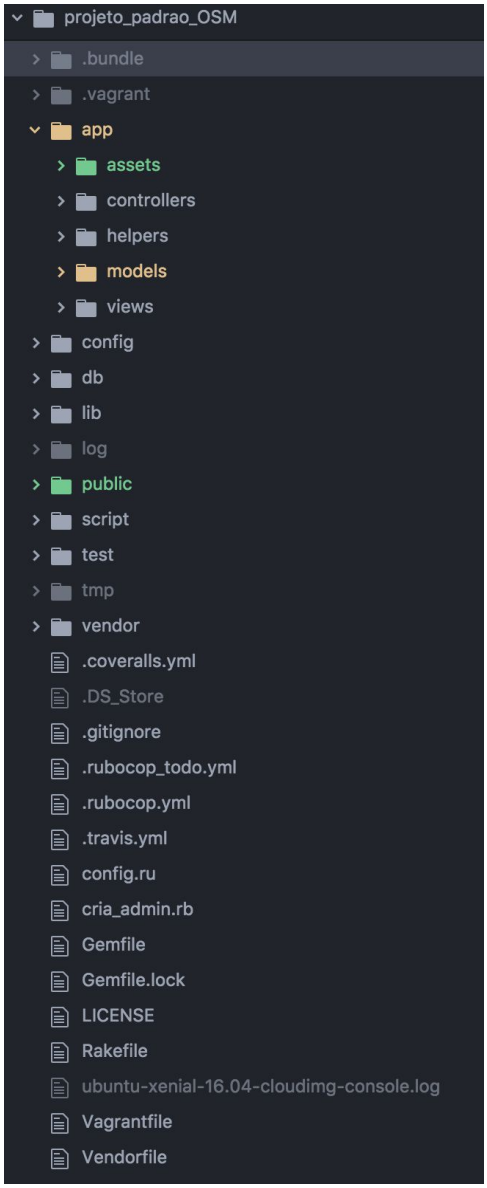
Grupo 3 - Código Fonte

Alunos:

Davi Grossi Hasuda
Eduardo Henrique Ferreira Silva
Gustavo Nahum Alvarez Ferreira

Organização do código

A imagem a seguir mostra a estrutura de pastas do código desenvolvido pelos alunos. A organização do código será explicada a partir de tal estrutura.



O código foi feito utilizando-se Ruby on Rails, que faz grande parte do trabalho de organização, sendo assim, a maior parte dessas pastas foi criada pelo próprio framwork utilizado.

Dentro da pasta app é possível encontrar o projeto de fato. A arquitetura de código utilizada é o MVC (Model View Controller), padrão seguido pelo Rails. Suas subpastas serão discutidas mais à frente com mais detalhes, mas o importante a se saber inicialmente é que a estrutura central do código está toda dentro da pasta app, todo o core do MVC implementado nos arquivos dentro dessa pasta.

A pasta config contém arquivos de configuração. Isso inclui arquivos de environment, que são três: desenvolvimento, teste e produção. Há também inicializadores e outros arquivos relacionados com configurações do projeto. A vantagem é que o Rails gerencia todos esses arquivos de forma automática e não é necessário preocupação humana com eles.

Dentro da pasta db existem arquivos relacionados com o banco de dados. O que se destaca dessa parte do código é o diretório chamado "migrate". Tal diretório contém todas as migrações realizadas no banco de dados, ou seja, todas as modificações realizadas na estrutura do banco de dados. Sempre que for feita alguma alteração naquilo que é salvo, surgirá aqui um novo arquivo. Esses arquivos também são gerados pelo Rails, mas isso ocorre no momento em que se chama o comando db migrate.

Há pastas como lib, log e script que são geradas automaticamente pelo framework e que não devem ser de grande preocupação aos desenvolvedores. Neste documento serão detalhados apenas os fragmentos do código mais importantes para o bom entendimento e continuidade do projeto.

A última pasta que será detalhada é a pasta test. Ela é fundamental para o bom uso de metodologias ágeis e bastante utilizada pelos programadores envolvidos no projeto. Nela encontram-se todos os testes automatizados implementados. Eles estão separados de acordo com a estrutura MVC dentro dos subdiretórios models, controllers. Sempre que um novo modelo ou controlador é criado, recomenda-se adicionar um teste a ele.

No momento de realizar os testes, principalmente testes de modelos, pode ser necessário criar instâncias de classes, mas escondendo-se a complexidade de tais objetos, utilizando apenas o essencial para o teste. Dentro de factories utiliza-se o "FactoryBot" que ajuda na criação de *mocks* para os testes.

Por fim, há também testes de integração que podem ser encontrados dentro da pasta "integration". É muito importante que seja mantida essa coerência e estrutura nas pastas para que o projeto se mantenha organizado e sua manutenibilidade não seja afetada.

Detalhes do MVC

A partir daqui e uma vez conhecida a estrutura mais geral do código, será explicado com um pouco mais de aprofundamento como o código se organiza a partir da ideia do MVC.

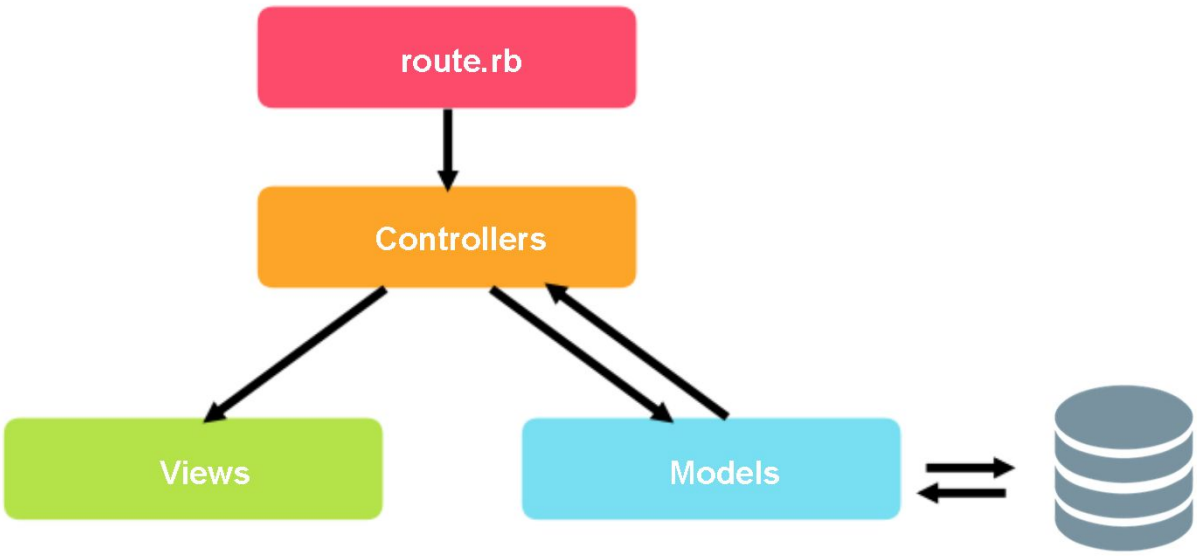
Dentro da pasta config há um arquivo chamado routes.rb. Tal arquivo é responsável por direcionar as requisições do usuário para um dos controladores implementados no código. A partir da ação tomada pelo usuário, escolhe-se um controlador e uma ação contida nele para ser executada. Esse arquivo pode ser visto como uma ponte entre o usuário e a estrutura MVC do código.

Uma vez selecionado um controlador e uma ação dele, o papel do controlador é utilizar os modelos para ter acesso aos dados necessários e também realizar as alterações pertinentes. É ele que indica as ações a serem tomadas. Ao final, o controlador também envia uma view, que será rederizada ao usuário.

O papel dos modelos é, além de criar uma estrutura para os dados envolvidos, servir de interface ao banco de dados. A partir da definição dos modelos, o Rails já possui informações suficientes para conseguir salvar, alterar e recuperar dados do banco de dados. Dessa forma, os modelos são úteis aos controladores por encapsularem os dados de forma lógica e estruturada.

Por fim as views possuem as páginas a serem rederizadas pelo browser. Elas são escolhidas pelo controlador e, portanto, seu único papel é o de prover o que o usuário possue acesso. Nesse pedaço do código esperavam-se arquivos no formato .html, mas o que se obtém são arquivos com terminação .erb. Essa terminação é a sigla para *"embeded ruby"*. Como muitas páginas renderizadas são dinâmicas, isto é, variam de acordo com o contexto, é necessário adicionar um pouco de lógica na montagem do html. Assim utiliza-se o Ruby para realizar a parte lógica na montagem de tais páginas. Nota-se que esses arquivos possuem uma estrutura muito parecida com a de um arquivo html, a diferença está nos códigos entre "<%" e "%>" e entre "<%=" e "%>", pois são linhas de código escritos em Ruby. O restante do código pode ser facilmente associado a um arquivo html.

Em resumo, o projeto utiliza a arquitetura MVC. Seus controladores são acessados a partir do *route* e eles utilizam modelos para suas ações e retornam views para o usuário. A imagem a seguir ilustra a arquitetura básica do projeto.



Tal organização do código é de fácil manutenção e é, ao mesmo tempo, bastante flexível, principalmente para se adicionar novos elementos ao projeto.

Padrões utilizados

Conforme será visto no item a seguir, a ferramenta utilizada na análise estática automatizada foi o RuboCop. Sendo assim, os padrões utilizados no projeto se baseiam naqueles testados por esse guia de estilo. Tais testes são realizados com base em algumas preferências dos desenvolvedores, e elas estão registradas em forma de código dentro do arquivo rubocop.yml. Cada uma das propriedades de estilo checadas são denominadas "cops", este vocabulário será usado no decorrer deste documento.

A documentação do *style guide* que foi utilizado como base para este projeto pode ser visitada em https://github.com/rubocop-hq/ruby-style-guide, e a versão em português está disponível em https://github.com/rubensmabueno/ruby-style-guide/blob/master/README-PT-BR.md.

Sendo assim, com o intuito de se evitar repetições desnecessárias, este arquivo conterá uma breve descrição dos elementos que divergem dos *links* apresentados. Isto é, as mudanças introduzidas pelos desenvolvedores no arquivo rubocop.yml. Um fragmento desse arquivo é mostrado a seguir.

```
27  Layout/ExtraSpacing:
28    AllowForAlignment: true
29
30  Lint/PercentStringArray:
31    Exclude:
32      - 'config/initializers/secure_headers.rb'
33      - 'app/controllers/site_controller.rb'
34
35  Naming/FileName:
36    Exclude:
37      - 'script/deliver-message'
38      - 'script/locale/reload-languages'
39      - 'script/update-spam-blocks'
40
41  Naming/UncommunicativeMethodParamName:
42    Enabled: false
43
44  Rails/ApplicationRecord:
45    Enabled: false
46
47  Rails/CreateTableWithTimestamps:
48    Enabled: false
49
50  Rails/HasManyOrHasOneDependent:
51    Enabled: false
52
53  Rails/HttpPositionalArguments:
54    Enabled: false
55
56  Rails/InverseOf:
57    Enabled: false
```

Como pode ser visto, nas linhas enumeradas 27 e 28, o que se está indicando é que são permitidos espaços em branco tidos como extras ou desnecessários, desde de que tenham o objetivo de alinhar múltiplas linhas no código. Dessa forma o código

```
name        = "RuboCop"
```

não é considerado ruim.

O comando "Exclude" presente nas linhas 31 e 36 indica exceções, ou seja, arquivos que não terão o mesmo dos demais nos quesitos especificados.

Nas linhas 41 e 42 o que é feito é desabilitar a parte do guia de estilo que checa nomes de parâmetros de métodos por quão descritivos eles são. Essa propriedade quando ativa é altamente configurável, mas neste caso não será considerada pela análise automatizada de estilo.

Em seguida, nas linhas 44 e 45, apenas checa se o modelo de subclasses é realizado com Rails 5.0 e, para o projeto em questão, foi desabilitado. Caso ainda estivesse ativa essa análise de estilo, a boa prática seria:

```ruby
class Rails5Model < ApplicationRecord
      # ...
end
```

enquanto que o código:

```ruby
class Rails4Model < ApplicationRecord::Base
      # ...
end
```

seria considerado ruim.

O parâmetro CreateTableWithTimestamps checa as *migrations* em que não há *timestamps* incluídas ao se criar uma nova tabela. Mas para o projeto desenvolvido esses *timestamps* não se faziam úteis de fato e não eram obrigatórios em todas as *migrations*.

Na cop HasManyOrHasOneDependent busca-se por associações de tabela, '*has_many*' ou '*has_one*' que não especificam a opção '*dependent*' ou ':*through*'. Mas mais uma vez esta cop está desabilitada.

Nas linhas 53 e 54 está sendo desabilitada uma cop que apenas se aplica para versões do Rails a partir da 5ª. Neles os métodos CRUD de requsições http são verificados a fim de se garantir que os argumentos possuem uma palavra-chave associada.

Por fim, a cop InverseOf busca por relações entre tabelas do banco de dados em que o Active Record não é capaz de identificar automaticamente a associação inversa. Contudo essa cop está desativada.

Em seguida há mais especificações de como o guia de estilos deve ser utilizado pelo código. Veja-se a imagem a seguir contendo o restante do arquivo rubocop.yml.

```
59  Rails/SkipsModelValidations:
60    Exclude:
61      - 'db/migrate/*.rb'
62      - 'app/controllers/user_controller.rb'
63
64  Style/BracesAroundHashParameters:
65    EnforcedStyle: context_dependent
66
67  Style/FormatStringToken:
68    EnforcedStyle: template
69
70  Style/IfInsideElse:
71    Enabled: false
72
73  Style/GlobalVars:
74    Exclude:
75      - 'lib/quad_tile/extconf.rb'
76
77  Style/GuardClause:
78    Enabled: false
79
80  Style/HashSyntax:
81    EnforcedStyle: hash_rockets
82    Exclude:
83      - 'lib/tasks/testing.rake'
84      - 'config/initializers/wrap_parameters.rb'
85
86  Style/StringLiterals:
87    EnforcedStyle: double_quotes
88
89  Style/SymbolArray:
90    EnforcedStyle: brackets
```

Nessas linhas finais o que se faz é excluir alguns arquivos de se checar algumas validações - que estão listadas em: http://guides.rubyonrails.org/active_record_validations.html#skipping-validations. Um outro arquivo (extconf.rb) não é checado quanto ao uso de variáveis globais. E outros dois arquivos não são checados quanto ao uso da sintaxe literal para hash.

Na cop BracesAroundHashParameters, utilizar EnforcedStyle como sendo context_dependent, faz com que o último parâmetro não tinha chaves em torno dele, a menos que do segundo ao último parâmetro sejam hash literais.

Nas linhas 67 e 68, segue-se o padrão exemplificado a seguir (bad representa o código ofensivo e good é o código com o melhor estilo).

```
# bad
format('%{greeting}', greeting: 'Hello')
format('%s', 'Hello')

# good
```

```
format('%<greeting>s', greeting: 'Hello')
```

Na cop das linhas 70 e 71 dá-se preferência para o *elsif* em relação a um segundo *if* dentro de um *else*.

Em GuardClauses, dá-se preferência ao uso de *guard clause* em relação ao condicional para se testar se deve ou não haver exceção.

Em HashSyntax, checa-se a sintexe de hash literal, e no caso é mandatório o uso de *hash rockets* para todos os *hashes*, exceto para dois dos arquivos do projeto.

Ao final, StringLiterals utilizando-se double_quotes faz com se utilize aspas duplas em vez de aspas simples. Em SymbolArray, ao se escolher brackets, utiliza-se a sintexe de colchetes (sem o símbolo "%") para expressar *arrays* de forma literal. Segue o exemplo

```
# good
[:foo, :bar, :baz]

# bad
%i[foo bar baz]
```

Esses são os principais destaques dos padrões utilizados no projeto, mas é sempre possível revisitá-los com mais detalhes a partir da documentação do Ruby on Rails e do RuboCop. Lembrando que a seguir está apresentada a análise estática, o que dá mais clareza e mostra na prática os padrões explorados até então.

Análise estática automatizada

Como explicado anteriormente, o projeto Gislene se utiliza do padrão OpenStreetMaps, que é um projeto open source. Originalmente, tal projeto já se utiliza de algumas ferramentas para análise de código fonte e uma delas e o RuboCop.

RuboCop é um análisador de código estático para Ruby que, por padrão, busca garantir que os *guidelines* estabelecidos pelo Guia de Estilo de Ruby sejam seguidos. Pode-se encontrar documentação sobre a ferramenta em https://github.com/rubocop-hq/rubocop .

Assim, decidiu-se por continuar utilizando o RuboCop como ferramenta de análise estática automatizada no projeto Gislene, de forma a aproveitar o trabalho feito anteriormente.

Basicamente, dois são os arquivos mais importantes para entender o funcionamento do RuboCop no projeto Gislene: os arquivos “**.rubocop.yml**” e “**.rubocop_todo.yml**”.

O arquivo “**.rubocop.yml**” contém os padrões procurados pelo RuboCop no código. Já o arquivo “**.rubocop_todo.yml**” basicamente exclui alguns arquivos da busca do RuboCop, funcionando como uma lista de tarefas para possíveis colaboradores do projeto OSM. Ademais, nesse documento mostra-se quantas infrações às regras estabelecidas existem em cada documento ignorado pela busca.

Dado que o arquivo “**.rubocop.yml**” foi explorado anteriormente, abaixo, somente uma fração do documento “**.rubocop_todo.yml**”:

```
# Offense count: 1
# Cop supports --auto-correct.
# Configuration parameters: EnforcedStyle, SupportedStyles, IndentationWidth.
# SupportedStyles: aligned, indented
Layout/MultilineOperationIndentation:
  Exclude:
    - 'lib/bounding_box.rb'

# Offense count: 34
Lint/AmbiguousOperator:
  Exclude:
    - 'test/controllers/amf_controller_test.rb'
    - 'test/controllers/changeset_controller_test.rb'
    - 'test/lib/bounding_box_test.rb'
    - 'test/lib/country_test.rb'

# Offense count: 124
Lint/AmbiguousRegexpLiteral:
  Enabled: false

# Offense count: 32
# Configuration parameters: AllowSafeAssignment.
Lint/AssignmentInCondition:
  Exclude:
    - 'app/controllers/application_controller.rb'
    - 'app/controllers/geocoder_controller.rb'
    - 'app/controllers/notes_controller.rb'
    - 'app/controllers/trace_controller.rb'
    - 'app/controllers/user_controller.rb'
    - 'app/controllers/user_preference_controller.rb'
    - 'app/helpers/application_helper.rb'
    - 'app/helpers/browse_helper.rb'
    - 'app/models/client_application.rb'
    - 'app/models/notifier.rb'
    - 'lib/nominatim.rb'
    - 'lib/osm.rb'
    - 'script/deliver-message'
```

Por simplicidade, o grupo decidiu não corrigir nenhum dos arquivos que haviam sido ignorados no projeto OSM opensource, dado que eles não são de autoria do grupo e existem

maiores prioridades no projeto. Abaixo, entretanto, veremos o processo adotado para identificar e corrigir as imperfeições contidas nos arquivos gerados pelo grupo.

Inicialmente, instalou-se a *gem rubocop*, utilizando-se do comando abaixo:

```
$ gem install rubocop
```

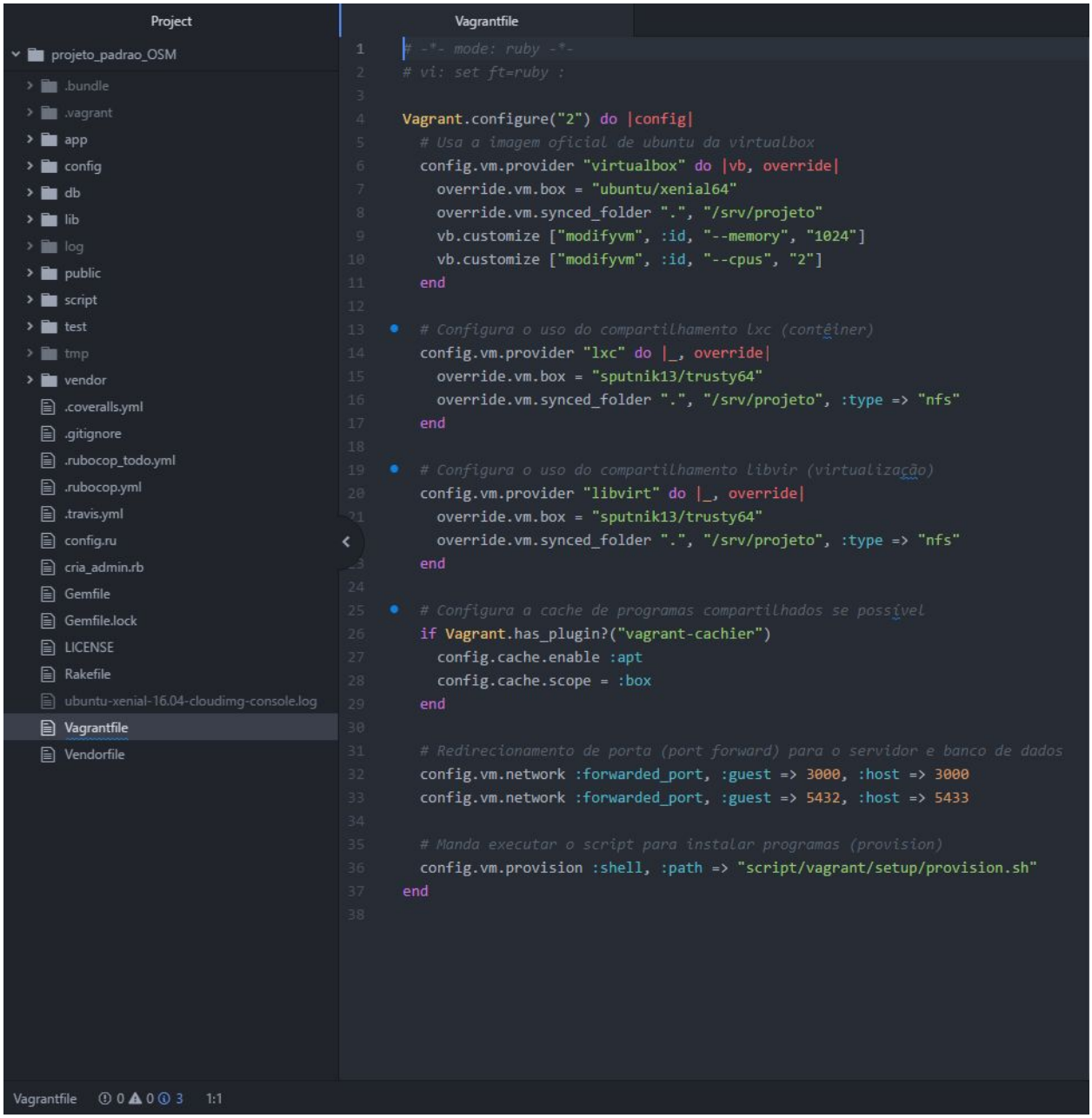
Uma vez instalado, para que se o RuboCop faça a busca nos arquivos não ignorados, basta executar o comando `rubocop` na pasta raiz do projeto. A imagem abaixo mostra quais as contravenções às regras estabelecidas foram encontradas em uma dada execução do comando:

```
C:\Users\Avell 1513\CES29_Gislene\projeto_padrao_OSM>rubocop
Inspecting 37 files
...C.................................

Offenses:

Vagrantfile:13:50: C: Style/AsciiComments: Use only ascii symbols in comments.
  # Configura o uso do compartilhamento lxc (contêiner)
                                                ^
Vagrantfile:19:59: C: Style/AsciiComments: Use only ascii symbols in comments.
  # Configura o uso do compartilhamento libvir (virtualização)
                                                          ^^
Vagrantfile:25:58: C: Style/AsciiComments: Use only ascii symbols in comments.
  # Configura a cache de programas compartilhados se possível
                                                        ^


37 files inspected, 3 offenses detected

C:\Users\Avell 1513\CES29_Gislene\projeto_padrao_OSM>cd ..
```

No exemplo acima, por exemplo, ele encontra três ofensas relacionadas à utilização de caracteres não ascii em comentários.

Para facilitar a produção de código seguindo os padrões definidos, ou corrigir erros encontrados pelos RuboCop, uma das melhores maneiras é integrar as análises estáticas a um editor de texto ou IDE. No caso do grupo, decidiu-se por utilizar para isso o editor de texto Atom, uma vez que é completamente gratuito. Para isso, basta seguir tutorial encontrado no link a seguir: https://www.escoladeecommerce.com/artigos/ruby/ .

Na imagem abaixo, podemos ver os erros encontrados na imagem acima sendo informados em tempo real no editor Atom.



Na imagem acima, os pontos azuis ao lado da numeração das linhas 13, 19 e 25 indicam alertas do RuboCop, e no canto inferior esquerdo, temos o número de alertas e erros encontrados pela ferramenta. Assim, podemos corrigir tais erros durante a própria programação, tornando o alinhamento aos padrões ainda mais fácil.

Observemos que existem três tipos de alertas, com diferentes gravidades. Buscou-se corrigir os mais graves, aqueles meramente estéticos, e que não prejudicam a leitura do código receberam uma baixa prioridade de correção. Por exemplo, na imagem abaixo, podemos ver

um ponto amarelo na linha 9, informando a infração cometida ao não se alinhar o *end* com o respectivo *def.* Tal ofensa é bem mais grave que aquelas assinaladas pelos pontos azuis, e foram o principal alvo das correções feitas pelo grupo.

```
1   class ObjetoValue < ApplicationRecord :: Model
2     #Representa as caracteristicas do Objeto
3     #Colecao de Property
4     has_many :propertys
5
6     #metodo para adicionar property em propertys
7     def << (property)
8       @propertys << property
9           end
10  end
```

O processo de busca e correção pode ser feito em cada uma das pastas do projeto, de forma a cobrir todos os arquivos não excluídos da busca.

Uma vez corrigidas as infrações indicadas, ao se executar o comando *rubocop* na pasta em questão, obtemos o seguinte resultado.

```
C:\Users\Avell 1513\CES29_Gislene\projeto_padrao_OSM>rubocop
Inspecting 37 files
.....................................

37 files inspected, no offenses detected

C:\Users\Avell 1513\CES29_Gislene\projeto_padrao_OSM>
```

No período em que esse relatório é escrito, ainda estão sendo escritas partes do código fonte autoral, logo ainda ocorreram algumas iterações da análise estática e da correção das infrações, mas a maior parte das correções já foi feita.

Dessa forma, vemos que a maior parte da Análise Estática foi reaproveitada do projeto opensource OSM, entretanto mantendo-a conseguiu-se continuar com um padrão elevado de código. Vê-se então a importância da utilização dessas ferramentas e o fato de a análise estática realmente ser uma prática adotada no desenvolvimento profissional. Logo, o aprendizado na prática de ferramentas do gênero colabora fortemente para o crescimento profissional dos alunos.