UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Uma abordagem para pontuação de exercícios de programação baseada em análise estática de código

Trabalho de Conclusão de Curso

Victor Souza Vieira

São Cristóvão – Sergipe

2022

UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Victor Souza Vieira

Uma abordagem para pontuação de exercícios de programação baseada em análise estática de código

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Professor Doutor Alberto Costa Neto

São Cristóvão – Sergipe
2022

Agradecimentos

Dedico este trabalho principalmente à minha noiva Evelyn Gabriele que me incentivou e esteve comigo em cada passo dessa longa jornada e à minha família, principalmente a minha mãe Cristiane Souza que também me incentivou desde pequeno aos estudos. Agradeço também a todos os meus amigos que fizeram parte dessa jornada de alguma maneira em especial ao Victor Santos que nos últimos anos demonstrou uma boa amizade. Agradeço também ao professor Alberto Costa por acreditar em mim e pelo apoio neste trabalho.

Resumo

Com o aumento massivo de estudantes em cursos introdutórios de computação ocorrendo nas últimas décadas, o trabalho dos professores para corrigir seus execícios aumentou significativamente. Com a criação dos analisadores automáticos de código-fonte tornou-se possível avaliar e analisar códigos-fonte de forma estática, dinâmica e híbrida (utiliza tanto análise estática quanto dinâmica). Juízes *Online* avaliam se, durante a execução do código-fonte, todos os casos de teste foram ou não aprovados. Caso aprovados, o exercício recebe nota máxima e, caso contrário, nota mínima. Desta forma, podem ser enquadrados como analisadores dinâmicos de código. Esta abordagem muitas das vezes acaba frustrando os alunos iniciantes, pois apenas é avaliado o resultado final do programa, isto é, sem considerar a qualidade da solução. Além disso, os professores precisam avaliar manualmente o código-fonte para identificar situações em que deveria alertar o aluno de má qualidade do seu código-fonte. Como solução, para que haja uma melhor avaliação destes estudantes, é proposta uma ferramenta baseada em análise estática de código, para a linguagem de programação *Python*, que pontua, com base em regras que podem ser definidas pelo professor, o código escrito pelos alunos, dando assim um melhor *feedback* para o resultado da avaliação.

Palavras-chave: Análise estática. Linguagem de programação *Python*. Código-fonte.

Abstract

With the massive increase of students in introductory computing courses taking place in recent decades, the work of teachers to correct their exercises has increased significantly. With the creation of automatic source code analyzers, it became possible to evaluate and analyze source code in a static, dynamic and hybrid way (it uses both static analysis and as dynamic). Online judges assess whether, during source code execution, all test cases pass or fail. If approved, the exercise is rated as a maximum grade and, if not, a minimum grade. In this way, they can be classified as dynamic code analyzers. This approach often ends up frustrating beginner students, as only the final result of the program is evaluated, that is, without considering the quality of the solution. In addition, teachers need to manually evaluate the source code to identify situations where they should alert the student of poor quality of their source code. As a solution, so that there is a better evaluation of these students, a tool based on static code analysis is proposed for the Python programming language, which scores, based on rules that can be defined by the teacher, the code written by the students, thus giving better feedback for the evaluation result.

Keywords: Static analysis. Python programming language. Source code.

Lista de ilustrações

Figura 1 – Processo da análise dinâmica . . . 16
Figura 2 – Processo da análise estática . . . 17
Figura 3 – Grafo de fluxo . . . 22
Figura 4 – Elementos para o cálculo da complexidade ciclomática . . . 24
Figura 5 – Distribuição percentual dos artigos encontrados por base de conhecimento . 28
Figura 6 – Distribuição quantitativa de artigos aceitos e rejeitados por base de conhecimento 28
Figura 7 – Quantidade de artigos aceitos lançados por ano . . . 29
Figura 8 – Estrutura de diretórios . . . 35
Figura 9 – Fluxo de dados . . . 36
Figura 10 – Estrutura de pastas da ferramenta . . . 40
Figura 11 – Diagrama de pacotes . . . 40
Figura 12 – Localização dos arquivos desenvolvidos . . . 41
Figura 13 – Definição inicial do arquivo de configurações . . . 43
Figura 14 – Diagrama de sequência . . . 46

Lista de tabelas

Tabela 1 – *Strings* de busca 27
Tabela 2 – Modelo de resultado 37
Tabela 3 – Modelo de resultado de alerta 37

Lista de códigos-fonte

Código-fonte 1 – Retornando taxa de desconto baseado na fidelidade de um cliente . . 21
Código-fonte 2 – Código *Python* sem *pycodestyle* 25
Código-fonte 3 – Código após a execução do *code formatter* 25
Código-fonte 4 – Código-fonte do arquivo *source_code.py* 42
Código-fonte 5 – Exemplo de código-fonte não totalmente modularizado 44
Código-fonte 6 – Exemplo de código-fonte modularizado após execução da solução criada 45
Código-fonte 7 – Função que calcula a complexidade ciclomática 47
Código-fonte 8 – Função que calcula as *raw metrics* 47
Código-fonte 9 – Pontuação das soluções 48

Lista de abreviaturas e siglas

JSON	*Java Script Object Notation*
UFS	Universidade Federal de Sergipe

Sumário

1 Introdução **11**
1.1 Objetivos 13
1.1.1 Gerais 13
1.1.2 Específicos 13
1.2 Estrutura do documento 13

2 Fundamentação teórica **15**
2.1 Análise dinâmica de código 15
2.2 Análise estática de código 17
2.2.1 Análise do estilo de programação 17
2.2.2 Detecção de erros semânticos 18
2.2.3 Atributos da qualidade do *software* 18
2.2.4 Análise de similaridade estrutural e não estrutural 19
2.3 Métricas de *software* 19
2.3.1 Métricas de procedimento 19
2.3.2 Métricas de projeto 19
2.3.3 Métricas de produto 20
2.3.3.1 Complexidade 20
2.3.4 Complexidade ciclomática de McCabe 20
2.4 Ferramentas 22
2.4.1 Radon 22
2.4.2 *Black* 24

3 Revisão sistemática **26**
3.1 Protocolo da revisão 26
3.2 Resultados encontrados 27
3.3 Análise dos resultados 29
3.3.1 *A software system for grading student computer programs* 29
3.3.2 *Ability-training-oriented automated assessment in introductory programming course* 30
3.3.3 *Semantic similarity based evaluation for C programs through the use of symbolic execution* 30

4 Descrição da ferramenta **31**
4.1 Métricas que serão utilizadas na ferramenta 31
4.2 Arquivo de configurações 32

4.3 Análise e cálculo da pontuação dos códigos-fonte 33
4.4 Resultados esperados 37
4.5 Linguagem de programação suportada pela ferramenta 37
4.6 Linguagem de programação que a ferramenta será desenvolvida 38

5 Desenvolvimento 39
5.1 Estrutura da ferramenta 39
5.2 Decisões tomadas 42
5.3 Comportamento interno da ferramenta 45
5.4 Testes da ferramenta 48

6 Conclusão 50
6.1 Trabalhos futuros 50

Referências 52

1 Introdução

Com o desenvolvimento constante das tecnologias da informação e comunicação (TIC), no que diz respeito à aprendizagem, configurações e métodos de avaliação vêm se tornando um imenso desafio (WESIAK; AL-SMADI; GÜTL, 2012). Em contrapartida, de acordo com (AMER; HAROUS, 2017), o rápido avanço tecnológico mudou a forma como os professores ensinam e como os alunos aprendem, desta maneira fazendo com que os processos de ensino e aprendizagem sejam continuamente melhorados para que seja possível a ambas as partes conviverem em harmonia.

Aprender programação não é uma tarefa trivial, pois, o ato de programar requer raciocínio e habilidades criativas para resolver problemas (RONGAS; KAARNA; KALVIAINEN, 2004), diversas ferramentas foram desenvolvidas ao longo do tempo para auxiliar professores e alunos em cursos introdutórios de programação e são divididas em quatro categorias:

a) Ambientes de Desenvolvimento Integrado (IDE);

b) Visualização;

c) Ambientes Virtuais de Aprendizagem (AVA);

d) Sistemas para apresentação, gerenciamento e teste de exercícios, conhecidos também como *online judges*.

Nos cursos introdutórios de programação os discentes geralmente são apresentados a algum ambiente que permite que eles desenvolvam seu raciocínio e escrevam programas. Várias vezes estes ambientes possibilitam que os estudantes possam: editar, depurar, compilar e executar os programas que foram construídos. Por se tratar de cursos introdutórios, muitas dessas informações, como, por exemplo, depurar e compilar o código, podem acabar sobrecarregando os alunos de tal forma que sua motivação e performance diminuam, em consequência causando o abandono do curso (BRANDãO; RIBEIRO; BRANDãO, 2012).

Sob a perspectiva do docente, o processo de avaliar os alunos de cursos introdutórios

de programação é de grande importância para que haja uma constatação do seu aprendizado ou não. Porém, com o aumento da quantidade de alunos nestes cursos, acaba sendo gerada uma sobrecarga maior nos professores que os avaliam (ARIFI et al., 2015). Com isso várias atividades que antes eram feitas de forma manual foram total ou parcialmente substituídas por métodos automatizados para que assim seja possível lidar com a demanda crescente de matriculados nestes tipos de curso (POžENEL; FüRST; MAHNIč, 2015).

Os métodos automatizados de avaliação de código, especialmente os *Online Judges*, utilizam-se basicamente da análise de casos de teste, independente da qualidade do código escrito ou se o código está de acordo com o que foi ensinado, levando assim aos alunos em diversos casos a ignorarem técnicas de codificação adequadas (C et al., 2019). No capítulo 2 veremos que a análise de casos de teste pertence ao grupo de análise dinâmica de código.

Para exemplificar, considere aqui um programa para calcular o n-ésimo termo da sequência de Fibonacci, é sabido que, existem formas iterativas e recursivas para escrever este programa e todas fornecerão o mesmo resultado. Agora imagine que o professor deseja que o aluno encontre a solução para este problema utilizando apenas a forma iterativa. Considerando este caso a análise dinâmica baseada em casos de testes será inútil, pois, tanto as soluções iterativas quanto as soluções recursivas fornecerão o mesmo resultado, sem fazer questão se a solução foi efetivamente iterativa. (C et al., 2019).

Seguindo o raciocínio descrito acima, o aluno que desenvolveu o programa para calcular o n-ésimo termo da sequência de qualquer uma das duas formas e tenha sido aprovado em todos os casos de teste, irá tirar a nota máxima neste exercício, enquanto que o aluno que desenvolveu utilizando a técnica pedida pelo professor, mas que por algum motivo seu código não foi aprovado em todos os casos de teste, receberá nota mínima. Dessa maneira não há um meio termo, ou o aluno recebe a nota máxima ou a nota miníma, independente da forma como o programa foi escrito, sendo assim, os alunos iniciantes em cursos introdutórios de programação, podem acabar sentindo-se frustrados por esta forma de avaliação binária e que não o bonifica por seguir boas práticas de programação.

De forma análoga, considere um programa que, dado um número entre 1 e 12, deverá imprimir o mês correspondente a este. Novamente existem diversas formas de solucionar esta questão, uma das mais óbvias seria fazer uma sequência de condicionais (*if* e *else*) onde seriam necessárias pelo menos onze checagens. A abordagem anterior traz consigo uma complexidade ciclomática[1] de no mínimo onze pontos. Em contrapartida, se o aluno utilizar uma estrutura de dados do tipo dicionário fazendo o mapeamento dos números para meses, a complexidade ciclomática neste caso cairia para apenas 1 ponto. A segunda solução apresentada tem menos desvios condicionais que a primeira, porém, durante a avaliação de casos de teste, ambas serão aceitas e dadas como corretas, mas a qualidade da primeira solução é baixa quando comparada à segunda.

[1] Será ilustrado na seção 2.3.4 do capítulo 2.

1.1 Objetivos

1.1.1 Gerais

Desenvolver um *software* de linha de comando capaz de realizar a análise estática de código e extrair métricas dos exercícios dos alunos, gerando um conjunto de relatórios ao final de sua execução. Um dos relatórios será responsável por conter os valores das métricas avaliadas juntamente com uma pontuação definida pela ferramenta para cada exercício. Outro relatório apresenta todas as submissões que necessitam de alguma atenção extra do docente, como exemplo submissões que possuem nota inferior a um limiar previamente definido pelo professor. Assim, facilitando o processo de avaliação dos exercícios e também possibilitando que o docente possa fornecer um melhor *feedback* aos discentes sobre a solução por eles adotadas nos exercícios.

1.1.2 Específicos

São objetivos específicos deste trabalho:

- Realizar uma busca sobre ferramentas que realizem algum tipo de análise estática de código-fonte;
- Especificar um arquivo de configurações para definir quais métricas serão analisadas, diretórios onde serão salvos os resultados entre outras;
- Propor um conjunto de métricas que podem servir para analisar a qualidade do código-fonte dos aprendizes de programação;
- Desenvolver uma ferramenta para realizar a análise do código-fonte baseando-se nas métricas coletadas a partir do mesmo e comparar com uma solução modelo.
- Realizar testes da ferramenta utilizando problemas e soluções extraídas do juiz *online The Huxley*.

1.2 Estrutura do documento

Para uma melhor navegabilidade e entendimento das divisões deste trabalho, o documento está dividido da seguinte maneira.

- Capítulo 1 - Introdução: é apresentada a situação problema que será abordada neste trabalho, suas causas e justificativas além dos objetivos deste documento;
- Capítulo 2 - Fundamentação teórica: neste capítulo todo o referencial teórico visto é abordado e conceitos e métricas que serão posteriormente utilizadas. Aqui se encontram também

as ferramentas e linguagem de programação que serão utilizadas no desenvolvimento do *software*.

- Capítulo 3 - Revisão sistemática: traz os resultados da revisão feita em artigos sobre ferramentas existentes que possam servir de embasamento para desenvolver a ferramenta;
- Capítulo 4 - Descrição da ferramenta: neste capítulo as métricas selecionadas, estrutura de pastas para funcionamento da ferramenta, descrição de fluxo de dados e modelo de resultados esperados é ilustrado;
- Capítulo 5 - Desenvolvimento: aqui todo o desenvolvimento da ferramenta é detalhado, decisões tomadas e diagramas também são relatados neste capítulo;
- Capítulo 6 - Conclusão: é apresentada a conclusão deste trabalho juntamente com possíveis trabalhos futuros.

2

Fundamentação teórica

Neste capítulo serão discutidos temas que mais a frente serão utilizados no desenvolvimento da ferramenta. Inicialmente será ilustrado como é o processo da análise dinâmica de código seguido pela análise estática (que é o tipo de análise que será utilizada na ferramenta aqui proposta), em seguida as métricas de *software* serão introduzidas, incluindo complexidade ciclomática. Após, teremos uma subseção específica sobre a linguagem escolhida tanto para codificar a ferramenta deste trabalho quanto para os códigos-fonte que serão avaliados, por fim teremos uma introdução ao analisador estático Radon (subseção 2.5.1) e o *code formatter Black* (subseção 2.5.2) que serão também utilizados no desenvolvimento da ferramenta.

2.1 Análise dinâmica de código

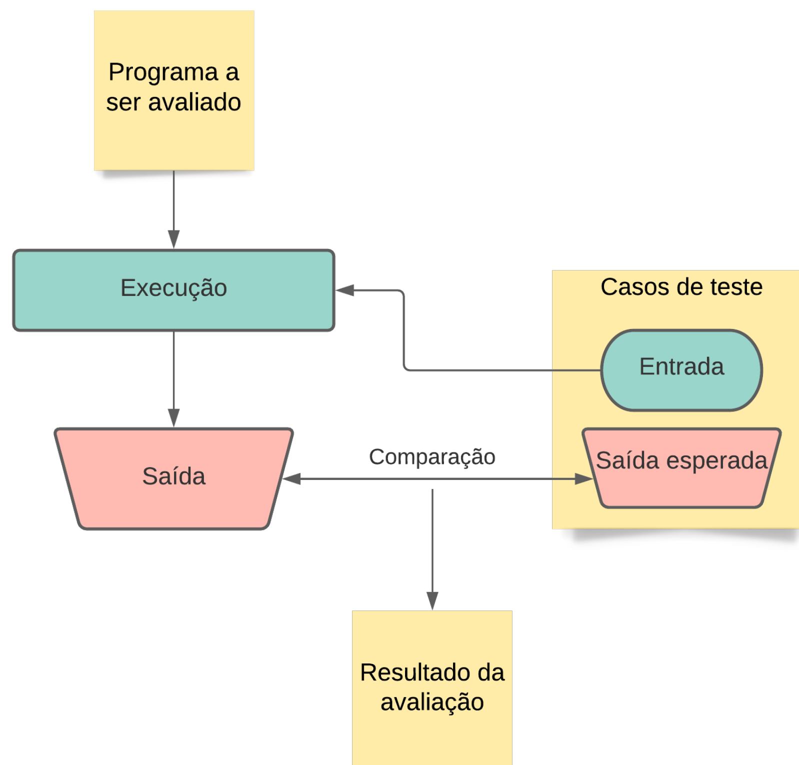
Segundo Arifi et al. (2015) a análise dinâmica consiste em executar o programa utilizando como entrada uma sequência de casos de teste onde, para cada caso, existe uma saída esperada e uma saída fornecida pelo programa, no fim estas são comparadas e caso sejam iguais o caso de teste foi um sucesso caso contrário, insucesso. Um programa modelo é construído e tem suas saídas coletadas de acordo com um conjunto de casos de teste e depois os programas que executarem sobre estes casos de teste terão suas saídas comparadas ao do programa modelo (ARIFI et al., 2016). A Figura 1 mostra o processo de funcionamento da análise dinâmica.

Ainda de acordo com (ARIFI et al., 2016), por não possuir uma implementação muito complexa, a maioria dos sistemas de avaliação automatizados utiliza a análise dinâmica, porém vale ressaltar que existem alguns problemas com esta abordagem, sendo o mais relevante deles, no contexto deste trabalho, a impossibilidade de analisar o programa caso ele não compile ou apresente um ou mais erros durante sua execução.

Existem alguns pontos fracos em utilizar a análise dinâmica para avaliar alunos iniciantes, pois, geralmente eles costumam cometer erros, como esquecer de colocar ponto e vírgula ao

final de uma sentença ou esquecer de fechar um parênteses que foi aberto. Como os sistemas de análise dinâmica são sensíveis a erro, cometer alguns dos citados anteriormente inviabilizaria a execução do programa, dessa forma impossibilitando a análise e consequentemente a nota zero será atribuída (ARIFI et al., 2015).

Figura 1 – Processo da análise dinâmica



Um outro ponto fraco da análise dinâmica é o *feedback* gerado ao término da execução, onde basicamente seria:

- "Correto": quando o programa foi executado sem erros e passou em todos os casos de teste;
- "Incorreto": quando o programa executou, mas não passou em todos os casos de teste;
- "Erro de compilação": quando o programa não teve sua compilação bem sucedida;
- "Erro de execução": quando ocorreu algum erro durante a execução do programa, impedindo-o de chegar ao final, como ocorre por exemplo, numa divisão por 0;

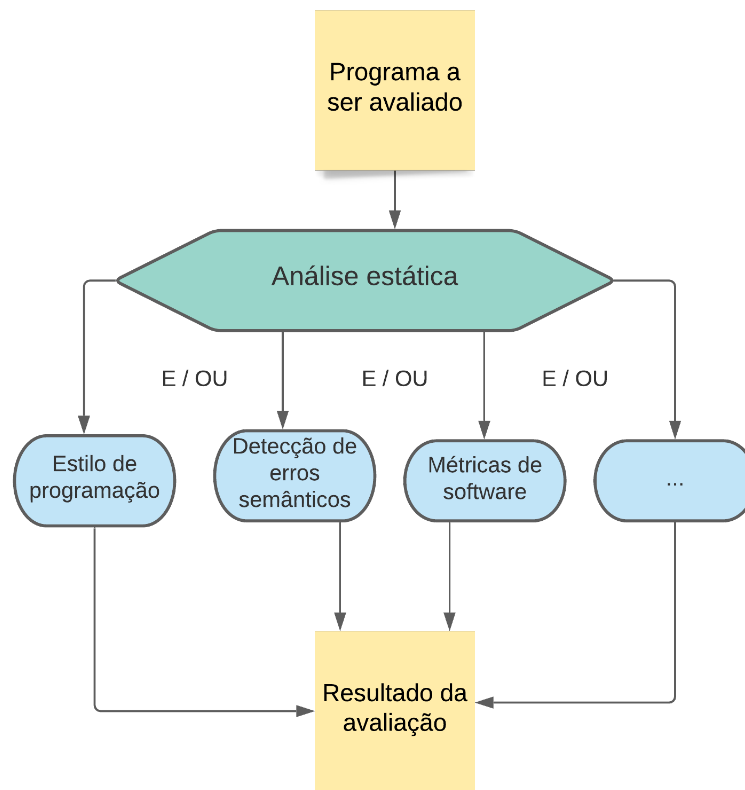
Desta maneira, os sistemas de avaliação que utilizam este tipo de análise de código-fonte são mais recomendados para alunos mais avançados em cursos de programação, pois

eles certamente possuirão uma maior compreensão sobre o que estão escrevendo, além de já possuírem uma familiaridade maior com o desenvolvimento de programas.

2.2 Análise estática de código

Diferentemente da análise dinâmica, que é feita executando o programa para que seja possível a comparação com os casos de teste, a análise estática é realizada sem a necessidade dessa execução, pois esta busca extrair informações e métricas a respeito do código escrito e não do resultado esperado (ARIFI et al., 2015). A Figura 2 ilustra o funcionamento do processo da análise estática.

Figura 2 – Processo da análise estática



As próximas subseções abordam alguns dos diversos métodos desta análise.

2.2.1 Análise do estilo de programação

Neste método são utilizadas técnicas que tentam explicitar o estilo que o programador escreve seus códigos-fonte, considerando a legibilidade para mensurar a qualidade, o que torna

esta análise útil também para auxiliar na verificação de plágio. Alguns elementos são considerados neste método, sendo alguns deles:

- Nomes de variáveis;
- Uso de constantes;
- Espaçamento entre linhas;
- Modularização do código;

2.2.2 Detecção de erros semânticos

Erros semânticos ocorrem quando existem sentenças que estão gramaticalmente corretas, mas que durante a execução do programa causarão algum erro podendo impedir o funcionamento correto do programa ou até mesmo interrompendo sua execução. Alunos iniciantes tendem a cometer vários erros semânticos como por exemplo a divisão por zero, alguma recursão infinita ou erros de tipagem. A priori erros semânticos parecem não ter grande impacto num programa, mas imagine este problema num sistema financeiro, por exemplo, no qual a quantidade de dinheiro perdido pode ser incalculável.

Atualmente diversos IDEs trabalham com a pré-compilação do código fonte, isto é, executam a compilação enquanto o programador está codificando, para verificar possíveis erros como os erros de tipagem para linguagens de programação que possuem declaração de variáveis com tipagem explícita.

2.2.3 Atributos da qualidade do *software*

De acordo com a (ISO-25010, 2011) a manutenibilidade de um *software* pode ser definida como uma característica que expressa a eficácia e eficiência com que um sistema pode ser modificado para melhoria, correção ou adaptação às mudanças no domínio ou nos próprios requisitos de determinado produto. Dentro do contexto de manutenibilidade temos algumas subcaracterísticas:

- **Modularidade**: decomposição em componentes ou módulos com interfaces bem definidas e, idealmente, independentes um dos outros. Pode ser expressa como o grau em que o *software* está modularizado, se seus componentes estão bem definidos de tal forma que, ao ser necessária alguma modificação, os impactos dela sejam mínimos no produto como um todo.
- **Reutilização**: é relativa ao quanto o código-fonte escrito pode ser reutilizado no mesmo ou por outros componentes do *software*.

- **Analisabilidade**: é a medida de eficiência e efetividade com o qual é possível avaliar quais são os impactos e efeitos colaterais de realizar determinada mudança num dado sistema ou produto, diagnosticar deficiências ou pontos de falha, ou identificar partes para modificação.
- **Modificabilidade**: é o grau onde o *software* pode ser modificado de forma eficiente e eficaz sem introdução de falhas ou problemas que possam degradá-lo.
- **Testabilidade**: é a medida de eficiência e efetividade onde critérios de testes podem ser definidos para produto de *software* e estes podem ser executados para verificar os critérios de aceite.

2.2.4 Análise de similaridade estrutural e não estrutural

Para esta análise se faz necessária a existência de uma ou mais soluções para serem utilizadas como base de comparação com as soluções posteriormente fornecidas pelos alunos, assim podendo determinar níveis de similaridade entre elas.Assumindo que se sabe que uma das soluções é adequada, esta análise pode indicar que quanto mais próximas forem as soluções, maior será a nota atribuída à solução que se deseja avaliar. Esta análise também pode servir para detecção de plágio nas soluções, porém isto não será abordado neste trabalho.

2.3 Métricas de *software*

De acordo com (HONGLEI; WEI; YANAN, 2009) as métricas de *software* fornecem medidas tanto para o *software* quanto para seu processo de produção, assim, dando valores numéricos aos atributos que envolvem o produto e o processo. De uma maneira mais generalizada, as métricas de *software* podem ser vistas como uma função que tenha como entrada os dados do *software* e a saída é um valor que pode decidir como determinado atributo afeta o *software*. Elas podem ser divididas em três classes, sendo elas: métricas de procedimento, projeto e produto.

2.3.1 Métricas de procedimento

Estão associadas aos processos de desenvolvimento de *software*. São mais voltadas para o processo de gestão, onde existe um maior foco no custo, eficácia dos métodos, tempo decorrido do procedimento e comparação com outros métodos.

2.3.2 Métricas de projeto

Têm como foco auxiliar na melhoria de qualidade do projeto, através de métricas que ajudem a controlar a situação e andamento do projeto. Aqui é possível incluir métricas como, carga de trabalho, custo, escala e satisfação do cliente.

2.3.3 Métricas de produto

As métricas de produto têm como foco obter o entendimento e controle da qualidade do produto. Das diversas métricas de produto existentes, podemos citar as que estão relacionadas à: confiabilidade, manutenibilidade, escalabilidade, complexidade tanto do *software* quanto do código-fonte e usabilidade.

2.3.3.1 Complexidade

Um software com alta complexidade é mais difícil de ser entendido, testado e modificado (HARRISON et al., 1982). O problema da confiabilidade está essencialmente ligado ao problema da complexidade do *software*. Ao coletar métricas de complexidade, quanto maior o limite atingido, mais problemas ou defeitos podem vir a surgir levando assim um maior tempo demandado para suas correções (HONGLEI; WEI; YANAN, 2009).

Métricas de complexidade ajudam a prever e manter projetos, por exemplo, quanto mais complexos forem os componentes do programa, mais difícil será evolui-los e mantê-los, havendo assim uma maior demanda de tempo e esforço para tal. Essas métricas auxiliam também a avaliar a carga de programação, custo de desenvolvimento e selecionar uma solução mais adequada para um problema, pois quanto menor a complexidade, melhor é aquela solução naquele contexto (HONGLEI; WEI; YANAN, 2009).

2.3.4 Complexidade ciclomática de McCabe

A complexidade ciclomática é uma métrica de *software* que pode ser utilizada para determinar o quão complexo um programa é. Thomas J. McCabe uma vez propôs, utilizando teoria dos grafos, uma forma para mensurar a complexidade ciclomática de um programa, esta proposta foi amplamente aceita pelo público em geral, muito provavelmente por ser fácil de ser calculada (HARRISON et al., 1982). McCabe sugeriu que a complexidade ciclomática pode ser aplicada para determinar quão difícil será testar e modificar um programa. O cálculo dessa complexidade se dá por:

$$V(G) = E - n + 2p \tag{2.1}$$

Onde **E** é o número de arestas do grafo, **n** é o número de nós e **p** é quantidade de componentes conexos do grafo de fluxo. Porém na maior parte das vezes o grafo é totalmente conexo, ou seja, p=1. De maneira mais simples a complexidade ciclomática pode ser calculada como o número de decisões contidas num bloco do programa + 1. Os programas com maior complexidade são mais suscetíveis a erro, por isso normalmente os programadores tentam definir um limite máximo de linhas de código em seus módulos, para que assim seja possível diminuir também sua complexidade (HONGLEI; WEI; YANAN, 2009).

Explicando melhor a Equação 2.1, os nós do grafo de fluxo representam um ou mais comandos onde os comandos sequenciais podem ser representados juntos, as arestas do grafo representam o fluxo de controle

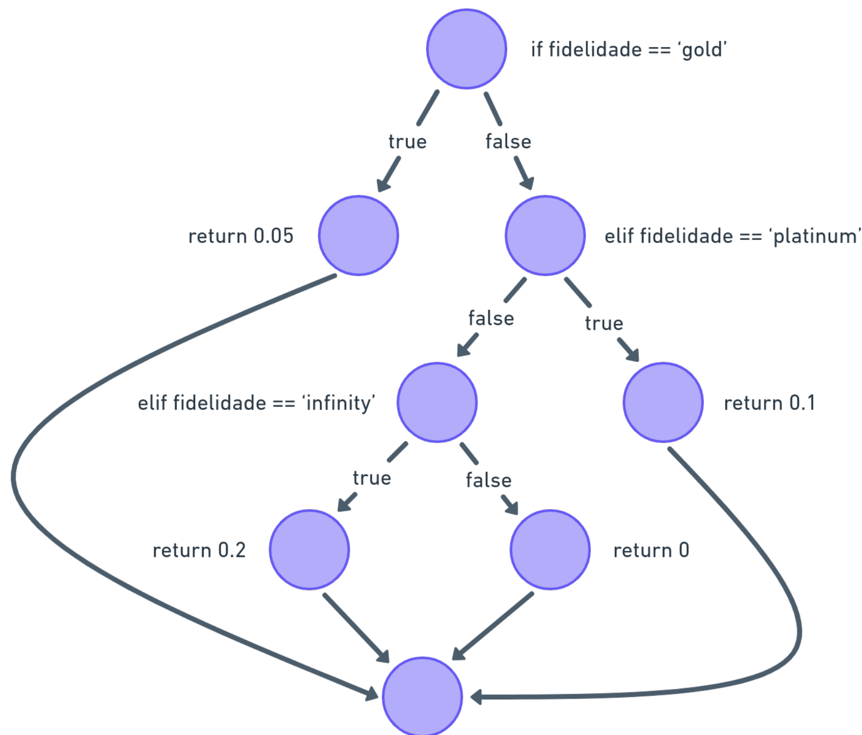
- Cada nó do grafo de fluxo representa um ou mais comandos;
- As arestas representam o fluxo de controle;
- Cada nó que contém uma condicional é chamado de nó predicado;
- Um nó predicado é representado com duas ou mais arestas partindo dele;
- Toda aresta deve chegar a um nó, mesmo que este nó não represente nenhum procedimento;

Aplicando a Equação 2.1 no grafo de fluxo da Figura 3, referente ao Código-fonte 1, temos: E = 10, n = 8, como o grafo é conexo p = 1, dado isso temos que a complexidade ciclomática do código é de 4 pontos.

Código-fonte 1 – Retornando taxa de desconto baseado na fidelidade de um cliente

```
1 if fidelidade == 'gold':
2     return 0.05
3 elif fidelidade == 'platinum':
4     return 0.1
5 elif fidelidade == 'infinity':
6     return 0.2
7 else:
8     return 0
```

Figura 3 – Grafo de fluxo



2.4 Ferramentas

Para a linguagem de programação *Python*, duas ferramentas muitos importantes, que serão utilizadas no desenvolvimento da ferramenta proposta, foram encontradas, são elas o analisador estático **Radon**[1] e o *code formatter* ***Black***[2] as próximas subseções ilustram o funcionamento delas.

2.4.1 Radon

Radon é uma ferramenta *Python* de código aberto que computa diversas métricas de código de forma estática. Atualmente encontra-se na sua versão 4.1.0 e consegue extrair as seguintes métricas dos códigos:

- ***Raw metrics***
  - ***Lines Of Code*** **- LOC**: quantidade de linhas de código, não é necessariamente a quantidade de linhas no arquivo;

[1] <https://radon.readthedocs.io/en/latest/index.html>
[2] <https://pypi.org/project/black/>

- ***Logical Lines Of Code* - LLOC**: número de linhas lógicas de código, cada linha lógica contém exatamente uma instrução;
- ***Source Lines Of Code* - SLOC**: número de linhas de código fonte, não corresponde necessariamente ao LLOC;
- ***Comment***: número de linhas de comentários, aqui vale ressaltar que comentários em bloco não são considerados nesta contagem, pois, para o interpretador *Python*, elas são apenas *strings*;
- ***Multi***: número de linhas que representam *strings* que ocupam mais de uma linha;
- ***Blanks***: número de linhas em branco ou que contenham apenas espaços em branco;

• ***Complexity Cyclomatic - McCabe's Complexity***: para calcular a complexidade ciclomática a ferramenta utiliza o cálculo de McCabe exemplificado na Equação 2.1, vale ressaltar que o resultado dessa equação é igual ao número de caminhos linearmente independentes através do código. A ferramenta analisa a Árvore Sintática Abstrata - AST de um código-fonte escrito em *Python* para calcular a complexidade ciclomática Lacchia (2020). A Figura 4 exemplifica de forma mais simples como é feito o cálculo da complexidade, sendo que:

- Cada ***if*** adiciona 1 ponto na complexidade ciclomática;
- Cada ***elif*** adiciona 1 ponto na complexidade ciclomática;
- Cada laço ***for, while*** adiciona 1 ponto na complexidade ciclomática;
- Cada ***except*** adiciona 1 ponto na complexidade ciclomática;
- Cada ***with*** adiciona 1 ponto na complexidade ciclomática pois, corresponde a um bloco *try/except*;
- Cada ***assert*** adiciona 1 ponto na complexidade ciclomática pois, corresponde internamente à uma condicional *if*;
- Cada ***comprehension*** adiciona 1 ponto na complexidade ciclomática pois, é equivalente à um laço *for*;
- Cada ***boolean operator*** adiciona 1 ponto na complexidade ciclomática para cada operador booleano utilizado.

• ***Halstead metrics***: a partir de quatro medidas

- n1 = quantidade de operadores distintos;
- n2 = quantidade de operandos distintos;
- N1 = número total de operadores;
- N2 = número total de operandos;

é possível obter uma série de novas métricas, sendo uma delas o volume que é definido por:

$$V = N.\log_2(n) \tag{2.2}$$

onde N = (N1+N2) e n = (n1+n2).

- ***The Maintainability Index***: o índice de manutenibilidade é uma métrica que diz o quanto o código é fácil de receber manutenção e suporte. Nesta ferramenta essa métrica ainda é experimental, mas utiliza um conjunto de outras métricas para realizar seu cálculo, por exemplo, SLOC, complexidade ciclomática e o volume de *Halstead* definido na Equação 2.2;

Figura 4 – Elementos para o cálculo da complexidade ciclomática

Construct	Effect on CC	Reasoning
if	+1	An `if` statement is a single decision.
elif	+1	The `elif` statement adds another decision.
else	+0	The `else` statement does not cause a new decision. The decision is at the `if`.
for	+1	There is a decision at the start of the loop.
while	+1	There is a decision at the `while` statement.
except	+1	Each `except` branch adds a new conditional path of execution.
finally	+0	The finally block is unconditionally executed.
with	+1	The `with` statement roughly corresponds to a try/except block (see PEP 343 for details).
assert	+1	The `assert` statement internally roughly equals a conditional statement.
Comprehension	+1	A list/set/dict comprehension of generator expression is equivalent to a for loop.
Boolean Operator	+1	Every boolean operator (and, or) adds a decision point.

Fonte: Lacchia (2020)

2.4.2 *Black*

O formatador de código *Black* tem como objetivo formatar códigos-fonte *Python* aplicando o *Python Code Style*[3], ou seja, o estilo de formatação definido nas boas práticas da linguagem. Possui uma fácil instalação e configuração e está sob a licença MIT onde é possível modificar, comercializar e distribuir a ferramenta. Como a formatação dos códigos-fonte, que serão avaliados pela ferramenta, não é conhecida, foi utilizada a *Black* para essa finalidade. O Código-fonte 2 exibe um código *Python* sem o *pycodestyle* aplicado, após a execução do *code formatter*, obtemos o Código-fonte 3, o qual segue as boas práticas de formatação de código da linguagem.

[3] Disponível em: <https://peps.python.org/pep-0008/>

Código-fonte 2 – Código *Python* sem *pycodestyle*

```
def␣AnalisarSituacao(media):
␣if␣media␣>=␣9␣:
␣␣return('aprovado␣com␣louvor')
␣if␣media␣>=␣7␣:
␣␣return('aprovado')
␣if␣media␣<␣3␣:
␣␣return('reprovado')
␣elif␣media␣>=␣3␣and␣media␣<7␣:
␣␣return('prova␣final')
```

Código-fonte 3 – Código após a execução do *code formatter*

```
def␣AnalisarSituacao(media):
␣␣␣␣if␣media␣>=␣9:
␣␣␣␣␣␣␣␣return␣('aprovado␣com␣louvor')
␣␣␣␣if␣media␣>=␣7:
␣␣␣␣␣␣␣␣return␣('aprovado')
␣␣␣␣if␣media␣<␣3:
␣␣␣␣␣␣␣␣return␣('reprovado')
␣␣␣␣elif␣media␣>=␣3␣and␣media␣<␣7:
␣␣␣␣␣␣␣␣return␣('prova␣final')
```

3 Revisão sistemática

3.1 Protocolo da revisão

Foi definido um protocolo de busca visando responder às seguintes questões:

- Q1 - Quais ferramentas de análise estática de código voltadas para auxiliar professores no processo de correção de avaliações de exercícios de programação em turmas introdutórias existem?
- Q2 - Quais linguagens de programação são suportadas pelas ferramentas encontradas?

As bases de conhecimento selecionadas para esta revisão foram: IEEE[1] sigla para *Institute of Electrical and Electronics Engineers*, a Scopus[2] e a Science Direct[3]. Para que fosse possível acessar os artigos completos destas bases foi utilizado o portal do periódicos CAPES[4].

Para a realização da consulta dos artigos, nas bases de conhecimento, algumas *strings* de busca foram criadas, pois, além de facilitar a procura desses artigos, elas também auxiliam no processo de filtragem deles. A Tabela 1 traz as *strings* utilizadas na pesquisa.

---

1 Disponível em: <https://ieeexplore.ieee.org/>
2 Disponível em: <https://www.scopus.com/>
3 Disponível em: <https://www.sciencedirect.com/>
4 Disponível em: <https://www-periodicos-capes-gov-br.ez20.periodicos.capes.gov.br/>

Tabela 1 – *Strings* de busca

Base	***String***	**Nº de resultados**
IEEE Xplore	(("All Metadata":Automatic Grading Of Programming Exercises) OR ("All Metadata":Static Analysis Of Metrics For Software Quality In The Academic Environment)) AND ("Index Terms":computer science education)	11
Scopus	("Automatic Grading Of Programming Exercises"OR "Static Analysis Of Metrics For Software Quality In The Academic Environment")	48
Sciente Direct	('Automatic Grading Of Programming Exercises') OR ('Static Analysis Of Metrics For Software Quality In The Academic Environment')	232

Com o intuito de refinar mais a filtragem dos 291 artigos encontrados, foram definidos critérios de inclusão e exclusão, para que assim fossem mantidos os artigos mais coerentes com o objetivo deste trabalho. Os critérios de inclusão definidos foram:

- Propor uma ferramenta de análise estática de código, configurável e voltada para avaliação de exercícios de programação.

Já os critérios de exclusão:

- Proposta de ferramenta que não utilize análise estática de código;
- Proposta que não deixe claro qual o método de análise;
- Proposta de ferramenta que não seja voltada para avaliação de exercícios de programação;
- Proposta que seja voltada para o ambiente profissional de desenvolvimento de software;

Após a coleta dos artigos científicos nas três bases citadas anteriormente foram aplicados os processos de inclusão e exclusão no título e resumo de cada um dos artigos encontrados. A próxima seção demonstra os resultados encontrados.

3.2 Resultados encontrados

Aplicando os critérios de inclusão e exclusão nos 291 artigos encontrados, nas 3 bases de conhecimento, foram considerados aceitos 3, duplicados 31 e rejeitados 257. A Figura 5 diz respeito à distribuição percentual dos artigos encontrados por base, a Figura 6 exibe a quantidade de artigos encontrados por base, já a Figura 7 traz os anos em que foram lançados os artigos que foram aceitos no critério de inclusão.

Figura 5 – Distribuição percentual dos artigos encontrados por base de conhecimento

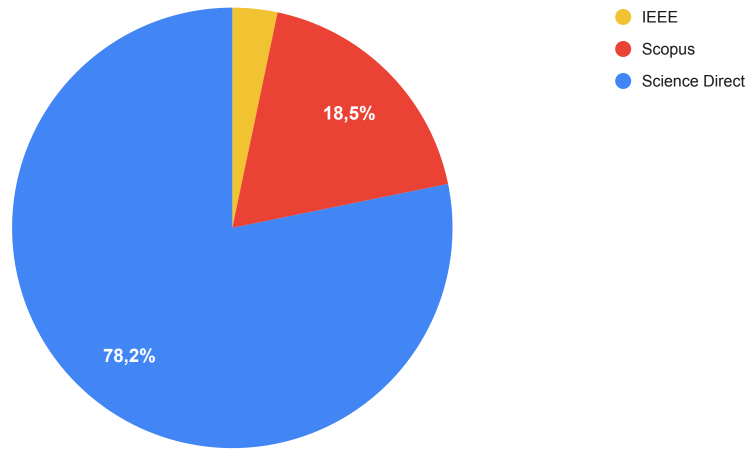


Figura 6 – Distribuição quantitativa de artigos aceitos e rejeitados por base de conhecimento

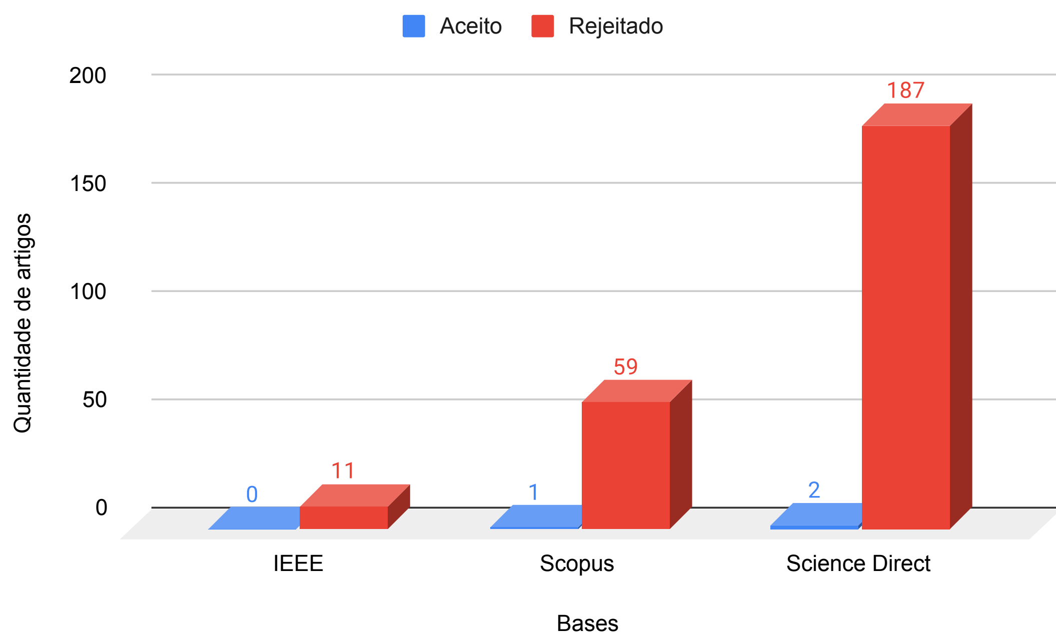
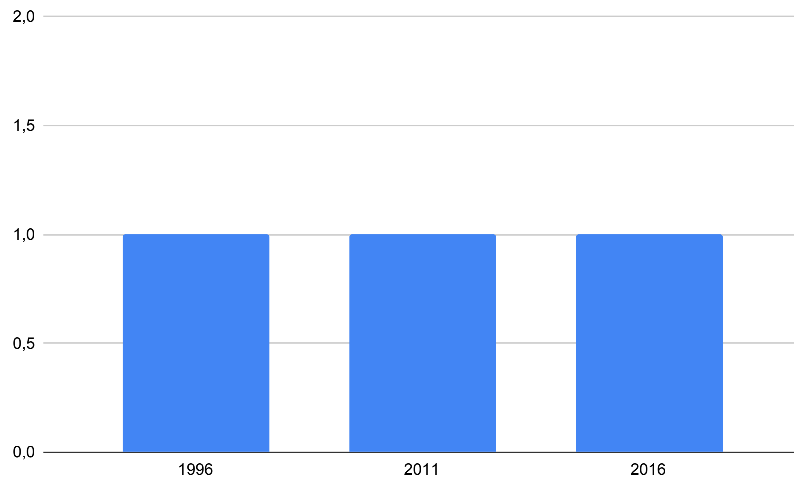


Figura 7 – Quantidade de artigos aceitos lançados por ano



3.3 Análise dos resultados

As próximas subseções dizem respeito aos artigos que foram selecionados no critério de inclusão, as duas primeiras propostas fogem um pouco do critério, mas foram consideradas visto que, seu conteúdo é de fato relevante para este trabalho.

3.3.1 *A software system for grading student computer programs*

No artigo produzido por (JACKSON, 1996), o autor discute sobre como as avaliações devem incluir julgamentos sobre a qualidade das soluções produzidas pelos alunos nos cursos de computação. O autor então propõe uma ferramenta de análise híbrida (que combina análise estática e análise dinâmica) capaz de avaliar os seguintes critérios:

- Corretude da solução apresentada
- Eficiência da solução
- Complexidade
- Análise de estilo de código contendo:
  - Percentual de linhas em branco
  - Percentual de linhas comentadas
  - Espaços por linhas
  - *Gotos*
  - Percentual de indentação

  - Tamanho de funções
  - Caracteres por linha
  - Palavras-chave utilizadas
- Taxa de acerto em casos de teste

A ferramenta proposta por (JACKSON, 1996) embora não utilize apenas análise estática, foi inclusa nos artigos aceitos. O motivo desta inclusão se deu pelo fato da ferramenta estar bem particionada entre as análises e os pontos de avaliação de cada uma delas, o autor também utilizou pesos entre cada um dos itens da lista acima descrita, tornando assim possível uma avaliação mais justa dos exercícios dos alunos.

3.3.2 *Ability-training-oriented automated assessment in introductory programming course*

A ferramenta proposta por (WANG et al., 2011), AutoLEP é bastante robusta no quesito de suas análises e feedback, novamente se trata de uma ferramenta de análise híbrida, mas, diferentemente da proposta anterior, esta normaliza os códigos fornecidos pelos estudantes e as soluções fornecidas pelo professores para assim poder remover variações entre os códigos e tornar o processo de análise mais simples. O AutoLEP foi projetado para fornecer testes suficientes para que seja possível testar todas as possibilidades de determinados problemas, verificar se os programas atendem as especificações ou não e conseguir funcionar mesmo quando os programas possuem erros sintáticos e semânticos, para que desta forma seja possível fornecer um feedback mais preciso, assim como uma pontuação condizente com as soluções aprensentadas.

3.3.3 *Semantic similarity based evaluation for C programs through the use of symbolic execution*

A proposta apresentada no trabalho de Arifi, Zahi e Benabbou (2016) traz como centro a análise estática e o uso de grafos de fluxo de controle para que seja possível realizar comparações entre uma solução fornecida pelo avaliador com outra solução fornecida pelo aluno que está sendo avaliado. O grande desafio deste trabalho foi conseguir lidar com a questão relacionada ao conjunto de soluções existentes para algum determinado exercício. Para contornar o problema, uma solução inovadora foi alcançada, sendo esta a comparação de programas de acordo com sua execução semântica como medida para verificar a similaridade entre as soluções fornecidas.

4

Descrição da ferramenta

Para que seja possível auxiliar professores de cursos introdutórios de computação no processo de correção de exercícios de programação, a ferramenta proposta utiliza a análise estática de código, fornecida pelo analisador estático Radon[1], dessa forma indo além da avaliação oferecida pelos juízes *online*.

A ferramenta conta também com uma forma de notificar o docente quando uma ou mais soluções possuam diferença maior que um limiar estabelecido previamente via arquivo de configuração, dessa maneira ajudando o professor a fornecer um *feedback* mais robusto acerca daquelas soluções.

Ao final de sua execução, a ferramenta gera três arquivos de resultado onde o primeiro arquivo ilustra um resultado geral sobre cada submissão analisada. O segundo arquivo exibe, de forma mais sucinta, detalhes sobre as soluções incluindo os resultados das métricas analisadas junto com uma pontuação que poderá crescer ou decrescer à medida que a ferramenta processa os dados. O terceiro arquivo terá a função de notificar o professor sobre soluções que ultrapassarem o limiar definido nas configurações do projeto.

A seguir será exibido todo o detalhamento da ferramenta proposta, desde as métricas selecionadas para análise, passando pela linguagem de programação escolhida e chegando aos resultados esperados da ferramenta.

4.1 Métricas que serão utilizadas na ferramenta

Todas as métricas selecionadas se encaixam na categoria de métricas de complexidade de código-fonte de acordo com o trabalho de Honglei, Wei e Yanan (2009), e elas são extraídas pelo analisador estático Radon. Segue a lista das métricas que serão utilizadas pela ferramenta.

[1] Disponível em: <https://radon.readthedocs.io/en/latest/>

- Complexidade ciclomática
- Linhas de código - LOC
- Linhas lógicas de código - LLOC
- Linhas lógicas de código-fonte - SLOC

Estas métricas foram selecionadas tomando como inspiração algumas das métricas citadas no trabalho de Jackson (1996) e os grafos de fluxo citados no trabalho de Arifi, Zahi e Benabbou (2016).

4.2 Arquivo de configurações

A ferramenta conta com um arquivo de configurações no formato *JSON (Java Script Object Notation)*, onde constam as principais definições a respeito do funcionamento interno da ferramenta. A seguir temos a lista com todas as 17 configurações disponíveis:

1. ***Root path***: Diretório raiz onde deverão estar os códigos-fonte que serão avaliados;
2. ***Calculate cyclomatic complexity***: Diz se a complexidade ciclomática deverá ser calculada;
3. ***Calculate raw metrics***: Diz se as métricas LOC, LLOC e SLOC deverão ser calculadas;
4. ***Diff cyclomatic complexity***: Diferença que deve existir entre o código-fonte do aluno e a solução fornecida pelo professor para que aquele código-fonte receba um alerta ao término da execução da ferramenta;
5. ***Diff raw metrics loc***: Análogo ao item anterior;
6. ***Diff raw metrics lloc***: Análogo ao item anterior;
7. ***Diff raw metrics sloc***: Análogo ao item anterior;
8. ***Initial score***: Pontuação inicial para todos os códigos-fonte. Não necessita de um cálculo para ser definido, ficando a cargo do usuário a definição deste valor;
9. ***Decrease factor***: Fator de diminuição de pontuação; A pontuação é decrementada por este fator, quando a diferença entre a pontuação atribuída ao código-fonte do aluno com a pontuação da solução fornecida pelo professor, ambas pela Radon, for maior que zero;
10. ***Increase factor***: Fator de aumento de pontuação; A pontuação é incrementada por este fator, quando a diferença entre a pontuação atribuída ao código-fonte do aluno com a solução fornecida pelo professor, ambas pela Radon, for maior que zero;

11. ***Rate decrease to raw metrics***: Taxa de diminuição da pontuação para as métricas LOC, LLOC e SLOC; Esta taxa é aplicada diretamente no *decrease factor* o que significa, que ao decrementar a pontuação do código-fonte do aluno ela será decrementada por um percentual do *decrease factor* em vez de pelo seu valor completo. Esta forma de realizar o cálculo foi criada pois, durante os testes ao decrementar a pontuação, mais de 90% dos códigos-fonte estavam sendo duramente penalizados;

12. ***Rate increase to raw metrics***: Taxa de aumento de pontuação para as métricas LOC, LLOC e SLOC;

13. ***Score to great solution***: Pontuação necessária para que uma solução seja considerada ótima;

14. ***Score to not so good solution***: Pontuação para que uma solução esteja abaixo do esperado;

15. ***Result output txt***: Diretório onde deverá ser salvo o resultado geral da execução da ferramenta;

16. ***Result output csv***: Diretório onde deverá ser salvo o resultado geral resumido em forma de planilha;

17. ***Result output alert csv***: Diretório onde deverão ser salvos os resultados cujas respectivas soluções fornecidas precisarão ser olhadas com um maior cuidado pelo professor;

Aqui vale ressaltar que as configurações acima descritas não se inspiraram em algum outro trabalho e foram surgindo a medida que a ferramenta foi sendo desenvolvida. As principais configurações apresentadas serão vistas novamente no capítulo 5;

4.3 Análise e cálculo da pontuação dos códigos-fonte

Antes da análise ser iniciada é necessário que uma estrutura de diretórios seja criada para que assim ela seja possível, conforme ilustrado na Figura 8. Para que os arquivos sejam analisados é recomendado que seja criado um diretório geral e para cada problema ou exercício exista um diretório com seu nome para que assim seja possível diferenciá-los, dentro do diretório de cada problema é obrigatório existir outras duas pastas, a pasta “professor” deve conter o arquivo com a solução do docente, a qual deve ser uma boa solução considerando o conhecimento esperado do aluno, e o diretório “alunos” deve conter todas as soluções dos discentes para aquele problema.

Utilizando o analisador estático Radon, as métricas anteriormente citadas serão extraídas de todos os arquivos contidos na estrutura de diretórios também definida. Com as métricas extraídas, a ferramenta divide todos os códigos-fonte por problema em seguida pontua cada um deles com base na solução fornecida pelo professor e por fim a ferramenta gera os três arquivos de saída. O diagrama de fluxo de dados na Figura 9 exibe de forma resumida o que foi descrito.

A pontuação dos códigos-fonte é calculada utilizando as seguintes configurações definidas no arquivo de configurações:

- *Initial score*;
- *Increase factor*;
- *Decrease factor*;
- *Rate increase to raw metrics*;
- *Rate decrease to raw metrics*;

Quando a Radon extrai as métricas dos códigos-fonte elas são armazenadas em memória para serem utilizadas durante o cálculo. O primeiro passo é calcular a diferença entre cada métrica extraída do código-fonte do aluno com o código-fonte do professor, em seguida o resultado é interpretado, sendo eles:

- **resultado da métrica do aluno - resultado da métrica do professor > 0**: significa que a solução do aluno obteve um resultado inferior à solução fornecida pelo docente;
- **resultado da métrica do aluno - resultado da métrica do professor = 0**: significa que a solução do aluno obteve o mesmo resultado da solução do professor;
- **resultado da métrica do aluno - resultado da métrica do professor < 0**: significa que a solução do aluno obteve um resultado superior à solução fornecida pelo professor.

Como é feito o cálculo da pontuação para cada código-fonte após o cálculo da diferença:

- **Caso a diferença seja maior que zero** então, o *initial score* é decrementado em *decrease factor* * diferença;
- **Caso a diferença seja menor que zero** então, o *initial score* é incrementado em *increase factor* * diferença.

É importante salientar que, ao calcular o incremento ou decremento da pontuação, quando a métrica for LOC, LLOC ou SLOC o *increase factor* e o *decrease factor* serão o resultado do cálculo percentual deles por sua respectiva taxa (*rate increase to raw metrics* ou *rate decrease to raw metrics*) definidas no arquivo de configurações, o Código-fonte 9 ilustra o processo acima descrito. Ao processar e calcular a pontuação de todos os códigos-fonte é necessário definir quais códigos-fonte vão para o arquivo de alerta, para isso são utilizadas as configurações *score to great solution* e *score to not so good solution*. Caso a pontuação final do código-fonte seja igual ou maior ao *score to great solution* aquele código-fonte será adicionado ao arquivo de alerta, o mesmo vale caso a pontuação final seja igual ou inferior à *score to not so good solution*.

Figura 8 – Estrutura de diretórios

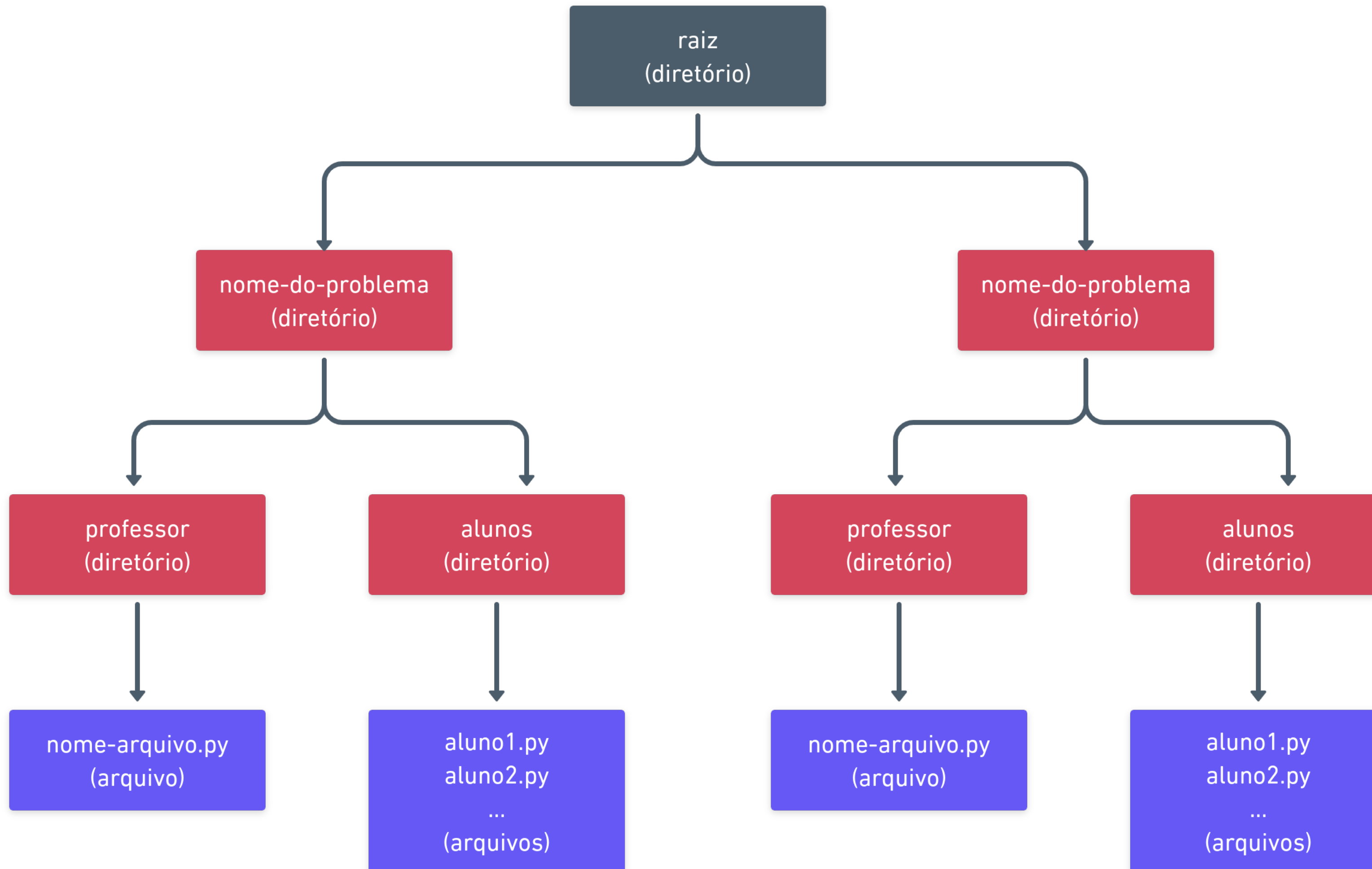
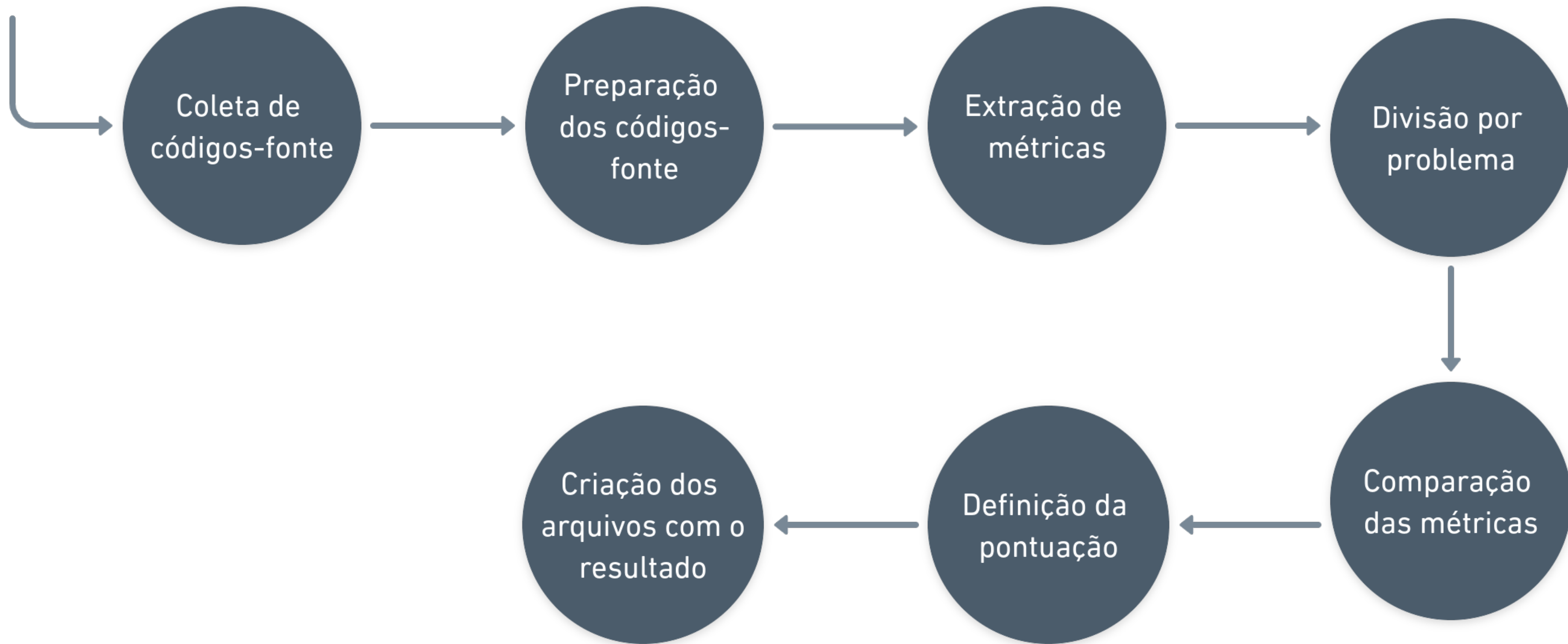


Figura 9 – Fluxo de dados



4.4 Resultados esperados

A partir deste momento toda solução do docente será chamada de solução base. Ao término da execução da ferramenta três arquivos serão gerados:

1. **result.txt**: Arquivo mais geral, dividido por problema onde para cada problema existe uma descrição contendo as métricas avaliadas da solução base seguida por um texto que faz comparação das métricas das soluções dos alunos com a base;

2. **result.csv**: Arquivo que possui uma forma mais resumida e tabelada sobre todas as métricas avaliadas, seus valores e uma pontuação final de cada submissão. O cabeçalho desse arquivo é exibido na Tabela 2;

3. **result-alert.csv**: Arquivo responsável por sinalizar ao docente submissões que necessitam de uma melhor atenção seja ela pela solução possuir uma pontuação muito alta ou baixa vide arquivo de configuração definido anteriormente. A Tabela 3 exibe o formato deste arquivo.

Vale a pena ressaltar que os nomes dos arquivos de saída citados acima são customizáveis e podem ser modificados via arquivo de configuração.

Tabela 2 – Modelo de resultado

PROBLEM	Nome do problema
SOLUTION	Nome do arquivo de solução
IS_TEACHER	Se a submissão é do professor ou não
CYCLOMATIC_COMPLEXITY	Valor (inteiro) da complexidade ciclomática
EXCEEDED LIMIT CC	Se excedeu o limite estabelecido de complexidade ciclomática
LINES OF CODE	Valor (inteiro) da quantidade de linhas de código
EXCEEDED LIMIT LOC	Se excedeu o limite da quantidade de linhas de código
LOGICAL LINES OF CODE	Valor (inteiro) da quantidade de linhas lógicas de código
EXCEEDED LIMIT LLOC	Se excedeu o limite da quantidade de linhas lógicas de código
SOURCE LINES OF CODE	Valor (inteiro) da quantidade de linhas de código-fonte
LIMIT SLOC	Se excedeu o limite da quantidade de linhas de código-fonte
FINAL SCORE	Pontuação final obtida após o processamento das métricas

Tabela 3 – Modelo de resultado de alerta

PROBLEM	Nome do problema
SOLUTION	Nome do arquivo de solução
ATTENTION_TYPE	Motivo da submissão estar neste arquivo (pontuação maior ou menor que o limiar)
SCORE	Pontuação

4.5 Linguagem de programação suportada pela ferramenta

Inicialmente, a linguagem de programação que será suportada pela ferramenta é a *Python*, visto que está sendo amplamente utilizada em cursos introdutórios de computação além de sua

popularidade. Dessa maneira tornará o processo de teste da ferramenta mais simples já que várias turmas introdutórias estão utilizando esta linguagem na Universidade Federal de Sergipe.

4.6 Linguagem de programação que a ferramenta será desenvolvida

Por conveniência a linguagem de programação escolhida para o desenvolvimento do analisador estático será também a *Python*, já que esta linguagem possui um conjunto grande de ferramentas e facilidades de desenvolvimento, tornando todo o processo relativamente mais rápido para que testes possam ser executados rapidamente.

5

Desenvolvimento

Este capítulo abordará os processos utilizados no desenvolvimento da ferramenta. Como descrito no objetivo geral, a ferramenta visa analisar estaticamente códigos-fonte extraindo métricas de *software* para que ao final de sua execução seja gerado um relatório contendo todas as submissões que necessitem de atenção extra por parte do professor e também gerando outros dois relatórios que descrevem de forma geral as métricas extraídas e pontua cada exercício. Utilizando o analisador de código-fonte Radon as métricas dos exercícios podem ser extraídas e posteriormente analisadas pela ferramenta. Todo o código-fonte da ferramenta está escrito na linguagem de programação *Python*, assim como os exercícios avaliados.

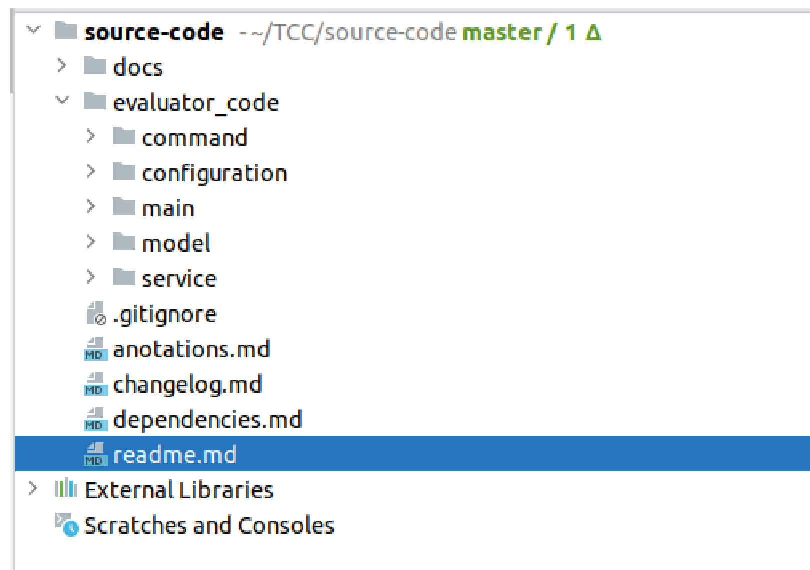
5.1 Estrutura da ferramenta

Esta seção tem como objetivo discorrer sobre toda a estrutura de pastas e arquivos da implementação da ferramenta, a Figura 10 exibe tal estrutura, sendo que:

- ***evaluator_code***: este diretório é onde estão contidos todos os códigos-fonte criados durante o desenvolvimento;
- ***command***: contém arquivos que encapsulam as informações necessárias para executar a ferramenta;
- ***configuration***: neste diretório está localizado o arquivo de configuração que guia a ferramenta durante sua execução;
- ***main***: aqui conta o arquivo *Main.py* que é responsável por chamar a função principal da ferramenta;
- ***model***: neste diretório temos a classe com os atributos que representam um código-fonte;

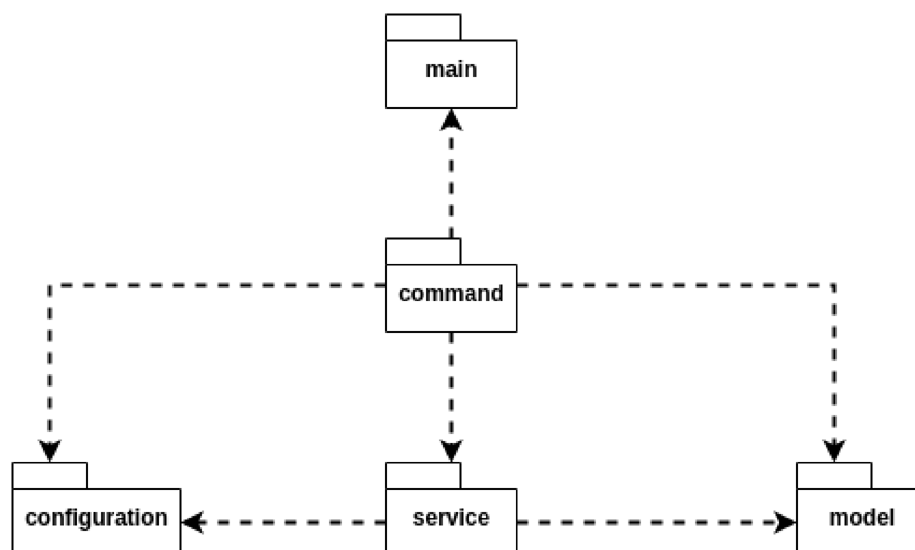
- ***service***: contém os módulos *Python* responsáveis por todas as regras do funcionamento da ferramenta, como extração e cálculo de métricas, comparador de submissões e um construtor de arquivos que é responsável por criar os arquivos de resultado.

Figura 10 – Estrutura de pastas da ferramenta



A Figura 11 traz o diagrama de pacotes da ferramenta. Em resumo os pacotes *main, model* e *configuration* são independentes entre si e os pacotes *command* e *service* possuem dependência com outros pacotes. O *command*, para funcionar, necessita do *service, model* e *configuration*. O *service* necessita do *model* e *configuration*.

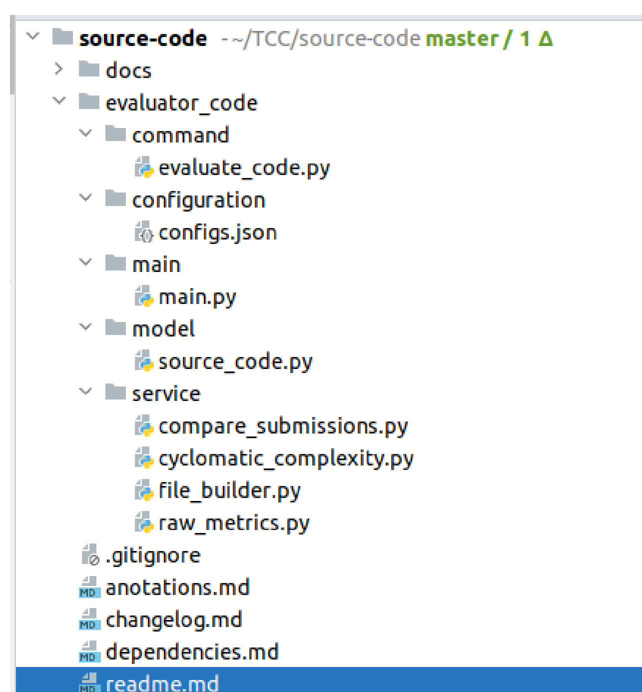
Figura 11 – Diagrama de pacotes



Na Figura 12 temos onde cada arquivo desenvolvido na ferramenta está localizado. A seguir uma breve descrição do que cada arquivo é responsável.

- ***evaluate_code.py***: implementa o fluxo de execução da ferramenta, é o responsável por realizar todas as chamadas aos serviços que extraem as métricas, avaliam e salvam os resultados;
- ***configs.json***: descreve as configurações exibidas na seção 4.2 deste trabalho;
- ***main.py***: responsável exclusivamente para chamar a função principal contida no arquivo ***evaluate_code.py***;
- ***source_code.py***: como a ferramenta avalia códigos-fonte, este arquivo implementa a classe ***SourceCode*** que é responsável por definir os atributos de um código-fonte, sua implementação está disponível no Código-fonte 4;
- ***compare_submissions.py***: tem como objetivo comparar cada métrica de cada código-fonte dos alunos com as métricas da solução base fornecida pelo professor;
- ***cyclomatic_complexity.py***: serviço responsável por extrair e calcular a complexidade ciclomática de todos os códigos-fonte;
- ***file_builder.py***: sua principal função é criar todos os arquivos de resultado da ferramenta;
- ***raw_metrics.py***: serviço que tem como função extrair e calcular todas as *raw metrics* (LOC, LLOC e SLOC) utilizadas na ferramenta.

Figura 12 – Localização dos arquivos desenvolvidos



Código-fonte 4 – Código-fonte do arquivo *source_code.py*

```
class SourceCode:
    def __init__(self, path):
        self.path = path
        self.content = []
        self.cyclomatic_complexity = []
        self.raw_metrics = []
        self.cyclomatic_complexity_result_txt = ''
        self.raw_metrics_result_txt = ''
        self.cyclomatic_complexity_result_csv = ''
        self.raw_metrics_result_csv = ''
        self.score = 0
        self.need_attention = False
        self.need_attention_type = ''

    def print_content(self):
        print(self.content)

    def extract_content(self):
        with open(self.path, 'r') as file:
            self.content = [line for line in file]
        return self.content

    def count_code_lines(self):
        return len(self.content)

    def print_attr(self):
        print(self.path)
        print(self.cyclomatic_complexity)
        print(self.raw_metrics)

    def extract_file_name(self):
        return self.path.split('/')[-1].replace('.py', '')

    def extract_problem_name(self):
        return self.path.split('/')[-3]

    def is_base_source_code(self):
        return self.path.split('/')[-2] == 'professor'
```

5.2 Decisões tomadas

Durante o desenvolvimento da ferramenta algumas decisões importantes foram tomadas e serão discutidas nos próximos parágrafos.

A princípio o arquivo de configurações teria apenas quatro ou cinco propriedades. Porém, com o decorrer do desenvolvimento, novas propriedades precisaram ser adicionadas

totalizando ao final dezessete propriedades de configuração. A Figura 13 traz a definição inicial das configurações, inclusive estas também foram utilizadas para realizar os testes da ferramenta. Vale lembrar que na seção 4.2 todas as configurações foram definidas.

Figura 13 – Definição inicial do arquivo de configurações

Todos os valores numéricos atribuídos às propriedades do arquivo de configuração foram definidos de forma empírica à medida que a ferramenta foi sendo testada, um melhor detalhamento sobre os testes é feito na seção 5.4. Para definir todos os valores numéricos, foi utilizada de tentativa e erro até que os resultados se tornassem mais homogêneos. Aqui é válido dizer que, para cada conjunto de valores numéricos atribuídos às propriedades do arquivo, um conjunto de resultados diferente será gerado, cabendo, assim, a quem vai executar a ferramenta defini-los da maneira que for melhor para seus objetivos, porém os valores definidos na Figura 13 são uma sugestão para utilização. Da maneira como foi definido o arquivo, a ferramenta consegue ter uma maior flexibilidade para ser executada.

Durante os testes da ferramenta, problemas foram encontrados na utilização do Radon, pois para que fosse possível extrair as métricas a partir do analisador, os códigos-fonte deveriam possuir uma formatação que seguisse o *Python Code Style*, porém ao se trabalhar com códigos-fonte de diversos estudantes iniciantes, cobrar que eles sigam o padrão definido não seria muito efetivo, por isso foi introduzido o formatador de código-fonte *Black* dessa forma sendo possível padronizar todos os códigos-fonte fornecidos para que fossem analisados pelo Radon.

O analisador de código-fonte Radon não consegue extrair a métrica de complexidade ciclomática de códigos-fonte caso estes não estejam modularizados, então foi decidido criar

uma estratégia para modularizar estes códigos-fonte para que assim essa métrica pudesse ser extraída. O Código-fonte 5 traz um código não modularizado antes da execução da ferramenta, já o Código-fonte 6 traz o mesmo código modularizado. Vale também dizer que esta abordagem foi criada apenas para o caso da complexidade ciclomática e ela não modifica os arquivos originais dos alunos, dessa maneira não afetando a coleta das *raw metrics*.

Código-fonte 5 – Exemplo de código-fonte não totalmente modularizado

```
1  def ordenar_imprimir(lista):
2      lista.sort()
3      for n in lista:
4          print(n)
5
6
7  maior_nome = None
8  nomes_yes = []
9  nomes_no = []
10 entrada = input()
11 while entrada != 'FIM':
12     nome, resp = entrada.split()
13     if resp == 'YES':
14         if nome not in nomes_yes:
15             nomes_yes.append(nome)
16         if maior_nome is None or len(maior_nome) < len(nome):
17             maior_nome = nome
18     else:
19         nomes_no.append(nome)
20     entrada = input()
21
22 ordenar_imprimir(nomes_yes)
23 ordenar_imprimir(nomes_no)
24
25 print('\nAmigo do Habay:')
26 print(maior_nome)
```

Código-fonte 6 – Exemplo de código-fonte modularizado após execução da solução criada

```
def ordenar_imprimir(lista):
    lista.sort()
    for n in lista:
        print(n)
def created_by_auto_code_1():
  maior_nome = None
  nomes_yes = []
  nomes_no = []
  entrada = input()
  while entrada != 'FIM':
      nome, resp = entrada.split()
      if resp == 'YES':
          if nome not in nomes_yes:
              nomes_yes.append(nome)
          if maior_nome is None or len(maior_nome) < len(nome):
              maior_nome = nome
      else:
          nomes_no.append(nome)
      entrada = input()
  ordenar_imprimir(nomes_yes)
  ordenar_imprimir(nomes_no)
  print('\nAmigo do Habay:')
  print(maior_nome)
```

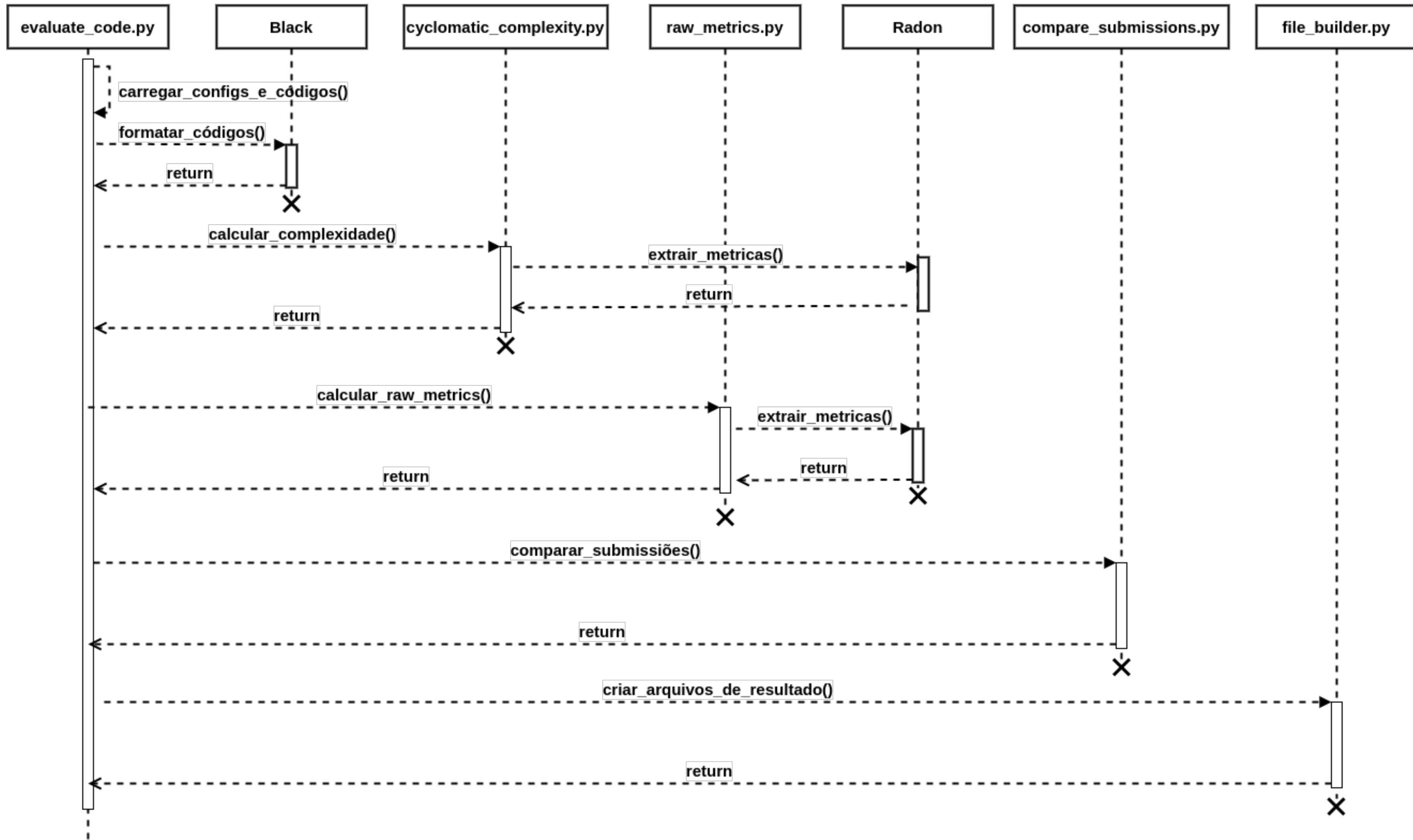
A complexidade ciclomática calculada pelo Radon é contabilizada para cada função definida em cada código-fonte, como não é possível definir que cada estudante crie funções com os mesmos nomes e em mesma quantidade. Assim, foi definido que para cada código-fonte, a complexidade ciclomática para cada função é somada e o valor final é a complexidade ciclomática daquele código-fonte.

5.3 Comportamento interno da ferramenta

Para explicar o funcionamento interno da ferramenta o diagrama de sequência da Figura 14 foi desenvolvido. No diagrama, o ator que dispara a ação principal foi omitido para minimizar o tamanho da imagem sem prejudicar seu entendimento.

Inicialmente o *script evaluate_code.py* responsável por coordenar todo o funcionamento da ferramenta, carrega as configurações definidas no arquivo *configs.json* e carrega também os códigos-fonte que serão posteriormente analisados. Em seguida todos os códigos-fonte são submetidos à ferramenta *Black* para que fiquem de acordo com o *Python code style*. O comando ***python -m black*** seguido do caminho para o arquivo faz com que aquele código-fonte seja estilizado seguindo o padrão da ferramenta *Black*.

Figura 14 – Diagrama de sequência



Para calcular a complexidade ciclomática o *script cyclomatic_complexity.py* é executado, a função que realiza esta ação é exibida no Código-fonte 7 o parâmetro -j utilizado na função indica ao Radon que retorne o resultado em formato JSON.

Código-fonte 7 – Função que calcula a complexidade ciclomática

```
1  def calculate_complexity(self):
2      """
3      Calculates the cyclomatic complexity of the source codes of a
          given directory
4      :return: List of the SourceCode class filling the cyclomatic
          complexity field
5      """
6
7      # Process all codes from directory and return json string with
          result
8      process_data_string = sp.getoutput('python3 -m radon cc ' +
          self.path + ' -j')
9
10     self.__extract_data(process_data_string)
11
12     self.__clean_data()
13
14     return self.sources_code
```

Para calcular as *raw metrics* o *script raw_metrics.py* é executado, o Código-fonte 8 ilustra a função que executa esta ação.

Código-fonte 8 – Função que calcula as *raw metrics*

```
1  def calculate_raw_metrics(self):
2      """
3      Calculate raw metrics:
4      loc = Total lines of code
5      lloc = Number of logical lines of code
6      sloc = Number of source lines of code
7      :return:
8      """
9      process_data_string = sp.getoutput('python3 -m radon raw ' +
          self.path + ' -j')
10
11     self.__extract_data(process_data_string)
12
13     return self.sources_code
```

O *script compare_submissions.py* é o responsável por comparar e pontuar cada uma das soluções fornecidas pelos alunos com a solução base fornecida pelo professor. A pontuação para as soluções é calculada de acordo com o Código-fonte 9 onde os parâmetros são:

- ***solution***: é uma solução do conjunto de soluções fornecidas pelos alunos;
- ***diff_in_points***: é a diferença em pontos da solução do aluno contra a solução base;
- ***is_raw_metrics***: booleano que indica se a pontuação a ser calculada é ou não uma *raw metric*.

Código-fonte 9 – Pontuação das soluções

```
1  def __calculate_score(self, solution, diff_in_points,
      is_raw_metrics=False):
2      increase_factor = self.configs['increase_factor']
3      decrease_factor = self.configs['decrease_factor']
4
5      if is_raw_metrics:
6          increase_factor *=
              (self.configs['rate_increase_to_raw_metrics'] / 100)
7          decrease_factor *=
              (self.configs['rate_decrease_to_raw_metrics'] / 100)
8
9      if diff_in_points > 0:
10         solution.score -= diff_in_points * decrease_factor
11     else:
12         solution.score += -diff_in_points * increase_factor
```

Ao término do processamento os resultados são gravados em arquivo pelo *script file_builder.py*, os dados utilizados para gravação em arquivo ficam armazenados em objetos da classe *source_code.py*. Todo o código-fonte da ferramenta está disponível no *GitHub*[1] através do endereço <https://github.com/victor-souza-vieira/tcc-ufs> lá é possível obter uma maior compreensão acerca do código-fonte e todo funcionamento da ferramenta.

5.4 Testes da ferramenta

Para realizar os testes da ferramenta, foi utilizada uma base com 100 soluções fornecidas por alunos ao juiz *online The Huxley*[2] para 10 problemas diferentes e para cada um desses problemas, uma solução base fornecida pelo professor. Inicialmente o arquivo de configurações foi definido de acordo com o ilustrado na Figura 13. Antes de executar a ferramenta, foi feita uma análise manual de todas as submissões, sem levar em consideração o funcionamento interno da ferramenta, com essa análise, foram extraídas soluções que foram consideradas precisarem de algum tipo especial de atenção. Das 100 soluções dos alunos, comparando manualmente com a solução base, 32 foram selecionadas para serem melhor avaliadas pelo professor. Após a execução da ferramenta, o arquivo final continha 36 soluções que foram consideradas pela

[1] Disponível em: <https://github.com/>
[2] Disponível em: <https://www.thehuxley.com>

ferramenta para serem melhor avaliadas. Todas as 32 selecionadas de forma manual estavam dentro do conjunto das 36 selecionadas pela ferramenta. Desta forma, mesmo com base em uma amostra relativamente pequena, a ferramenta se mostrou capaz de indicar quais soluções merecem uma atenção do professor com uma baixa taxa de erro.

6 Conclusão

O processo de avaliar códigos-fonte de alunos em cursos introdutórios de programação vem se tornando cada vez mais desafiador, visto que está havendo um aumento significativo de alunos nestes cursos. Por esta razão, o projeto desenvolvido neste trabalho vem para auxiliar os professores no processo de avaliar e fornecer um melhor *feedback* aos alunos acerca de seus códigos-fonte.

A ferramenta foi desenvolvida utilizando a linguagem de programação *Python* e os códigos-fonte que podem ser analisados por ela também estão escritos nesta linguagem. Um arquivo de configuração foi elaborado para que seja possível: definir local dos códigos-fonte que serão avaliados e onde serão armazenados arquivos de resultado, definir quais métricas serão avaliadas pela ferramenta e definir pontuações e taxas de incremento e decremento de pontos. Toda a ferramenta foi projetada para que seja possível adicionar novos tipos de métricas sem ser necessário alterar diversas partes diferentes do código-fonte original, dessa forma facilitando o processo de ampliação da ferramenta.

6.1 Trabalhos futuros

Visando a melhoria tanto da ferramenta quanto do processo de avaliação e *feedback* vários cenários de incremento da ferramenta são possíveis:

- Otimizar as variáveis definidas no arquivo de configurações para uma avaliação mais precisa, que pode ser baseada por problema;
- Adicionar novas métricas de qualidade de código-fonte para incrementar a avaliação e o *feedback* fornecido pelo professor aos alunos;
- Adicionar novas linguagens de programação à ferramenta para que seja possível ampliar a diversidade de linguagens;

- Automatizar a extração dos códigos-fonte do *The Huxley* removendo assim mais uma etapa manual do processo;
- Integrar a ferramenta ao *The Huxley*.

Referências

AMER, H.; HAROUS, S. Smart-learning course transformation for an introductory programming course. In: *2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT)*. [S.l.: s.n.], 2017. p. 463–465. Citado na página 11.

ARIFI, S.; ZAHI, A.; BENABBOU, R. Semantic similarity based evaluation for c programs through the use of symbolic execution. In: . [s.n.], 2016. v. 10-13-April-2016, p. 826–833. Cited By 3. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84994560588&doi=10.1109%2fEDUCON.2016.7474648&partnerID=40&md5=a90ebcef0a9b12e65ec0d2d645092630>. Citado 2 vezes nas páginas 30 e 32.

ARIFI, S. M. et al. Automatic program assessment using static and dynamic analysis. In: *2015 Third World Conference on Complex Systems (WCCS)*. [S.l.: s.n.], 2015. p. 1–6. Citado 4 vezes nas páginas 12, 15, 16 e 17.

ARIFI, S. M. et al. Automated fault localizing and correction in dynamically analyzed programs. In: *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*. [S.l.: s.n.], 2016. p. 587–592. Citado na página 15.

BRANDãO, L. de O.; RIBEIRO, R. da S.; BRANDãO, A. A. F. A system to help teaching and learning algorithms. In: *2012 Frontiers in Education Conference Proceedings*. [S.l.: s.n.], 2012. p. 1–6. Citado na página 11.

C, S. V. et al. Assessment of quality of program based on static analysis. In: *2019 IEEE Tenth International Conference on Technology for Education (T4E)*. [S.l.: s.n.], 2019. p. 276–277. Citado na página 12.

HARRISON et al. Applying software complexity metrics to program maintenance. *Computer*, v. 15, n. 9, p. 65–79, 1982. Citado na página 20.

HONGLEI, T.; WEI, S.; YANAN, Z. The research on software metrics and software complexity metrics. In: *2009 International Forum on Computer Science-Technology and Applications*. [S.l.: s.n.], 2009. v. 1, p. 131–136. Citado 3 vezes nas páginas 19, 20 e 31.

ISO-25010. *The ISO/IEC 25000 series of standards*. [S.l.], 2011. Disponível em: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=6>. Citado na página 18.

JACKSON, D. A software system for grading student computer programs. *Computers Education*, v. 27, n. 3, p. 171–180, 1996. ISSN 0360-1315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0360131596000255>. Citado 3 vezes nas páginas 29, 30 e 32.

LACCHIA, M. *Introduction to Code Metrics*. [S.l.], 2020. Disponível em: <https://radon.readthedocs.io/en/latest/intro.html#cyclomatic-complexity>. Citado 2 vezes nas páginas 23 e 24.

POžENEL, M.; FüRST, L.; MAHNIč, V. Introduction of the automated assessment of homework assignments in a university-level programming course. In: *2015 38th International Convention*

on Information and Communication Technology, Electronics and Microelectronics (MIPRO). [S.l.: s.n.], 2015. p. 761–766. Citado na página 12.

RONGAS, T.; KAARNA, A.; KALVIAINEN, H. Classification of computerized learning tools for introductory programming courses: learning approach. In: *IEEE International Conference on Advanced Learning Technologies, 2004. Proceedings*. [S.l.: s.n.], 2004. p. 678–680. Citado na página 11.

WANG, T. et al. Ability-training-oriented automated assessment in introductory programming course. *Computers Education*, v. 56, n. 1, p. 220–226, 2011. ISSN 0360-1315. Serious Games. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0360131510002241>. Citado na página 30.

WESIAK, G.; AL-SMADI, M.; GÜTL, C. Towards an integrated assessment model for complex learning ressources: Findings from an expert validation. In: *2012 15th International Conference on Interactive Collaborative Learning (ICL)*. [S.l.: s.n.], 2012. p. 1–7. Citado na página 11.