gerou-se um novo método e a complexidade foi eliminada. Os métodos podem ser observados na Figura 19.

```
1  def update
2    self.resource = resource_class.reset_password_by_token(resource_params)
3
4    if resource.errors.empty?
5      set_flash_message_and_redirect resource
6    else
7      respond_with_resource
```

```
8     end
9   end
10
11 def set_flash_message_and_redirect resource
12     resource.unlock_access! if unlockable?(resource)
13     flash_message = resource.active_for_authentication? ? :updated :
:updated_not_active
14     set_flash_message(:notice, flash_message) if is_navigational_format?
15     sign_in(resource_name, resource)
16     redirect_to admin_usuarios_path, :notice => "Passwords has been change"
17 end
```

Figura 19. Novos métodos do controlador *Passwords* após eliminação da complexidade

Após a extração do método *create* do controlador *Registrations*, obteve-se um novo método, como pode ser observado na Figura 20, tendo sido eliminada a complexidade. A extração resultou na melhoria da nota do controlador, alterando de B para A, conforme a Figura 21.

```
1 def create
2   build_resource(sign_up_params)
3   resource_saved = resource.save
4   yield resource if block_given?
5   if resource_saved
6     active_message_and_redirect resource
7   else
8     clean_up_passwords resource
9     respond_with resource
10 end
11end
12
13 def active_message_and_redirect resource
14     if resource.active_for_authentication?
15       set_flash_message :notice, :signed_up if is_flashing_format?
16       flash[:estado] = "Inserido-sucesso"
17       redirect_to admin_usuarios_path
18     else
19       set_flash_message :notice,:"signed_up_but_#{resource.inactive_message}"
if is_flashing_format?
20       expire_data_after_sign_in!
21       respond_with resource, location:
after_inactive_sign_up_path_for(resource)
22     end
23 end
```

Figura 20. Novos métodos do controlador *Registrations* após a extração

Figura 21. Melhoria de nota do controlador *Registrations*

Após a eliminação das complexidades restantes e consequente melhoria da nota do controlador *Registrations*, alcançou-se um novo GPA conforme a Figura 22.

Figura 22. GPA final

5. Considerações Finais

Este trabalho apresentou uma análise técnica da ferramenta *Code Climate*, com objetivo de melhorar a qualidade do código desenvolvido em *Ruby on Rails*, eliminando itens de complexidade e duplicação que foram apontados.

O trabalho expôs ainda conceitos sobre ferramentas de análise estática de código, definiu as métricas utilizadas pela ferramenta para classificar os projetos analisados e apresentou uma ferramenta similar, *PullReview*, utilizada para analisar código escrito em *Ruby on Rails*.

Após a primeira análise realizada pela ferramenta, foram apontados problemas de código complexo e duplicado. Os itens identificados foram solucionados após a atribuição de responsabilidades a um único local na aplicação, pela utilização da biblioteca *Responders*, e após aplicar a técnica de refatoração, extração de método.

O resultado apresentado sugere que o uso de ferramentas como Code Climate e PullReview podem ser recomendadas para uso em ambientes de desenvolvimento de software uma vez que oferecem recursos interessantes relacionados à melhoria do código fonte, devido principalmente à redução e simplificação, o que contribui para facilitar os trabalhos de manutenção.

Como trabalhos futuros recomenda-se o desenvolvimento de uma aplicação que utilize outra linguagem suportada pelo *Code Climate* e analise as métricas avaliadas para a linguagem escolhida. Além disso, pode-se realizar uma análise comparativa entre softwares, desenvolvidos em linguagens diferentes, que possuem o mesmo foco e avaliar os resultados e sugestões apresentadas pela ferramenta.

Referências

Akita, F. (2014) “CodeClimate, Qualidade de Código e os Rubistas Sádicos”, http://www.akitaonrails.com/2014/01/30/codeclimate-qualidade-de-codigo-e-os-rubistas-sadicos#.VdZ-lLSppHg, acessado em Setembro de 2015.

Baca, D., Petersen, K., Carlsson, B., Lundberg, L. (2009) “Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?” International Conference on Availability, Reliability and Security.

Batista, C. F. Alves (2007) “Métricas de Segurança de Software”, Pontifícia Universidade Católica, Rio de Janeiro, RJ.

Chacon, S., Straub B. (2014), Pro Git: Everything You Need To Know About Git, Apress, 2ª Edição.

Code Climate, disponível em: http://docs.codeclimate.com, acessado em Setembro de 2015.

Fowler, M. (1999), Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1ª Edição.

Helmkamp, B. (2015) "Introducing the Code Climate Platform", https://www.youtube.com/watch?v=lukVYdENeNY, New York City, acessado em Setembro de 2015.

Kannavara, R. (2012) "Securing Opensource Code via Static Analysis" IEEE Fifth International Conference on Software Testing, Verification and Validation.

Lima, Johann Gomes B. (2014) "Uma aplicação de impacto social com aprendizagem de máquina", Monografia de Conclusão de Curso, UFRPE, Recife.

Muske, T., Bokil, P. (2015) "On Implementational Variations in Static Analysis Tools" IEEE International Conference on Software Testing, Verification and Validation, Montréal, Canada.

Panichella, S., Arnaoudova, V., D. Penta, M., Antoniol, G. (2015) "Would Static Analysis Tools Help Developers with Code Reviews?" IEEE International Conference on Software Testing, Verification and Validation, Montréal, Canada.

Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H., Grillenberger, F. (2012) "Opportunities and Challenges of Static Code Analysis of IEC 61131-3 Programs", Austria.

Pressman, Roger S. (2011), Engenharia de Software: Uma Abordagem Profissional, Porto Alegre, AMGH, 7ª Edição.

PullReview, disponível em: http://docs.pullreview.com, acessado em Setembro de 2015.

RedMonk (2015) "The RedMonk Programming Language Rankings: June 2015", disponível em http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/, acessado em Setembro de 2015.

Ribeiro, A. Coimbra (2015) "Análise Estática de Código-Fonte com Foco em Segurança: Metodologia Para Avaliação de Ferramentas", Universidade de Brasília UnB, Brasília, DF.

Robert, C. Martin (2008), Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall PTR.