Testes de Software

Verificação de Código

16/0122180	Geovana Ramos Sousa Silva
15/0011903	Igor Aragão Gomes
13/0122254	Lucas Midlhey Cardoso Naves
15/0078692	Caio César de Almeida Beleza
16/0123186	Guilherme Guy de Andrade
15/0035756	Geovanne Santos Saraiva

Sumário

1. Introdução

Neste trabalho será desenvolvida uma checklist para verificar as conformidades aos padrões de codificação estabelecidos e às boas técnicas de programação usados no projeto DrDown, que foi desenvolvido na disciplina de Metodologias de Desenvolvimento de Software no primeiro semestre de 2018 no framework Django.

O módulo escolhido para a avaliação de código foi o de marcação de consulta, o qual é estruturado em pedidos de marcação e a própria marcação de consulta. Para fins didáticos, apenas o código python será analisado, deixando de lado o código dos templates, ou seja, os códigos HTMLs, CSS e JavaScript. A principal fonte de informação para a elaboração da checklist será a documentação PEP8, a qual foi proposta como padrão de código pela equipe de desenvolvimento, além de artigos que trazem métricas de qualidade de código.

2. Métricas de Código

Para a análise do código, serão utilizadas métricas voltadas para manutenibilidade de código, propostas pelo Software Improvement Group e analisadas por BAGGEN, ROBERT & CORREIA (2011), além das boas práticas de programação propostas pelo padrão PEP8. É importante ressaltar que a métrica de volume não será utilizada, pois no modelo proposto ela faz uso da Produtividade em Ponto de Função, a qual necessita do tempo gasto pelos desenvolvedores na entrega das *"*features*"* e que não é disponibilizado na documentação do DrDown. Segue abaixo, a descrição das métricas e seus subindicadores.

2.1 Redundância

Redundância é a presença de elementos que implementam a mesma funcionalidade em um software, podendo ser com um código diferente, ou com o mesmo código, mas com diferentes parâmetros ou um contexto diferente. A redundância pode estar presente em formas diferentes, desde pequenos blocos de código, até sistemas inteiros. O que define a redundância entre elementos é que tem formas diferentes de execução, mas chegam ao mesmo resultado.

A redundância pode ser introduzida no código por design, já que possui algumas aplicações positivas, como aumentar a tolerância a falhas do sistema. Por outro lado, existem alguns pontos negativos, quando não se entende a magnitude de quantas redundâncias existem no sistema, como a propagação de um mesmo erro, em pedaços de código redundantes.

2.2 Tamanho e Interface da Unidade

Separadamente da complexidade, o tamanho das unidades de qual um sistema é composto, pode mostrar resultados sobre sua manutenibilidade. Intuitivamente, unidades maiores tendem a ser mais difíceis de realizar manutenção, pois são mais difíceis de analisar e testar.

Existe uma forte correlação entre o tamanho e a complexidade ciclomática. Utilizar o tamanho de uma unidade, para completar a análise de complexidade ciclomática permite conclusões que podem ajudar uma equipe a repensar se aquela unidade pode ser separada em pequenas partes, aumentando a capacidade de análise, teste e manutenção do mesmo.

Para medir o tamanho da unidade, utilizamos o SLOC (Source Lines of Code), que funciona como o LOC (Lines of Code), porém eliminando linhas vazias de sua contagem.

Então, teremos 4 micro serviços do software *DrDown* que serão considerados como uma unidade, na qual, cada unidade terá as métricas de SLOC e parâmetros levantados. Os arquivos analisados de cada unidade, serão dos tipos: model, form, view e urls.

2.3 Complexidade

A complexidade mede a facilidade de entendimento do código e se a solução proposta é a mais simples possível, envolvendo um menor número de linhas de execução. O cálculo dessa métrica que é referência foi desenvolvido por MCCABE (1976). Porém, algumas bibliografias mais recentes dividem a complexidade em duas vertentes, pois nem sempre um código com boa complexidade terá a solução mais compreensiva.

Complexidade Cognitiva

Essa complexidade mede o quanto o código é intuitivo para um *ser humano.* O cálculo descrito por CAMPBELL (2016-2018) é:

a. Some um para cada quebra de fluxo linear (while, if, for ..)
b. Some um quando estruturas de quebra de fluxo estão aninhadas

A soma dos resultados dos dois itens corresponde a complexidade cognitiva total.

Complexidade Ciclomática
Mede o quão difícil será testar um código executado pelo *computador.* Aqui se encontra a proposta de MCCABE (1976), a qual faz um grafo que é a representação gráfica do código:

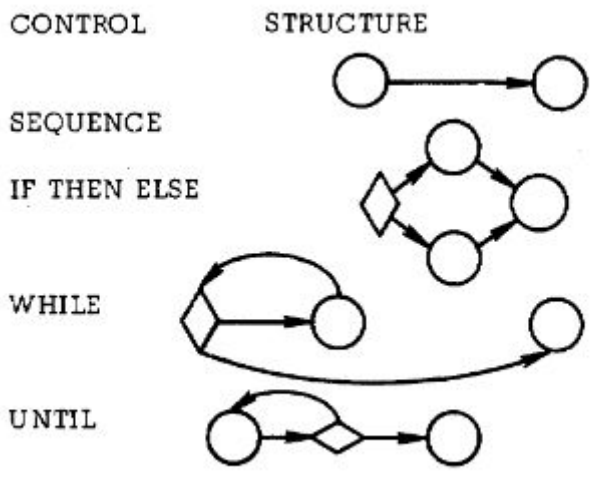
E o valor matemático da complexidade é dado por:

$$V(G) = A - N + 2$$

ou

$$V(G) = C + 1$$

Onde:
G -> Grafo
A -> número de arestas
N -> número de nós
C - > nós condicionais (Ex: if)



Levando isso em consideração e tendo como base as duas referências e os cálculos por elas apresentados, as questões 1 e 2 foram inseridas na checklist de avaliação.

2.4 Acoplamento

Um sistema com paradigma orientado a objetos possui as classes como os elementos principais de sua estrutura. Essas classes podem ser organizadas de diversas formas, dependendo da arquitetura do sistema e de seus objetivos.

Uma boa arquitetura de classes permite economias consideráveis nos gastos de recursos com manutenção e desenvolvimento. Um fator que auxilia na medição da qualidade da arquitetura e, por sua vez, na qualidade do produto de software é o nível de acoplamento (DE OLIVEIRA RODRIGUES, 2014).

O acoplamento mede o quanto duas classes dentro de um software orientado a objetos estão conectadas, dependendo uma da outra. Se alguma entidade é muito acoplada aumenta a dificuldade de se fazer modificações no sistema, já que surgem efeitos colaterais em outras partes do software (MEIRELLES, 2013).

Para medir o nível de acoplamento serão utilizadas as métricas definidas por Meirelles em “Monitoramento de métricas de código-fonte
em projetos de software livre”. Onde ele estabelece três métricas relacionadas ao acoplamento:

- *Afferent Connection per Class (ACC)*: mede quantas classes dependem da classe atual. Para cada classe que depende da classe atual é adicionado um valor unitário ao ACC. Sendo que quanto menor o ACC, melhor.
- *Coupling Between Objects (CBO)*: mede o quanto a classe atual depende de outras classes. É o inverso de ACC.
- *Coupling Factor (CFO)*: Medida geral de quão acoplado é o sistema. Utiliza a métrica ACC, citada anteriormente, para ser calculado de acordo com a fórmula:

$$CFO = \frac{\sum_{i=1}^{n} ACC(C_i)}{n^2 - n}$$

> O autor indica que a faixa de 0 até 0,02 é nomeada como *boa*, 0,02 a 0,14 como *regular* e acima de 0,14 como *ruim*.

Para a checklist será calculado o CFO do módulo a ser analisado, desconsiderando classes externas ao módulo. Serão consideradas também apenas dependências entre classes, já que o projeto foi executado na linguagem Python/Django é possível que haja execução de código fora de qualquer classe e estas serão desconsideradas para o cálculo ACC.

Após isso o CFO será comparado em uma das faixas indicadas pelo autor das métricas, sendo considerado resultados desejáveis a classificação *boa* ou *regular*.

2.5 Boas Práticas

Existem boas práticas que atendem não necessariamente a nível de código e sim de equipe que tornam o rendimento da equipe melhor. É possível utilizar várias ferramentas que podem entregar melhor versionamento, nivelar conhecimento entre a equipe, distribuir funções, direcionar conhecimentos, entre outras.
Boas práticas a nível de código são ações praticadas pelos programadores e tem evolução direta no software pois relacionam ao código fonte. O objetivo de se utilizar estas práticas é melhorar a legibilidade do código ou a arquitetura do mesmo.
Utilizar boas práticas eleva a qualidade do código na busca de melhor compreensão de outros desenvolvedores, além de facilitar em outras fases como, aquisição, codificação, testes e integração com o repositório. Desta forma o profissional levará muito menos tempo para evoluir e corrigir problemas ou erros que porventura podem ocorrer.

Um código fonte ideal se relaciona com fácil coesão, não possui duplicidade, eficiência e é direto em suas dependências entre outras.

Adotar apenas um idioma

O código para ser de fácil entendimento deve ser escrito em apenas um idioma e que todos os desenvolvedores tenham conhecimento para facilitar o entendimento dos mesmos. É notório também salientar que há várias bibliotecas, as mais utilizadas no meio são em inglês, logo acaba se tornando um padrão e o mais recomendado.

Indentação

Um software de fácil compreensão segue um padrão de indentação afim de tornar mais fácil a leitura do código. É necessário definir um padrão entre os desenvolvedores, a fim de melhorar o aspecto visual.

Nomes consistentes

Deve ser possível entender basicamente o que a função, variável, classe, constante atua. É muito interessante comentar o código, porém comentarios demais poluem o mesmo e contendo informações desnecessárias. Desta forma comentar para que serve tal função não é uma boa prática.

Código objetivo

Esta boa prática explica o motivo de código funcional porém desnecessário não devem existem no código fonte. Dificulta o entendimento por parte da equipe de desenvolvimento pois a mesma se atenta a algo que não persiste no código.

Legibilidade de Código

Classes, funções, métodos bem escritos demonstram boa prática, representam objetividade com suas características e comportamentos, desta forma respeitam o relacionamento entre elas tornando o código limpo e de fácil entendimento.

Processamentos Objetivos

É importante evitar processos desnecessários para se criar um produto de qualidade, Desta forma o tempo de execução e a sobrecarga do dispositivo utilizado tornam o software usual.

3. Checklist

Número	Pergunta	Métrica	Referência
1	O código possui complexidade cognitiva abaixo de 5?	Complexidade	[2]

2	O código possui complexidade ciclomática abaixo de 5?	Complexidade	[9]
3	A classificação de CFO é regular ou boa?	Acoplamento	[8]
4	O código está em apenas um idioma?	Boas práticas	[10]
5	O código tem um padrão de recuo?	Boas práticas	[10]
6	O código tem limite por linha de 79 caracteres?	Boas práticas	[10]
7	Os nomes das funções, classes e métodos estão padronizados?	Boas práticas	[10]
8	Existe duplicidade de código?	Redundância	[3], [4]
9	O nível de avaliação de risco do código é menor ou igual a 10 (classificação simples)?	Tamanho da unidade	[7]
10	O rank do código é superior ao rank ‘+’?	Tamanho da unidade	[7]
11	O código tem quantidade de linhas equilibradas por unidade?	Tamanho da unidade	[7]

4. Avaliação

4.1 Arquivos Avaliados

Nome	**Número de Linhas**
appointments_form.py	25
requests_form.py	38
model_appointment.py	84
model_request.py	155
view_appointment.py	233
view_request.py	160
test_model_appointment.py	125

test_model_request.py	131
test_view_appointment.py	285
test_view_request.py	487
admin.py	7
apps.py	6
urls.py	89
Total	1386

4.2 Unidades Avaliadas

4.2.1 SLOC (Source lines of code)

Medical Records	
Nome do arquivo	**SLOC (Source lines of code)**
Forms	63
Models	366
Views	422
Urls	93
Total	**944**

Users	
Nome do arquivo	**SLOC (Source lines of code)**
Forms	8
Models	718
Views	171
Urls	47
Total	**944**

Forum

Nome do arquivo	**SLOC (Source lines of code)**
Forms	14
Models	100
Views	198
Urls	54
Total	**366**

Appointments	
Nome do arquivo	**SLOC (Source lines of code)**
Medical Records	51
Users	193
Views	316
Urls	86
Total	**646**

No total da amostra, temos 2900 SLOC.

Medidas anteriores nos mostram que o nível de complexidade ciclomática do código, não passa de 5, portanto temos que o código quase não apresenta riscos, seguindo a tabela abaixo:

CC	Risk evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
> 50	untestable, very high risk

Respondendo a pergunta 9: O código é 100% simples, não havendo riscos maiores.

Em relação à pergunta 10, tendo em base a resposta da pergunta 10, e a tabela abaixo, chegamos à seguinte conclusão: O ranking do código é ++.

	maximum relative LOC		
rank	moderate	high	very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
--	-	-	-

Ao calcular a quantidade de SLOC por unidade, temos:

Unidade	**SLOC (unidade) / SLOC (total)**
Medical Records	944 / 2900 = ~0,325
Users	944 / 2900 = ~0,325
Fórum	366 / 2900 = ~0,126
Appointments	646 / 2900 = ~0,224

O desvio padrão das unidades amostradas, é aproximadamente 9,54.

Portanto, a resposta da pergunta 11 é: Sim, a quantidade de linhas por unidade é equilibrada.

4.2.3 Parâmetros

Medical Records	
Nome do arquivo	**Número de parâmetros**
Forms	20
Models	48
Views	101
Urls	42
Total	**211**

Users

Nome do arquivo	**SLOC (Source lines of code)**
Forms	1
Models	126
Views	25
Urls	24
Total	**176**

Forum	
Nome do arquivo	**SLOC (Source lines of code)**
Forms	6
Models	55
Views	33
Urls	27
Total	**121**

Appointments	
Nome do arquivo	**SLOC (Source lines of code)**
Forms	7
Models	57
Views	56
Urls	39
Total	**159**

4.3 Coupling Factor (CFO)

Classe	**ACC**
AppointmentsConfig	0

RequestFilter	1
RequestListView	0
RequestCreateView	0
RequestUpdateView	0
RequestDeleteView	0
RequestUpdateStatusView	0
RequestAfterResultDeleteView	0
AppointmentFilter	1
AppointmentListView	1
AppointmentCreateView	0
AppointmentMonthArchiveView	0
AppointmentUpdateView	0
AppointmentUpdateStatusView	0
AppointmentFromRequestCreateView	0
TestViewAppointment	0
TestViewRequest	0
TestModelRequest	0
TestModelAppointment	0
Appointment	8
AppointmentRequest	11
Total	22

Seja *n* = 21 (número de classes avaliadas):

$$CFO = \frac{\sum_{i=1}^{n} ACC(C_i)}{n^2 - n}$$

$$CFO = \frac{22}{21^2 - 21}$$

$$CFO = 0,05238095238$$

Conclusão: O CFO pertence à faixa *regular*.

4.4 Boas Práticas

Em cordialidade com a pep8 analisamos o código e sugerimos essas métricas.

Idioma	Inglês
Indentação	4 recuos por linha de continuação
Indentação	79 caracteres por linha
Nomes consistentes	convenção capsWord

Todas as métricas foram analisadas manualmente, percebendo se haviam seguido os padrões impostos.

4.5 Resultado

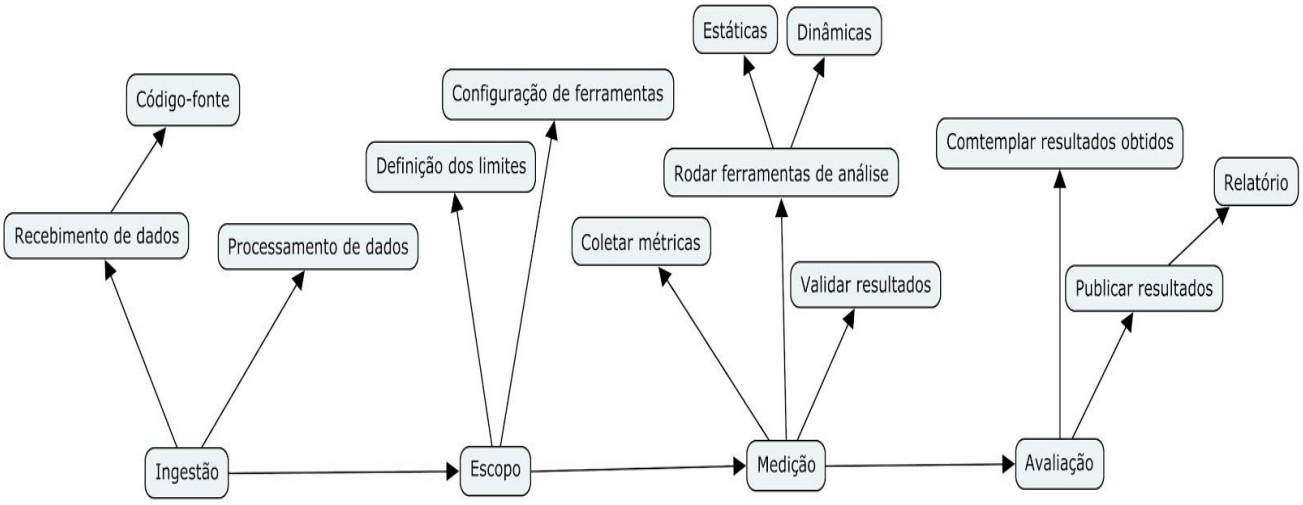
Número da Pergunta	**Porcentagem de Linhas que Atendem**
1	97.98%
2	100%
3	100%
4	100%
5	100%
6	100%
7	100%
8	~0.02%
9	100%

10	100%
11	100%

5. Mapa Conceitual

O seguinte mapa conceitual representa o processo de análise de um produto de software, feito baseado na teoria descrita por Baggen em Standardized code quality benchmarking for improving software maintainability.

Dando detalhes de todas suas etapas até o momento final em que se é gerado um artefato como um relatório.



6. Referências

[1] BAGGEN, ROBERT & CORREIA, José Pedro & Schill, Katrin & Visser, Joost. (2011). **Standardized code quality benchmarking for improving software maintainability**. Software Quality Journal. 20. 1-21. 10.1007/s11219-011-9144-9.

[2] CAMPBELL, G. Ann. **Cognitive Complexity. A new way of measuring understandability**. SonarSource S.A., 2016-2018, Switzerland.

[3] CARZINGA, A.; MATTAVELLI, A.; PEZZE, M. **Measuring software redundancy.** IEEE/ACM 37th IEEE International Conference on Software Engineering. 2015.

[4] CHI, D.-H., LIN, H.-H., & KUO, W. (n.d.). **Software reliability and redundancy optimization. Proceedings.**, Annual Reliability and Maintainability Symposium. doi:10.1109/arms.1989.49570

[5] DE OLIVEIRA RODRIGUES, Bruno Rafael; DE SOUZA, Daniel Edilson; FIGUEIREDO, Eduardo Magno Lages. Medindo Acoplamento em Software Orientado a Objeto: Uma Perspectiva do Desenvolvedor. **Abakós**, v. 3, n. 1, p. 3-17, 2014.

[6] GUY, Guilherme; MAGALHÃES, Elias; MAIKE, Daniel; MEDEIROS, Gabriela; RAMOS, Geovana; ROGERS, Joberth **Dr.Down**: Módulo de Consultas. Disponível em: <https://github.com/fga-eps-mds/2018.1-Dr-Down/tree/develop/drdown/appointments>. Acesso em: 27 abr. 2019

[7] HEITLAGER, I.; KUIPERS, T.; VISSER, J. **A practical model for measuring maintainability.** In 6th international conference on the quality of information and communications technology. IEEE Computer Society. pp. 30–39. 2007.

[8] MEIRELLES, Paulo Roberto Miranda. **Monitoramento de métricas de código-fonte em projetos de software livre**. 2013. Tese de Doutorado. Universidade de São Paulo.

[9] MCCABE, T. J. **A Complexity Measure. IEEE Transactions on Software Engineering**. vol. 2. no. 4. pp. 308-320, 1976.

[10] VAN ROSSUM, Guido; WARSAW, Barry; COGHLAN, Nick. **PEP 8**: Style Guide for Python Code. Disponível em: <https://www.python.org/dev/peps/pep-0008/>. Acesso em: 27 abr. 2019.

7. Tabela de Contribuição

Aluno	**Tópicos**	**Referências**
Geovana	Introdução; Métricas de Código; Complexidade; Arquivos Avaliados;	[1], [2], [9], [10]

Guilherme	Acoplamento; Coupling Factor (CFO); Mapa conceitual;	[5], [8]
Caio	Redundância.	[3], [4]
Lucas Midlhey	Boas práticas	[10]
Geovanne; Igor	Tamanho e Interface da Unidade; Quantidade SLOC	[7]