Algoritmo avalidor de código-fonte Racket

Felipe Diniz Tomás, 110752

[1]Departamento de informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

Paradigma de Programação Lógica e Funcional – 6902

Ra110752@uem.br

04, Maio 2021

1. Introdução

A avaliação de código é um processo árduo e não trivial que muitas vezes consome muito tempo quando feita manualmente. Esse processo necessita de uma análise de todas as linhas do código-fonte, aplicando métricas para avaliar o mesmo. O estudo de uma codificação pode ser complexa, já que obrigatóriamente o analisador deve ter um conhecimento aprofundado na linguagem, para identificar por exemplo, comentários, definição de funções e variáveis, etc...

Quando se trata da avaliação, é necessário estabelecer métricas que serão utilizadas como parâmetros para análise do código-fonte. Defini-las não é uma tarefa fácil, é necessário encontrar métricas padronizadas, as quais podem fornecer uma verificação precisa.

Dado tal contexto, este trabalho tem como objetivo realizar uma avaliação de um ou mais arquivos de código-fonte na linguagem Racket. Para que essa avalição seja feita, foi necessário pesquisar métricas para serem aplicadas e selecionados códigos com o mesmo propósito para compará-los. Por fim também foi realizado testes unitários para cada função.

2. Métricas

A ISO/IEC 9126[1], reunida agora na ISO/IEC 25000[2], apresenta características de qualidade e um guia para o uso dessas características, de forma a auxiliar e padronizar o processo de avaliação de qualidade de produtos de software. Métricas de código fonte caracterizam bem o produto de software em qualquer estado do seu desenvolvimento. Existem vários tipos de métricas de código fonte. Entre as principais, temos métricas de tamanho, quantidade de funções (*define*), quantidade de imports (*requires*), quantidade de linhas comentadas, quantidade de testes e quantidade de linhas maiores que 80 caracteres.

A métrica de tamanho pode ser utilizada para avaliar a quantidade de código fonte, como por exemplo Linhas de código - *Lines of Code* (LOC). Manter o monitoramento de métricas de volume de código em conjunto com outras métricas é importante para que comparações sejam feitas para sistemas de tamanho semelhante.

---

[1] https://pt.wikipedia.org/wiki/ISO/IEC_9126

[2] https://pt.wikipedia.org/wiki/ISO/IEC_25000

A métrica de quantidade de definições (funções), é um parâmetro que contribui de certa forma a análise de organização do código, já que um uso maior de funções denota-se uma maior fragmentação de funcionalidades aplicada ao código. Por exemplo, se considerarmos um sistema complexo mas que possui poucas funções, conclui-se que a organização e fragmentação do código não foi das melhores.

A métrica de quantidade de requires (imports) contribui para análise de depências externas e organização do código-fonte. Ou seja, se um código possui muitos imports, denota-se que necessita de muitas bibliotecas externas ou arquivos externos (que podem ser arquivos do próprio autor ou de terceiros) para executar seu propósito, isso enfere em sua organização, já que o sistema pode estar fragmentado em vários arquivos.

A métrica de quantidade de linhas comentadas, é um parâmetro que contribui para a avaliação de legibilidade do código e é uma boa prática, já que muitos comentários indica uma preocupação do programador em tornar seu código mais legível, explicando o que determinada lógica empregue na implementação.

A métrica de quantidade de testes unitários, oferece uma análise sobre a boa prática e coesão do código fonte, já que indica que houveram testes de funções e portanto as mesmas estarão com seus resultados testados.

Por fim a métrica de quantidade de linhas maiores que 80 caracteres é um dos fatores que verifica se o código possui uma boa legibilidade. Apesar do ambiente Dr Racket ter um limete de caracteres por linha de 120, em sua documentação não é descartado o uso de 70 a 80 caracteres por linha[3]. 80 caracteres é um bom número por vários motivos: impressão de código em modo de texto, exibição de código em tamanhos de fonte razoáveis, comparação de várias partes diferentes de código em um monitor, etc.

3. Implementação

Todo o código está comentado seguindo a receita de projeto, portanto cada função está bem explicada no próprio código-fonte. Sendo assim, nessa seção darei uma visão geral de como cada métrica foi implementada.

De ínico é necessário carregar os arquivos para avaliação, o racket possui a biblioteca *batch-io*[4], que será útil. Essa biblioteca tem a definição *read-lines* que é aplicavel ao nosso problema, tornando possível ler cada linha do arquivo. Para carregar todos os arquivos de um diretório a função *directory-list* padrão do próprio racket foi usada.

Para implementar a métrica de quantidade de teste unitários, foi verificado primeiro se o arquivo possui a biblioteca *rackunit*[5]caso possua, começa a verificar cada linha do arquivo. Se a linha não for um comentário, verificará se na linha existe alguma

[3] https://docs.racket-lang.org/style/Textual_Matters.html

[4] https://docs.racket-lang.org/teachpack/2htdpbatch-io.html

[5] https://docs.racket-lang.org/rackunit/api.html

palavra reservada para teste (presentes na biblioteca *rackunit).* Caso houver contará a linha.

Para implementar a métrica de quantidade de definições (*define*), foi iterado linha por linha do arquivo, aquelas que não são comentários, passam por uma verificação se contém "(*define"* em seu corpo. Caso houver contará a linha.

Na implementação da métrica de quantidade de *requires* (imports), foi iterado linha por linha do arquivo, aquelas que começam com "(*require*" serão contadas.

Para implementar a métrica de quantidade de linhas maiores que 80 caracteres, foi verificado linha por linha do arquivo, e contada aquelas com tamanho maior que 80 caracteres.

Na implementação da métrica de quantidade de comentários, foi iterado linha por linha do arquivo, verificando se contém o símbolo reservado que indica comentário. O símbo pode estar em qualquer parte da linha. Caso houver contará a linha.

Para implementar a métrica de quantidade de linhas de código, foi iterado linha por linha do arquivo, desconsiderando linhas de comentários e linahs vazias, contando as demais.

3.1 Pesos das métricas

A distribuição de peso das métricas foi definida da seguinte forma:

- Quantidade de linhas de código: 0,1 pontos por linha.
- Quantidade de testes unitários: 0,4 pontos por teste.
- Quantidade de definições (funções): 0,25 pontos por definição.
- Quantidade de requires : 0,2 pontos por requires.
- Quantidade de linhas maiores que 80 caracteres: - 0,25 pontos por linha.
- Quantidade de comentários: 0,3 pontos por comentário.

3.2 Testes unitários

Foram implementados um total de 9 testes unitários, sendo eles para as funções que executam cada métrica, função da pontuação final e demais funções úteis. Os testes são executados automaticamente quando o código é rodado.

5. Casos de teste

Foi selecionado três códigos-fontes racket da internet, que implementam jogo flappy bird. Os aquivos estão disponíveis na pasta *files*, é neste diretório que os códigos-fontes a serem avaliados devem ficar. São eles:

- Exemplo – 1.rkt[6]
- Exemplo – 2.rkt[7]
- Exemplo – 3.rkt[8]

[6] https://github.com/Zhenya750/FlappyBird/

[7] https://github.com/kkspeed/racket-flappy

Os três códigos fontes apesar de terem o mesmo objetivo se diferem bastante em sua composição, portanto irão garantir uma análise variada sobre as métricas.

6. Resultados

A seguir podemos conferir os resultados de cada arquivo considerando cada métrica aplicada e sua respectiva pontuação. Figura 1, indica os resultados do arquivo "Exemplo-1.rkt". A Figura 2, indica os resultados do arquivo "Exemplo-2.rkt". A Figura 3, indica os resultados do arquivo "Exemplo-3.rkt"

```
Nome do arquivo: "Exemplo - 1.rkt"

• Quantidade de linhas que contêm código:         177 linhas.
• Quantidade de testes unitários no código:       2 testes.
• Quantidade de definições no código:             24 definições.
• Quantidade de requires no código:               4 requires.
• Quantidade de linhas maiores que 80 caracteres: 0 linhas.
• Quantidade de comentários no código:            49 comentários.

Pontuação: 40 pontos
```

Figura 1. Resultado do arquivo "Exemplo-1.rkt".

```
Nome do arquivo: "Exemplo - 2.rkt"

• Quantidade de linhas que contêm código:         100 linhas.
• Quantidade de testes unitários no código:       0 testes.
• Quantidade de definições no código:             32 definições.
• Quantidade de requires no código:               1 requires.
• Quantidade de linhas maiores que 80 caracteres: 2 linhas.
• Quantidade de comentários no código:            3 comentários.

Pontuação: 18.6 pontos
```

Figura 2. Resultado do arquivo "Exemplo-2.rkt".

```
Nome do arquivo: "Exemplo - 3.rkt"

• Quantidade de linhas que contêm código:         154 linhas.
• Quantidade de testes unitários no código:       0 testes.
• Quantidade de definições no código:             51 definições.
• Quantidade de requires no código:               4 requires.
• Quantidade de linhas maiores que 80 caracteres: 41 linhas.
• Quantidade de comentários no código:            99 comentários.

Pontuação: 48.4 pontos
```

Figura 3. Resultado do arquivo "Exemplo-3.rkt".

---

[8] https://github.com/Alexapostol2000/Flappy-Bird

7. Análise dos resultados

7.1. Quantidade de linhas de código

Houve uma variação considerável do número de linhas de código em cada arquivo. No Gráfico 1, percebe-se que o "Exemplo-1.rkt" foi o arquivo que teve a maior quantidade de linhas de código com 177 linhas, recebendo a maior pontuação de 17,7 pontos.

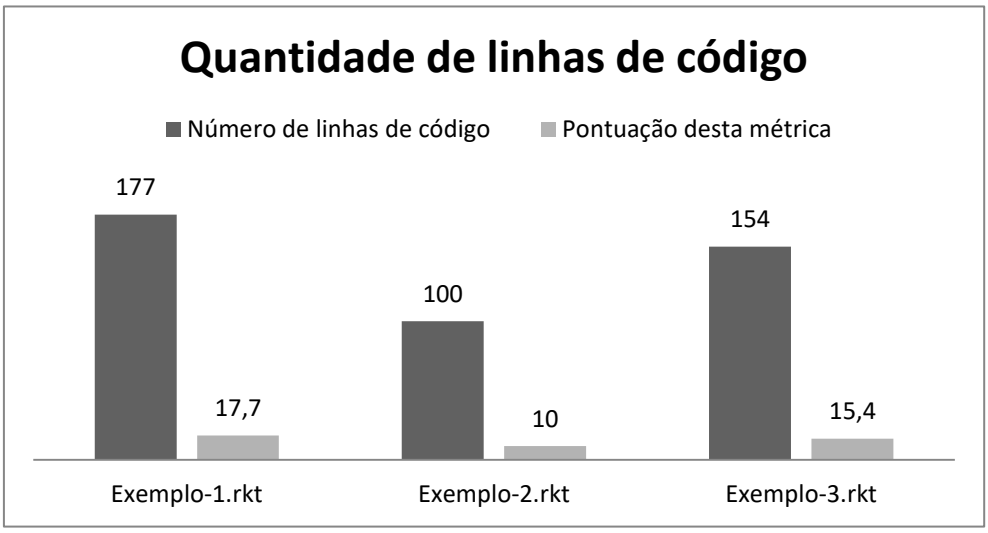

Gráfico 1. Resultado da métrica quantidade de linhas de código.

7.2. Quantidade de testes unitários

Não houveram muitos testes unitários nos arquivos. No Gráfico 2, percebe-se que o "Exemplo-1.rkt" foi o arquivo que teve a maior quantidade de testes unitários com 2 testes, recebendo a maior pontuação de 0,8 pontos.

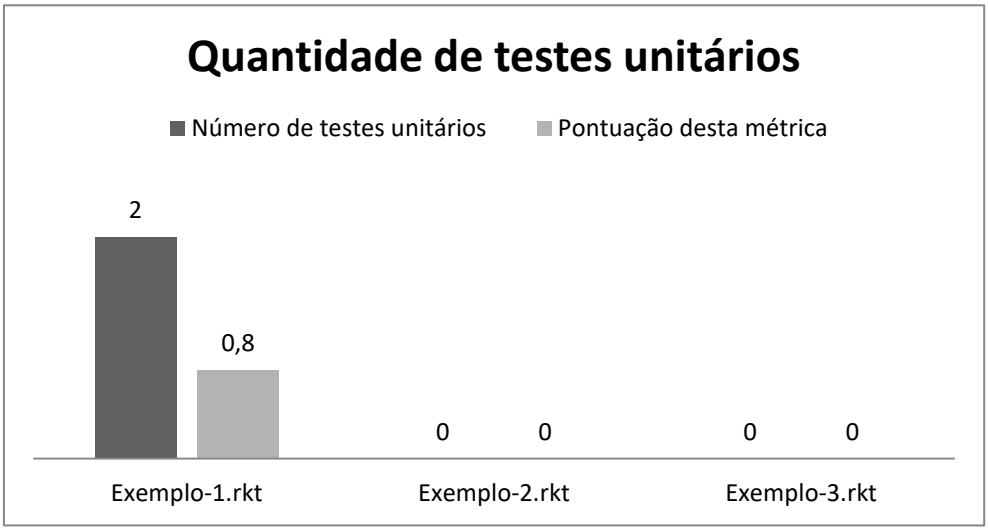

Gráfico 2. Resultado da métrica quantidade de testes unitários.

7.3. Quantidade de definições

Houve uma diferença considerável no número de definições (funções) entre os arquivos. No Gráfico 3, percebe-se que o “Exemplo-3.rkt” foi o arquivo que teve a maior quantidade de definições com 51, recebendo a maior pontuação de 12,5 pontos.

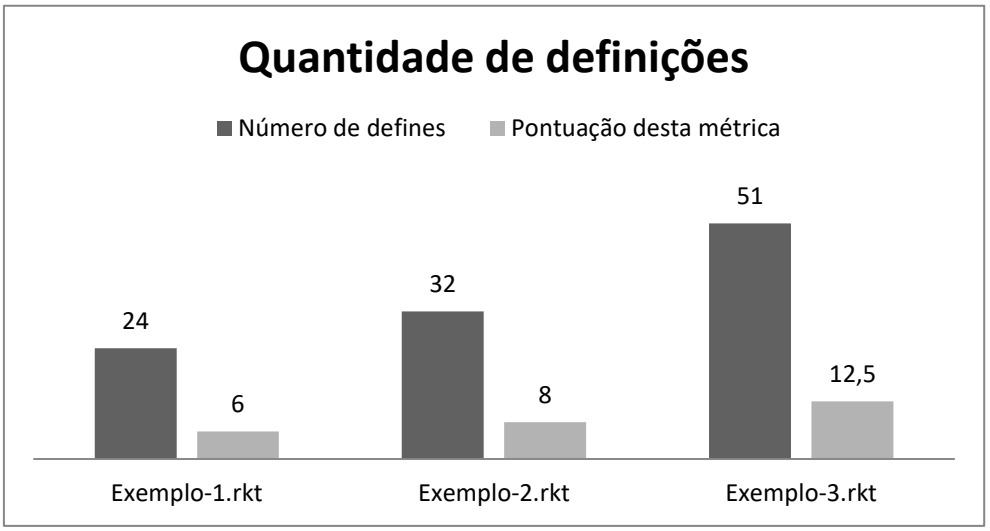

Gráfico 3. Resultado da métrica quantidade de definições.

7.4. Quantidade de requires

Não houve uma diferença grande no número de requires (imports) entre os arquivos. No Gráfico 4, percebe-se que ocorreu um empate entre o “Exemplo-1.rkt” e o “Exemplo-3.rkt” , ambos pontuando 0,8.

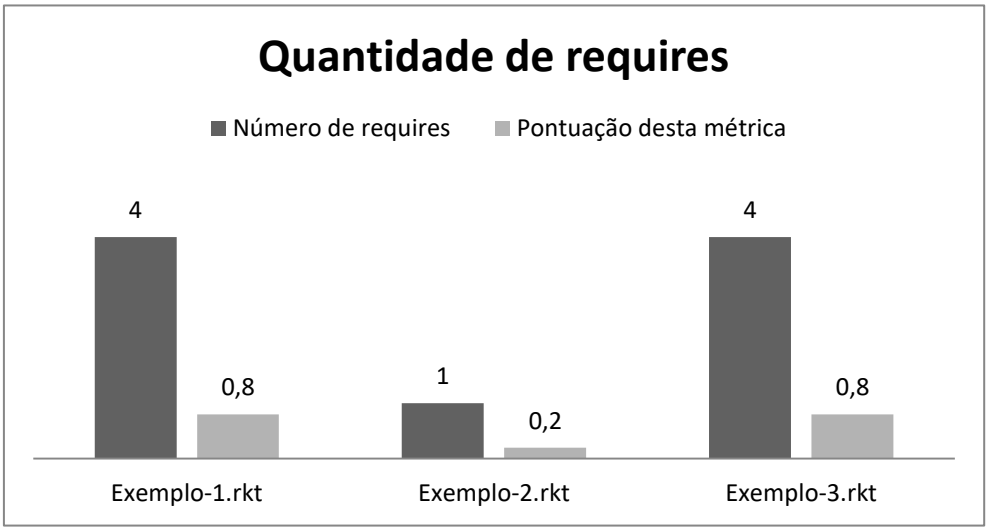

Gráfico 4. Resultado da métrica quantidade de requires.

7.5. Linhas maiores que 80 caracteres

Houve uma diferença grande no número de linhas maiores que 80 caracteres entre os arquivos. No Gráfico 5, percebe-se que o “Exemplo-3.rkt” foi o maior penalizado, perdendo 10,25 pontos.

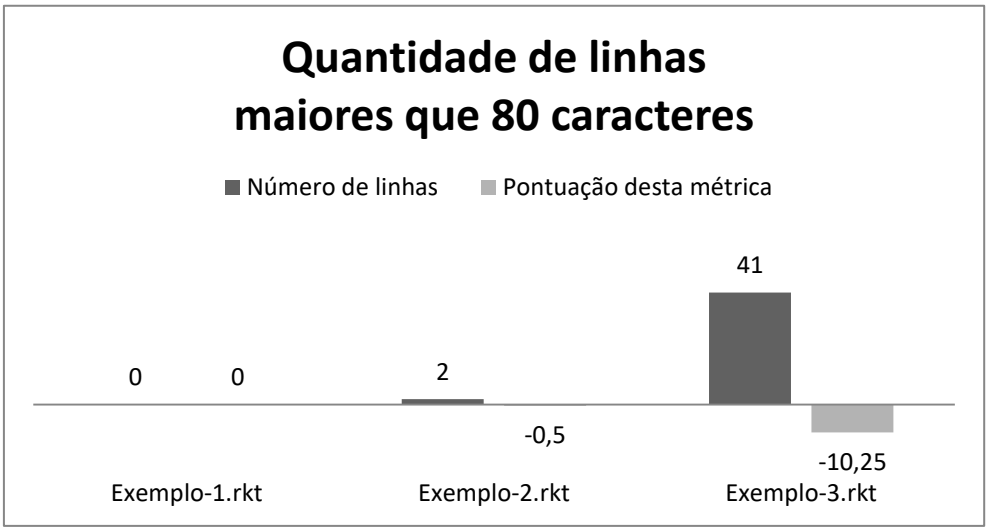

Gráfico 5. Resultado da métrica de linhas maiores que 80 caracteres.

7.6. Quantidade de comentários

Houve uma diferença grande no número de comentários, e foi uma métrica decisiva. No Gráfico 6, percebe-se que o “Exemplo-3.rkt, foi o arquivo com a maior quantidade de comentários.

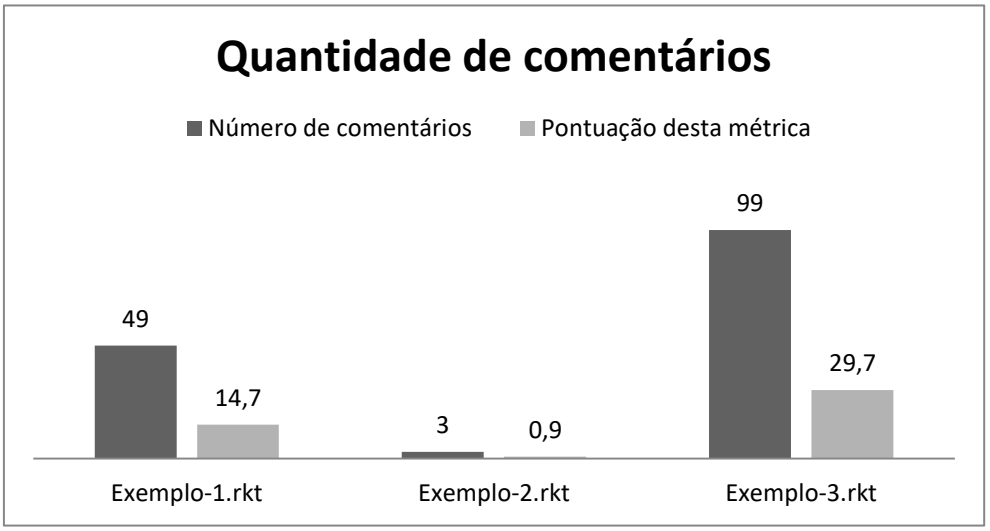

Gráfico 6. Resultado da métrica quantidade de comentários.

7.7. Pontuação final

Sendo assim, o código “Exemplo-3.rkt” foi o código fonte que obteve a maior pontuação final, com 48,4 pontos, seguido do “Exemplo-1.rkt” com 40 pontos, e em último o “Exemplo-2.rkt” com apenas 18,6 pontos.

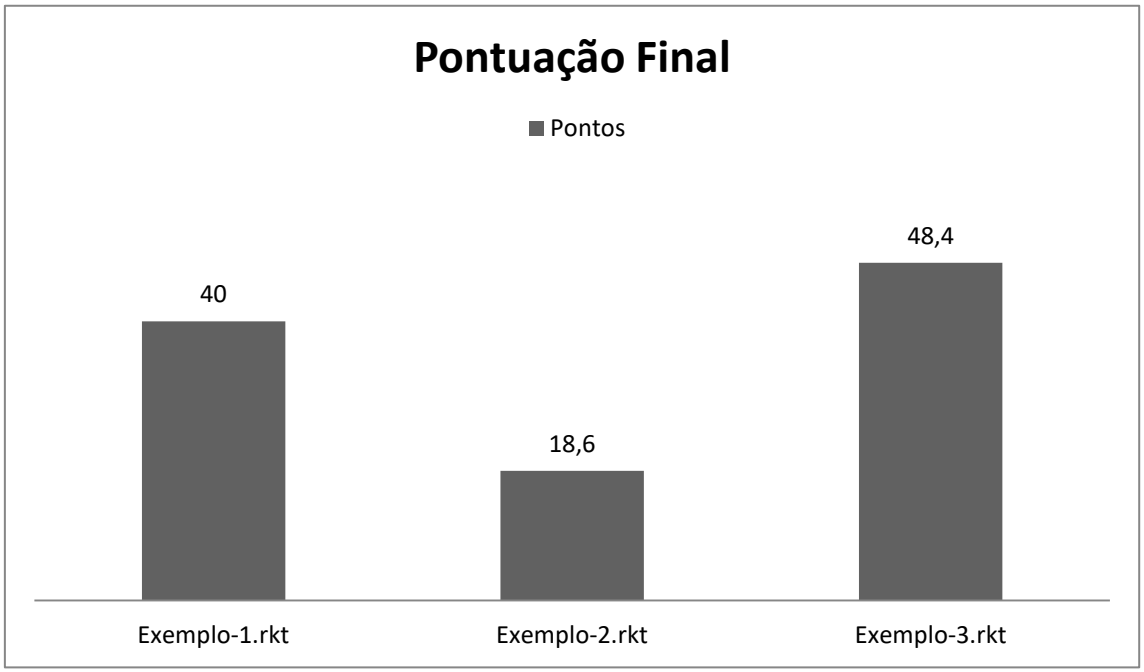

Gráfico 7. Pontuação final.

8. Conclusão

A implementação de um algoritmo de avaliação de código fonte pode proporcionar um entendimento maior de métricas de avaliação e como elas podem ser aplicadas, além de um aprendizado interessante da linguagem racket. É claro, em termos de utilidade, um avaliador de código não é 100% preciso, e um sistema básico como este está longe de substituir uma avaliação manual.

Considerando implementações futuras, poderia ser desenvolvido mais métricas analisando outros fatores no código, além de uma melhor definição de pesos para cada métrica, já que algo fundamental na avaliação.

Referências

Racket Documentation. Autor deconhecido. Disponível em < https://docs.racket-lang.org/>. Acessado em: 05 de maio de 2021.