UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Pedro Henrique de Andrade Gomes

Inspeção de Código-Fonte como Subsídio para o Processo de Ensino e Aprendizagem de Qualidade de Software

São José do Rio Preto
2021

Pedro Henrique de Andrade Gomes

Inspeção de Código-Fonte como Subsídio para o Processo de Ensino e Aprendizagem de Qualidade de Software

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: CAPES

Orientador: Prof. Dr. Rogério Eduardo Garcia

São José do Rio Preto
2021

G633i

Gomes, Pedro Henrique de Andrade

Inspeção de Código-Fonte como Subsídio para o Processo de Ensino e Aprendizagem de Qualidade de Software / Pedro Henrique de Andrade Gomes. -- São José do Rio Preto, 2021

145 f. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto

Orientador: Rogério Eduardo Garcia

1. Ciências Exatas e da Terra. 2. Engenharia de Software. 3. Qualidade de Código. 4. Ensino da Qualidade. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Pedro Henrique de Andrade Gomes

Inspeção de Código-Fonte como Subsídio para o Processo de Ensino e Aprendizagem de Qualidade de Software

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: CAPES

BANCA EXAMINADORA

Prof. Dr. Rogério Eduardo Garcia
UNESP–Presidente Prudente
Orientador

Profª. Drª. Ellen Francine Barbosa
Professora Associada Doutora
Universidade de São Paulo

Prof. Dr. Marcelo Medeiros Eler
Professor Assistente Doutor
Universidade de São Paulo

São José do Rio Preto
22 de junho de 2021

Dedicatória

Ao meu orientador, meu maior incentivador.
À minha mãe e irmão, minha base.
À minha (quase) esposa, que esta comigo e com “meu mestrado”.
Ao meu pai, com carinho.
Aos que eu dedico, também agradeço.

Agradecimentos

Primeiramente à Deus, *"Porque Dele e por Ele, e para Ele, são todas as coisas"* – Rm 11:36.

Agradeço aos colegas pesquisadores e participantes e ex-participantes do Laboratório de Pesquisa em Engenharia de Software Aplicada – LaPESA e que estiveram comigo durante o período em que estive nesta Universidade (em especial ao Darlan Murilo Nakamura de Araújo, quem deu inicio ao projeto *Teacher Mate* e todo o suporte que precisei).

Também sou grato ao Departamento de Matemática e Computação (em especial ao Prof.Dr. Danilo Medeiros Eler e ao Assessor de Suporte Acadêmico Ms.Fernando Pacanelli Martins) e aos servidores da Seção Técnica de Pós-Graduação dos *campi* da Unesp de Presidente Prudente e de São José do Rio Preto. Ainda agradeço ao Programa de Pós-Graduação em Ciência da Computação pela oportunidade.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Um trecho de código em desacordo com boas práticas de programação pode não acarretar em problemas em um primeiro momento, mas a falta de clareza ocasionada por códigos mal escritos e/ou muito complexos, compromete o entendimento do mesmo. Essa dificuldade na compreensão do código torna custosas as atividades relacionadas. O código bem escrito é frequentemente associado à experiência do programador, razão pela qual as empresas têm procurado profissionais cada vez mais qualificados. Pesquisas apontam que o egresso não se sente preparado para o ingresso no mercado de trabalho, enfrentando dificuldade na hora de concorrer pelas melhores oportunidades de trabalho. Essa dificuldade evidencia uma lacuna entre a indústria e a academia. Essa lacuna é identificada por diversos pesquisadores, que propõem melhorias para o processo de ensino-aprendizagem, a partir da utilização, em ambiente acadêmico, de conceitos e ferramentas utilizados pela indústria de software. Ao analisar a condução do ensino de programação nas instituições de ensino, fica evidente um distanciamento entre o ensino de programação e o ensino de qualidade de código. Nesse cenário, este projeto tem como objetivo propor melhorias no processo de ensino–aprendizagem de programação e de qualidade, propondo uma abordagem que utilize a qualidade de software como subsídio para o ensino de programação, provendo ao professor diretrizes para o ensino de programação com foco em qualidade interna de código-fonte.

Palavras-chave: Engenharia de Software. Qualidade de Código. Ensino da Qualidade.

Abstract

A code snippet that disagrees with coding best practices may not cause problems at first, but the lack of clarity mainly caused by poorly written and/or very complex code, compromises the understanding of it. This difficulty in understanding code makes related activities costly. Well-written code is often related to the programmer's experience, which is why companies are looking for increasingly skilled professionals. Research indicates that graduates do not feel prepared for entry into the labor market, facing difficulty in competing for the best job opportunities. This difficulty highlights a gap between industry and academia. This gap is identified by several researchers, who propose improvements to the teaching-learning process, based on the use, in an academic environment, of concepts and tools used by the software industry. By analyzing the conduct of programming teaching in educational institutions, a gap is evident between programming teaching and code quality teaching. In this scenario, this project aims to propose improvements in the teaching–learning process of programming and quality, proposing an approach that uses software quality as a subsidy for programming teaching, providing the teacher with guidelines for programming teaching, focused on the source code internal quality.

Keywords: Software Engineering. Code Quality. Teaching of Quality.

Lista de Figuras

Lista de Tabelas

Sumário

CAPÍTULO

1

Introdução

A Qualidade de Código está relacionada à compreensibilidade, e é possível associá-la ao seguinte questionamento: quão fácil é para outros programadores entender um trecho de código e quão bem ele pode ser estendido e reutilizado?

Esse conceito é discutido e difundido na literatura (Fowler et al., 1999, Hunt et al., 2000, McConnell, 2004, Martin, 2009, Bloch, 2017), mostrando sua relevância no desenvolvimento de software. O grande interesse pelo tema se justifica na perspectiva de negócios, em que os custos associados a problemas de qualidade exercem impacto nos custos do projeto em desenvolvimento, não sendo raro casos em que as implicações orçamentárias associados à extrapolação de prazos corroboram para o insucesso do projeto.

Estes estudos mostram que um código mal estruturado, fora dos padrões e com vícios do programador afeta diretamente a qualidade do projeto e, consequentemente, os custos de desenvolvimento. Um código mal escrito dificulta a manutenção impactando negativamente no custo – o custo da manutenção pode chegar a 90% do custo total estimado (Dehaghani e Hajrahimi, 2013). A busca por alternativas para minimizar os efeitos de um código fonte com baixa qualidade interna é contante na indústria de software.

Apoiando-se no processo de Inspeção Contínua, ferramentas como CheckStyle[1], SonarQube™[2] e SQuORE[3] possibilitam ao desenvolvedor identificar trechos de código-fonte considerados mal escritos. Ao encontrar uma violação, as ferramentas informam o problema, sua localização, e em alguns casos, sugerem melhorias.

[1]Disponível em: https://checkstyle.sourceforge.io/
[2]Disponível em: https://www.sonarqube.org/
[3]Disponível em: https://www.squoring.com

1.1 Contexto

As disciplinas relacionadas ao ensino de programação não têm o foco na qualidade de código-fonte. Nos casos em que existe um direcionamento para a qualidade interna de software, estes estão associados com o docente, que aborda conceitos além do conteúdo programático da disciplina, proposto em ementas previamente aprovadas em conselhos de docentes.

As ementas delimitam o escopo das disciplinas oferecidas, indicando conteúdos apresentados nas disciplinas. O escopo está diretamente relacionado aos currículos de referência, que por sua vez, estão em conformidade com diretrizes curriculares nacionais. Esse contexto é consequência de não ter claramente definido em diretrizes curriculares e, consequentemente, em ementas de disciplinas de programação, a apresentação (ensino-aprendizagem) de qualidade interna de código-fonte juntamente com a programação.

1.2 Motivação

A literatura aponta que as exigências do mercado de trabalho na área de Tecnologia da Informação não são plenamente atendidas. Empresas relatam dificuldades em encontrar profissionais que cumpram os requisitos das vagas disponíveis. Por outro lado, profissionais relatam dificuldades em cumprir os requisitos das vagas (Sami et al., 2017).

Nesse sentido, Bailey e Mitchell (2006), Matturro (2013), Sami et al. (2017) têm se dedicado na identificação do que o mercado de trabalho espera de um aluno formado nos cursos relacionados ao desenvolvimento de software. Esses três estudos apontam a existência uma lacuna entre o perfil do egresso e o esperado pelas empresas.

Por se tratar de uma questão recorrente, a preparação do aluno para ingressar no mercado de trabalho tem chamado a atenção de pesquisadores (de Andrade Gomes et al., 2017, Dietz et al., 2018, Scatalon et al., 2019). Fica evidente que a crescente necessidade de preparar o aluno para produzir um software com qualidade tem se mostrado um desafio aos educadores.

Embora alunos recém ingressos na universidade sejam encorajados a melhorar a qualidade interna de seus programas desenvolvidos, só nos anos posteriores são apresentados às disciplinas de Engenharia de Software e ensinados conceitos fundamentais como a Garantia da Qualidade.

A partir da análise dos conceitos ministrados nos cursos de Bacharelado em Ciência da Computação nas principais instituições de ensino do país, evidencia-se uma lacuna entre o ensino de programação e o ensino de qualidade de código. Estudantes aprendem os conceitos básicos de programação antes de entender a importância da aplicação correta dos mesmos, sendo comum a produção de código com baixa qualidade (Dietz et al., 2018).

1.3 Justificativa

O fornecimento de informações a respeito da qualidade interna de código-fonte ao educador possibilita identificar as faltas mais cometidas e as mais impactantes na qualidade de um software, permitindo ao professor uma visão detalhada do nível de competência técnica de seus alunos. Assim, permite ao professor elaborar uma estratégia para sanar dificuldades.

Na perspectiva do ensino de programação, é possível identificar trechos de código em desacordo com conceitos ministrados pelo professor, como problemas de projeto e arquitetura de código, de padronização e estilo, dentre outros, indicando conteúdos ministrados que não foram suficientemente esclarecedores ao aluno.

Concomitantemente, na perspectiva de qualidade de código, torna-se possível identificar as faltas mais cometidas e as mais impactantes na qualidade de um software, permitindo ao educador uma visão mais detalhada da competência técnica dos alunos, dando possibilidade de elaborar uma estratégia para guiar o ensino de maneira a sanar essas dificuldades.

Ademais, possibilitar ao aluno entender problemas ocasionados por suas ações incorretas, e quão grande é o impacto em um projeto de software, pode levar ao não cometimento da falta identificada, gerando um olhar mais crítico e colaborando para o ganho de habilidades como programador.

A antecipação desse processo de compreensão da qualidade interna permite, ao ainda aluno, um maior período dentro da acadêmia para desenvolver competências relacionadas, consideradas importantes para a indústria.

Mais concretamente, a justificativa consiste em introduzir conceitos de qualidade interna de código-fonte em disciplinas introdutórias de programação sem que acarrete trabalho extra ao docente em demasiado. A melhoria (ou adequação) do material de ensino utilizado pelo docente juntamente com o *feedback* das avaliações dos trabalhos também justificam esta proposta, uma vez que isso permite ao discente uma melhor compreensão e, consequentemente, aplicação dos conceitos.

1.4 Formulação do Problema

Diante do exposto, é possível fragmentar o problema na perspectiva do processo de ensino-aprendizagem, do professor e do aluno. Nesse sentido, o problema consiste em:

1. introduzir o ensino de qualidade interna em disciplinas de programação;
2. o material utilizado atualmente não trata do assunto;
3. o trabalho necessário (e extra) do docente para que essa introdução seja efetivada;

4. o aluno não tem um *feedback* sobre a qualidade interna do seu código produzido nas disciplinas de programação.

O desafio é tornar possível ao educador detectar de maneira mais eficiente e eficaz as faltas cometidas por um aluno e/ou por um grupo de alunos a partir da análise de código-fonte.

1.5 Objetivo

O projeto tem por objetivo apoiar docentes que atuam em disciplinas iniciais de programação para, além de tratar de conceitos básicos inerentes às suas disciplinas, também tratem de qualidade interna de código-fonte.

Como objetivos específicos, tem-se:

- criar uma ferramenta computacional de análise e registro de faltas cometidas por alunos de programação, seja por uso não adequado de algum conceito teórico, de alguma tecnologia ou organização interna do código-fonte. Com essa ferramenta espera-se minimizar o esforço adicional do docente para avaliar os trabalhos dos alunos, na perspectiva de qualidade interna.
- criar um conjunto de diretrizes, de modo a subsidiar o aprimoramento do material didático em programação e para evitar as faltas identificadas durante a análise de códigos-fonte produzidos por alunos.
- prover ao aluno a lista de faltas cometidas, na perspectiva de qualidade interna, juntamente com um subsídio para sua correção.

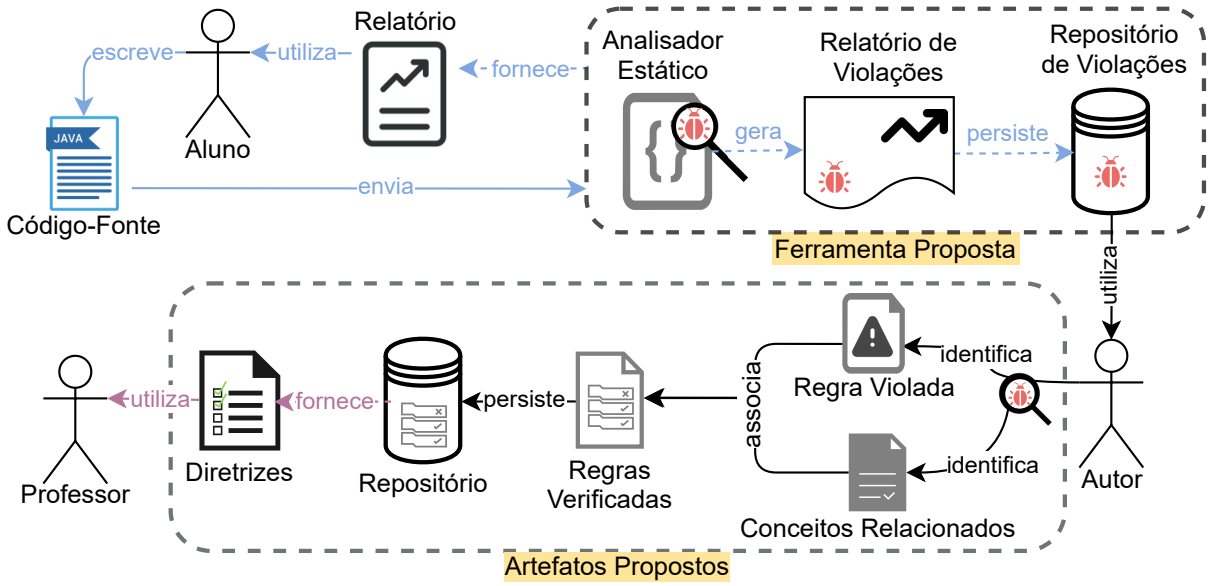


Figura 1.1: Esquema representando o objetivo deste trabalho (Produção Própria)

O objetivo é alcançado com a criação de um ambiente de trabalho como esquematizado na Figura 1.1. Nesse ambiente, o aluno submete o código-fonte

(CF) produzido à ferramenta de análise estática. A ferramenta analisa o CF para identificar trechos de código em desacordo com regras de qualidade e, ao finalizar as análises, gera um relatório contendo as violações encontradas, que por sua vez são persistidas em um repositório de violações. Em seguida, as regras violadas devem ser analisadas para identificar conceitos de programação incorretamente empregados pelos alunos. A partir dessa análise, o mapeamento entre as regras violadas e os conceitos relacionados deve ser estabelecido e mantido. O envio dessas diretrizes para o professor subsidia o ensino direcionado às dificuldades observadas na turma. Os artefatos produzidos são armazenados em repositórios, sendo utilizados como base de conhecimento para o processo de ensino-aprendizagem.

1.6 Organização

Para apresentar a proposta de utilização do código-fonte como subsídio para o ensino de qualidade de código associada ao ensino de programação, esta monografia encontra-se dividida em 6 capítulos, como segue:

- No Capítulo 2 é apresentado um panorama do mercado da Indústria de Software, destacando os principais desafios encontrados pela indústria e pelo desenvolvedor. São também apresentados os conceitos de Qualidade de Código e alguns padrões e normas que ajudam a definir a qualidade e a apoiar o desenvolvimento de software com qualidade.
- No Capítulo 3 é apresentado um panorama do ensino de programação nos principais cursos de Bacharelado em Ciência da Computação. Sendo apresentadas propostas para sanar eventuais desafios no processo de ensino-aprendizagem de programação e qualidade de código.
- Baseado nos conceitos apresentados nos Capítulos 2 e 3, o Capítulo 4 contém os detalhes do desenvolvimento do trabalho.
- No Capítulo 5 são apresentados resultados da aplicação da metodologia proposta.
- Por último, no Capítulo 6 são fornecidas algumas considerações sobre o projeto.

CAPÍTULO

2

Mercado de Tecnologia da Informação e a Qualidade de Código

O profissional de Tecnologia da Informação (T.I.) tem ocupado funções estratégicas dentro de empresas em diferentes segmentos de atuação. O Setor de T.I. deixou de ser uma área da empresa relacionada com a manutenção e passou a fazer parte das estratégias de negócio (Melville et al., 2004, Luftman et al., 2017).

Essa mudança no cenário é percebida tanto nas empresas de Tecnologia da Informação, como nas que a utilizam enquanto atividade meio, refletindo em um aumento na demanda por profissionais da área com habilidades bem desenvolvidas. Neste estudo, considera-se parte da industria de software as empresas que possuem na tecnologia da informação a sua atividade fim, com destaque para o desenvolvimento de software.

Por outro lado, empresas que se apoiam em alguns conceitos e/ou ferramentas como suporte ao processo ou parte do negócio não possuem nomenclatura definida, sendo neste estudos tratadas como parte do mercado.

Na primeira parte deste capítulo, são apresentados estudos que demonstram o panorama do mercado de tecnologia da informação de diferentes perspectivas, destacando os principais desafios encontrados tanto pelos empregadores, quanto pelos profissionais atuantes na área.

Na parte seguinte, são apresentados os principais conceitos de qualidade, a importância da qualidade de código, seus atributos e aplicações.

2.1 Competências do Profissional de T.I.

Conforme o setor de T.I. se expande dentro das empresas, a demanda por profissionais aumenta. Surgem novas ofertas de emprego, ocasionando um aumento na disputa pelas melhores posições do mercado de trabalho (Bailey e Mitchell, 2006).

Analisando o comportamento do mercado e da indústria e baseados no tipo de exigência de ambos, estudiosos dividem as competências do profissional de T.I. em duas categorias: competências técnicas (*hard skills*) e competências interpessoais (*soft skills*) (Bailey e Mitchell, 2006, Matturro, 2013, Sami et al., 2017).

2.1.1 Competências Técnicas

No grupo das competências técnicas estão presentes habilidades adquiridas na maioria dos cursos relacionados ao desenvolvimento de software. São competências relacionadas com a utilização de ferramentas, processos e métodos, como saber modelar, entender algoritmos e os fluxos de execução, programar em determinada linguagem de programação, testar, analisar, projetar e definir a arquitetura de sistemas, de redes de computadores, etc.

2.1.2 Competências Interpessoais

O grupo das competências interpessoais é composto por características dificilmente exploradas na academia. Essas características são associadas ao perfil psicológico e social dos alunos, como capacidade de trabalhar em equipe, de gerenciar projetos, de resolver problemas e de se comunicar.

Na tentativa de apoiar o aluno no desenvolvimento dessas competências pessoais, alguns estudiosos propuseram abordagens de ensino com foco em permitir ao o aluno o desenvolvimento dessas competências, como apresentado por Garcia et al. (2015).

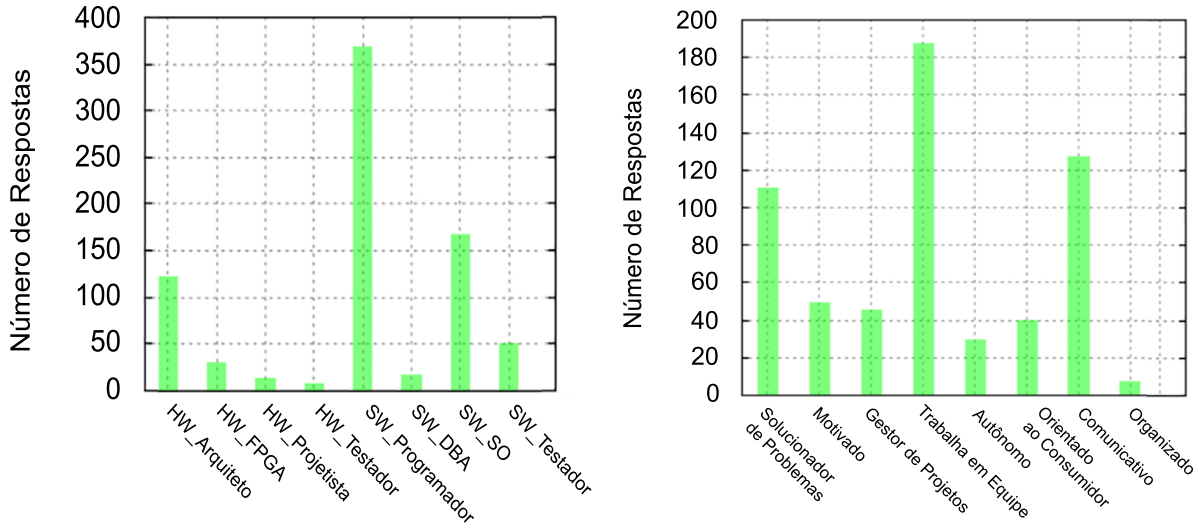
2.2 Análise de Mercado

O interesse na inserção de recém graduados na área de T.I. motiva diversos pesquisadores a analisar o mercado de tecnologia da informação. A seguir, são apresentados estudos sobre o perfil de profissionais requisitados pelas empresas, tipos de vagas ofertadas e dificuldades apontadas na transição da academia para o mercado.

2.2.1 Estudo de Caso I

Sami et al. (2017) analisam as oportunidades baseados nas vagas de trabalho ofertadas, no ramo da tecnologia da informação, no mercado internacional e no mercado suíço. Na primeira fase da pesquisa, os autores analisam as

competências requeridas pelas empresas. As competências são agrupadas em dois grupos: competências técnicas e competências interpessoais. Cada grupo define um conjunto de habilidades especificas.



(a) Competências técnicas.

(b) Competências interpessoais.

Figura 2.1: Competências identificadas em vagas oferecidas no mercado internacional.Adaptado de: Sami et al. (2017).

Metade das vagas analisadas são provenientes de um banco de ofertas de uma conhecida instituição de ensino. Já a outra metade era composta por vagas oferecidas em *websites* conhecidos como a rede social *Linkedin*, representando as ofertas internacionais de emprego. Nos dois casos, foram consideradas apenas ofertas de trabalho para estudantes recém graduados.

Ao analisar a Figura 2.1a, fica evidente a alta procura por programadores, aproximadamente três vezes maior que o segundo tipo de competência técnica mais buscada.

No caso das competências interpessoais, se destacam a busca das empresas por pessoas capazes de trabalhar em equipe, que saibam resolver problemas e consigam se comunicar, conforme demonstrado na Figura 2.1b.

Ao avaliar o mercado na perspectiva dos empregadores, Sami et al. (2017) evidenciam a preferência dos empregadores em não submeter o novo empregado a extensivos treinamentos. Ao contratarem, espera-se que o nível técnico do recém contratado seja suficiente, assim como a capacidade de se adaptar à equipe, podendo começar rapidamente a desempenhar as atividades para as quais foi contratado.

2.2.2 Estudo de Caso II

Matturro (2013) analisam as competências requeridas pelas empresas para preencher uma vaga de emprego na área de Engenharia de Software. Após essa análise os autores comparam os resultados obtidos com os encontrados

em outros estudos.

Parte dos dados coletados pertencem a banco de oportunidades da instituição de realização do estudo, sendo também coletados, dados de vagas oferecidas em classificados de jornais locais.

Tabela 2.1: Frequência de vagas em que aparecem as competências técnicas com/sem competências interpessoais (H.I.). Adaptado de Matturro (2013).

Processo de Software	**Com H.I.**	**Sem H.I.**	**Total**
Engenharia de Requisitos	35	14	49
Arquitetura de Software	15	13	28
Codificação/Manutenção	395	146	541
Garantia de Qualidade	43	17	60
	488	**190**	**678**

Segundo os autores, as empresas manifestam significativo interesse em competências interpessoais; – alguns casos, avaliadas com o mesmo grau de importância, se comparadas as competências técnicas.

Nesse segundo estudo, a demanda por programadores é maior, comparada ao estudo feito por Sami et al. (2017). Observa-se na Tabela 2.1, que das 678 vagas analisadas, 541 delas são vagas destinadas para programadores, representando quase 80% das vagas para a área de Engenharia de Software.

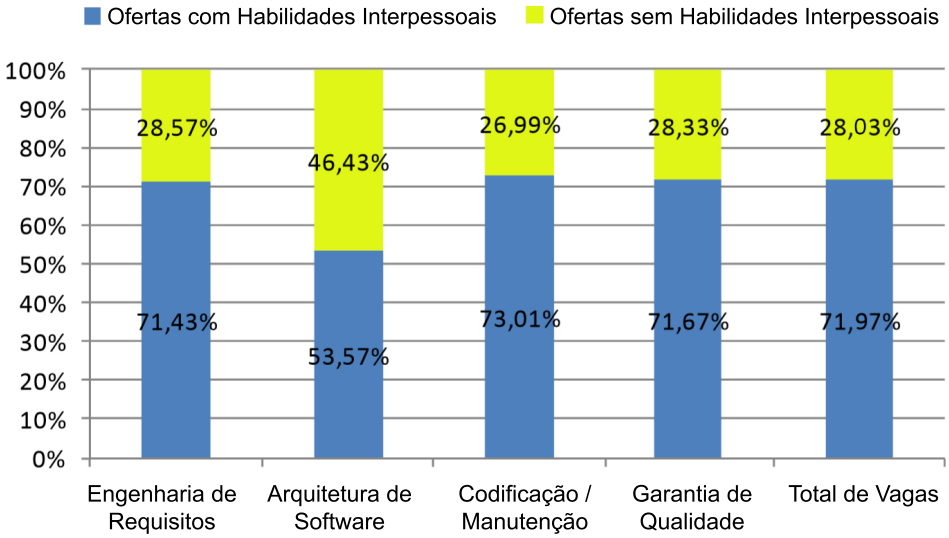


Figura 2.2: Relação das vagas com e sem competências interpessoais descritas. Adaptado de: Matturro (2013).

Na Figura 2.2, as oportunidades de emprego na área do desenvolvimento destacam competências interpessoais em sua descrição. Salvo o projetista de software, as demais vagas ultrapassam em 70% os casos em que são exigidas ao menos uma competência interpessoal, totalizando 71,97% das vagas analisadas. Fica evidente uma similaridade entre as competências exigidas em

oportunidades para trabalhar com desenvolvimento de software.

É importante que as universidades e cursos da área de T.I. se preocupem cada vez mais com o desenvolvimento da capacidade técnica de seus alunos, além do desenvolvimento de competências interpessoais.

2.2.3 Estudo de Caso III

Bailey e Mitchell (2006) coletam informações de 227 profissionais da área de Tecnologia da Informação e analisam qual o conhecimento, competências e habilidades necessárias para um programador. O diferencial desse estudo é a análise do ponto de vista dos próprios programadores. Os resultados podem ser observados na Tabela 2.2.

Tabela 2.2: Competências consideradas mais importantes de acordo com os profissionais. Adaptado de Bailey e Mitchell (2006)

Competência Exigida	Tipo	%
Ler, Entender e Modificar Programas de Terceiros	Técnica	96,89%
Codificar Programas	Técnica	96,48%
Debugar Softwares	Técnica	96,43%
Habilidade em Compreender	Pessoal	92,09%
Solucionador de Problemas	Pessoal	91,30%
Trabalhar em Equipe	Pessoal	89,93%
Conhecimento em Programação Estruturada	Técnica	89,82%
Projetista de Software	Técnica	88,99%
Habilidade em várias Linguagens de Programação	Técnica	88,94%

Nota-se que quase a totalidade dos profissionais consideram importante as habilidades de ler, entender, modificar, codificar e debugar (depurar) programas. Todas essas habilidades são associadas ao código-fonte. São destacadas ainda, habilidades como a capacidade de compreensão, saber resolver problemas e trabalhar em equipe, todas no âmbito das competências interpessoais.

2.3 Conceitos de Qualidade

Qualidade é definida por Crosby (1980) como: “a conformidade aos requisitos”. De fato, para determinar se algo possui ou não qualidade, é necessário uma referência, algo que se pode utilizar para comparar. A busca pelo entendimento e pela correta definição do que é qualidade continuou pelos anos seguintes. É possível encontrar similaridade do que foi afirmado por Crosby (1980) e o que consta na NBR/ISO-9000 (2015), “qualidade é o grau no qual um conjunto de características inerentes satisfaz aos requisitos".

No Processo de Software, **Qualidade de Software** é uma área da Engenharia de Software que, segundo Bourque et al. (2014), se refere “às características desejadas de produtos de software, à extensão em que um produto de

software em particular possui essas características e aos processos, ferramentas e técnicas que são usadas para garantir essas características."

A definição de Qualidade de Software é abrangente, apoiando o processo de software e seus produtos. Uma subárea de grande interesse deste estudo é a Qualidade de Código.

A definição formal de Qualidade de Código envolve diversos conceitos. Não existe uma padronização específica para qualidade de código, mas um padrão amplamente aceito é definido na ISO/IEC 9126-1 (ISO/IEC-9126-1, 2003). O modelo define a qualidade do produto de software usando seis atributos: funcionalidade, confiabilidade, usabilidade, eficiência, capacidade de manutenção (manutenibilidade) e portabilidade.

A ISO/IEC-9126 foi substituída pela ISO/IEC-25010 (2011), não sendo mais recomendada a sua utilização pelo órgão responsável. Também chamada de *SQuaRE*, acrônimo para Requisitos e Avaliação de Qualidade de Produto de Software, a nova norma define mais algumas características que um software deve possuir para ser considerado de qualidade. Foram definidos dois novos atributos de qualidade: a Compatibilidade e a Segurança.

Nenhuma alteração ocorre em desacordo com a ISO/IEC-9126, sendo que a substituição teve como principal motivo para adaptação as novas tecnologias, as ferramentas e as tendências que acompanham o desenvolvimento de software.

Além da pesquisa, o estudo da Qualidade de Software na academia é normalmente baseado nas normas ISO/IEC-9126 e ISO/IEC-25010, nas quais há a definição de qualidade de software mais difundida no meio acadêmico: "Qualidade de Software é a capacidade do produto de software em satisfazer as necessidades implícitas e explícitas, quando usado em condições específicas".

Por possuir ampla fundamentação teórica e ser mais consolidada tanto na indústria como na academia, sendo amplamente utilizada na literatura (Suwawi et al., 2015, Rochimah et al., 2015, Djouab e Bari, 2016, Al-Obthani e Ameen, 2018, Pérez et al., 2018, Andry et al., 2018), o trabalho proposto neste projeto de mestrado se apoia nas definições presentes na ISO/IEC-9126. Essa norma divide a qualidade do produto de software em qualidade interna, externa e de uso, além da relação entre elas. O foco desta proposta é a qualidade interna, detalhada adiante, neste capítulo. A seguir, destaca-se a importância da qualidade de código, suas classificações e atributos mais desejáveis.

2.3.1 Importância da Qualidade de Código

Com objetivo de diminuir a complexidade das atividades inerentes à escrita e leitura de código, a indústria de software e a academia constantemente propõem melhorias para a qualidade do código-fonte. Pesquisas relacionadas a Qualidade de Código demonstram as vantagens de investir na mesma a médio

e longo prazo. Um código bem escrito é de fácil entendimento, ocasionando uma diminuição no esforço para entende-lo, que por razões descritas a seguir, podem impactar nos prazos e custos do projeto (Lewis, 2016, Dietz et al., 2018).

Crosby (1980) definem qualidade como "a conformidade aos requisitos". Apoiado por normas como a ISO/IEC-9126-1 (2003) e ISO/IEC-25010 (2011), o estudo de qualidade de software permite o entendimento de como se obtém um produto de software com qualidade. As normas classificam os atributos de qualidade do software conforme a percepção dos mesmos, sendo providas métricas para cara um dos grupos.

No código-fonte, é possível medir atributos de qualidade como a manutenibilidade, coesão, adequação ao estilo e padrões, entre outro, sendo possível evoluir essas características a partir de processos de melhoria contínua. A falta de padrões de projeto, de arquitetura e de estilo tornam o código difícil de manter e impactam na complexidade e custos do projeto (Baggen et al., 2012).

O uso correto de técnicas, métodos e ferramentas propostos pela disciplina de Engenharia de Software permitem ao desenvolvedor minimizar a maioria dos problemas enfrentados durante o desenvolvimento. A seguir são apresentados alguns conceitos e soluções propostas pela Engenharia de Software relacionados com o escopo deste estudo.

A tarefa de identificar um software com baixa qualidade de código, nem sempre é trivial. O usuário comum pode nem entender o que é codificação, mas existem diversas razões pelas quais a qualidade do código é importante. Com o tempo, o usuário pode se deparar com situações em que o software em execução realiza um comportamento inesperado, desde lentidão até situações de total inoperabilidade com risco de perda de informação.

Fowler, M. (2019) afirma que o papel fundamental da qualidade interna é a redução de custos para problemas futuros. O esforço adicional para escrever um bom código implica em um aumento nos custos a curto prazo.

O desenvolvimento inicial é mais rápido quando não há preocupação com a qualidade de código, mas com o passar do tempo fica mais difícil adicionar novos recursos. Mesmo pequenas mudanças exigem que os programadores entendam grandes áreas de código, sendo esses códigos difíceis de entender. As alterações geram problemas, levando a longos períodos de teste e diversos defeitos que precisam ser corrigidos.

Ainda segundo Fowler, M. (2019), a dificuldade em medir precisamente as funcionalidades desenvolvidas por uma equipe. Essa incapacidade de medir a produção e, portanto, a produtividade, torna difícil precisar as consequências da baixa qualidade interna. Ao conversar com diversos programadores

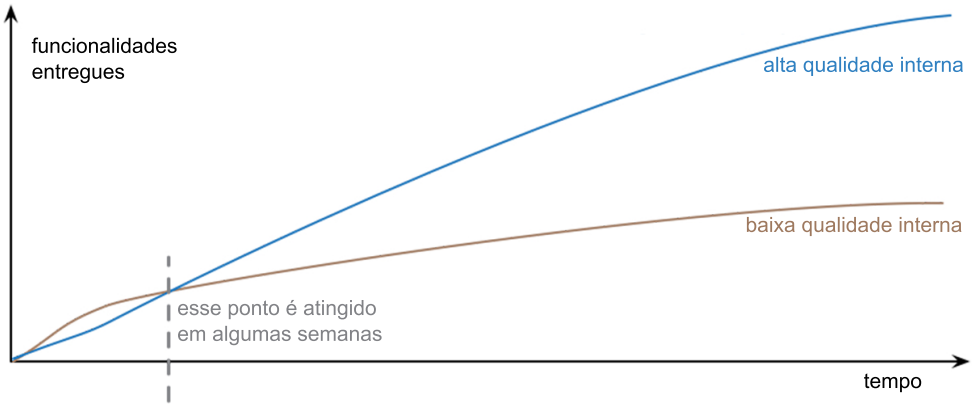


Figura 2.3: Exemplo da relação entre o desenvolvimento de funcionalidades e o tempo para códigos com e sem qualidade interna. Adaptado de: Fowler, M. (2019)

experientes, a constatação do autor foi de que em poucas semanas de desenvolvimento os participantes começam a identificar uma maior lentidão para execução das tarefas de escrita de código (Figura 2.3).

2.3.2 Qualidade Interna, Externa e de Uso

Desenvolvedores e usuários estão constantemente preocupados com a qualidade, mas a percebem de perspectivas diferentes. Os usuários identificam características de qualidade enquanto estão fazendo uso do software. Já os desenvolvedores podem ser afetados enquanto em etapas de manutenção e de adição de novas funcionalidades.

A segunda, terceira e quarta parte da ISO/IEC-9126 definem métricas de qualidade interna, externa e de uso, respectivamente, a serem utilizadas com o modelo de qualidade definido na primeira parte da norma. As definições a seguir foram adaptadas da ISO/IEC-9126-3 (2003), sendo as mesmas definições encontrada na segunda e quarta parte da norma.

- **Métricas Internas** - podem ser aplicadas a um produto de software não executável durante seu desenvolvimento (como na definição de requisitos, especificação da arquitetura ou no código-fonte). As métricas internas fornecem aos usuários a capacidade de medir a qualidade dos produtos intermediários e, assim, prever a qualidade do produto final. Isso permite que o usuário identifique problemas de qualidade e inicie a ação corretiva o mais cedo possível no ciclo de vida do desenvolvimento do software.

- **Métricas Externas** - podem ser usadas para medir a qualidade do produto de software, medindo o comportamento do sistema do qual faz parte. As métricas externas só podem ser usadas durante os estágios de teste do ciclo de vida do software e durante qualquer estágio operacional. A

medição é obtida ao executar o produto de software no ambiente do sistema no qual ele se destina a operar.

- **Métricas de Qualidade em Uso** - medem se um produto atende às necessidades de usuários especificados para atingir metas especificadas com eficácia, produtividade, segurança e satisfação em um contexto de uso especificado. Tais métricas são obtidas com o software em utilização no ambiente para o qual foi pretendido.

É possível observar na Figura 2.4 a relação entre os tipos de métricas descritos pela ISO/IEC-9126-3 (2003). Fica evidente a influência da qualidade interna na qualidade externa do produto de software, sendo a qualidade em uso influenciada pela qualidade externa e consequentemente, também pela qualidade interna.

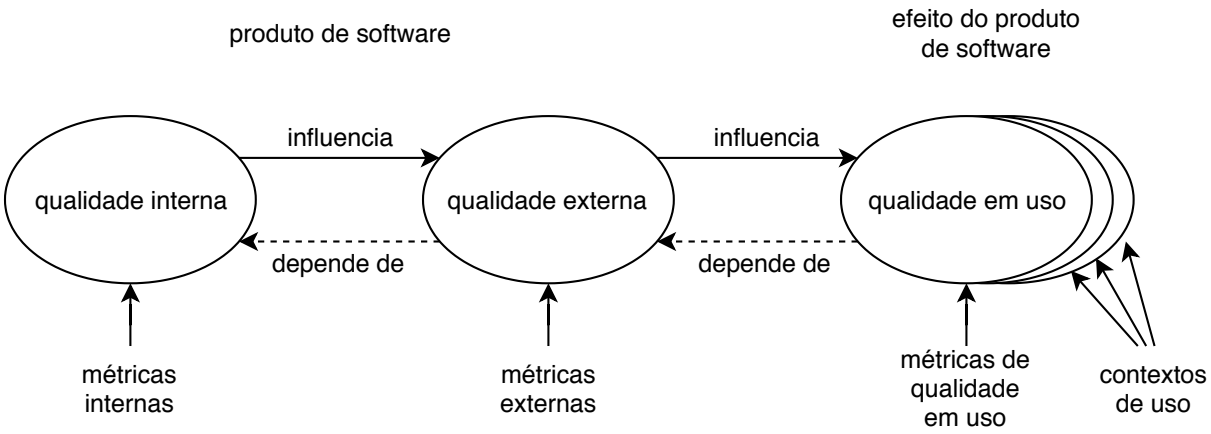


Figura 2.4: Relação entre os tipos de métricas. Adaptado de: ISO/IEC-9126-3 (2003)

A qualidade externa pode ser percebida por desenvolvedores principalmente durante o teste de software. Ao executar o produto de software em um ambiente de testes, são analisados os valores de entrada e saída e o desempenho afim de verificar se o comportamento do produto está de acordo com o que foi definido (Tonella e Lemma Abebe, 2008).

A qualidade interna, relacionada com atributos como a capacidade de manutenção, coesão, adequação ao estilo e padrões de linguagem são mais propensas a se refletir nas estruturas estáticas do software. Algumas faltas cometidas durante o desenvolvimento podem parecer irrelevantes, porém após sucessivas alterações, advindas de processos de manutenção, é comum que o software comece a sofrer os efeitos da deterioração. Esses efeitos começam a impactar no processo de manutenção, tornando-o complexo, custoso e muitas vezes não satisfatório (Glass, 2001).

Tendo foco no produto, Mari et al. (2003) utilizam termos descritivos alternativos sendo qualidades internas definidas como qualidades de evolução e qualidades externas como qualidades de execução.

Embora o conjunto de normas ISO/IEC-9126 forneçam um importante conjunto de parâmetros com o objetivo de padronizar a avaliação da qualidade de software (o que inclui a qualidade de código), a própria norma explica que os requisitos de qualidade devem ser fornecidos de acordo com o projeto, o negócio e os requisitos, não existindo assim um modelo específico de código com qualidade.

Embora seja um conceito bem definido, a qualidade de código não possui uma norma específica. Neste sentido, foi realizada uma revisão bibliográfica com o intuito de identificar os atributos relacionados com a qualidade interna mais desejáveis e qual o impacto deles no projeto, tanto no processo, quanto no produto, sendo esta apresentada a seguir.

2.3.3 Atributos de Qualidade de Código

A ISO/IEC-9126-3 (2003) define um conjunto de métricas relacionadas com a qualidade interna do produto. A mesma norma define que a utilização das métricas pode se alterar para se adequar as necessidades do projeto, indicando novamente que a definição de qualidade varia de acordo com os atributos desejáveis.

Nesse sentido, estudiosos como Sullivan et al. (2001), Baxter et al. (2004), Myers et al. (2004), Baggen et al. (2012) concentram-se na identificação dos atributos de qualidade interna de código, mais desejáveis pelos desenvolvedores. A seguir são descritos os principais atributos de qualidade de código identificados na literatura.

- **Sustentável** - dado que a maioria dos desenvolvedores precisa manter o código escrito por outra pessoa (ou seu próprio código), é importante que o código seja sustentável (Martin, 2009, Baggen et al., 2012, Bloch, 2017).

- **Funcional** - Todo trecho código precisa ter um propósito, devendo cumprir algum requisito ou especificação do sistema. Eder et al. (2012) mostram que a grande maioria dos softwares possuem trechos de códigos desnecessários e a presença deles impacta nos custos de manutenção. Uma das maneiras de contornar esse problema é utiliando testes de cobertura, que podem identificar trechos de código que não são executados (Myers et al., 2004).

- **Eficiente** - A grande maioria das linguagens de programação permitem que uma mesma funcionalidade pode ser escrita de diversas maneiras. Um código mal escrito pode demorar mais para ser executado, afetando o desempenho da aplicação (Simunic et al., 2000). Alguns procedimentos de garantia da qualidade (seção 2.5) incluem testes para problemas de desempenho.

- **Escalável** - sem escalabilidade, a manutenção evolutiva, visando adição de novas funcionalidades, pode ser muito impactada, ocorrendo situações em que o software precisa ser completamente reprojetado e recodificado para quaisquer alterações significativas, impactando diretamente na complexidade e no tempo de codificação. A utilização correta de padrões de projeto pode contribuir muito para que esse tipo de situação não ocorra (Baxter et al., 2004).

- **Modular** - É comum que uma mesma funcionalidade componha diversos *softwares*, sendo muito difícil o reaproveitamento de código quando o mesmo é de baixa qualidade. Quando bem escrito, o mesmo pode ser aproveitado em diversas soluções. Alguns princípios da Engenharia de Software, como a alta coesão e o acoplamento fraco, providos pela utilização correta de padrões de projeto, facilitam a modularidade (Sullivan et al., 2001).

- **Legível** - Ler um trecho de código requer prática, mas, assim como a literatura convencional, o código mal escrito pode ser de difícil entendimento, colaborando para o aumento na complexidade da tarefa de codificar, bem como num maior risco da inserção de defeitos no código, motivados por uma falha na compreensão por parte dos programadores. Padrões e convenções de código são constantemente definidos com o intuito de melhorar a legibilidade do código sendo um dos mais conhecidos o *Clean Code*, definido por Martin (2009) e apresentado na seção 2.4.2.

De maneira geral, pode-se considerar esses atributos como de alto nível, sendo necessária a definição de maneiras para identificar situações em que trechos de código-fonte estão em não conformidade com esses atributos.

A seguir, são apresentadas duas abordagens que têm por objetivo prover subsídios para que seja possível adequar a escrita de código aos atributos de qualidade esperados.

2.4 Convenções de Código

Conforme demonstrado na seção 2.3, não existe uma norma específica para a qualidade de código. Nesse contexto, alguns grupos de pesquisadores e organizações têm promovido debates com intuito de proporem convenções para a escrita de código.

As convenções de código têm evoluído junto com as linguagens de programação. A maioria delas partem da premissa de que o código escrito respeitando acordos e seguindo padrões garante uma melhor legibilidade e entendibilidade resultando em uma menor deterioração durante o seu ciclo de vida (Allamanis et al., 2014).

É importante destacar que as convenções de código não tratam apenas de questões relacionadas ao estilo de programar, sendo também definidos padrões de arquitetura e *design* de código além de padrões de projeto e documentação, entre outros.

Algumas convenções são generalistas, outras são específicas para uma linguagem de programação (Java, C#, Python, etc) ou para um paradigma (estruturado, orientado a objeto, etc), sendo comum a utilização de diversas convenções simultaneamente para compor um conjunto de parâmetros de qualidade do projeto.

Diversas ferramentas disponíveis no mercado foram desenvolvidas com o intuito de facilitar o processo de análise de código podendo ser usadas para impor essas convenções em um projeto de desenvolvimento de software. Além das citadas *FindBugs*, *CheckStyle* e *Jtest*, outras ferramentas são capazes de realizar um processo de inspeção no código fonte tentando identificar trechos de código em desacordo com regras pré-definidas.

O termo *convenções de código* é abrangente, e inclui as melhores práticas sobre estilo, nomeação, sintaxe e comentários. Nem todas essas convenções são igualmente relevantes no que diz respeito à legibilidade, à compreensibilidade e à facilidade de manutenção do código (Smit et al., 2011).

2.4.1 SQALE

Desenvolvido pela *inspearit - France*, o *Software Quality Assessment based on Lifecycle Expectations*, (Avaliação da Qualidade de Software Baseada em Expectativas do Ciclo de Vida) , conhecido como SQALE, é definido pelos seus autores como um método para a avaliação da qualidade de um software (Letouzey, 2012a).

O uso do *SQALE* é recomendado pelos seus criadores para: 1) apoiar a avaliação do código-fonte de um software da forma mais objetiva, precisa, reprodutível e automatizada possível; 2) prover um método eficiente para gerenciar a dívida técnica (Letouzey, 2012b).

O método é baseado em quatro componentes principais, contendo definições de um Modelo de Qualidade, de um Modelo de Análise além de um conjunto de Índices e Indicadores, sendo brevemente descritos a seguir:

1. **Modelo de Qualidade *SQALE*** - é usado para formular e organizar os requisitos não funcionais relacionados à qualidade do código que são definidos na ISO/IEC-9126-1 (2003). Possuí três subdivisões hierárquicas, sendo o nível mais alto contendo as características definidas na norma, um segundo nível, intermediário, contendo sub-características e um terceiro nível, mais baixo, contendo os requisitos específicos da linguagem e do contexto do projeto;

2. **Modelo de Análise *SQALE*** - sendo dividido em dois grupos, um primeiro grupo contém as regras que são usadas para normalizar as medidas e os controles relacionados ao código e, um segundo grupo contendo as regras para agregar os valores normalizados. Os relatórios resultantes das ferramentas de análise de código-fonte calculam a não conformidade a partir do custo empregado para corrigi-las associado a um fator de não conformidade.
3. **Índices *SQALE*** - representam custos que podem ser calculados em unidades de trabalho, de tempo, ou monetária. O custo de correção relacionado a uma determinada não conformidade com uma característica pode ser estimado adicionando-se todos os custos de remediação vinculados aos requisitos da característica.
4. **Indicadores *SQALE*** - relacionados às características de qualidade, permitem uma representação altamente sintetizada da qualidade de um software em desenvolvimento. O método SQALE define três indicadores, e possibilita aos desenvolvedores definir indicadores de acordo com suas necessidades.

Um critério de análise pode ser estabelecido a partir de um conjunto de regras definidas pelo gerente de projetos. Cada regra é definida para satisfazer uma ou mais normas e/ou recomendações fornecidas a partir de estudos e/ou diversas convenções realizadas relacionadas com Qualidade de Código.

Um trecho de código em desacordo com qualquer uma dessas regras definidas gera uma violação (*issue*). Um mesmo trecho de código pode violar várias regras, introduzindo várias *issues* no projeto assim como uma regra pode ser violada em partes diferentes do código.

Embora toda regra exista para um propósito claro e alguma violação a essas regras deve ser resolvida o quanto antes, o impacto que a violação pode causar na projeto é diferente. Algumas violações podem impactar no processo de manutenção enquanto outras podem impactar diretamente na execução do programa. Essas últimas devem ser resolvidas o quanto antes, visto que o programa pode não funcionar corretamente enquanto o trecho de código não for corrigido.

Pensando nisso, os idealizadores do SQALE propuseram uma classificação das violações de acordo com o grau de impacto que uma não conformidade pode causar no projeto, sendo a recomendação para que sejam resolvidas as violações de grau mais severo imediatamente, enquanto que as de graus menos severos podem ser adiadas conforme necessidade.

Além da classificação, foi atribuído para cada uma das classes um fator de não remediação. Trata-se de um valor discreto que auxilia nos cálculos para

Tabela 2.3: Classificação das violações (*issues*) quanto à severidade. Adaptado de Letouzey (2012a)

Severidade	Descrição	Exemplo	Fator
BLOQUEANTE	Pode ou vai resultar em um defeito	Divisão por zero	5000
ALTO	Gera um alto impacto no custo de manutenção	*SQL Injection*	250
MÉDIO	Gera impacto no custo de manutenção	Lógica complexa	50
BAIXO	Gera um pequeno custo de manutenção	Linhas longas	15
INFORMAÇÃO	É apenas uma recomendação	Anotações/*TODO*	2

indicar ao desenvolvedor qual o custo em unidades de trabalho (UT) que a não resolução imediata dessa violação pode adicionar ao projeto.

É possível observar na Tabela 2.3 as cinco categorias de severidade de uma violação, sendo a primeira, a de maior severidade. Na última coluna da tabela estão dispostos os valores do fator de não conformidade, utilizados por diversas métricas de qualidade de código.

A ferramenta SonarQube™ e a ferramenta SQuORE são exemplos de ferramentas conhecidas que se baseiam no método *SQALE* para prover ao desenvolvedor um conjunto de informações sobre qual é a qualidade do código e quanto o seu projeto está deficitário.

As análises realizadas e métricas utilizadas pelas ferramentas são muito similares, como a duplicação de código, a Complexidade de MCCABE, contagem de linhas de código, entre outras (Hervé, 2011, SQuORE, 2016). A principal diferença é que a ferramenta SQuORE é uma ferramenta corporativa, paga e de código fechado, enquanto que a ferramenta SonarQube™ é gratuita e de código-aberto, sendo suportada pela comunidade.

Além do método *SQALE*, outra abordagem amplamente utilizada pela indústria com o objetivo de garantir melhorias na escrita de código é o *Clean Code*, definido a seguir.

2.4.2 Clean Code

Fowler et al. (1999), McConnell (2004), Martin (2009), Bloch (2017) têm focado seus esforços nos últimos anos no entendimento de como um código mal estruturado, fora dos padrões e com vícios do programador impacta diretamente no desenvolvimento do software. Esses estudos mostram o quão importante é seguir os padrões na hora do desenvolvimento e propõem alternativas para melhorar e aperfeiçoar o que esta sendo produzido.

Uma das alternativas mais utilizadas para auxiliar o programador a adquirir boas práticas de programação consiste num conjunto de sugestões denominado *Clean Code*. Trata-se de um estilo de desenvolvimento de software

com foco na leitura do código, possibilitando a produção de um código fácil de escrever, ler e manter.

Martin (2009) propõem um conjunto de sugestões ao programador sobre como agir e quais decisões pode tomar visando a escrever linhas de código de maneira modularizada e sistemática para, que ele ou outro programador possam interpretar e modificar o programa.

Essa ideia reforça uma definição de Engenharia de Software, "*a construção por muitas pessoas de um software de múltiplas versões*". Se um código, deve ser lido e compreendido por diversas pessoas, logo, é importante que a codificação seja simples possível.

Se aplicado corretamente, o *Clean Code* permite escrever códigos levando em consideração padrões e convenções de *código limpo* adotados pela indústria e pelas comunidades de desenvolvimento de software livre.

Nas disciplinas de programação é comum ensinar ao aluno que *"programar é dizer ao computador o que você quer que ele faça"*. Ná prática, a ideia por trás do *Clean Code* é que "*programar é dizer para outra pessoa o que se quer que o computador faça*" (Knuth, 1997). O foco está sempre na pessoa e não na máquina.

A adoção do *Clean Code* tem resultado em uma diminuição considerável nos custos de manutenção, o que conforme demonstrado em 2.6, é o maior custo de um projeto. Além disso, é possível estimar de uma maneira mais simples e precisa custos e tempo de desenvolvimento de novas funcionalidades, o que impacta positivamente na gestão de contratos e gestão de processos.

Com o objetivo de investigar os efeitos da utilização do *Clean Code* no entendimento do código, Koller (2016) realizaram um estudo com dois grupos de estudantes, atribuindo ao primeiro grupo a tarefa de corrigir problemas em um código legado "sujo", enquanto um segundo grupo teve a mesma tarefa, só que em um código refatorado de acordo com as práticas do *Clean Code*.

De acordo com os autores, tarefas como a alteração de funcionalidades são executadas em tempo inferior pelo grupo que pôde realizar a refatoração no código.

Definidos os atributos e métodos para atingir a qualidade interna de código, torna necessário a definição de processos, métodos e ferramentas. A seguir são descritas soluções para a construção de software com qualidade, focado na qualidade de código, objetivo deste estudo.

2.5 Garantia da Qualidade

A Garantia da Qualidade do Software, pode ser definida como "o conjunto de atividades que garantem que os processos estão estabelecidos e são continuamente melhorados com objetivo de desenvolver produtos que atendam às

especificações e que sejam adequados ao uso pretendido" (Lewis, 2004).

SQA não é um processo definido e também não é um arcabouço. Esse conjunto de atividades sistemáticas envolve o ciclo de vida do software, para que os padrões e procedimentos sejam seguidos de modo a fornecer evidências da construção correta do produto.

A Garantia da Qualidade também auxilia o desenvolvedor na identificação de problemas e possíveis correções, sejam elas corretivas, paliativas ou ainda evolutivas. A *SQA* pode estar presente no processo e no produto. No processo é comum que se realizem auditorias a partir da utilização de processos definidos, como as normas ISO/IEC-9126-1 (2003) e ISO/IEC-25010 (2011).

São inclusas atividades de auditorias nos próprios processos de avaliação, garantindo assim a efetividade da avaliação em identificar a conformidade do produto avaliado.

Sua definição formal está presente na NBR/ISO-12207 (2017), fazendo parte do conjunto de processos de apoio que acompanham todo o ciclo de vida do software, desde a sua definição até a descontinuação. Sua estrutura é apresentada na Figura 2.5.

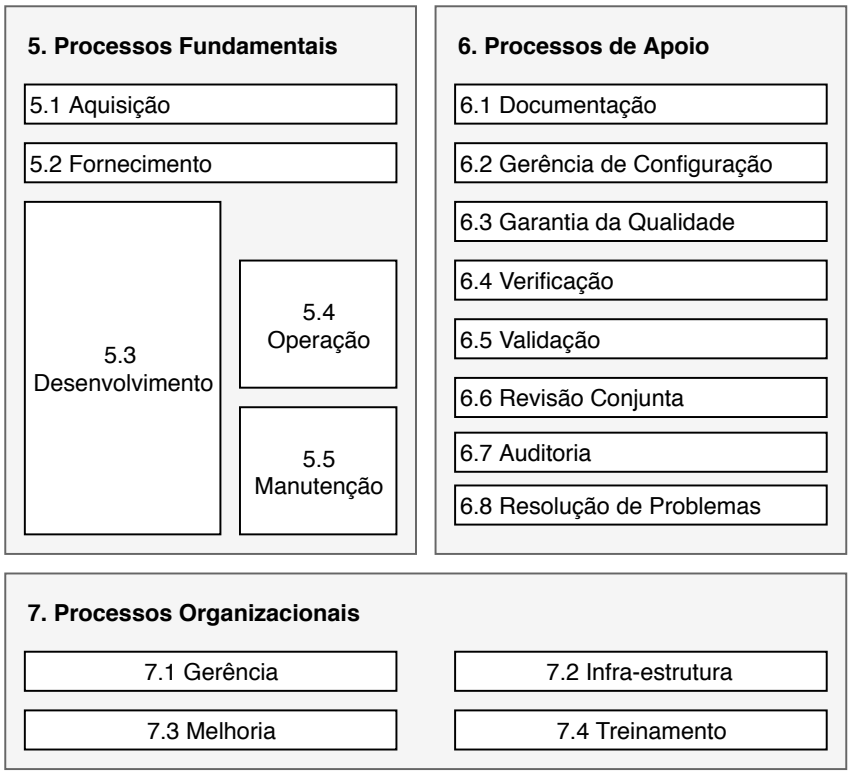


Figura 2.5: Estrutura da norma NBR/ISO-12207 (2017).

A norma também define outros processos de apoio, como a documentação, gerência de configuração, verificação, validação, revisão conjunta, auditoria e resolução de problemas. Estes processos, associados à garantia da qualidade apoiam os processos fundamentais, organizacionais e os próprios processos de apoio. Cada processo dispõe de parâmetros e diretrizes que definem padrões,

procedimentos, convenções e especificações.

Assim como no processo de software, as atividades relacionadas ao produto de software também estão em conformidade com as normas ISO/IEC-9126 e ISO/IEC-25010, apresentadas na Seção 2.3, nas quais são propostas atividades para a avaliação da qualidade interna, externa e de uso. Após a realização de cada atividade, é comum a realização de auditorias e análise dos resultados.

Dentre os processos de apoio, a Gerência de Configuração define as atividades de controle de mudanças, responsável por identificar, rastrear e controlar mudanças, definindo o mecanismo para o gerenciamento de diferentes versões destes produtos, controlando as mudanças impostas, e auditando e relatando mudanças realizadas.

O processo de Verificação define atividades, relacionadas a análise de um artefato estático para certificar requisitos funcionais e não-funcionais. Dentre as atividades, são inclusas a análise estática e as atividades de inspeção.

Na Validação, o objetivo é assegurar que o produto de software atende as necessidades e as expectativas para a qual o software foi projetado. Dentre as atividades, são inclusas a análise dinâmica e o teste de software – testes unitários e de integração.

Enquanto o processo de verificação apoia a avaliação da qualidade interna, a verificação apoia a avaliação da qualidade externa do produto, conceitos definidos na seção 2.3.2. A seguir são descritas as principais atividades propostas pela Engenharia de Software, relacionadas com este estudo.

2.5.1 Inspeção de Código

Fagan (1986) define a inspeção de software como "*um tipo específico de revisão que tem a função de detectar defeitos antes que a fase de teste seja iniciada, contribuindo para a melhoria da qualidade geral do software.*"

Profissionais e pesquisadores da área de Engenharia de Software estão em uma busca contínua por melhorias na qualidade do software. Parnas e Lawford (2003) afirmam que o processo de inspeção é a única maneira de maximizar a qualidade do software.

A definição formal de Inspeção de Código, presente na IEEE (2008) é: "*A inspeção, mais formal que a revisão técnica, tem o objetivo principal de identificação e remoção de defeitos. É obrigatória a geração de uma lista de defeitos com a classificação padronizada, requerendo-se a ação dos autores para a remoção desses defeitos. No Praxis padrão, são aplicadas aos artefatos de desenho, implementação e testes, focando a correção desses em relação aos respectivos padrões e especificações, enquanto as revisões técnicas têm maior enfoque na qualidade da documentação.*"

Na "inspeção de software" é aplicada uma abordagem sistemática com ob-

jetivo de analisar um programa minuciosamente. O resultado final de um processo de inspeção mostra ao desenvolvedor se um produto está ou não adequado (Pressman e Maxim, 2016).

Este processo pode ser aplicado a todos os artefatos estáticos do software, possuindo um processo de detecção de defeitos bem definido. É possível observar na Figura 2.6 a inspeção em diferentes artefatos.

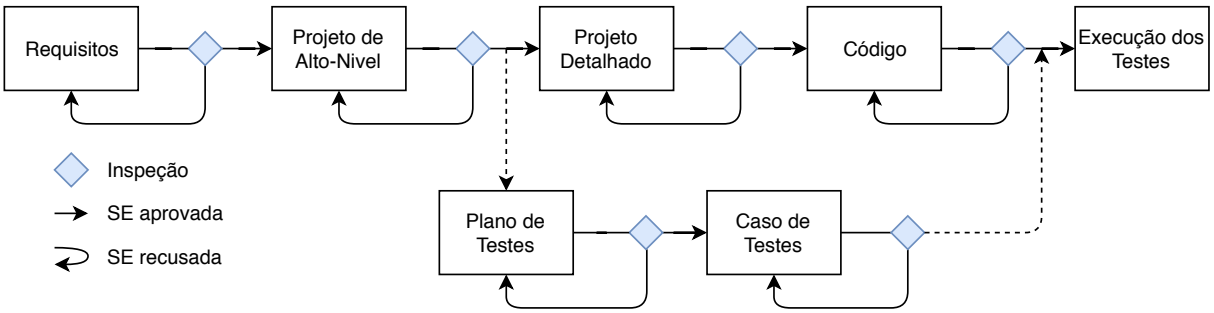


Figura 2.6: Inspeções de Software nos Diferentes Artefatos (Produção Própria)

Em se tratando da inspeção de código, foco deste estudo, Laitenberger e DeBaud (2000) mostram que é o tipo mais comum de inspeção, ocorrendo em um número bem maior de vezes, se comparada com as inspeções realizadas nos requisitos, no projeto e nos artefatos de teste, podendo ser observado na Figura 2.7.

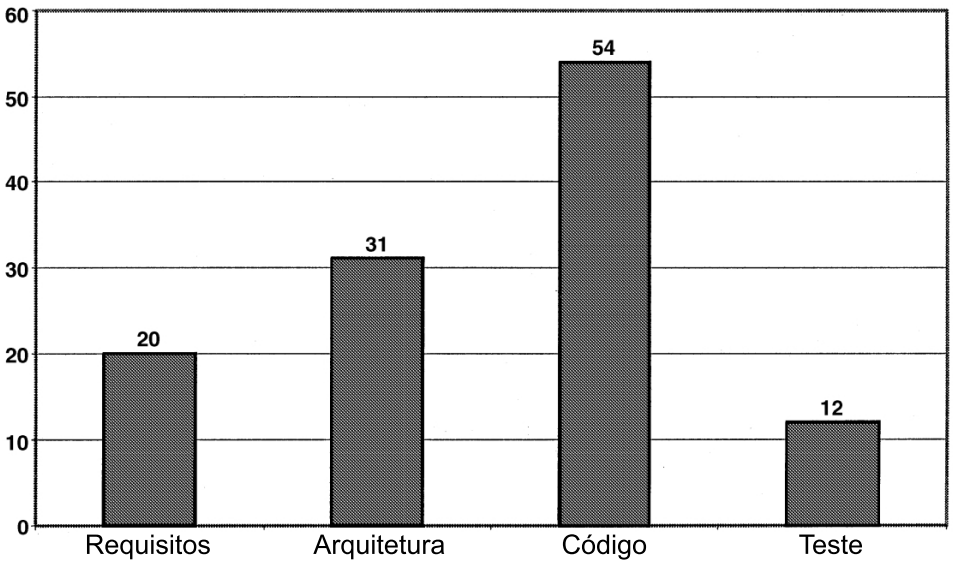


Figura 2.7: Exemplo de distribuição do Uso de Inspeção nos Diferentes Artefatos - Adaptado de: Laitenberger e DeBaud (2000)

A inspeção nos artefatos de código identifica defeitos no código-fonte, realizando uma análise estática do código com objetivo de tornar os programas menos complexos, pois os subprogramas são escritos em um estilo consistente e obedecem padrões estabelecidos, permitindo uma maior legibilidade e inteligibilidade de código.

Embora a inspeção ocorra no produto, benefícios da aplicação correta da inspeção de código podem refletir no processo de desenvolvimento de software. Um exemplo é a utilização de medidas e métricas de software, tornando o desenvolvimento mais transparente e possibilitando o aumento da confiabilidade das estimativas e do planejamento.

A produção de artefatos mais inteligíveis, trará benefícios para as fases seguintes do processo de desenvolvimento, principalmente na fase de manutenção.

Gilb et al. (1993) mostram que as inspeções aumentam a produtividade do projeto entre 30% a 50%. Ainda segundo o autor, o tempo de desenvolvimento é diminuído entre 10% a 30% do esforço total empregado no desenvolvimento.

Embora bem definido e apoiado, realizar uma inspeção de código pode demandar um certo esforço, convertido em aumento de custo na indústria. Essa é uma das razões que motivam empresas a utilizarem ferramentas de inspeção de código para automatizar a análise. As principais ferramentas são descritas a seguir.

2.5.2 Ferramentas de Inspeção de Código

A inspeção de código é realizada a partir da análise estática. É possível encontrar uma grande variedade de ferramentas que têm por objetivo a automatização do processo de inspeção de código-fonte. Dentre essas, destacam-se ferramentas mais enxutas como o FindBugs [1], CheckStyle [2] e Jtest [3] e mais robustas como o SonarQube™ [4] e a ferramenta SQuORE[5].

Muitas são as razões para empregar esse tipo de análise em um projeto, dentre elas destacam-se (Ackerman et al., 1989):

- Considerável facilidade em identificar trechos de código com defeito, com falhas de segurança e/ou com problemas de estilo;
- Ferramentas de análise estática costumam fornecer uma visão analítica mais objetiva do código, sendo mais fácil ao desenvolvedor identificar uma falta cometida por desatenção, por exemplo;
- Gerentes de projeto podem estudar os relatórios e identificar padrões de falhas técnicas e perfis técnicos de codificadores;
- Apoio aos processos de melhoria contínua.

Em geral, ferramentas de análise estática de código-fonte podem ser classificadas entre três categorias: os analisadores de código baseados em métricas

[1] Disponível em: http://findbugs.sourceforge.net/
[2] Disponível em: https://checkstyle.sourceforge.io/
[3] Disponível em: https://www.parasoft.com/products/jtest
[4] Disponível em: https://www.sonarqube.com/
[5] Disponível em: https://www.squoring.com

de qualidade (*Metric Tools*), os verificadores de estilo (*Style Checkers*) e os *Linters*.

2.5.2.1 *Metric Tools* - Analisadores baseados em métricas

Esse tipo de ferramenta de software (ou *plugin*) tem objetivo de medir aspectos de qualidade do código como:

- **Complexidade** - consideram o número de linhas de código, de métodos, de classes, de heranças e de arquivos. A análise de complexidade identifica se o sistema tem muitos desvios de fluxo (*if*, *switch*, *for* ou *while*), conhecidos com complexidade ciclomática. As medidas mais comuns são: LOC (*Lines of Code*), NOM (*Number of Methods*), NOA (*Number of Attributes*), DIT (*Depth of Inheritance Tree*), CC (*Cyclomatic Complexity*) e WMC (*Weight Methods per Class*).
- **Acoplamento** - são avaliadas as relações de dependência entre módulos ou componentes de software. Quanto mais independente um módulo/componente é de outro, mais fraco o seu acoplamento, sendo mais fácil o seu entendimento e manutenção. As medidas mais comuns são: RFC (*Response for a Class*) e CBO (*Coupling between objects*).
- **Coesão** - outro atributo importante de qualidade, a análise de coesão permite identificar se os membros de um módulo estão ligados entre si com um mesmo propósito. Trata-se da análise de responsabilidade. As medidas mais comuns são: LCOM (*Lack of Cohesion of Methods*) em várias versões, TCC (*Tight Class Cohesion*) e LCC (*Loose Class Cohesion*).

2.5.2.2 *Style Checkers* - Analisadores de estilo

Analisadores de estilo são verificadores de regras de estilo de programação. Essas ferramentas analisam se um dado código está em conformidade com as convenções de uma determinada linguagem. Por exemplo, para o Java, a verificação deve ser feita baseada no *Java Code Conventions*, tais como abertura de chaves, regras de declaração, javadoc e outros.

Dentre as ferramentas mais conhecidas nessa categoria está o *Checkstyle*. Nessa ferramenta são checadas a ordem de declaração nos métodos, padrão de nomenclatura, entre outras. É importante destacar que violações em regras de estilo normalmente não ocasionam falhas no software. Entretanto, podem ser importantes para evitar problemas quando o trecho de código precisa ser alterado nas futuras etapas de manutenção.

2.5.2.3 *Linters*

São verificadores de qualidade estrutural do código. Os *linters* ajudam a identificar e corrigir problemas sem ser necessária a execução da aplicação. As

ferramentas dessa categoria têm como função relatar os problemas encontrados baseado no nível de severidade e descrevendo do que se trata o problema.

As violações identificadas nos linters, geralmente são classificadas em 7 categorias: Arquitetura e Design, Comentários, Duplicação de código, Padrões de codificação, Testes (cobertura de código), Complexidade ciclomática e Bugs em potencial.

A maioria das IDE's modernas (Visual Studio Code[6], NetBeans[7], Eclipse[8], IntelliJ-IDEA[9], etc) possuem *linters* integrados, estes com objetivos mais específicos, como sinalizar erros de sintaxe, utilização de variáveis não declaradas, espaçamento e formatação além de identificar problemas no escopo e utilização de tecnologias (bibliotecas, métodos e API's) depreciadas.

2.5.3 Integração Contínua

No processo de desenvolvimento de software, a integração se faz necessária quando vários programadores trabalham em um mesmo projeto. O desenvolvimento de projetos complexos requer a utilização de várias tecnologias diferentes, como *frameworks* e bibliotecas que precisam interagir ao mesmo tempo.

Booch e Grady (1994) afirmam que o processo de integração contínua é "*um processo que regularmente integra os componentes de um sistema e os testa*". Segundo os autores, uma iteração deste processo libera versões executáveis do sistema, ou seja, a iteração representa um crescimento nas funcionalidades do sistema assim como correção de problemas detectados em iterações anteriores.

De maneira similar, Fowler e Foemmel (2006) definem a integração contínua como uma prática de desenvolvimento de software da qual membros de uma equipe integram seus trabalhos com certa frequência.

Cada desenvolvedor trabalha em uma cópia local de uma base de código compartilhada. Depois de implementar o código-fonte, o desenvolvedor integra as alterações na base do código compartilhado. Uma compilação automatizada (incluindo casos de teste) verifica cada integração para detectar erros de compilação, teste e integração o mais rápido possível.

Para tornar possível a análise seguida de um *feedback* imediato, a integração contínua depende de um servidor de integração contínua, responsável por identificar alterações no repositório, gatilho para um conjunto de ações que envolve a análise, teste, validação e retorno para o desenvolvedor.

A sequência comum de atividades esta presente na Figura 2.8 e é descrita

[6]Disponível em: https://code.visualstudio.com/docs/java/java-linting
[7]Disponível em: https://netbeans.apache.org/kb/docs/java/code-inspect.html
[8]Disponível em: https://marketplace.eclipse.org/category/free-tagging/linter
[9]Disponível em: https://www.jetbrains.com/help/idea/linters.html

a seguir:

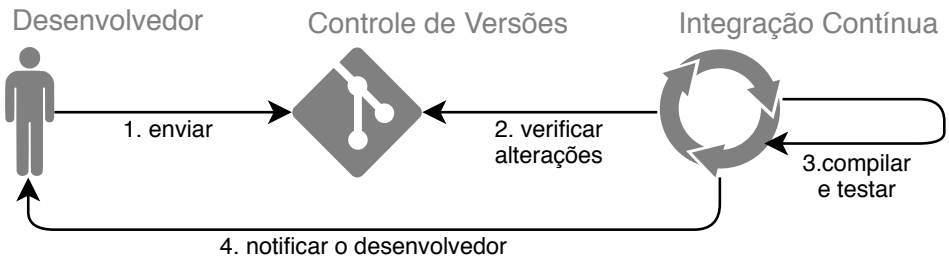


Figura 2.8: Fluxo típico da Integração Contínua (Produção Própria)

1. O processo inicia com um desenvolvedor submetendo as alterações de código em um sistema de controle de versão;
2. O servidor de integração contínua identifica alterações no repositório;
3. Gatilhos presentes no servidor de integração contínua disparam o processo de compilação, montagem e teste em cada submissão;
4. Ao finalizar os testes, relatórios de qualidade são gerados e ficam disponíveis para os participantes do projeto.

Os autores destacam diversas vantagens da utilização da integração contínua no ambiente profissional. Os benefícios impactam positivamente a qualidade do projeto, sendo identificados não só pelos codificadores, como também os demais participantes do projeto, incluindo o cliente e o gerente do projeto, que conseguem relatórios mais rápidos e precisos sobre a "saúde" do projeto. Muitas dessas vantagens podem ser aproveitadas para o ambiente acadêmico.

Bowyer e Hughes (2006) provam que a partir da utilização de conceitos de Engenharia de Software Experimental e dentro de um ambiente controlado é possível ensinar os conceitos de integração contínua aos alunos, mostrando na prática como funcionam os métodos ágeis.

Ferramentas como o SonarQube
textsuperscriptTM possibilitam a realização da inspeção de código-fonte no repositório de maneira contínua. Isso quer dizer que quando um trecho de código é adicionado ou modificado, a ferramenta realiza a análise na parte da alteração, sendo possível identificar se uma alteração corrigiu ou não um defeito e ainda se não adicionou novos defeitos no código.

A cada análise a ferramenta fornece um relatório detalhado das modificações, sendo possível então acompanhar a evolução da qualidade do projeto durante o desenvolvimento do mesmo, tanto na etapa de codificação como de manutenção, auxiliando então os programadores a desenvolver soluções mais elegantes, como indicado na garantia da qualidade.

2.6 A Qualidade de Código e a Manutenção

Entende-se por manutenibilidade a aptidão que um produto de software tem em receber manutenção. Essa capacidade de modificação de um software tem como principais parâmetros de referência a Analisabilidade que busca verificar o quão simples é de se diagnosticar problemas no código-fonte e a Modificabilidade, que busca verificar a facilidade com que um software pode ser modificado. Esse último está fortemente ligado a forte coesão e ao baixo acoplamento. Pode-se dizer que a Manutenibilidade é diretamente afetada por um código-fonte de baixa qualidade.

A manutenção é uma propriedade desejável do software, e uma variedade de métricas foi proposta para medi-lo, todas baseadas em diferentes noções de complexidade. Embora essas métricas sejam úteis, a complexidade é apenas um fator que influencia a sustentabilidade. A experiência prática em desenvolvimento de software leva a um conjunto de práticas recomendadas e convenções de codificação que fazem com que o código-fonte seja mais fácil de ler, entender e manter.

Smit et al. (2011) conduzem uma pesquisa com engenheiros de software, identificando a importância de 71 convenções de codificação para a manutenção. Os autores propõem uma métrica que oferece uma perspectiva diferente de manutenção, ou seja, uma métrica de "adesão à convenção" com base no número e na severidade das violações dessas convenções de codificação.

A maioria das métricas de manutenibilidade estão relacionadas com a complexidade. Embora valiosas, os autores sugerem que essas métricas seriam beneficiadas pela inclusão de fatores relacionados com melhores práticas de desenvolvimento de software e convenções de código que evoluíram ao longo do tempo.

A ideia de que a escrita de um código limpo facilita a manutenção se mostra válida quando consideramos diversos exemplos, como: strings codificadas e constantes numéricas tornam o código mais difícil de atualizar; comentários bem formatados ajudam os novos desenvolvedores a entender o código ou usar uma API; um estilo de nomenclatura bem definido que corresponda a bibliotecas existentes ajuda a associar a sintaxe ao significado semântico.

Os autores analisaram diversos repositórios de programas de código aberto afim de identificar a relação entre a adesão à convenções de código e a manutenibilidade. Foi descoberto que, quando conscientes das convenções, ou seja, quando o projeto tem uma política explícita de aderência à convenção de código imposta pelos *softwares* analisadores de convenções de código, os desenvolvedores estão dispostos a empregar certo esforço para adequar o código a um conjunto de regras previamente definidos; quando não, as violações a essas regras são predominantes.

Em um outro estudo, feito por de Andrade Gomes et al. (2017), os autores identificam que mesmo dentro de um ambiente acadêmico, os alunos são capazes de identificar questões problemáticas de qualidade em seus códigos ao utilizar ferramentas de análise de código, contribuindo para a melhoria da qualidade do que desenvolveram.

Nesse segundo estudo também são propostas métricas para avaliar a qualidade da manutenção. Os autores compararam o código presente no repositório local com o código presente no repositório do projeto afim de identificar trechos de código das quais são realizadas modificações para cumprir determinada tarefa de manutenção.

Identificados esses trechos, foi verificado se a modificação contribuiu para uma melhoria na qualidade ou não, levando em conta fatores como a quantidade de violações corrigidas/introduzidas, a severidade destas violações e qual o impacto em relação ao projeto levando em conta todas as violações presentes.

2.7 Considerações Finais

Profissionais da área de tecnologia da informação têm sido requisitados por empresas que não têm como atividade fim o desenvolvimento de soluções que envolvam a tecnologia da informação. Essas empresas buscam no profissional uma forma de consolidar o seu negócio, colocando o profissional da área diretamente em contato com as decisões estratégicas, táticas e operacionais da empresa (Melville et al., 2004, Luftman et al., 2017).

Essa mudança no cenário faz com que empresas demandem de profissionais com habilidades técnicas e interpessoais bem desenvolvidas. Nesse contexto, as universidades precisam avaliar se estão fornecendo para o mercado profissionais capazes de atender essas exigências.

Bailey e Stefaniak (2001), Bailey e Mitchell (2006), Matturro (2013), Sami et al. (2017) evidenciam o interesse das empresas por bons desenvolvedores, sendo o maior interesse, por bons programadores.

Dentre competências esperadas de um profissional, destacam-se as atividades de leitura, entendimento, modificação, codificação(escrita) e depuração de código-fonte, todas relacionadas ao código-fonte (Bailey e Mitchell, 2006).

A execução de todas essas atividades em conformidade com os requisitos definidos para as mesmas, sendo estes corretamente definidos, propicia um artefato de código com qualidade (Bourque et al., 2014) logo, fica evidente que para a indústria, o programador experiente deve ter domínio da escrita de código com qualidade, de maneira que contribua para o desenvolvimento do projeto.

O estudo da Qualidade de Software possibilita o entendimento de como se

obtém um produto de software com qualidade. A Qualidade de Software não está associada apenas ao resultado final, o produto de software, mas também aos processos, ferramentas e técnicas utilizados para garantir a qualidade (Bourque et al., 2014).

Normas como a ISO/IEC-9126-1 (2003) e posteriormente a ISO/IEC-25010 (2011), definem uma série de requisitos que o software deve possuir para ser considerado de qualidade. As normas dividem os atributos em três grupos, sendo eles a qualidade em uso, a qualidade externa e a qualidade interna, sendo a última, foco deste trabalho.

Na qualidade interna, são definidos atributos como a capacidade de se realizar manutenção, coesão, adequação ao estilo e padrões, entre outros. Essas características são evoluídas a partir de processos de melhoria contínua, definidos pela Garantia de Qualidade.

Um importante processo de melhoria contínua é o processo de Inspeção de Código, uma abordagem sistemática onde se analisa um programa de maneira minuciosa, com o objetivo de identificar defeitos no código-fonte que podem ocasionar falhas no software.

Para apoiar o processo de Inspeção, Letouzey (2012a) propõem o *SQALE*™. Definido pelos seus autores como um método focado na melhoria de código, o *SQALE* consiste em uma série de indicadores, classificações e métricas com o objetivo de permitir ao desenvolvedor avaliar as deficiências encontradas no projeto, fornecendo subsídios para a realização do processo de inspeção.

Além do *SQALE*, outra abordagem com objetivo de garantir melhorias na escrita de código é o *Clean Code*. Trata-se de um conjunto de recomendações que definem um estilo de desenvolvimento, focado na leitura de código.

As duas abordagens apresentadas, traduzem os conceitos definidos nas normas para algo mais tangível, facilitando a adoção e empregabilidade das mesmas, e por consequência a ampla utilização na industria, contribuindo com a melhoria contínua.

Sendo apresentado o panorama do mercado para o profissional de tecnologia da informação, com destaque as competências requeridas pelas empresas, a dificuldade por parte das empresas em encontrar profissionais capacitados e a dificuldade dos egressos em adentrar no ambiente profissional. Ao definir os principais conceitos de qualidade e as principais abordagens utilizadas pela indústria de software para garantir que a produção permaneça em conformidade com o estabelecido, para entender como os egressos são preparados para ocupar suas funções, assim como os principais desafios enfrentados nesse processo, torna-se necessário adentrar no contexto da educação. O escopo é restringido ao ingressante no curso curso de Bacharelado em Ciência da Computação, explorado a seguir.

CAPÍTULO

3

Ensino de Programação e da Qualidade de Código

3.1 Considerações Iniciais

O desenvolvimento de software é uma tarefa complexa e requer um grande conhecimento técnico, além de boa capacidade de abstração (Banker et al., 1998). Desenvolver as competências que tornam o ingressante capaz de entender e dominar a complexidade inerente ao processo de codificação demanda tempo do professor e do aluno e em muitos casos, pode não surtir os resultados esperados.

Na universidade, o processo de ensino-aprendizagem de programação normalmente está relacionado a resolução de problemas computacionais propostos e avaliação dos resultados obtidos durante a execução do programa (Dietz et al., 2018).

Essa avaliação das atividades realizadas baseada nos resultados alcançados durante a execução das soluções propostas é denominada de correção funcional e está associada a qualidade externa do artefato de código (Seção 2.3.2). O foco em correção funcional não abrange aspectos de qualidade interna presentes no código-fonte, como o desenvolvimento de código limpo e sustentável.

Como resultado, nota-se uma falta de estímulo para o desenvolvimento de códigos em conformidade com atributos de qualidade interna é fundamental para a indústria de software, cuja competência está muitas vezes relacionadas ao bom programador (Bailey e Mitchell, 2006).

A maneira como transcorre o processo de ensino-aprendizagem de programação dentro das universidades está em consonância com currículos de referência propostos tanto no Brasil (Rocha et al., 2005, Zorzo et al., 2017)

quanto no exterior (Association for Computing Machinery e Society, 2020).

A seguir são apresentados os currículos de referência propostos pela Sociedade Brasileira de Computação[1], a aplicação deles nas matrizes curriculares do curso de Bacharelado em Ciência da Computação de algumas das principais instituições de ensino do país, além de estudos que apresentam propostas de melhorias para o processo de ensino-aprendizagem de programação.

3.2 Diretrizes para o Ensino de Computação

No Brasil, a Sociedade Brasileira de Computação (SBC) tem papel fundamental no direcionamento do ensino de computação. Periodicamente são definidas comissões que conduzem grupos de discussões nos quais são definidas diretrizes para os cursos de graduação na área da Computação, como as Diretrizes Curriculares Nacionais (DCN)[2].

Além das diretrizes, são definidas comissões para a elaboração de currículos de referência, utilizados como base para a criação e avaliação em todo o país de cursos como Bacharelado em Ciência da Computação, Bacharelado em Sistemas de Informação, Bacharelado em Engenharia de Software, entre outros cursos na área.

Os currículos de referência são definidos com o intuito de sumarizar as diretrizes visando a facilitar a construção de cursos. A seguir são demonstrados aspectos relacionados ao escopo deste estudo de dois currículos de referência.

3.2.1 Currículo de Referência de 2005

Rocha et al. (2005) organizam o conteúdo considerado importante para a formação do profissional em matérias de computação. Essas matérias podem ser utilizadas pelos cursos em todo país como um referencial para a definição de como, quando e quais conteúdos serão abordados nas diferentes disciplinas oferecidas.

As matérias são organizadas em seis núcleos, sendo que cada matéria abrange um campo específico do conhecimento. As matérias da área da Computação estão organizadas em dois núcleos, *Fundamentos da Computação* e *Tecnologia da Computação*, definindo os dois núcleos e as matérias que os compõem.

Por se tratar da versão mais antiga do Referencial de Formação (embora ainda amplamente utilizada), para facilitar a leitura, a lista de matérias foi inclusa no Anexo A.

Dentro do núcleo de *Fundamentos da Computação*, a matéria de Algoritmos

[1] Portal da SBC: https://www.sbc.org.br/

[2] Disponível em: http://portal.mec.gov.br/index.php?option=com_docman&view=download&alias=52101-rces005-16-pdf&category_slug=novembro-2016-pdf&Itemid=30192

e Estrutura de Dados compõe as primeiras disciplinas oferecidas aos alunos recém-ingressos na universidade. Tendo seus principais tópicos sugeridos, os seguintes:

> "***Metodologia de Desenvolvimento de Algoritmos***. *Tipos de Dados Básicos e Estruturados. Comandos de uma Linguagem de Programação. Recursividade: (...). Modularidade e Abstração. Estratégias de Depuração. Cadeias e Processamento de Cadeias. Estruturas de Dados Lineares e suas Generalizações: (...). Árvores e suas Generalizações: (...). Tabelas Hash. Algoritmos para Pesquisa e Ordenação. Algoritmos para Garbage Collection. Técnicas de Projeto de Algoritmos: (...).*"

É possível observar que entre os tópicos propostos que definem a matéria não existe nenhum tópico com referência direta a qualidade. Embora os conceitos de Modularidade e de Abstração definidos na matéria estão relacionados com a qualidade de software, nesse contexto o foco do primeiro é a utilização de funções e rotinas, enquanto que o segundo está relacionado com a abstração de dados.

Os conceitos de qualidade são explorados separadamente na matéria de Engenharia de Software, dentro do núcleo de Tecnologia da Computação, sendo os tópicos propostos para essa matéria apresentados a seguir:

> "*Processo de Desenvolvimento de Software. Ciclo de Vida de Desenvolvimento de Software.* ***Qualidade de Software***. *Técnicas de Planejamento e Gerenciamento de Software. Gerenciamento de Configuração de Software. Engenharia de Requisitos. Métodos de Análise e de Projeto de Software. Garantia de Qualidade de Software. Verificação, Validação e Teste. Manutenção. Documentação. Padrões de Desenvolvimento. Reuso. Engenharia Reversa. Reengenharia. Ambientes de Desenvolvimento de Software.*"

Ao analisar os tópicos que compõem a matéria de Engenharia de Software, fica evidente uma abordagem mais conceitual, voltada à especificação, ao desenvolvimento, à manutenção e à criação de software. Na matéria são explorados conceitos como a Qualidade de Software que, conforme apresentado na seção 2.5, apoia os demais conceitos presentes.

A matéria de Engenharia de Software, como definida pelo Currículo de Referência da SBC não tem como objetivo o ensino da programação, mas o ensino da utilização e da aplicação correta dos processos, dos métodos e das ferramentas relacionadas com o processo de software.

O documento descreve o perfil do profissional com forte caracterização técnica. O foco principal está na definição de conteúdos (matérias) a serem ofe-

recidos pelos cursos. Por outro lado, visando a adequação às Diretrizes Curriculares Nacionais, o currículo de referência proposto em 2017 é definido a partir da noção de competências organizas por eixo de formação (Zorzo et al., 2017).

3.2.2 Currículo de Referência de 2017

O Currículo de Referência para Cursos de Graduação em Computação - 2017 (RF-CC-17), é definido a partir das Diretrizes Curriculares Nacionais (2016) do Currículo de Referência - 2005 e do *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Esse último proposto pela Association for Computing Machinery e Society (2013), sendo a principal referência para definição de projetos pedagógicos e matrizes curriculares em cursos de Ciência da Computação no Mundo todo (Zorzo et al., 2017).

Cada eixo de formação corresponde a uma macro competência que se relaciona com um grupo de competências derivadas, oriundas das diretrizes curriculares. Essas macro competências derivam em competências mais específicas que derivam em conteúdos (ou matérias no currículo proposto por Rocha et al. (2005)) subsídios necessários para o domínio do eixo de formação. Essa organização pode ser observada na Figura 3.1.

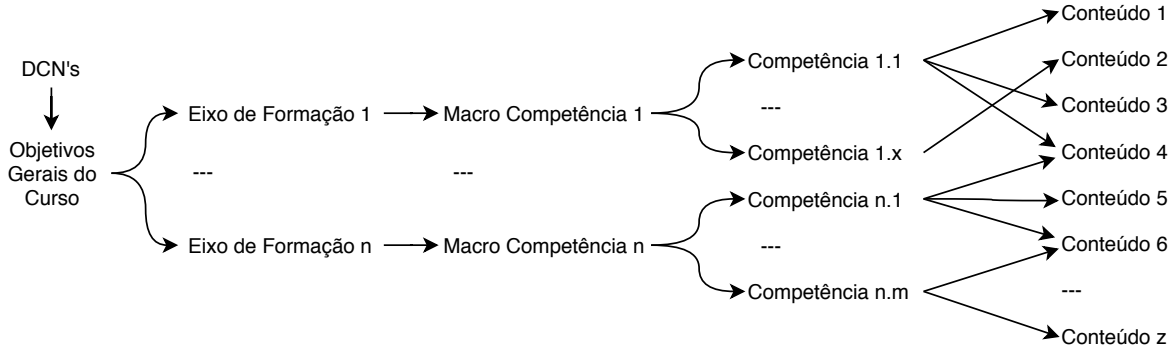


Figura 3.1: Estrutura do Currículo de Referência da SBC. Adaptado de: Zorzo et al. (2017).

De acordo com os autores, os eixos de formação são definidos da seguinte maneira:

> "*Produzem o entendimento de que tal formação deve levar em conta: a capacidade de atuar em todas as fases que envolvem a aplicação da ciência da computação em problemas diversos, desde a concepção de sistemas computacionais até a efetiva implementação de soluções adequadas; a capacidade de se reciclar e buscar novos conhecimentos; e a capacidade de seguir estudos avançados visando ao desenvolvimento da ciência e da tecnologia.*"

Em resumo, os eixos de formação são os seguintes:

1. Resolução de Problemas
2. **Desenvolvimento de Sistemas**
3. Desenvolvimento de Projetos
4. Implantação de Sistemas
5. Gestão de Infraestrutura
6. Aprendizado Contínuo e Autônomo
7. Ciência, Tecnologia e Inovação

Sacristán et al. (2016) e Scallon (2017) definem competência como a capacidade de um indivíduo em mobilizar recursos, como conhecimentos, habilidades, atitudes e valores para a sua atuação em situações reais complexas. Nesse sentido, as competências adquiridas por um Bacharel em Ciência da Computação para desenvolver sistemas computacionais, por exemplo, requer um conjunto de conteúdos necessários para que possa aplicar tais conhecimentos e habilidades, como a matéria de Algoritmos e Estrutura de Dados, definida em Rocha et al. (2005).

De acordo com Rocha et al. (2005), o *desenvolvimento de sistemas computacionais*, segundo eixo de entendimento da RF-CC-17, inclui a criação e a adaptação de sistemas existentes. Nesse processo, devem-se empregar teorias, métodos, técnicas e ferramentas para garantia e controle de qualidade do processo e do produto. As competências do profissional que domina esse eixo de formação inclui saber desenvolver sistemas que atendam a qualidade de processo e de produto, considerando boas práticas da engenharia de software e da engenharia de sistemas.

É importante destacar que os eixos de formação, conforme definidos pelo RF-CC-17, são comumente apresentados na sequência em que estão definidos. Por conseguinte, o aluno é primeiramente apresentado ao eixo de formação de *resolução de problemas*, no qual são apresentados os conceitos de Algoritmos e Estrutura de Dados.

É possível resolver diversos problemas computacionais com a execução de passos finitos e bem definidos. Nesse sentido, as Diretrizes Curriculares Nacionais determinam que os egressos devem ser "*capazes de criar soluções, individualmente ou em equipe, para problemas complexos caracterizados por relações entre domínios de conhecimento e de aplicação*" (DCN 2012)[3].

3.3 O Ensino de Programação

Zorzo et al. (2017), em conformidade com as diretrizes, propõem que a resolução de problemas, primeiro eixo de formação definido na RF-CC-17, deve

[3]Disponível em: http://portal.mec.gov.br/index.php?option=com_docman&view=download&alias=11205-pces136-11-pdf&category_slug=julho-2012-pdf&Itemid=30192

prover subsídios ao egresso para que seja capaz de resolver problemas que tenham solução algorítmica, considerando os limites da computação; o que inclui a identificação do problema, seleção ou criação do algoritmo apropriado e implementação da solução.

Nesse modelo, o aluno aprende a programar a partir da realização de tarefas de resolução de problemas baseadas em problemas computacionais conhecidos, sendo considerado bem sucedida a tarefa na qual o código produzido pelo aluno atinge os valores de saída esperados, desconsiderando como o aluno chegou ao resultado.

Para demonstrar a relação direta entre as diretrizes e as matrizes curriculares dos cursos de Ciência da Computação espalhados pelo país, este estudo apresenta uma relação das disciplinas que atendam aos dois primeiros eixos de formação definidos por Zorzo et al. (2017), ou seja, disciplinas oferecidas por cinco instituições de renome, que fornecem embasamento aos alunos para que os mesmos adquiram competências na resolução de problemas computacionais e no desenvolvimento de sistemas.

As instituições foram escolhidas levando em consideração a avaliação das mesmas nos principais indicadores de referência, para o curso de Ciência da Computação (Tabela 3.1). Todas possuem conceito máximo no Guia do Estudante[4], ocupando as primeiras posições no ranking do Exame Nacional de Desempenho de Estudantes (Enade)[5], no Ranking Universitário Folha (RUF)[6] e no *QS World University Rankings (QSU)*[7], índice internacional considerado um dos mais relevantes para a área.

Tabela 3.1: Avaliação das melhores instituições do país para o curso de Bacharel em Ciência da Computação (Produção Própria)

Instituição	**Avaliação**				
	Guia do Estudante	**Enade**	**RUF**	**QSU**	**Esfera**
USP-ICMC	5 de 5 estrelas	–	4°	1°	Pública
UFSCAR	5 de 5 estrelas	16°	10°	11°	Pública
PUC-RIO	5 de 5 estrelas	1°	11°	6°	Privada
UFMG	5 de 5 estrelas	2°	1°	3°	Pública
UFC	5 de 5 estrelas	3°	12°	–	Pública

Com base no exposto acima, na Tabela 3.2 é apresentada a organização

[4]Disponível em: https://guiadoestudante.abril.com.br/busca/?termo=computa%C3%A7%C3%A3o&filtro%5B%5D=graduacao&orderby=date&list-search-result=true

[5]A Universidade de São Paulo (USP) não participa do Enade, sendo selecionada levando em consideração os demais rankings e sua inegável excelência. Disponível em: https://cienciacomputacao.com.br/ranking-computacao/ranking-cc/

[6]Disponível em: https://ruf.folha.uol.com.br/2018/ranking-de-cursos/computacao/

[7]Considerando a posição das instituições nacionais. Disponível em: https://cacm.acm.org/careers/235063-qs-world-niversity-rankings-rate-top-computer-science-schools-for-2019/fulltext

das disciplinas para os cursos de Ciência da Computação da Universidade de São Paulo (ICMP-USP)[8], da Universidade Federal de São Carlos (UFSCAR)[9], da Pontífia Universidade Católica (PUC-RIO)[10], da Universidade Federal de Minas Gerais (UFMG)[11] e da Universidade Federal do Ceará (UFC)[12].

Tabela 3.2: Disciplinas relacionadas ao ensino de programação por termo (semestre) para os cursos de Bacharel em Ciência da Computação nas principais universidades do país (Produção Própria)

Termo	**Instituição**	**Disciplina(s)**
1°	ICMC-USP	SCC0221 - Introdução à Ciência de Computação I
		SCC0222 - Lab. de Introdução à Ciência de Computação
	UFSCAR	IPA - Introdução ao Pensamento Algorítmico
		CAP - Construção de Algoritmos e Programação
	PUC-RIO	INF1025 - Introdução à Programação
	UFMG	DCC003 - Algoritmos e Estruturas de Dados I
	UFC	CK0211 - Fundamentos de Programação
2°	ICMC-USP	SCC0201 - Introdução à Ciência de Computação II
		SCC0202 - Algoritmos e Estruturas de Dados I
	UFSCAR	AED1 - Algoritmos e Estruturas de Dados I
		POO - Programação Orientada a Objetos
	PUC-RIO	INF1007 - Programação II
	UFMG	DCC004 - Algoritmos e Estruturas de Dados II
	UFC	CK0226 - Programação
		CK0209 - Estruturas de Dados
3°	ICMC-USP	SSC0103 - Programação Orientada a Objetos
	UFSCAR	AED2 - Algoritmos e Estruturas de Dados 2
	PUC-RIO	INF1010 - Estruturas de Dados Avançadas
		INF1301 - Programação Modular
	UFMG	DCC005 - Algoritmos e Estruturas de Dados III
	UFC	CK0235 - Técnicas de Programação
4°	ICMC-USP	SSC0124 - Análise e Projeto Orientados a Objetos
	UFSCAR	ES1 - Engenharia de Software I
	PUC-RIO	INF1636 - Programação Orientada Objetos
		INF1629 - Princípios de Engenharia de Software
5°	ICMC-USP	SSC0130 - Engenharia de Software
	UFMG	DCC052 - Programação Modular
	UFC	CK0247 - Engenharia de Software
7°	UFMG	DCC603 - Engenharia de Software

As disciplinas são apresentadas de acordo com o semestre em que são ofe-

[8]Disponível em: https://uspdigital.usp.br/jupiterweb/listarGradeCurricular?codcg=55&codcur=55041&codhab=0&tipo=N

[9]Disponível em: http://bcc.dc.ufscar.br/index.php/grade-curricular/

[10]Disponível em: https://www.puc-rio.br/ensinopesq/ccg/ciencia_computacao.html#2018

[11]Disponível em: https://www.dcc.ufmg.br/dcc/sites/default/files/arquivos_diversos/EstruturaCurricularBCC2012.1.pdf

[12]Disponível em: https://cc.ufc.br/curso/matriz-curricular/

recidas, separadas conforme a instituição. As ementas de todas as disciplinas foram analisadas sendo desconsideras as disciplinas que não fazem parte do escopo deste estudo, restando as relacionadas com o ensino de programação e/ou com o ensino de qualidade.

Ao analisar os dados presentes na Tabela 3.2, fica evidente a similaridade da organização das disciplinas com os currículos de referência (Rocha et al., 2005, Zorzo et al., 2017). Os primeiros semestres dos cursos de todas as instituições selecionadas concentram disciplinas relacionadas com o eixo de formação de resolução de problemas computacionais, enquanto que as disciplinas focadas no desenvolvimento de sistemas se concentram nos termos posteriores.

Essa estrutura semelhante entre as matrizes curriculares de diversas instituições, indicam a tendência de todos os cursos de Ciência da Computação no país em acatar as recomendações do CR-05 e do RF-CC-17 para se adequarem as Diretrizes Curriculares Nacionais.

3.4 O Ensino de Qualidade de Código

A adoção dos currículos de referência dentro das instituições do país não é obrigatória, tampouco a definição de quais conceitos serão apresentados em quais disciplinas, ou ainda o período de oferta da disciplina.

A similaridade entre as matrizes curriculares permite traçar um fluxo de como o aluno adquire suas competências dentro de uma instituição de ensino. O ingressante aprende os conceitos básicos de programação a partir do estudo e reprodução de algoritmos para solucionar alguns problemas computacionais. O ensino de programação é muitas vezes relacionado apenas com a resolução desse conjunto de problemas, normalmente chamado de Lógica de Programação.

Nas primeiras disciplinas de programação, as competências de solução de problemas são desenvolvidas a partir da construção de programas que para uma determinada entrada, a solução desenvolvida seja capaz de fornecer uma saída esperada (resultado). O foco, nesse caso é a correção funcional, sendo considerados corretos todos os programas que fornecerem as saídas esperadas para os casos de teste.

Nesse tipo de análise, é comum que sejam negligenciadas questões como legibilidade do código, modularidade, complexidade e outras características, integram as competências que se espera do egresso, sendo essas capacidades técnicas importantíssimas para a indústria de software.

Nos semestres posteriores, o aluno é exposto aos conceitos de Estrutura de Dados e alguns paradigmas de programação, como a Orientação a Objetos. O aluno passa a ser desafiado a implementar soluções que resolvam problemas

da área da Ciência da Computação, como armazenamento e recuperação de dados, indexação e busca, etc.

Ao término das disciplinas, todo o material produzido pelos alunos não é mais utilizado, isso o motiva a resolver os desafios propostos da maneira conveniente e não da maneira correta. É comum ao aluno que analisa o código criado por ele próprio nos anos seguintes, não consiga justificar determinado trecho de código presente.

3.5 A Indústria no Contexto do Ensino

A seguir, são apresentados estudos que evidenciam a utilização de subsídios disponíveis na industria de software dentro do ambiente acadêmico, com o objetivo de apoiar o processo de ensino-aprendizagem.

3.5.1 *Clean Code* como ferramenta de Ensino

Dietz et al. (2018) apresentam um estudo sobre o ensino de qualidade de código para alunos de graduação. Em seu estudo, os autores propõem uma metodologia para o ensino da qualidade de código associada ao ensino de programação, baseada nos conceitos de *Clean Code*, apresentados em 2.4.2.

Os autores propõem uma abordagem, chamada de "ensino orientado ao *feedback*". Essa abordagem descreve como ensinar a programação de maneira eficaz aos alunos de graduação com ênfase na escrita e revisão de código.

O ensino de programação tradicionalmente segue um modelo prático-expositivo, no qual o professor diversos conceitos teóricos como algoritmos, estruturas de dados, paradigmas e técnicas de programação, seguidos de um conjunto de atividades práticas focadas na consolidação dos conceitos abordados.

Nesse modelo, o aluno é desafiado a provar que consegue aplicar os conceitos teóricos. No entanto, os artefatos produzidos pelos alunos para cumprir com os requisitos propostos nas atividades são geralmente descartados após a avaliação funcional por parte do professor. Como consequência, os alunos têm poucos incentivos para desenvolverem suas habilidades de escrita de código funcional seguindo padrões de qualidade.

Dietz et al. (2018) apontam que o ensino da escrita e revisão de código aliada ao ensino da programação de maneira eficaz pode ser desafiador ao professor. Os autores mencionam dificuldades em fornecer aos alunos subsídios necessários para alcançar alta qualidade de código, sendo necessária a realização de uma avaliação prévia do que a indústria dispõe e da viabilidade de aplicação da solução na academia.

Após avaliar conceitos de qualidade de código empregados na indústria com o objetivo de identificar quais poderiam ser aplicados na academia, optou-se pela aplicação do *Clean Code*, um conjunto de recomendações que se asse-

melha a um arcabouço, definido por Martin (2009) e focado na legibilidade do código-fonte. Essa estrutura contém diversas regras, definições e até mesmo sugestões, sendo definida pelo autor como um estilo de programação.

Considerando as dificuldades para ensinar qualidade de código na academia e após a avaliação do que a indústria utiliza, os autores se deparam com duas questões motivadoras relacionadas ao ensino da qualidade de código:

- Como ensinar *Clean Code* na universidade?
- Como ensinar *Clean Code* em larga escala?

Definindo-se qual subsídio seria fornecido aos alunos, tornou-se necessário encontrar uma forma de prover essas informações aos mesmos. Um pequeno projeto pode facilmente ultrapassar centenas de linhas de código, uma análise manual de todos os artefatos produzidos é bem custosa em relação ao tempo. Além disso, o aluno que ainda não possui o domínio sobre o assunto está propenso a avaliar incorretamente as não conformidades presentes em seu código.

Ao recorrer novamente a indústria, optou-se por fazer uso de ferramentas de inspeção contínua no código-fonte consolidadas no mercado. Empresas costumam utilizar ferramentas de análise estática dentro ciclo de desenvolvimento para assegurar que o código esteja de acordo com um conjunto de regras pré-definidos. Seu uso em um ambiente acadêmico pode permitir aos alunos identificar a partir dos relatórios fornecidos por essas ferramentas, quais práticas adotadas pelos mesmos estão em desacordo com as boas práticas de programação. A utilização combinada com processos de análise automatizada possibilita ao aluno ter um *feedback* da qualidade do seu código quase instantaneamente.

A automatização do processo de análise e a análise contínua e automatizada permite ao professor avaliar e fornecer ao aluno informações mais precisas sobre o que é necessário para que o aluno possa melhorar o estilo de codificação.

Os autores propõem uma divisão da disciplina em três principais atividades, Instruir, Atribuir e Avaliar. Durante o período de instrução, o professor define algum conceito de programação e propõe um ou mais exercícios relacionados ao tema abordado. Os alunos são desafiados a realizar os exercícios propostos e ao final da aula o professor demonstra um exemplo correto e funcional, para que os alunos possam se basear. submeter seu código para análise, recebendo um *feedback* de qualidade com as violações cometidas.

Na atividade de atribuição, o professor propõe um projeto maior, em grupo, que envolve diversos conceitos demonstrados, organizando a sala em grupos, permitindo assim que sejam utilizadas outras soluções da Engenharia de Software focadas em melhoria de código, como *Pair Programming* (Salleh et al.,

2011) e o *Peer Review* (Garousi, 2010). Os códigos devem ser submetidos em um determinado período de tempo utilizando um sistema de controle de versão. Os alunos recebem um relatório de qualidade do código enviado. Nessa etapa o professor atua como um facilitador, sugerindo aos alunos os impactos de suas decisões.

Por último, o professor propõe uma avaliação oral e individual aos alunos, questionando-os sobre as violações identificadas que foram mais comuns e que geraram mais dúvida nos alunos, assegurando assim que eles realmente entenderam os conceitos abordados.

Com o intuito de apoiar o professor, quando um aluno submete um trecho de código para análise, as violações cometidas ficam armazenadas em um repositório, podendo esses dados serem utilizados pelo professor, para que este possa direcionar o conteúdo da disciplina. Utilizando esses dados, os autores foram capazes de identificar quais as violações mais presentes nos códigos dos alunos, conforme consta na Tabela 3.3.

Tabela 3.3: Violações mais cometidas pelos alunos. (Adaptado de: Dietz et al. (2018))

Nome da Violação	**Frequência**
MagicNumberCheck	1382
AbbreviationAsWordInNameCheck	452
ParameterAssignmentCheck	150
PreserveStackTrace	144
UnnecessaryConstructor	143
UselessParentheses	124
VisibilityModifierCheck	120
ConfusingTernary	107
HideUtilityClassConstructorCheck	96
SingularField	93

Ao término da disciplina, os relatórios de qualidade fornecidos pela ferramenta de análise estática são novamente verificados afim de comparar a frequência com que as violações eram cometidas pelos alunos no começo e no fim do semestre, sendo possível então avaliar a efetividade da abordagem proposta pelo autor. Os resultados são demonstrados na Tabela 3.4.

A Tabela 3.4, encontra-se dividida de acordo com a frequência de violações. Na primeira parte são destacados casos em que as violações desaparecem completamente. Em seguida, são demonstrados os casos que ocorre uma diminuição significativa. Ao centro, a tabela lista as violações em que os números permaneceram estáveis, seguidos de crescentes e por último, novas violações. De um total de 107 regras, 18 não são violadas. O número de violações de três regras manteve estável e 44 apresentam diminuição, das quais 21 não são mais violadas.

Tabela 3.4: Frequência antes e depois do estudo conduzido pelos autores. (Adaptado de: Dietz et al. (2018))

Nome da Violação	Frequência	Variação (%)
CyclomaticComplexityCheck	43 -> 0	−100
UnnecessaryFinalModifier	37 -> 0	−100
ModifierOrderCheck	33 -> 0	−100
EqualsAvoidNullCheck	23 -> 0	−100
CollapsibleIfStatements	13 -> 0	−100
AvoidFieldNameMatchingMethodName	128 -> 3	−98
NeedBracesCheck	67 -> 2	−97
UselessParentheses	226 -> 9	−96
AvoidInstantiatingObjectsInLoops	37 -> 2	−95
PrematureDeclaration	16 -> 1	−94
UnnecessaryConstructor	67 -> 63	−6
LogicInversion	3 -> 3	0
MultipleVariableDeclarationsCheck	6 -> 6	0
AvoidCatchingNPE	10 -> 10	0
MagicNumberCheck	580 -> 633	+9
SingularField	27 -> 82	+203
HiddenFieldCheck	13 -> 50	+284
AbbreviationAsWordInNameCheck	43 -> 200	+365
VariableDeclarationUsageDistanceCheck	13 -> 72	+453
VisibilityModifierCheck	6 -> 54	+800
InnerAssignmentCheck	0 ->8	+∞
ClassFanOutComplexityCheck	0 ->10	+∞
SignatureDeclareThrowsException	0 ->17	+∞
UncommentedEmptyMethodBody	0 ->19	+∞
CompareObjectsWithEquals	0 ->19	+∞

Ficam evidentes os resultados positivos para diversas violações, sendo que em muitos casos, foi possível observar uma diminuição de 100%, mostrando que o objeto de estudo foi eficaz em diversas ocasiões. Porém, Dietz et al. (2018) também destacam que não há diferença significativa em alguns casos, em outros há um aumento considerável. Ainda são identificados um conjunto de violações ausentes no começo do semestre, sendo cometidas depois do inicio do estudo.

O aumento é justificado como parte do processo de especialização, por parte dos alunos. Dos 42 aumentos registrados, 32 não ocorreram na primeira tarefa, indicando que sua incidência pode estar relacionada com a utilização de novos conceitos e tecnologias, como a própria utilização de interfaces gráficas de usuário (*GUI*).

Em resumo, ocorre uma melhoria na estrutura do código. Os problemas com relação à complexidade ciclomática desapareceram e apenas poucas va-

riáveis são prematuramente declaradas. As chaves quase sempre são colocadas e a maioria dos parênteses desnecessários foi removida. Além disso, maus hábitos, como verificar referências com o comando *.equals()* e instanciar objetos em *loops*, despencaram. No lado crescente, foi possível notar que ainda há problemas com nomeação, declaração, posição e utilização de variáveis.

Dietz et al. (2018) propõem duas abordagens para o ensino de qualidade de código em larga escala. A primeira proposta é a criação de uma base de conhecimento de violações mais cometidas pelos alunos. A utilização pode ser justificada visto que existe um conjunto de falhas cometidas pelos alunos que foram comuns a um grande grupo de alunos. A proposta é que a estrutura da base possa comportar informações como textos explicativos, exemplos de códigos (com problema e corrigido), além dos nomes das violações. Provendo ao aluno um *feedback* mais rápido.

A segunda proposta está relacionada com a automatização do processo de revisão de código. Os autores recomendam o uso de um grande conjunto de testes de integração. Os mesmos desenvolveram uma ferramenta que consegue automatizar a análise utilizando recursos providos pelas ferramentas *PMD* e *Checkstyle* e geram uma saída (*.csv*) contendo a regra violada, o autor e a localização da violação (arquivo, linha, coluna).

A proposta de Dietz et al. (2018) se mostra eficaz, sugerindo melhorias no processo de aprendizagem de qualidade de código. A escolha dos autores de utilizar o *Clean Code* como um modelo de referência para avaliar a qualidade de código associada a utilização de ferramentas de análise estática, comum na indústria, também se mostra eficiente em um ambiente acadêmico de ensino de qualidade de código.

O interesse dos autores é voltado para o ensino da qualidade de código em paralelo com o ensino da programação porém, a proposta dos autores não prevê a utilização da mesma para o ensino e sim associada ao mesmo, sendo considerado um dos principais pontos fracos da proposta.

3.5.2 Integração Contínua em Ambiente Acadêmico

Krusche e Seitz (2018) apresentam estudo que mostra a efetividade da utilização da integração contínua associada à análise de código em ambiente acadêmico. É proposto um ambiente online e interativo focado no ensino de programação. Nesse ambiente, um professor pode disponibilizar exercícios a serem resolvidos pelos alunos de forma interativa.

Uma das principais justificativas para realização do trabalho, segundo os autores, é a demanda significativa que a avaliação manual exige de um instrutor, principalmente com relação ao tempo. Para resolver essa questão, é proposta a automatização deste processo. Os alunos que usam ferramentas de avaliação automática atingem metas importantes de aprendizado: desen-

volvem um estilo de código limpo e reutilizável, refletem criticamente sobre erros e estabelecem uma cultura de teste.

Ao considerar finalizada a atividade, o aluno pode enviar o código para análise, sendo fornecido um *feedback* instantâneo aos alunos contendo informações referentes aos aspectos funcionais do código. Além da Integração Contínua e do próprio ensino da programação, os autores também indicaram que os alunos adquiram experiência na utilização de gerenciadores de dependência e sistemas de controle de versão.

A Integração Contínua foi descrita por Booch e Grady (1994) como conceito para evitar integrações tardias e arriscadas. Fowler e Foemmel (2006) fornecem mais detalhes em sua definição, destacando que se trata de uma prática de desenvolvimento de software da qual os membros participantes integram seu trabalho com frequência considerável, que em muitos casos chega a ser diariamente. Os detalhes de sua utilização e aplicação são descritos na Sessão 2.5.3.

Embora a prática da integração contínua e a análise de código sejam estabelecidas na indústria, sua aplicação dentro da academia não é comum. Existem diversas ferramentas para avaliação de código e classificação, mas a maioria delas são soluções focadas em uma linguagem de programação específica, sendo limitada sua utilização apenas a determinados contextos.

A utilização da integração contínua também não é tarefa fácil, sua implantação em um ambiente educacional depende de um conjunto de ferramentas e processos, exigindo do aluno um rápido entendimento de diversos conceitos dos quais ele não possui domínio.

Os códigos são submetidos para uma ferramenta capaz de realizar testes dinâmicos. Seus principais aspectos são avaliar a funcionalidade, a eficiência e as habilidades de teste, executando um programa com dados de entrada de teste e verificando a saída para correção. Com a utilização de testes, é possível verificar se a solução desenvolvida pelo aluno provê as funcionalidades exigidas pelo professor, sendo possível a avaliação e até classificação do código, ao comparar quantas vezes o código atinge os resultados esperados em relação ao conjunto de testes definido.

Em um modelo denominado de aprendizagem interativa, os autores dividem as aulas em períodos de aula teórica, momento em que o professor apresenta e define um ou mais conceitos de programação e aula interativa, na qual alunos utilizam um ambiente virtual contendo um pouco de teoria, exemplos ministrados, exercícios e suas soluções e pontos de discussão.

São apresentados os conceitos de forma gradativa, seguidos de exemplos, atividades e discussões, focando na ideia de uma evolução incremental do conhecimento, tentando garantir assim uma maior absorção por parte do aluno.

Atividades práticas em sala de aula aumentam a motivação e engajamento do aluno e essa abordagem permite a participação mais ativa dos alunos, e a avaliação continua do aprendizado.

Os autores descrevem quatro tarefas típicas da indústria, normalmente presente nas atribuições de engenheiros de software que estão associadas ao processo de integração contínua.

T1 - **Programação**: escrita de código-fonte para solucionar problemas definidos pelo instrutor ou implementar modelos UML fornecidos pelo mesmo.

T2 - **Teste**: escrita de casos de teste. O instrutor cria casos de teste que avaliam os casos de teste dos alunos.

T3 - **Conflitos de Mesclagem**: adequação de trechos de códigos conflitantes durante o processo de mesclagem.

T4 - **Controle de Liberação**: entender como os gatilhos interagem com o servidor de integração contínua no momento da submissão.

Tabela 3.5: Número de estudantes participantes e de atividades realizadas em cada tarefa (* em média) (Adaptado de: Krusche e Seitz (2018))

	T1	**T2**	**T3**	**T4**
Estudantes Participantes	317	167	224	248
Estudantes que Submeteram	209(66%)	109(65%)	211(94%)	149(60%)
Estudantes com Êxito	200(96%)	108(99%)	183(87%)	135(91%)
Total de Submissões	340	340	904	285
Submissões Corretas	236	236	291	198
Casos de Teste	12	12	2	0
Tempo de Avaliação*	$10.3s$	$8.3s$	$5.1s$	$9.6s$
Submissões por Estudante*	1.6	3.1	4.3	1.9

Na Tabela 3.5 são apresentados os números de alunos e submissões para cada exercício, juntamente com o tempo médio de avaliação.

Os autores demonstram interesse em aprimorar o processo de ensino e aprendizagem da programação com destaque para a qualidade. A proposta de utilização de um ambiente virtual de ensino, *on-line* e interativo, se mostra relevante para o professor e para os alunos.

O foco apenas na análise dinâmica dificulta a avaliação dos atributos internos de qualidade do código, sendo este o principal ponto fraco do trabalho.

3.5.3 Inspeção Contínua no Ambiente Acadêmico

Conforme demonstrado na Seção 2.5.1, a inspeção é um processo aplicável a todos os artefatos estáticos do software, sendo considerado fundamental para a identificação de defeitos de maneira precoce, para que o mesmo não impacte em fases posteriores do ciclo de vida do software (Fagan, 1986).

Além da detecção de defeitos, normalmente ocasionadas por alguma falta cometida pelo programador, a inspeção de código é realizada com objetivo de tornar os trechos de código menos complexos e mais consistentes com padrões estabelecidos de legibilidade de código (Parnas e Lawford, 2003).

Um processo de inspeção de código manual demanda um grande esforço dos desenvolvedores, sendo comum a utilização de ferramentas que permitem a análise e avaliação do código de maneira automatizada. É comum que nessa análise automatizada um mesmo trecho de código seja exaustivamente inspecionado, com o intuito de identificar se ele está em desacordo com algum atributo de qualidade interna definido.

Essa prática faz com que em um grande projeto de desenvolvimento de software, contendo centenas de milhares e até milhões de linhas de código, analisar todo o código-fonte acaba requerendo um alto uso de recursos computacionais, podendo inviabilizar até mesmo a análise automatizada.

Na tentativa de contornar essa dificuldade, algumas ferramentas implementam a inspeção baseada em amostragem, enquanto que outras soluções realizam a inspeção em blocos especificos de código-fonte.

Para garantir que a inspeção ocorra em todo o código e com o mínimo de esforço, ferramentas como o SonarQube™ identificam alterações no código-fonte a cada submissão ao repositório e analisam apenas os trechos de código que sofreram alterações. Esse tipo de inspeção é conhecida como "inspeção contínua" do código-fonte e possibilita uma análise mais rápida e precisa, se comparada a análise em todo o código.

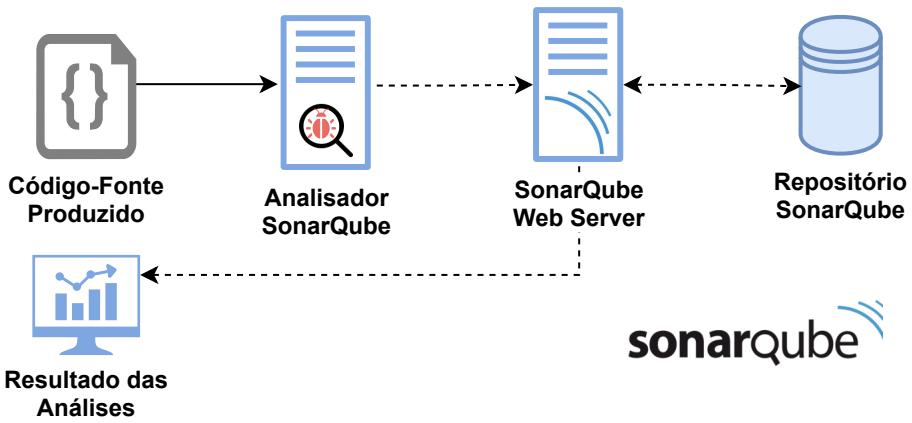


Figura 3.2: Arquitetura da plataforma SonarQube™. Adaptado de: SonarSourCe S.A. (2019)

A plataforma SonarQube™ é composta por três principais elementos distintos: (1) Servidor SonarQube – um servidor WEB, composto por um *Search Server* e uma *Compute Engine*, responsáveis por permitir que os usuários definam perfis de qualidade, naveguem e avaliem a qualidade do projeto; (2) Repositório do SonarQube – para armazenar estatísticas fornecidas pela *Com-*

pute Engine; (3) Analisador SonarQube – para analisar o código-fonte e enviar os resultados para o servidor. A interação entre os principais componentes é mostrada na Figura 3.2.

A ferramenta realiza a inspeção contínua a partir de gatilhos definidos no servidor de integração contínua, descrito na Seção 2.5.3. Diversas convenções de código foram utilizadas na definição das regras presentes na ferramenta. Conforme apresentado na Seção 2.4, o SonarQube™ se baseia no SQALE para definir métricas e classificações relacionadas com a qualidade de código.

Um trecho de código em desacordo com pelo menos uma regra é considerado uma violação, podendo comprometer a qualidade do produto de software em desenvolvimento. O mesmo trecho de código pode violar várias regras. Nesse caso, para cada violação, ele produz uma questão a ser resolvida. É importante observar que essa questão a ser resolvida é uma violação a uma regra e é possível escolher quais regras devem (ou não) ser levadas em consideração pelo analizador durante o processo de análise. O conjunto de regras escolhido para analisar um projeto de software é denominado *Perfil de Qualidade* e cada projeto tem apenas um *Perfil de Qualidade*.

Huo et al. (2004), Duvall (2007), Miller (2008), Prause e Augustin (2008), Lewis et al. (2009) e outros autores mostram que é possível realizar uma Inspeção Contínua no código fonte, permitindo a análise ininterrupta do que é produzido, a fim de aumentar a possibilidade de detectar possíveis problemas durante o desenvolvimento.

A Inspeção Contínua visa a prevenção da degradação do código fonte após sucessivas alterações (Manutenção Adaptativa, Perfectiva, Corretiva e Preventiva) evitando a introdução de novos defeitos e permitindo o desenvolvimento de software com qualidade.

3.5.3.1 Inspeção e o Ensino de Qualidade de Software

A ferramenta proposta por de Andrade Gomes et al. (2017) faz uso de relatórios de qualidade, gerados por ferramentas como o SonarQube™ para fornecer um *feedback* ao aluno sobre como está a qualidade do código produzido pelo mesmo.

Os autores propuseram um *plug-in* capaz de integrar e aproveitar os recursos fornecidos pela plataforma SonarQube™ e pela *Eclipse IDE*, para comparar a qualidade do código presente no repositório local com o repositório do projeto para permitir que o aluno avalie dentro do ambiente de desenvolvimento se determinada mudança contribui para a melhoria ou degradação da qualidade do projeto, antes submeter ao repositório do projeto.

Para avaliar a qualidade da manutenção realizada pelo aluno, a ferramenta compara o código modificado presente no repositório do aluno com as informações de qualidade do repositório do projeto. É necessário realizar a análise

do código no repositório local a partir da utilização de um segundo analisador, também integrado ao servidor, afim de poupar recursos computacionais do servidor de integração.

Após a análise local, o relatório de qualidade do repositório local é comparado com o relatório do projeto, sendo comparadas as violações cometidas e corrigidas durante a realização de determinada tarefa de manutenção. Os autores destacam que, devido às limitações das ferramentas, a comparação ocorre entre os relatórios de qualidade, artefato gerado após um processo de inspeção, e não entre os artefatos de código.

Um novo relatório é gerado e fornecido ao aluno dentro do ambiente de desenvolvimento. Este novo relatório contém métricas definidas pelos autores e apoiam o aluno no entendimento de como uma determinada ação pode impactar a qualidade geral do projeto (Figura 3.3). Além disso, a utilização de visualizações permitem explorar todos os trechos de código em desacordo com o perfil de qualidade definido pelo professor.

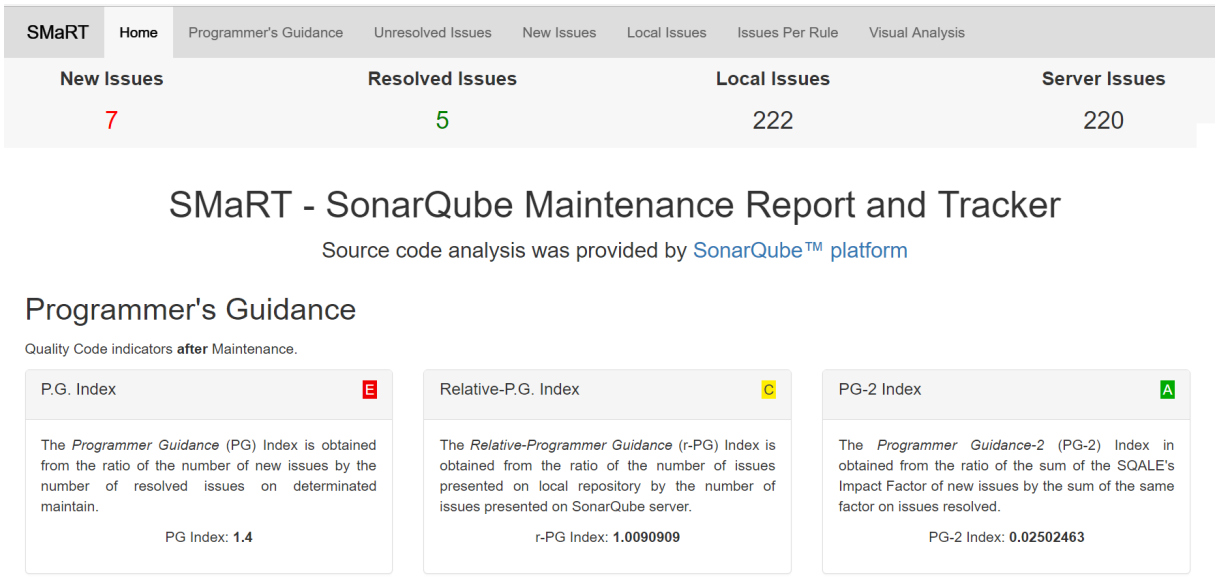


Figura 3.3: Exemplo de relatório fornecido pela ferramenta.
Adaptado de: de Andrade Gomes et al. (2017)

Os autores conduziram um estudo piloto de um experimento controlado, no qual um grupo de estudantes foi aleatoriamente dividido em dois grupos. Dois projetos foram preparados com a introdução de trechos de código que violavam alguns conceitos de qualidade.

Os alunos foram instruídos a resolver essas violações, sendo que na primeira parte do experimento, um grupo teve acesso a ferramenta proposta para corrigir as violações presentes em um dos projetos. Na segunda parte do experimento, os grupos tiveram que corrigir as violações do outro projeto. Nesta etapa, os alunos que haviam utilizado a ferramenta proposta não puderam fazer uso da mesma, enquanto que o outro grupo, pode utilizar a ferramenta.

Foram aplicados questionários antes e depois de cada atividade e, a partir da análise qualitativa dos dados fornecidos pelos participantes, os autores evidenciam que a ferramenta foi capaz de cumprir com a finalidade proposta, fornecendo aos alunos subsídios para que os mesmos pudessem avaliar a qualidade da sua manutenção antes de submeter o código ao repositório do projeto.

Os autores mostram que é possível a utilização de ferramentas de inspeção contínua, comum na indústria, dentro de um ambiente acadêmico. A partir dos questionários aplicados, são fornecidas evidências de que a ferramenta proposta ajuda os alunos a entenderem o que é um código-fonte com qualidade.

Com o foco no processo de manutenção, os autores restringem a aplicação da proposta ao contexto do ensino de qualidade, não sendo objeto do estudo o ensino da programação.

3.5.3.2 Inspeção e o Ensino de Programação

Desenvolvida e mantida pelo grupo de pesquisa vinculado ao Laboratório de Pesquisa em Engenharia de Software Aplicada (LaPESA), a ferramenta Teacher Mate é uma aplicação WEB focada em apoiar o professor na identificação das dificuldades por parte dos alunos em adquirir boas práticas de programação através da identificação de trechos de código que violam convenções e padrões de programação.

Essa ferramenta implementa uma abordagem que permite ao professor a criação de turmas e agrupamento dos relatórios de qualidade gerados pela ferramenta SonarQube™. Esse agrupamento resolve uma das dificuldades apontadas na utilização do (SQ) para fins de educação, permitindo a comparação de diversos trechos de código produzidos por diferentes alunos.

De acordo com os autores, a ferramenta Teacher Mate foi desenvolvida para atender dois desafios: 1) explicitar as dificuldades dos alunos em seguir boas práticas de programação; 2) possibilitar ao professor o acompanhamento do progresso da turma em função do tempo, através da possibilidade de agrupamento e comparação (de Araújo et al., 2020).

Os autores realizaram um estudo de caso com foco na identificação das dificuldades entre seis turmas da disciplina de Programação Orientada a Objetos, oferecidas pelo Departamento de Matemática e Computação (DMC) da Faculdade de Ciências e Tecnologia – FCT/UNESP – Campus de Presidente Prudente.

Na Figura 3.4, são apresentadas as principais violações identificadas para os trabalhos desenvolvidos pela turma do ano de 2019. É possível notar que as regras violadas não estão ordenadas pelo total de ocorrências (valor absoluto) e sim pela porcentagem em relação a Classe da violação (Defeito, Vulnerabili-

dade ou Código-Sujo).

Most Common Issues:

#	Name	Average	Amount	Class	Severity
1	Resources should be closed	64,44%	87	BUG	BLOCKER
2	Class variable fields should not have public accessibility	50,00%	25	VULNERABILITY	MINOR
3	"public static" fields should be constant	22,00%	11	VULNERABILITY	MINOR
4	Private fields only used as local variables in methods should become local variables	16,11%	836	CODE_SMELL	MINOR
5	Throwable.printStackTrace(...) should not be called	16,00%	8	VULNERABILITY	MINOR

Figura 3.4: Exemplo de Violações identificadas para os trabalhos produzidos pela turma de 2019 (Produção Própria)

A ferramenta também permite uma avaliação dos montantes de violações cometidas agrupados por Classes de violação, conforme pode ser observado na Figura 3.5.

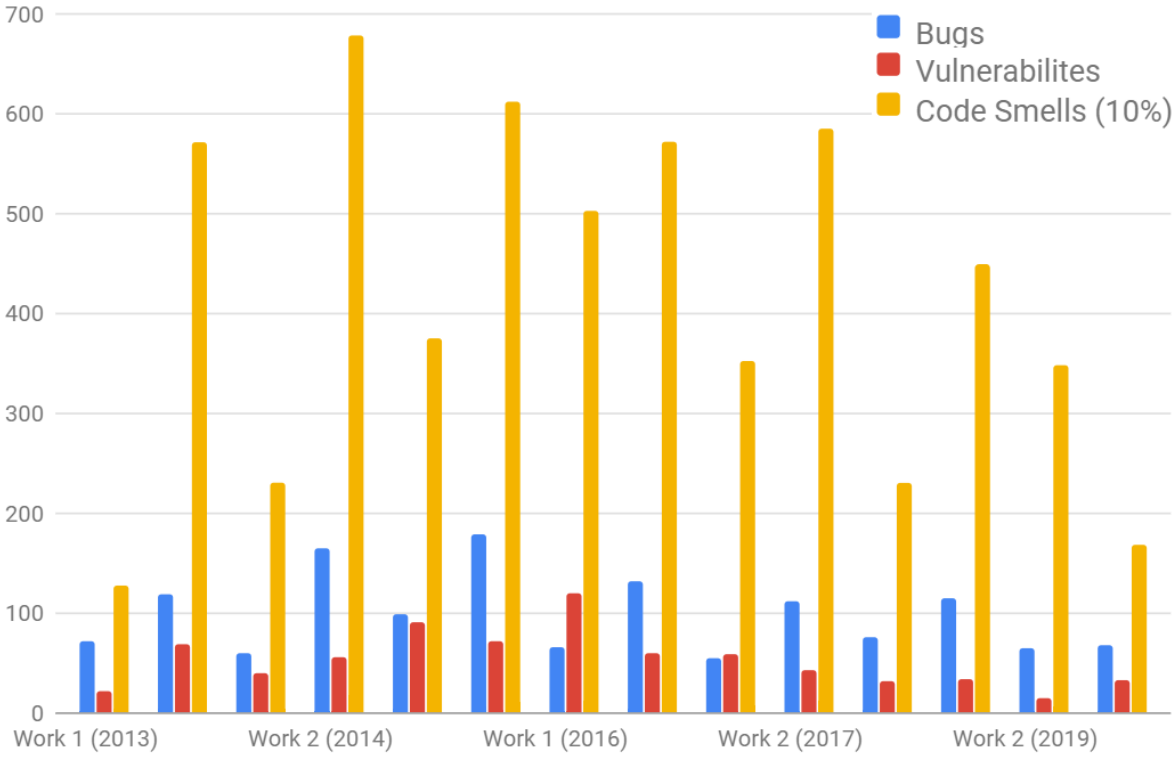
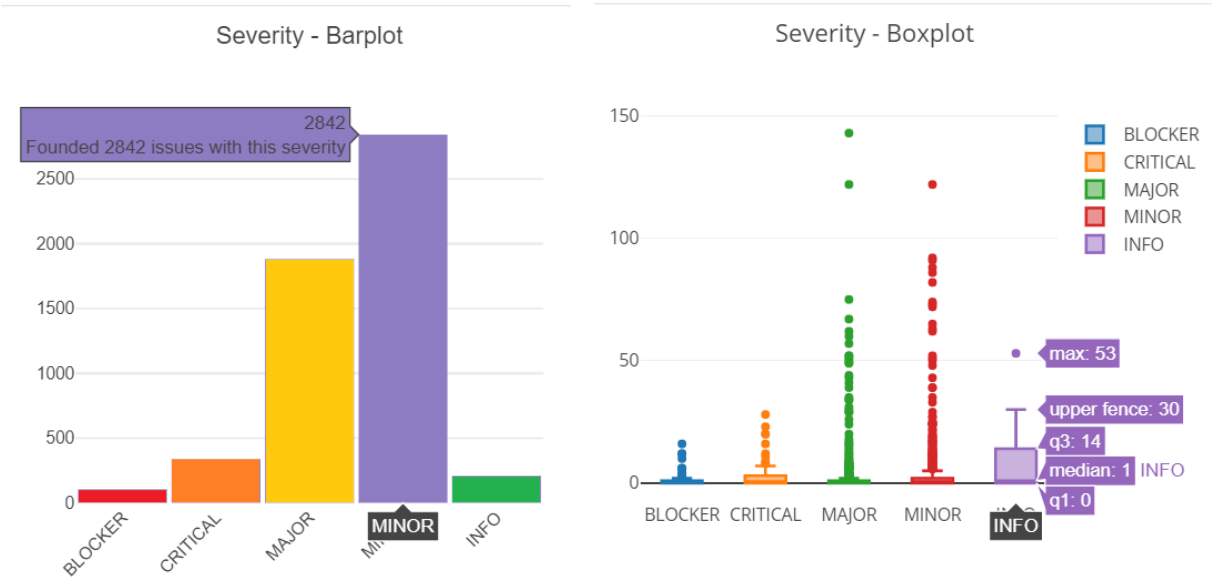


Figura 3.5: Exemplo de relatório fornecido pela ferramenta. Adaptado de: de Araújo et al. (2020)

Também, é disponibilizado ao professor um conjunto de análises em função da Severidade da violação, conforme o Índice *SQALE* (Letouzey, 2012a,b) apresentado na Seção 2.4.1 e demonstrado Tabela 2.3.



(a) Gráfico de Barras das Violações

(b) Diagrama de Caixa das Violações

Figura 3.6: Exemplo de relatório fornecido pela ferramenta, para o agrupamento dos trabalhos produzidos no ano de 2019. Adaptado de: de Araújo et al. (2020)

A ferramenta TM cumpre muito bem o seu objetivo no que diz respeito a permitir o agrupamento de diversos projetos. E fornecer ao professor um conjunto de subsídios que o permita uma avaliação da turma em relação aos conceitos fornecidos pela indústria, como agrupamento por classes e por severidade.

Ao se deparar com as informações fornecidas pela Figura 3.4, é possível identificar a necessidade por parte do professor de entender de fato o que a regra violada representa em relação a disciplina oferecida. Característica apontada como um dos pontos fracos da ferramente por avaliadores, utilizadores e revisores.

3.6 Considerações Finais

Profissionais e pesquisadores da área de computação, membros participantes da Sociedade Brasileira de Computação, em um esforço conjunto definiram comissões para discutir o direcionamento do ensino de computação no Brasil. A partir dessas discussões, foram definidas as Diretrizes Curriculares Nacionais, na qual são descritos os objetivos e áreas de atuação dos cursos de computação no país.

Sendo definidas as Diretrizes Curriculares Nacionais para os cursos de Computação, participantes da Sociedade Brasileira de Computação periodicamente definem grupos de discussão com o objetivo de propor uma matriz curricular a ser utilizada como referência para a definição e avaliação da qualidade dos cursos de computação em todo o país, sendo denominado de Cur-

rículo de Referência.

Na versão mais recente proposta, são definidos eixos de formação que descrevem as competências necessárias ao egresso para que seja capaz de atuar em uma das áreas de atuação definidas pela DCN (Zorzo et al., 2017) A adoção dos Currículos de Referência é opcional aos cursos, mas conforme demonstrado na Tabela 3.2, as matrizes curriculares de diversas instituições de ensino possuem grande similaridade com a proposta da SBC.

No modelo proposto, o aluno ingresso aprende a programar resolvendo problemas computacionais, sendo postergado o ensino de qualidade.

Diversos autores (de Andrade Gomes et al., 2017, Dietz et al., 2018, Krusche e Seitz, 2018) propõem melhorias para o processo de ensino-aprendizagem, focados no ensino dos conceitos de Engenharia de Software.

Dietz et al. (2018) propõem melhorias para o ensino de qualidade de código a partir da utilização dos conceitos de *Clean Code*, demonstrados na seção 2.4.2.

Em outro estudo, Krusche e Seitz (2018) propõem melhorias no processo de avaliação dos artefatos de código produzido pelos alunos, a partir da utilização da integração contínua (seção 2.5.3), associada a análise dinâmica para avaliar a qualidade externa do código-fonte produzido. Os autores definem uma abordagem para o ensino de programação baseado na análise funcional, associada a conceitos da Engenharia de Software.

de Andrade Gomes et al. (2017) trazem ferramentas profissionais de análise estática para dentro do ambiente acadêmico, e a partir da utilização do processo de análise continua, propõem uma ferramenta na qual o aluno pode avaliar a qualidade interna do código presente em seu repositório. Com a utilização da ferramenta, os autores mostram que o aluno foi capaz de entender que certas decisões incorretas tomadas podem impactar negativamente na qualidade do código-fonte.

Ao se deparar com as informações fornecidas na Figura 3.4, é possível identificar a necessidade por parte do professor de entender de fato o que a regra violada representa em relação a disciplina oferecida. Característica apontada como um dos pontos fracos da *Teacher Mate* por avaliadores, utilizadores e revisores.

CAPÍTULO

4

Código-Fonte como Subsídio para o Ensino da Qualidade

4.1 Considerações Iniciais

Como parte das atividades deste trabalho, para a validação da proposta, torna-se necessária a implementação da mesma. Nesse capitulo é apresentada a abordagem utilizada para a implementação do projeto.

A metodologia aqui apresentada restringe-se ao desenvolvimento do projeto, omitindo etapas já cumpridas para a obtenção do título de Mestre (como créditos e exame de proficiência em língua estrangeira).

Para descrever essas atividades assim como o fluxo de trabalho, este seção está dividida em 5 seções, sendo a primeira, esta introdutória, seguida das seguintes:

- A Seção 4.2 apresenta as ferramentas utilizadas para apoiar o desenvolvimento deste projeto.
- Na Seção 4.3, são apresentados os artefatos utilizados como facilitadores ou referenciais teóricos para a efetivação da proposta.
- Na Seção 4.4, são apresentados os artefatos originados para apoiar a proposta.
- Por último, a Seção 4.6 contém algumas considerações sobre o capitulo.

4.2 Ferramentas de Apoio

São primeiramente descritas as ferramentas que apoiaram o processo de implementação e suas respectivas utilizações dentro do escopo do problema,

sendo desconsideradas outras funcionalidades presentes e que não se relacionam ao projeto.

Neste cenário, são desconsideradas ferramentas de controle de versão, ambientes de desenvolvimento integrado, depuradores, navegadores e *frameworks* de desenvolvimento.

4.2.1 Utilização da Plataforma SonarQube™

A plataforma SonarQube™ (SQ) é uma solução, utilizada em larga escala pela indústria de software e algumas de suas aplicações dentro de um ambiente de ensino foram demonstradas na Seção 3.5.3.

Apesar da ferramenta realizar a inspeção de milhares de linhas de código em um tempo extremamente reduzido, a granularidade desse processo de inspeção gera uma grande quantidade de informação, tornando a análise manual dos relatórios gerados consideravelmente custosa ao professor.

Ao submeter o código-fonte produzido por um aluno para a inspeção automatizada, o resultado é justamente o relatório de conformidade desta submissão. Tanto o envio quanto a análise manual de um conjunto de relatórios é uma árdua tarefa ao professor, não sendo essa a proposta do trabalho.

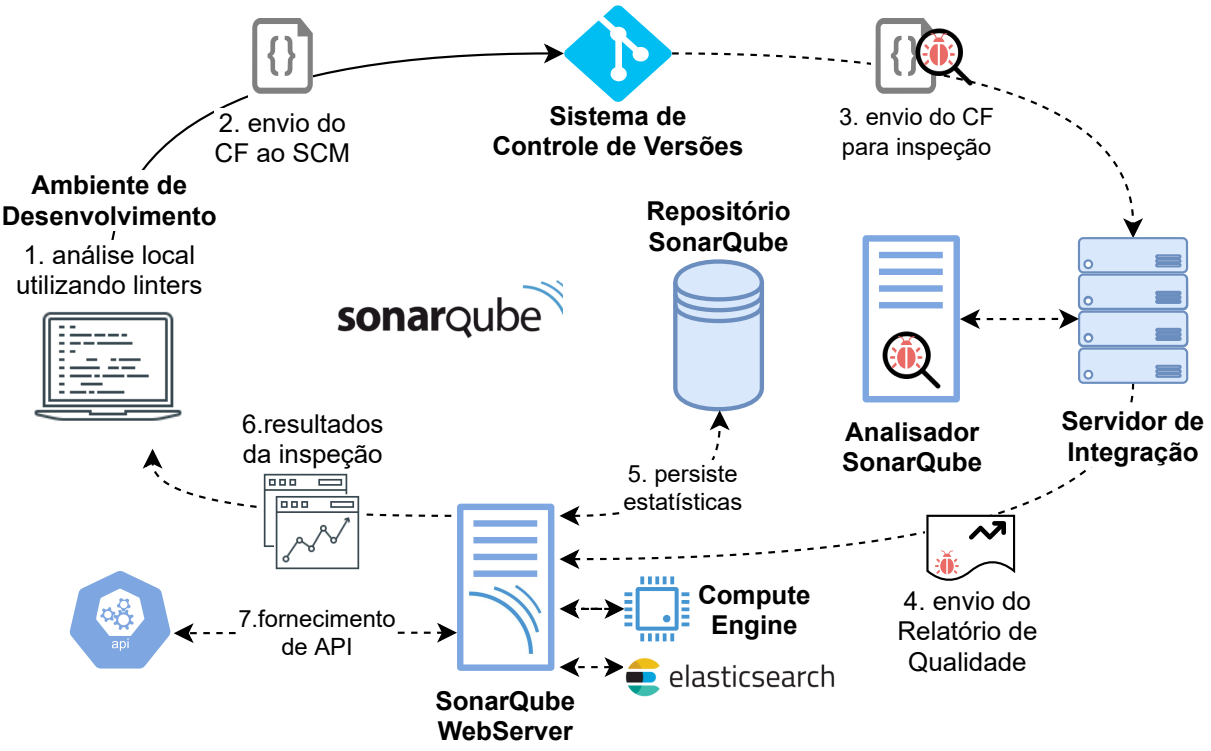


Figura 4.1: Plataforma SonarQube™ em um Ambiente de Desenvolvimento. Fonte: SonarSource S.A, Switzerland. SonarQube™ Docs.

1. Desenvolvedores utilizam IDEs para codificar e realizar análises locais a partir do plug-in SonarLint, compatível com várias IDEs como Eclipse, IntelliJ IDEA e mais recentemente o Visual Studio.

2. Desenvolvedores submetem o fonte para um Gerente de Configuração, como o Git, Subversion, Mercurial, entre outros.
3. Os servidores de integração contínua realizam automaticamente a análise do código submetido através do SonarQube Scaner.
4. Os resultados das análises são submetidos ao servidor do SonarQube para processamento.
5. O servidor do SonarQube processa os dados e os armazena, disponibilizando ao usuário as estatísticas através da interface com usuário.
6. Os desenvolvedores podem revisar e interagir com os resultados apresentados, como adicionar anotações, informar que será corrigido depois, entre outras funcionalidades.
7. Nessa etapa, APIs podem minerar e extrair informação dos dados das análises do SonarQube.

A Figura 4.1 evidencia algumas características interessantes para a aplicação deste projeto. 1) trata-se de ferramenta de inspeção de código que implementa um repositório interno de violações; 2) fornece um analisador e uma base de dados com os problemas encontrados; 3) a ferramenta também dispõe de uma interface web para interação com esse ambiente, 4) a ferramenta fornece uma API bem documentada[1], para fácil utilização em outras soluções.

Considerando estas características associadas ao projeto, bem como a comprovada bem-sucedida utilização da mesma em ambiente acadêmico, optou-se pela sua adoção neste projeto.

Todavia, como um dos desafios do trabalho é garantir que o professor não tenha um aumento na carga de trabalho, fez-se necessária a disponibilização de um outro ambiente de interação, apoiado pela ferramenta Teacher Mate.

4.2.2 Utilização da Ferramenta Teacher Mate

A ferramenta desenvolvida utiliza dos relatórios de inspeção de código-fonte fornecidos pelo analisador estático da ferramenta, *SQ Scanner*, estes disponibilizados via *SQ API*, para fornecer relatórios próprios além de uma série de visualizações (conforme demonstrado na Seção 3.5.3.2) para apoiar tanto o professor (a nível de atividades agrupadas), quanto o aluno (que visualiza a própria submissão).

Considerando a submissão dos trabalhos por parte dos alunos, como uma etapa anterior a demonstrada na Figura 4.2, as próximas etapas são detalhadas a seguir:

1. O professor seleciona quais agrupamentos (turmas e/ou trabalhos) de uma disciplina e envia a requisição ao servidor TM;

[1]Saiba mais em: https://docs.sonarqube.org/latest/extend/web-api/

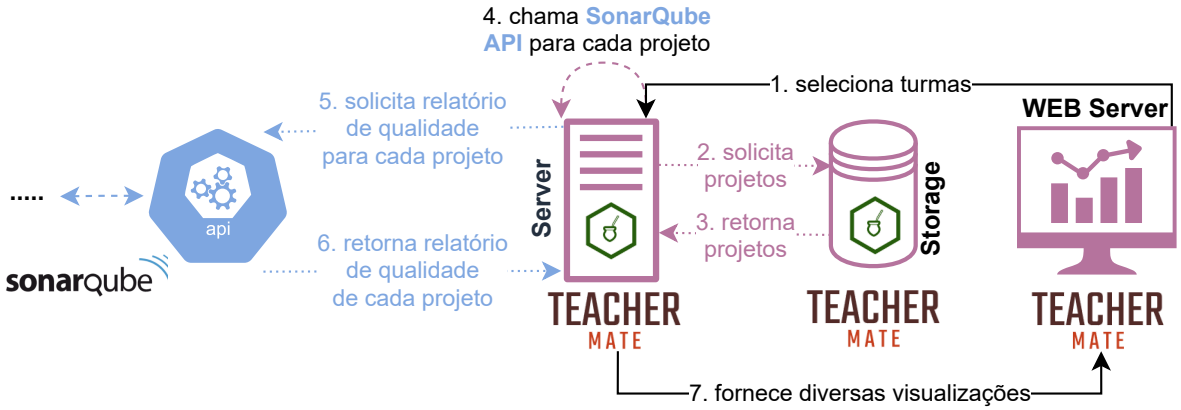


Figura 4.2: Arquitetura da ferramenta Teacher Mate (Adaptação: de Araújo et al. (2020))

2. O servidor busca no repositório quais projetos pertencem ao agrupamento selecionado pelo professor;
3. O repositório devolve ao servidor a lista de identificadores únicos dos projetos selecionados no formato compatível com a **SQ API**;
4. O servidor TM realiza diversas requisições à **SQ API** para cada projeto selecionado, enviando o identificador único do projeto.
5. Com a chamada de API, os relatórios de qualidade gerados no momento em que o aluno realiza o envio do trabalho são solicitados pelo Servidor TM.
6. A **SQ API** segue o fluxo definido na Figura 4.1 e retorna os relatórios de qualidade via *REST API*[2].
7. O servidor TM persiste os resultados no *Web Storage API*[3], realizando a computação e gerando as visualizações diretamente no computador do utilizador.

Ademais, tornou-se possível comparar grupos diferentes, dando ao professor subsídios para comparação através de períodos distintos.

Preservando as características da ferramenta de inspeção, ainda é garantida ao professor a opção de escolher quais regras comporão o perfil de qualidade que será utilizado como parâmetro para a inspeção no código, sendo possível ao professor definir critérios mais ou menos rígidos, a partir do servidor Web SQ.

No escopo deste projeto, a ferramenta permite a análise de cada problema reportado e identificação de conceitos violados. Considerando as características destacadas, torna-se vantajoso a escolha da ferramenta Teacher Mate para apoiar o desenvolvimento da proposta.

[2]Saiba mais em: https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api
[3]Saiba mais em: https://www.w3.org/TR/webstorage/

Uma vantagem da utilização desta abordagem que merece ser destacada, é que conforme indicado na Figura 4.2, a integração do TM com a SQ API garante que os projetos sejam inspecionados apenas uma vez, no momento em que o aluno realiza a submissão do mesmo. Quando a utilização ocorre na perspectiva do professor, a ferramenta SQ API retorna os resultados persistidos no Storage.

4.2.3 Evolução da Abordagem Definida

Em conformidade com a proposta definida, opta-se pela continuidade da utilização das ferramentas apresentadas. Faz-se necessária a evolução da ferramenta Teacher Mate, sendo idealizadas, desenvolvidas e implantadas novas funcionalidades, apresentadas a seguir.

Além das características apresentadas, foram considerados os seguintes fatores na avaliação e consequente escolha das ferramentas: 1) ambas são ferramentas gratuitas; 2) de código-aberto; 3) disponíveis em repositórios de código-fonte; 4) com comprovações de sucesso na utilização em ambiente acadêmico; 5) a plataforma SonarQube™ vem sendo utilizada desde 2015 em projetos dentro desta Universidade; 6) construída e mantida dentro desta Universidade, a ferramenta Teacher Mate é o terceiro projeto que envolve a utilização da plataforma SQ, indicando certa maturidade dos participantes envolvidos na utilização da tecnologia.

Para facilitar a demonstração da solução implementada, a Figura 4.3 demonstra diagrama da modelagem relacional que foi incluída na aplicação já existente.

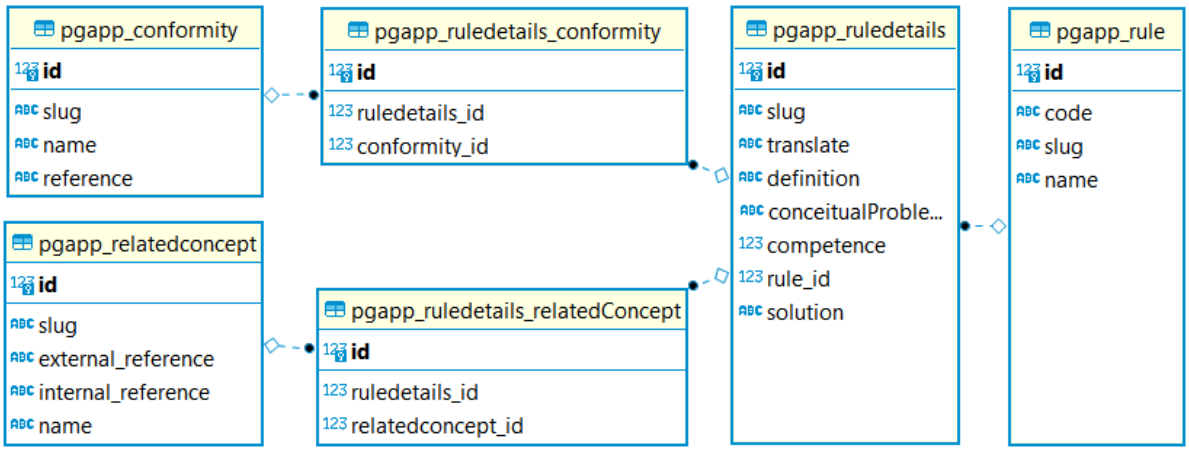


Figura 4.3: Modelagem da Nova Estrutura da Ferramenta TeacherMate (Produção Própria)

Na Figura 4.4, são representadas as principais atividades desenvolvidas no intuito de atingir o objetivo almejado. Além destas atividades, são destacadas ferramentas de apoio e itens que foram adotados ou fornecidos a partir da proposta.

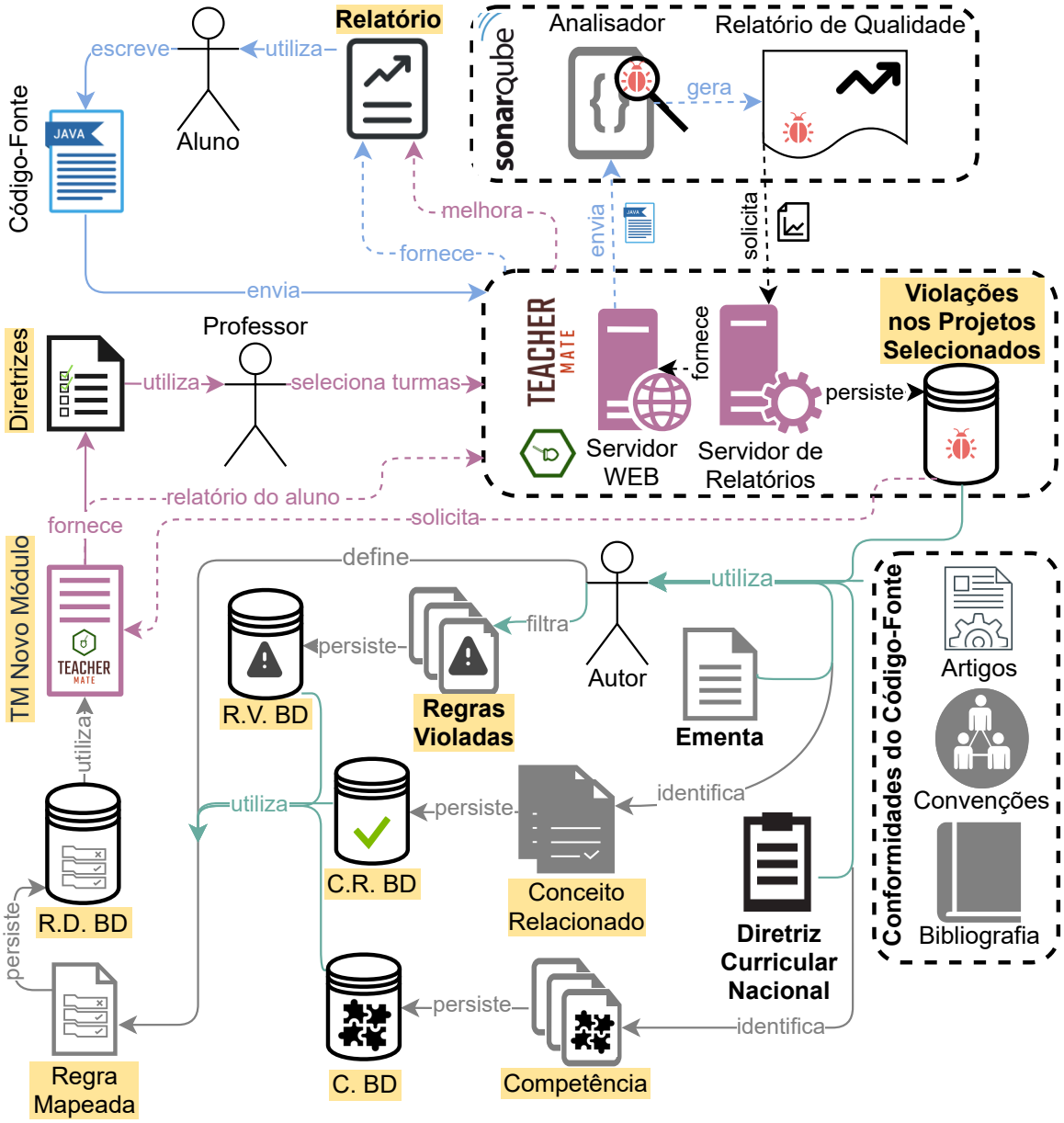


Figura 4.4: Metodologia desenvolvida para este trabalho (Produção Própria)

Para que a aplicação da metodologia fosse possível, foi necessária a utilização de diversos itens que contribuíram para o desenvolvimento do projeto.

Consideram-se artefatos todos os itens conceituais, físicos ou lógicos utilizados pelo autor, seja a partir da adoção de artefatos existentes e validados ou desenvolvimento de novos artefatos.

No escopo deste trabalho, são considerados artefatos os seguintes itens:

- Artefatos Adotados
  - Lista de Regras

    Conforme objetivo demonstrado na Seção 1.5, para identificar trechos de código em desacordo com convenções e padrões, faz-se ne-

cessária a adoção de um conjunto de regras. No escopo deste projeto, definiu-se pela utilização das regras providas pela plataforma SonarQube e mantidas pela SonarSource.

- □ Repositório de Violações

  Este repositório contém os resultados das análises executadas pela ferramenta SonarQube (Seção 4.2.1) nos trabalhos realizados por alunos na disciplinas de P.O.O., totalizando mais de 60 mil violações.

- □ Lista de Conformidades com as Regras

  Com o objetivo de justificar e motivar o docente sobre as razões pelas quais a violação não deve ser cometida, além de prover embasamento teórico ao docente, alunos e autor. Tornando este último apto a realizar demais etapas da proposta. O artefato utilizado esta disponível no Anexo C.

- □ Lista de Habilidades Adquiridas (Competências)

  Conforme apresentado na Seção 2.1, utilizou-se os Referenciais de Formação Rocha et al. (2005) e Zorzo et al. (2017) descritos na Seção 3.2, para identificar a lista de habilidades transversais relacionadas ao escopo do projeto e vincula-las com as regras.

■ Artefatos Produzidos

- □ Lista de Conceitos Relacionados com as Regras

  Para auxiliar professor e aluno a identificar mais rapidamente em quais conceitos ministrados o aluno apresentou alguma dificuldade, realizou-se o vinculo dos conceitos relacionados com a disciplina (Anexo B) com a Lista de Regras, produzindo o artefato disponível no Anexo E.

- □ Lista de Habilidades Associadas com as Regras.

  Com o objetivo de diminuir a lacuna entre a indústria e acadêmia, utilizando o exposto na Seção 4.3.4, realizou-se a vinculação das conformidades com a Lista de Regras. Disponível no Anexo C.

- □ Lista de Regras Mapeadas

  Tendo como objetivo fornecer mais autonomia ao aluno, motiva-lo a corrigir e não cometer mais determinada falta, o mapeamento das regras é um importante artefato que conta com a explicação da regra no escopo da disciplina. Estando este artefato disponível no Anexo F.

- □ Diretrizes Propostas

A próxima seção descreve os artefatos utilizados pelo autor com o objetivo de apoiar o desenvolvimento da proposta.

4.3 Artefatos Adotados

A seguir são apresentados os objetos de valor (artefatos) disponíveis e que foram utilizados para o desenvolvimento do projeto.

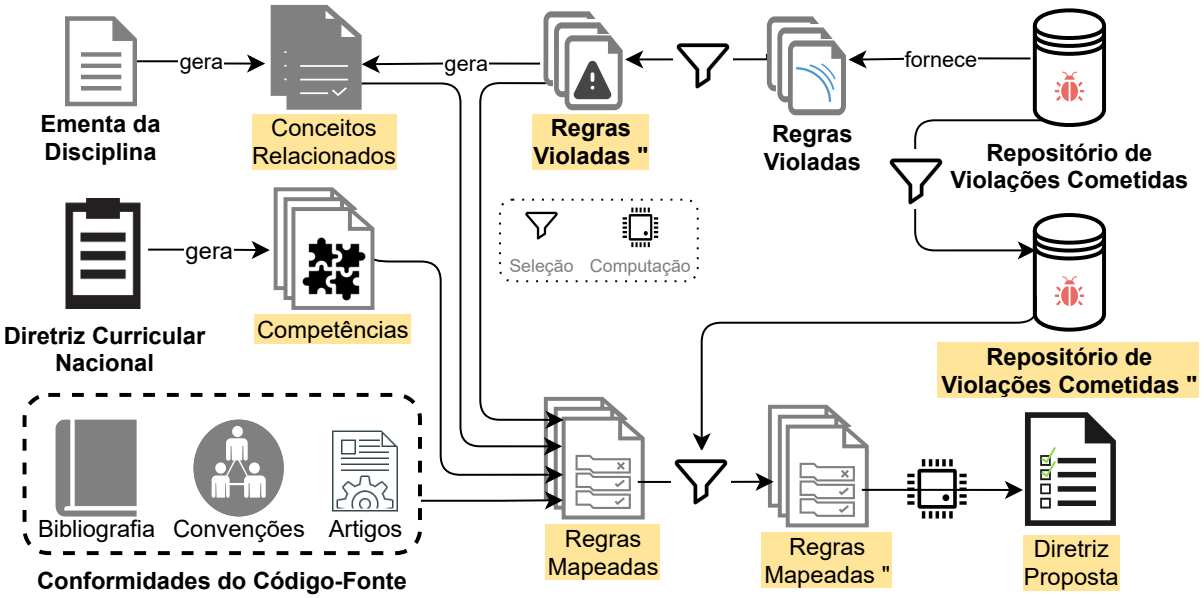


Figura 4.5: Artefatos relacionados com o desenvolvimento do projeto (Produção Própria)

4.3.1 Lista de Regras

Como primeira parte da evolução, tendo como objetivo permitir ao professor a definição do conjunto de regras a ser utilizado sem que para isso ele precisasse utilizar utilizar o painel gerencial provido pela plataforma SonarQube™, foi desenvolvida uma solução na qual é possível incluir e/ou remover regras que serão consideradas na hora da análise dos código-fontes submetidos. Torna-se possível a definição de um processo de inspeção (análise estática) mais ou menos rigoroso, conforme escolha arbitrária do professor, dentro da ferramenta TM. Essa solução pode ser observada na Figura 4.6.

É importante destacar que novas regras não podem ser criadas dentro da ferramenta, sendo possível apenas a adição ou remoção (seleção) de regras dentre as milhares que compõem a plataforma, desenvolvidas pela comunidade e mantidas pela SonarSource[4].

Para o escopo deste projeto, foram importadas as 430 regras ativas (dentre as 643 disponíveis)[5] no perfil padrão para a Linguagem Java (SonarWay[6]), utilizada na disciplina de Programação Orientada a Objetos.

[4]Disponível em: https://rules.sonarsource.com/
[5]Saiba mais em: https://rules.sonarsource.com/java
[6]Saiba mais em: https://docs.sonarqube.org/latest/instance-administration/quality-profiles/

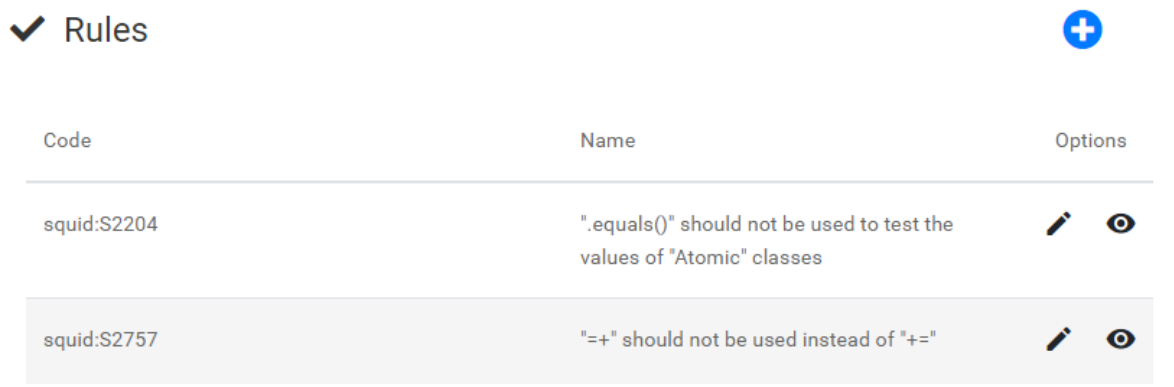


Figura 4.6: Exemplo de Listagem das Regras fornecidas pelo SonarQube™ , mantidas pela SonarSource sendo utilizadas dentro da plataforma Teacher Mate.

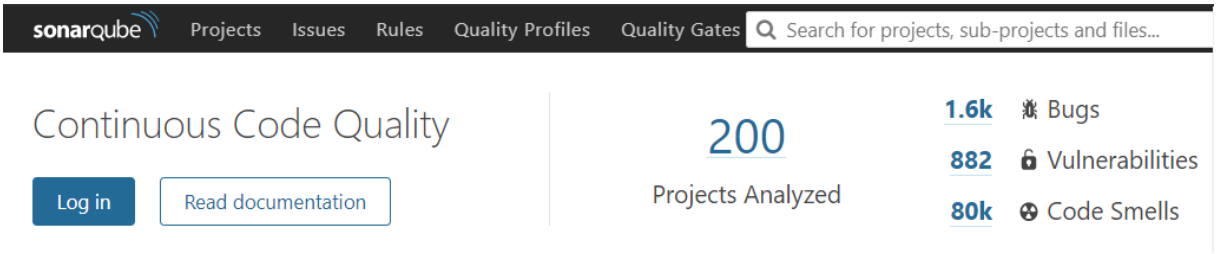


Figura 4.7: Interface inicial do SQ WebServer, destacando o número de projetos analisados, bem como a quantidade de defeitos, vulnerabilidades e *code smells* (Produção Própria)

Para a aplicação da metodologia, utilizou-se um repositório de projetos previamente analisados, contendo um total de 200 projetos (Figura 4.7) em Java, sendo a maioria dos projetos, trabalhos acadêmicos enviados por alunos pertencentes a turmas da disciplina de Programação Orientada a Objetos. Trabalhos de outras disciplinas foram desconsiderados, totalizando 180 projetos analisados.

Foram submetidas para análise os trabalhos acadêmicos realizados pelos alunos das turmas entre 2013 e 2019, totalizando 7 turmas, cada uma com dois projetos (trabalho 1 e trabalho 2). Ao realizar uma análise preliminar, identificou-se que um pequeno grupo de regras representa a grande maioria das violações. Neste sentido, foram feitas as devidas análises de todas as regras nas quais o quantitativo das violações é igual ou superior a 1% das ocorrências identificadas, utilizando dois diferentes critérios de seleção: 1) a porcentagem em função da classe da regra violada (Defeito, Falha de Segurança ou Código Sujo); 2) a porcentagem em função do total absoluto de violações.

As regras que não fizeram parte deste agrupamento não foram analisadas devido ao baixo impacto nos cálculos das diretrizes e alta complexidade do processo análise. Porém, a ferramenta possibilita que as mesmas possam ser

futuramente vinculadas.

Considerando o agrupamento desses dois critérios (porcentagem relativa em função a classe da violação e porcentagem relativa ao valor absoluto), um total 37 regras foram analisadas, seguindo o procedimento definido e exemplificado na Seção 4.4.4, sendo essas regras listadas no Anexo F.

4.3.2 Repositório de Violações

Para o desenvolvimento da solução, foi necessária a utilização de um repositório de violações. Aproveitando a estrutura previamente disponibilizada pela ferramenta Teacher Mate, optou-se pela utilização desse repositório em tempo de execução.

Quando o professor seleciona a(s) turma(s) da qual deseja realizar a análise, o Servidor TM faz um conjunto de requisições via API e persiste dinamicamente o resultado dessas requisições nos arquivos temporários do computador solicitante (cliente), utilizando a *Web Storage API*.

Com algumas melhorias na estrutura do repositório, tornou-se mais simples trabalhar com as informações disponibilizadas.

Dentre as regras utilizadas no perfil *SonarWay*, a grande maioria não foi violada por nenhum aluno. A quantidade de regras violadas considerados os projetos de todas as turmas, desde 2013 até 2019 é de 136 (31,63%), sendo identificados 64.179 trechos de códigos em desacordo (violando) com uma ou mais regras. A quantidade de violações por turma e o tipo de violação estão disponíveis na Tabela 4.1.

Tabela 4.1: Violações cometidas pelas turmas dos anos de 2013 a 2019, separadas por classe de violação, considerando os dois trabalhos (Produção Própria)

Ano	Defeitos	Vulnerabilidades	Código Sujo	Total
2013	193	93	7338	*7624*
2014	227	98	9981	*10306*
2015	280	165	10744	*11189*
2016	200	182	11302	*11684*
2017	169	104	10513	*10786*
2018	193	68	6941	*7202*
2019	135	50	5203	*5388*
Total	*1397*	*760*	*62022*	***64179***

4.3.3 Lista de Conformidades com as Regras

Toda regra mantida pela plataforma SonarQube™ é advinda de algum padrão estabelecido e relacionado com os conceitos de qualidade interna de código-fonte. Não existindo uma norma especifica, faz-se necessário a aplicação de convenções existentes e mencionadas na Seção 2.4.

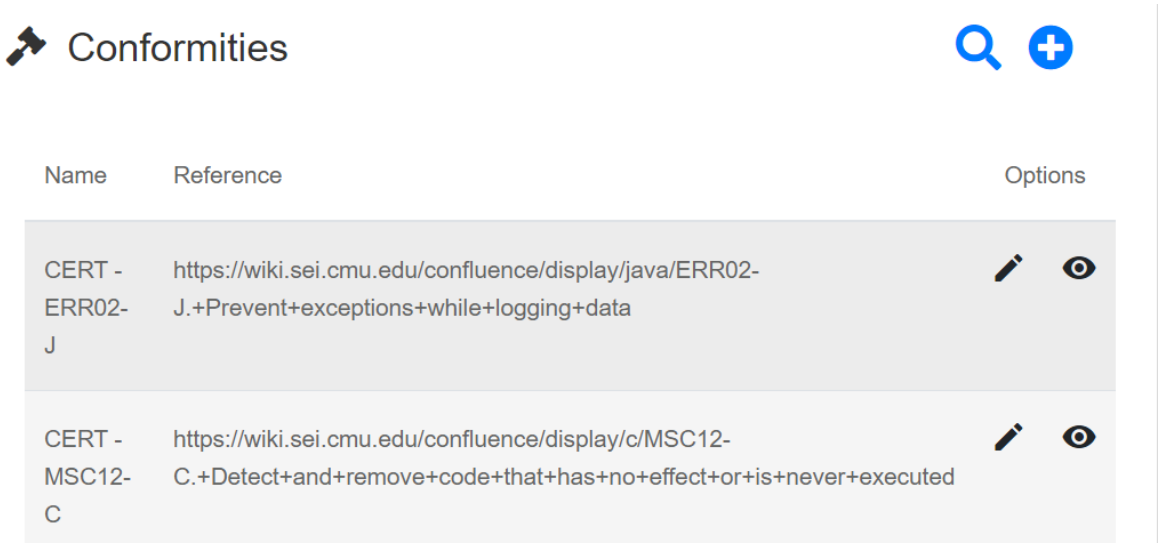


Figura 4.8: Exemplo Listagem das Conformidades (Produção Própria)

A adição das convenções e vínculo das mesmas com as regras, possibilita tanto ao educador quanto ao aluno o entendimento da importância daquela regra e os motivos pelos quais a violação deve ser evitada. Esse vínculo, na maioria dos casos foi obtido a partir de um processo de análise das regras, dentro dos repositórios e fóruns de discussão dedicados ao assunto e mantidos pela SonarSource.

Figura 4.9: Adequação das Regras as convenções, padrões e documentações (Produção Própria)

Ao acessar o conteúdo disponibilizado como referência, o utilizador da ferramenta (professor ou aluno) irá se deparar com um material mais técnico, que incluem exemplos de código em conformidade e não-conformidade; risco associado, que inclui o grau de severidade e custo da remediação; vulnerabilidades relacionadas e referências bibliográficas. Em alguns casos, existem discussões promovidas por colaboradores e que permitem ao interessado postar suas dúvidas ou sugestões, contribuindo assim para a maturidade do indivíduo e dos projetos envolvidos.

Ao todo, um total de 35 itens de conformidade foram avaliados e inclusos no escopo da ferramenta, sendo que cada regra está de acordo com um destes itens e cada item pode agrupar mais de uma regra, conforme podemos observar na Figura 4.9.

A lista dos itens de conformidade, sendo eles documentações da linguagem, convenções de código, convenções de segurança, além de alguns padrões definidos, estão disponíveis no Anexo C.

4.3.4 Lista de Habilidades Adquiridas (Competências)

As definições de competência abordadas neste projeto fazem parte dos Referenciais de Formação na área de Computação para os cursos de Bacharelado em Ciência da Computação.

Cada regra está associada a apenas uma competência. As competências no escopo deste projeto podem ser divididas em apenas três categorias, sendo implementada diretamente como um campo na janela de detalhes da regra. Conforme será visto a seguir.

Competências são habilidades que podem ser adquiridas pelo aluno a partir do entendimento da regra e estão em consonância com as Diretrizes Curriculares Nacionais (DCN16) e Referenciais de Formação para Ciência da Computação (RF-CC-17) abordadas na Seção 3.2.

4.4 Artefatos Produzidos

A seguir são apresentados os artefatos produzidos pelo autor, sendo estes, considerados como resultados de processos necessários para atingir o objetivo do trabalho.

4.4.1 Lista de Conceitos Relacionados com as Regras

Para associar as violações cometidas por alunos ao conteúdo ministrado na disciplina, foi desenvolvida uma solução na qual o professor torna-se capaz de manter um conjunto de conceitos relacionados com a disciplina ofertada (Figura 4.10).

Conceitos Relacionados, são conteúdos ministrados em disciplinas, que em conjunto, permitem ao aluno desenvolver aptidão em determinada competên-

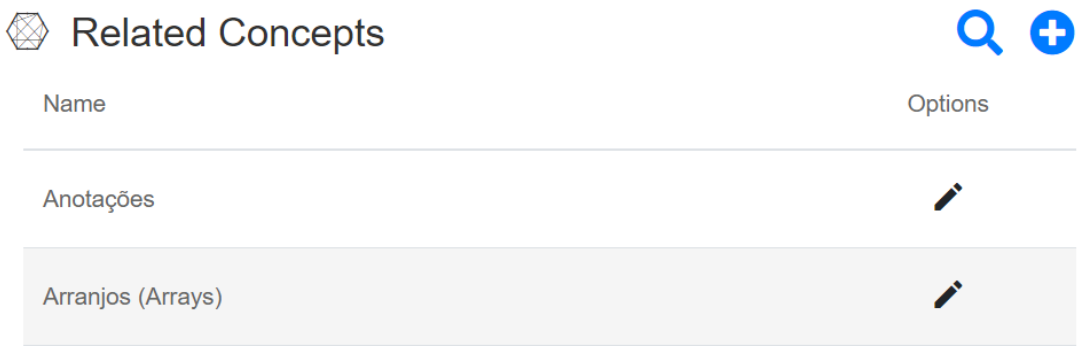


Figura 4.10: Exemplo de Conceitos Relacionados com a Disciplina de Programação Orientada a Objetos (Produção Própria)

cia. Esses conteúdos são comumente identificados nas ementas das disciplinas e nos planos de aula. Esse modelo segue a ideia proposta por Zorzo et al. (2017) demonstrado na Seção 3.2.

Um mesmo conceito pode estar relacionado com mais de uma regra, assim como uma violação pode ser resultado de uma falta de conhecimento ou habilidade em um ou mais conceitos, sendo possível vincular vários conceitos com a mesma regra e várias regras ao mesmo conceito.

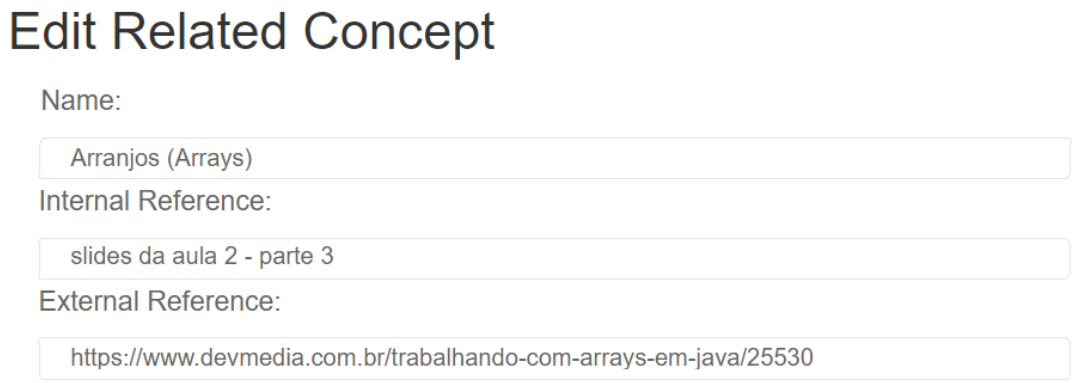


Figura 4.11: Manutenção dos Conceitos Relacionados a Disciplina (Produção Própria)

Na Figura 4.11, é possível notar que além do nome do conceito, o professor pode incluir referências internas (como slides, apostilas e material de apoio) e referências externas (como artigos, documentação, tutoriais e dicas).

Para validar a metodologia, foi necessário identificar todos os conteúdos ministrados na disciplina de P.O.O. Concomitantemente com o processo de identificação dos conceitos relacionados com a disciplina, foi realizado o processo de vinculação destes conceitos ao conjunto de regras definidos de acordo com subseção anterior (4.3.1). A descrição do processo para obtenção dos conceitos e vinculação dos mesmos com as regras é apresenta a seguir. A lista destes conceitos é demonstrada na Tabela 4.2.

1. **Identificar os Conceitos Relacionados com a Disciplina**

Para essa primeira etapa, foi feita a análise do Conteúdo Programático da Disciplina de Programação Orientada a Objeto oferecida na FCT/UNESP e aprovada pelo Departamento de Matemática e Computação (DMC) em 14/12/2017. Nesta etapa, foram identificados 20 conceitos da disciplina, considerados teórico-expositivos, conceitos que são apresentados em sala de aula pelo professor.
O Conteúdo Programático está disponível no Anexo B.

2. **Aprofundamento teórico no Conceito Relacionado identificado**
   Para cada Conceito Relacionado com a Disciplina de POO identificado, foram pesquisadas e analisadas as principais referências sobre o assunto; documentação, artigos, fóruns de discussão conceituados, etc. Sendo disponibilizadas essas referências externas no ambiente da ferramenta, conforme Figura 4.11.

3. **Vinculação com a Regra**
   Nesta etapa, para cada regra entre as selecionadas de acordo com a taxonomia definida na Seção 4.3.1, foram vinculados os conceitos dos quais o domínio do conteúdo permite ao aluno não cometer a violação. Para o apoio bibliográfico, foram considerados tanto as referências bibliográficas da disciplinas como Deitel (2016), artigos na área (de Araújo et al., 2020, Dietz et al., 2018), documentação da linguagem Java, e demais itens mencionados na Subseção 4.3.3.

4. **Refinamento dos Conceitos Relacionados**
   Algumas regras da etapa anterior evidenciaram outros conceitos, comumente associados a prática da disciplina, ou seja, a própria realização de atividades a partir da escrita de código. Em contrapartida dos conceitos inicialmente identificados, os conceitos práticos-construtivos não fazem parte da ementa, nem do conteúdo programático da disciplina, estando associados com as dificuldades que os alunos enfrentam enquanto desenvolvem as tarefas e tentam colocar em prática os conceitos teóricos ministrados.
   Ao identificar um desses conceitos, é realizada novamente a etapa de aprofundamento teórico (2) e ao confirmar que se trata do escopo da disciplina, o conceito é incluído no repositório, junto com as devidas referências.
   Nessa etapa foram identificados outros 16 conceitos, totalizando 36 conceitos relacionados com a disciplina.

5. **Pair-Review**
   Após a identificação dos conceitos relacionados com a disciplina, vinculação com as regras e refinamento do processo, este ultimo permitindo a

vinculação de novos conceitos. O resultado dos estudos foi enviado para que um professor, especialista na disciplina, pudesse validar a proposta.

Ao final do processo descrito, foram identificados os seguintes conceitos, conforme observa-se na Tabela 4.2.

Tabela 4.2: Conceitos Relacionados com a disciplina de Programação Orientada a Objetos de acordo com Projeto Pedagógico disponível no Anexo B (Produção Própria)

#	Conceitos Teórico-Expositivos	#	Conceitos Prático-Construtivistas
39	Abstração (Classes e Métodos)	31	Anotações
12	Atribuição	20	Bloco de Instrução
13	Atributos e Variáveis	22	Classes Anônimas
03	Classes	32	Comentários
24	Coleções (Arranjos)	33	Contexto Estático (static)
10	Construtores e Destrutores	14	Escopo
11	Controles de Acesso (getters e setters)	07	Fluxo de Execução
30	Conversões (Cast)	06	Garbage Collection
21	Declaração e Definição	35	Importação
08	Encapsulamento	16	Loggers
15	Entrada/Saída de Dados	02	Null Pointers
29	Estruturas de Controle	25	Pacotes
40	Fluxo de Repetição (Iteradores)	09	Padrão de Projeto
27	Função	28	Serialização
05	Herança	18	StackTrace
36	Métodos	01	Tratamento de Exceções (Throwable)
04	Objetos		
41	Operadores e Expressões		
38	Polimorfismo (Sobrecarga e Sobreposição)		
37	Tipos de Dados		

Ao analisar a Figura 4.12 podemos constatar que de fato, um conceito pode conter várias regras. Ao explorar o gráfico, é possível observar que as regras também fazem parte de vários conceitos. Essas informações estão disponíveis no Anexo E.

4.4.2 Lista de Habilidades Associadas com as Regras

As competências foram separadas em três grupos distintos, todos associados ao 2º EIXO DE FORMAÇÃO: DESENVOLVIMENTO DE SISTEMAS. Este eixo de formação incluí tanto o conjunto de competências associados a criação de sistemas, quando manutenção e adaptação de sistemas existentes. O aluno é capacitado para a realizar a elicitação, análise, especificação e validação dos requisitos. Fazem parte das habilidades a definição do projeto (design) arquitetônico e detalhado de um software para a sua construção (codificação), a escrita de código e o correto emprego de Ferramentas e Técnicas visando a garantia do controle de qualidade.

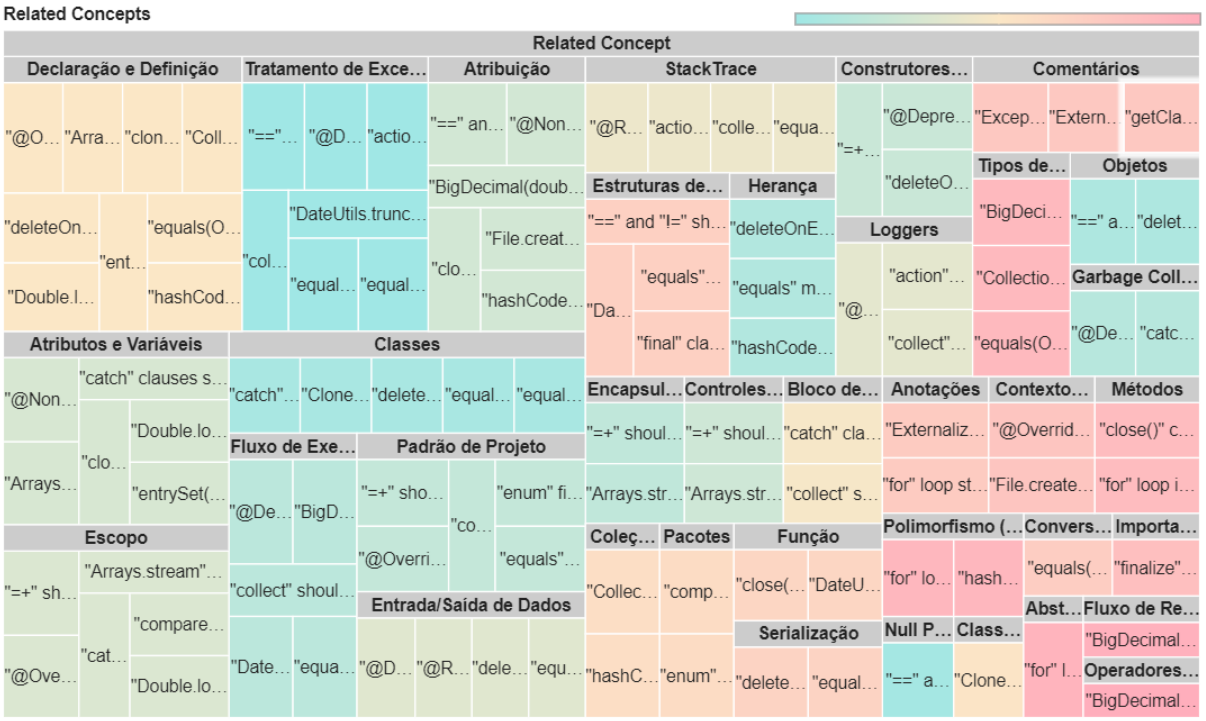


Figura 4.12: Conceitos Relacionados com a Disciplina de POO (Produção Própria)

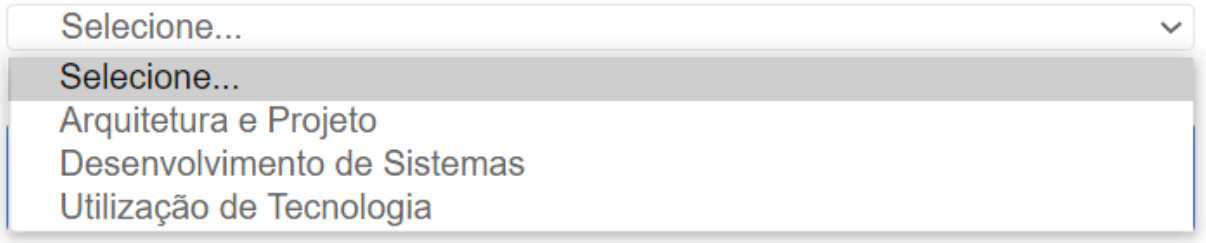


Figura 4.13: Exemplo Listagem de Competências (Produção Própria)

- **Arquitetura e Projeto**
  Esta competência está associada com as regras que garantem a aplicação correta de conceitos como padrões criacionais (Construtores, Singleton); padrões estruturais (MVC, MVVM,...); padrões comportamentais (atribuição de responsabilidade). Um exemplo de violação é o acesso a base de dados a partir da camada de visualização.

- **Desenvolvimento de Sistemas**
  Esta competência está associada com convenções genéricas de escrita de código, como a nomenclatura, alinhamento e endentação do código-fonte, cobertura e testes unitários, declarações, e até utilização de comentários e documentação de código.

- **Utilização de Tecnologia**
  Esta competência está associada com recursos específicos providos pela Linguagem de Programação adotada em classe. No escopo deste pro-

jeto, foram mapeadas as regras em que a recomendação para correção da violação está associada a correta utilização das ferramentas e técnicas providas pelo Java, como o uso de **Coleções**, **Iteradores**, **Pacotes**, entre outros.

4.4.3 Lista de Regras Mapeadas

Em conformidade a proposta deste trabalho, foi desenvolvida a funcionalidade denominada Detalhes da Regra. Trata-se da união entre tudo que foi apresentado nos itens (4.3.1, 4.4.1, 4.3.3 e 4.3.4) desta seção.

O relacionamento lógico entre Regra, Conceito Relacionado possibilitou a classificação da violação em relação ao conteúdo da disciplina. Já o vínculo da Regra com a Conformidade, permitiu explorar a criação de sugestões de solução para o trecho de código que contém a falta cometida pelo aluno.

Por último, o vínculo de Regra com a Competência tenta alinhar a metodologia definida aos itens dos Referenciais de Formação para Ciência da Computação (RF-CC-17) de acordo com o escopo proposto deste trabalho.

Esse último item não é o foco principal do projeto, trata-se de uma preparação para que em trabalhos futuros seja possível a análise do perfil do aluno ou turma a partir de macro-competências, que fazem parte das habilidades adquiridas de forma transversal, a partir da análise multidisciplinar, conforme definido por Zorzo et al. (2017).

Para o mapeamento e vinculação das regras, adotou-se a seguinte taxonomia:

1. **Selecionar uma regra ainda não mapeada:**

   Primeira etapa do processo, a regra escolhida deve estar entre as definidas seguindo o procedimento descrito na Seção 4.3.1.

2. **Estudar a Regra na plataforma SonarSource**[7]:

   Tanto a instância do SonarQube, quanto a página mantida pela SonarSource, contém informações relevantes sobre a regra em questão, como a contextualização, exemplos de trechos de código de acordo e que violam a regra, além de referências externas para itens de conformidade (conforme explicado na Seção 4.3.3.

3. **Identificar os itens de conformidade que estão associados a regra**:

   Conforme mencionado no item anterior, a própria plataforma fornece algumas referências nas quais a regra se baseia. Em muitos casos, essas informações não foram suficientes para justificar a importância da mesma, ou qual o problema relacionado com a violação. Neste caso, buscou-se referências externas, que complementassem a análise.

[7]Disponível em: https://rules.sonarsource.com/java

4. **Estudar cada referência de conformidade e como a não adequação impacta no código-fonte**:

   Após a identificação das referências externas, foram identificados quais continham informações que que auxiliavam o autor a associar a regra com os conceitos da disciplina, sendo associada a referência mais relevante. Todas essas referências estão disponíveis no Anexo C.

5. **Analisar exemplos de trechos de códigos produzidos pelos alunos**:

   Com o intuito de aprofundar nos conceitos estudados, a partir da instancia montada do SonarQube foram analisadas amostras trechos de código produzido pelos alunos e que dispararam a violação. Com isso foi possível adentrar ao escopo da disciplina. Nesta etapa identificou-se qual a macro-competência a que a mesma se refere, de acordo com a Seção 4.3.4.

6. **Identificar os conceitos relacionados com a regra**:

   Por requerer o nível mais alto de embasamento teórico, a ultima etapa do processo de análise é a vinculação dos conceitos relacionados com as regras. Nesta etapa, dada a lista de conceitos disponíveis, verifica-se quais se relacionam com a regra e se necessário, adicionam-se novos conceitos, conforme a taxonomia descrita na Seção 4.4.1.

7. **Peer-review**:

   Após a escolha, contextualização, definição, proposta de solução, assim como vinculação com a conformidade, competência e conceitos relacionados, as regras mapeadas foram submetidas para o docente especialista na disciplina. O docente pôde realizar as devidas considerações e possíveis correções com o intuito de colaborar e validar a proposta e vinculação.

Na Figura 4.14, é possível observar um exemplo da regra ***'Switch' statements should end with 'default' clauses*** detalhada, com todos os vínculos realizados. O campo *Portuguese* adequa o nome das regras fornecidas pela ferramenta ao escopo do projeto, considerando que nem todos os alunos tem o pleno domínio da língua inglesa. Os demais campos foram preenchidos de acordo com o procedimento descrito no próximo capítulo.

A vinculação das regras aos itens em questão é requisito para que a ferramenta possa auxiliar o professor na tarefa de criar as diretrizes a partir das violações encontradas e fornecer subsídios para a melhoria na condução do ensino de programação, de acordo com o Item 3 da metodologia proposta.

A demonstração da aplicação da ferramenta bem como a validação da mesma é demonstrada a seguir.

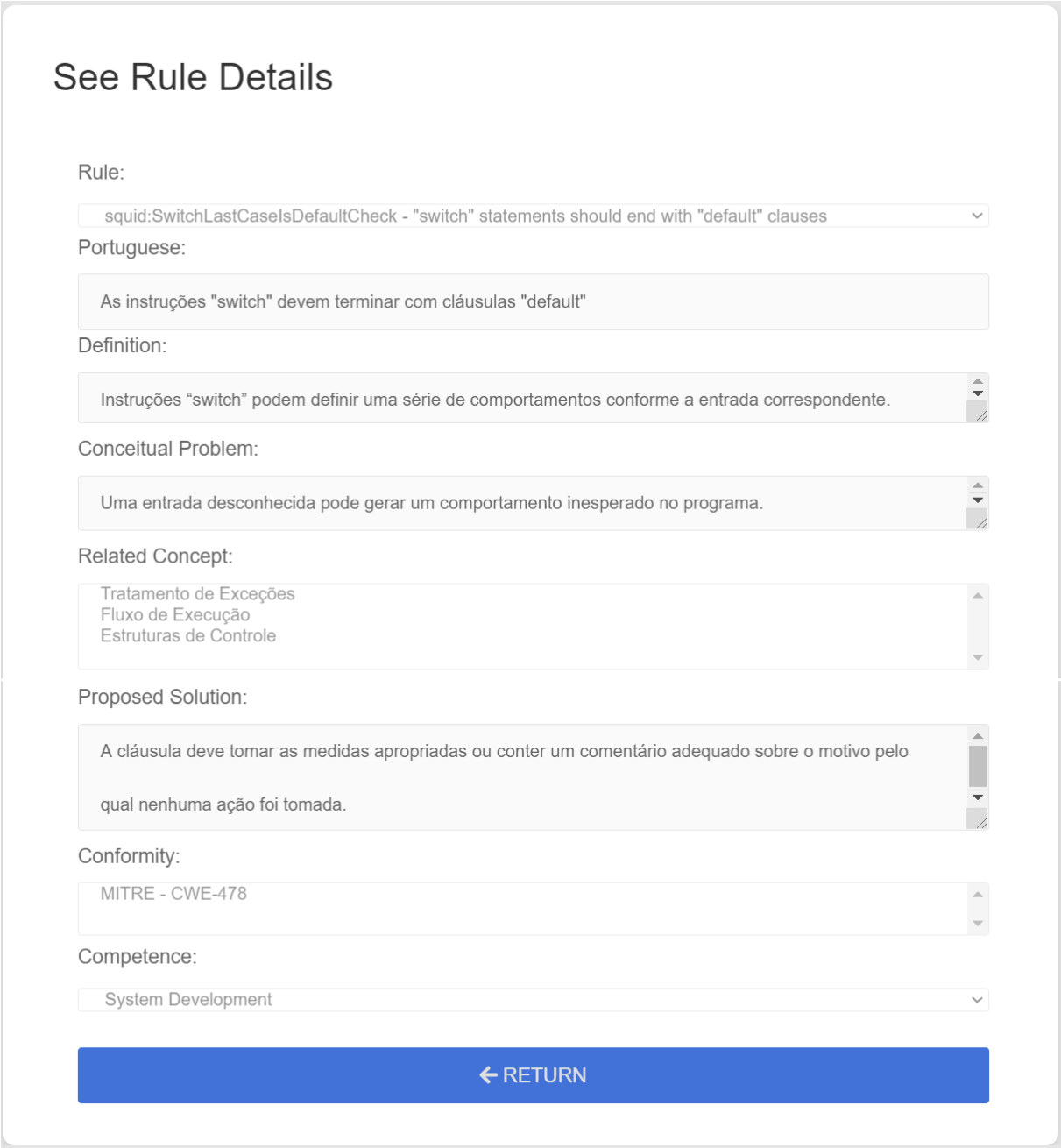


Figura 4.14: Exemplo dos detalhes da regra *squid:S1444* (Produção Própria)

4.4.4 Exemplos de Regras Mapeadas

A seguir são apresentados exemplos de regras avaliadas que podem compor o repositório de Regras Avaliadas. As regras foram arbitrariamente selecionadas com o intuito de facilitar ao leitor a exemplificação de como as diretrizes são geradas. O critério para escolha deu-se pela capacidade do conjunto representar o montante de regras, violações e conceitos relacionados.

A lista de todas as regras avaliadas, de acordo com os critérios adotados na Seção 4.3.1, está disponível no Anexo F.

R1 – Resources should be closed

Recursos devem ser encerrados

Definição

Conexões, streams, Arquivos e classes que implementam a interface Closable, devem ser encerrados após o uso.

Problema Conceitual

A falta de encerramento dos recursos implica em vazamento de recursos.

Conceitos Relacionados

Garbage Collector, Fluxo de Execução, Construtores e Destrutores, Entrada/Saída de Dados, Tratamento de Exceções

Exemplos de Código

Não Conformidade

```
private void doSomething() {
  OutputStream stream = null;
  try {
    for (String property : propertyList) {
      stream = new FileOutputStream("myfile.txt");
    /* ... */
    }
  } catch (Exception e) {
    /* ... */
  } finally {
    stream.close();
   //Multiple streams were opened. Only the last is closed.
  }
}
```

Em Conformidade

```
private void doSomething() {
  OutputStream stream = null;
  try {
    stream = new FileOutputStream("myfile.txt");
    for (String property : propertyList) {
    /* ... */
    }
 } catch (Exception e) {
    /* ... */
 } finally {
```

```
            stream.close();
        }
    }
```

Solução Proposta

Utilizar o bloco finally ou o padrão try-with-resources.

Conformidade

MITRE, CWE-459 - Incomplete Cleanup

Competência

Utilização de Tecnologia

R5 – Standard outputs should not be used directly to log anything

As saídas padrão não devem ser utilizadas para registrar nada

Definição

O uso de saídas padrão para registrar os logs é altamente desaconselhável.

Problema Conceitual

Registros de logs precisam ser facilmente recuperáveis, devem possuir uma estrutura uniforme e devem restringir o acesso, quando necessário.

Conceitos Relacionados

Entrada/Saída de Dados, Loggers, StackTrace

Exemplos de Código

Não Conformidade

```
private void doSomething() {
    System.out.println("Passou aqui!");
}
```

Em Conformidade

```
private void doSomething() {
    logger.log("Entrou no método doSomething()");
}
```

Solução Proposta

Substituir as saídas padrão para Loggers

Conformidade

CERT, ERR02-J. - Prevent exceptions while logging data

Competência

Utilização de Tecnologia

R6 – Throwable.printStackTrace(...) should not be called

Throwable.printStackTrace(...) não deve ser chamado

Definição

a utilização deste recurso imprime a pilha de execução de algum fluxo.

Problema Conceitual

por padrão, esse fluxo pode expor informações sensíveis

Conceitos Relacionados

Tratamento de Exceções, Loggers, StackTrace

Exemplos de Código

Não Conformidade

```
private void doSomething() {
try {
    /* ... */
} catch(Exception e) {
    e.printStackTrace();
}
```

Em Conformidade

```
private void doSomething() {
try {
    /* ... */
} catch(Exception e) {
    LOGGER.log("context", e);
}
```

Solução Proposta

é recomendada a utilização de Loggers para a definição de mensagens de erro.

Conformidade

OWASP Top 10 2017 Category A3 - Sensitive Data Exposure, MITRE, CWE-489 - Leftover Debug Code

Competência

Utilização de Tecnologia

4.4.5 Diretrizes Propostas

Principal objetivo deste trabalho, conforme apresentado na Seção 1.5, o apoio ao ensino de programação com qualidade a partir do fornecimento de diretrizes que possam ser utilizados como subsídio na condução do ensino de programação, além do aprimoramento do material didático da disciplina.

Conforme exposto na Seção 4.4.3, as diretrizes são geradas a partir das violações identificadas em trechos de código-fonte submetido pelos alunos à regras que foram analisadas e vinculadas com os conceitos relacionados com a disciplina em questão.

Uma violação a uma mesma regra analisada e vinculada com conceitos de outra disciplina, produziria diretrizes diferentes, de acordo com a disciplina em questão. As regras analisadas que atendem aos interesses deste projeto estão disponíveis no Anexo F.

Na Figura 4.15 é possível observar a interface da ferramenta Teacher Mate para seleção dos conjuntos de trabalho para análise.

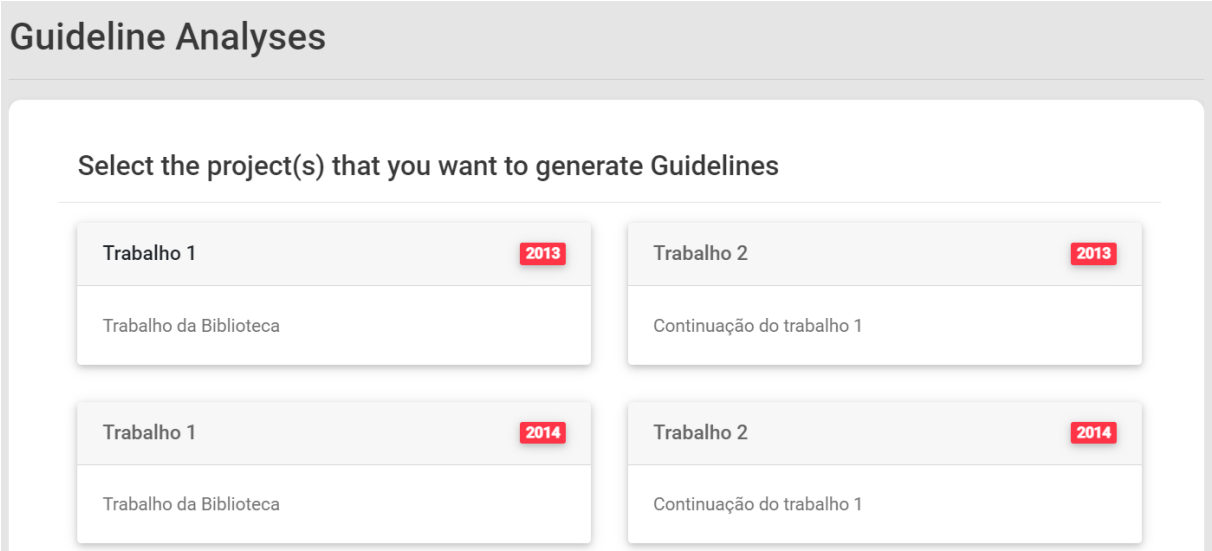


Figura 4.15: Interface da ferramenta Teacher Mate, desenvolvida para criação das Diretrizes – Destaque para as turmas de 2013 e 2014 (Produção Própria)

Conforme demonstrado na Figura 4.4, considerando o envio prévio do código-fonte por parte do aluno, assim como as demais etapas apresentadas neste capítulo já definidas.

Dado uma ou mais turmas selecionadas, considerando todos os trabalhos que fazem parte daquele conjunto. Para cada violação identificada em cada trecho de código, de cada um desses trabalhos, cada conceito relacionado que pertence a regra violada recebe incremento de um. Ao final da avaliação, a diretriz é fornecida ordenando-se os conceitos relacionados conforme incidência (número de violações identificadas).

Dentro do escopo do projeto, o processo de geração das diretrizes segue o

seguinte fluxo:

1. Professor Seleciona 1 ou mais grupos de projetos dos quais deseja realizar gerar as diretrizes;
2. Ferramenta TM identifica quais projetos fazem parte do agrupamento escolhido.
3. Ferramenta TM requisita quais foram as violações cometidas para a seleção à ferramenta SQ.
4. Ferramenta SQ retorna dados das violações cometidas, nos trechos de código inspecionados quando o aluno fez o envio do CF.
5. O Novo Módulo(NM) desenvolvido utiliza o repositório de violações gerado em tempo de execução e compara as violações com o repositório de regras mapeadas.
6. O NM realiza a soma das incidências dos conceitos relacionados e fornece ao professor a lista dos conceitos ordenadas pela maior incidência.

4.5 Exemplo de Diretriz Proposta

A seguir é apresentado um exemplo de diretriz proposta considerando as regras $R1$, $R5$ e $R6$, demonstradas na Seção 4.4.4. O conjunto de violações selecionadas refere-se a todos os trabalhos de todas as turmas.

Tabela 4.3: Quantidade de violações cometidas por regra (Produção Própria)

ID	Regra	Total	%
2	As saídas padrão não devem ser utilizadas para registrar nada	1158	65%
1	Recursos devem ser encerrados	546	31%
3	Throwable.printStackTrace(...) não deve ser chamado	74	04%

Tabela 4.4: Exemplo de Diretriz Proposta conforme violações apresentadas na Tabela 4.3 (Produção Própria)

ID	Conceito Relacionado	R1	R5	R6	Total	%
15	Entrada/Saída de Dados	546	1158		1704	96%
16	Loggers		1158	74	1232	69%
18	StackTrace		1158	74	1232	69%
01	Tratamento de Exceções	546		74	620	35%
06	Garbage Collector	546			546	31%
07	Fluxo de Execução	546			546	31%
10	Construtores e Destrutores	546			546	31%

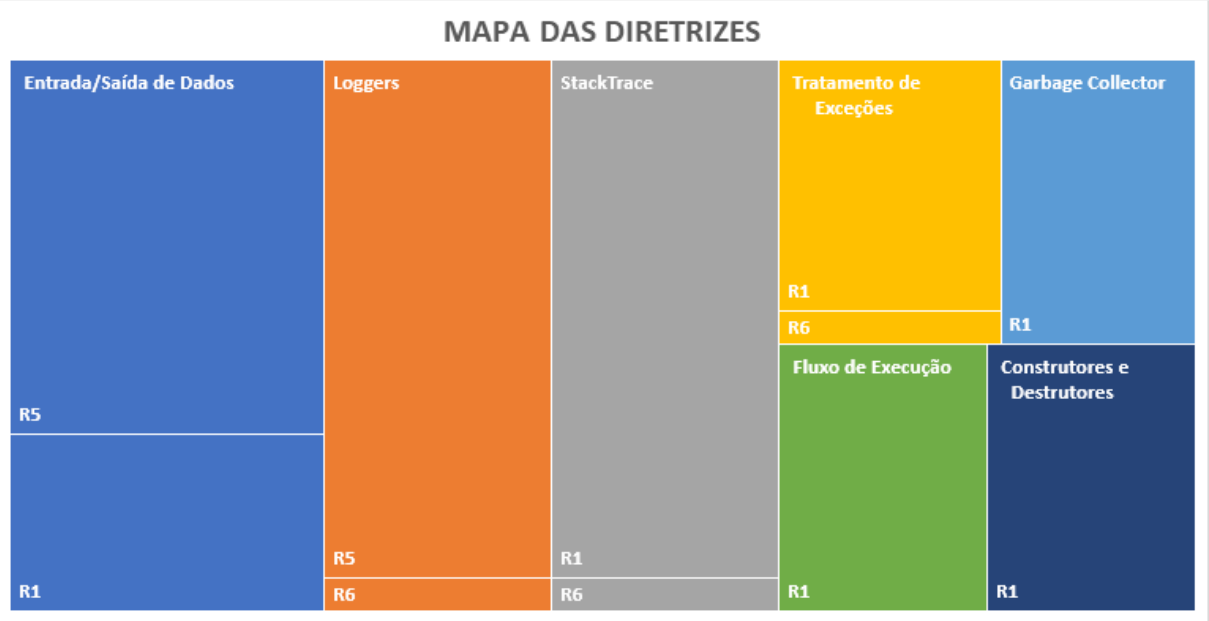


Figura 4.16: Mapa das Diretrizes propostas para as regras R1, R5 e R6, conforme exemplo na Seção 4.4.4 (Produção Própria)

4.6 Considerações Finais

A Engenharia de Software provê uma série de subsídios para apoiar processo de software. Utilizando ferramentas de análise contínua de código-fonte, o educador pode ter um panoramas geral ou específico da qualidade do código dos alunos sem que para isso despeda de uma grande quantidade de tempo realizando uma análise similar, de forma manual.

A ferramenta Teacher Mate, apresentada na Seção 4.2.2, foi desenvolvida com o intuito de prover ao professor esse panorama, utilizando de recursos fornecidos pela plataforma SonarQube, apresentada na Seção 4.2.1.

Por se tratar de uma ferramenta com desenvolvimento modular, e desenvolvida e mantida por participantes do grupo de pesquisa do autor, a adoção deste conjunto de ferramentas mostrou-se vantajosa tanto para o autor, quanto para o grupo de pesquisa.

Um dos pontos fracos da ferramenta TM, é que os resultados fornecidos pela mesma são considerados complexos, sendo necessário grande conhecimento e interpretação das regras por parte dos seus utilizadores.

Neste sentido, a proposta foi justamente trazer os resultados ao contexto da disciplina, diminuindo a carga de trabalho do professor, sendo o procedimento descrito em detalhes na Seção 4.4. Os resultados preliminares das analises se mostraram promissores.

A seguir, é conduzido um estudo piloto contendo estudos de caso da utilização da ferramenta na prática.

CAPÍTULO

5

Estudo Piloto

Para demonstrar a aplicação da metodologia, assim como a utilização da ferramenta, definiu-se pela realização de dois estudos de caso, analisando os dados das turmas de 2019 e 2018, além de uma pequena comparação entre eles. As análises foram validadas pelo professor especialista e responsável pela disciplina de P.O.O., de maneira qualitativa.

5.1 Estudo de Caso 1

Para o primeiro estudo, foram geradas as diretrizes para o Trabalho 1 (T1), produzido no primeiro trimestre e para o Trabalho 2 (T2), produzido no segundo trimestre, pela turma de 2019.

Na primeira análise, é possível observar na Tabela 5.1 que entre as regras com o maior número de violações, apenas uma esteve presente tanto na listagem das principais regras violadas do T1 quanto T2. Um aumento expressivo das violações à regra *"Private fields only used as local variables in methods should become local variables"* pode ser identificado do T1 para o T2.

Uma justificativa plausível para este aumento é a complexidade do T2 se comparada ao T1, sendo exigido um maior domínio dos conceitos de encapsulamento e escopo de variáveis. O estudante que não dominar bem esses conceitos, cometerá essa violação diversas vezes.

Embora a ferramenta SonarQube™ seja capaz de avaliar a complexidade do código-fonte, a maneira como a funcionalidade foi implementada impede essa comparação de maneira direta. Entretanto, na Figura 5.1 é possível observar o tamanho do T2 em relação ao T1, considerando a métrica *Lines of Code*, apresentada na Seção 2.5.2.

A regra *"Catches should be combined"* relacionada a utilização do comando **Try-Catch** sugere que o aluno começou a realizar o tratamento de exceções,

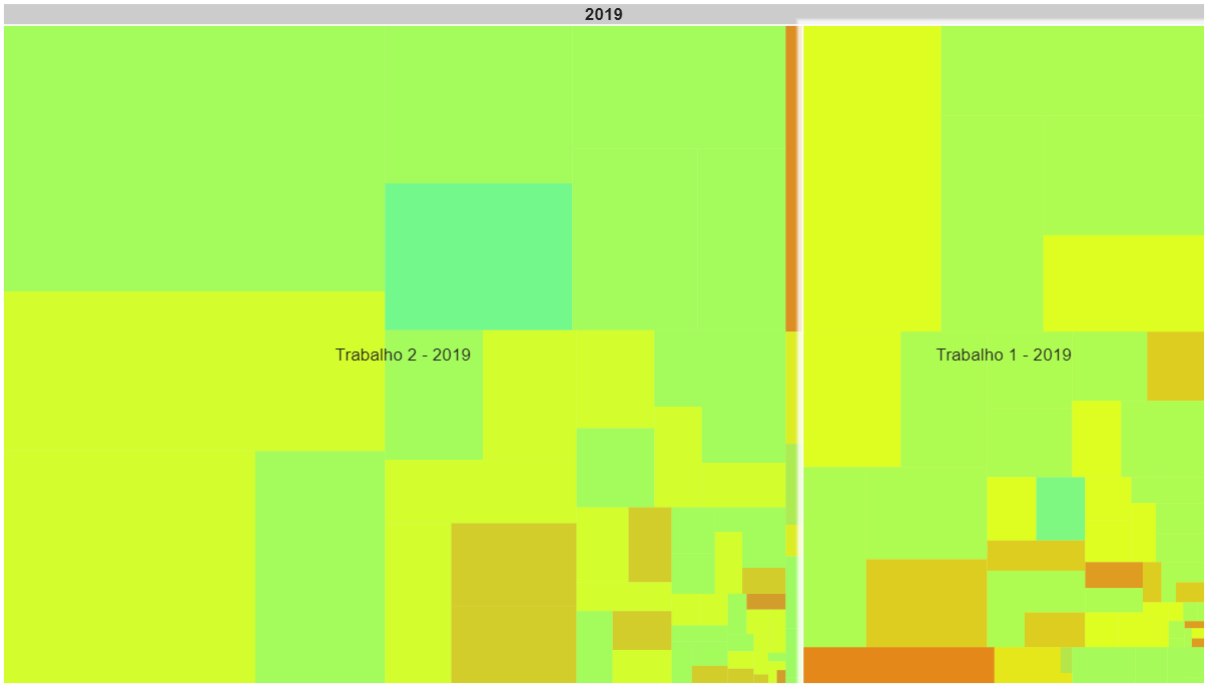


Figura 5.1: Diagrama hierárquico baseando-se na métrica *Lines of Code* (LOC) (Produção Própria)

Tabela 5.1: Regras mais Violadas - Trabalho 1 e 2 da Turma de 2019 (Produção Própria)

(a) Regras - Trabalho 1

#	Regra	%	Total
1	Standard outputs should not be used directly to log anything	15,91%	286
2	Strings should not be concatenated using '+' in a loop	8,95%	161
3	Private fields only used as local variables in methods should become local variables	8,29%	149
4	Multiple variables should not be declared on the same line	7,29%	131
5	Anonymous inner classes containing only one method should become lambdas	6,01%	108

(b) Regras - Trabalho 2

#	Regra	%	Total
1	Private fields only used as local variables in methods should become local variables	19,22%	678
2	Anonymous inner classes containing only one method should become lambdas	11,52%	412
3	Unused method parameters should be removed	11,13%	398
4	Catches should be combined	5,73%	205
5	Array designators "[]"should be on the type, not the variable	5,54%	198

mas ainda não de uma maneira adequada. Um outro destaque é a diminuição na ocorrência da primeira regra da Tabela 5.1, remetem a adoção de alternativas para o uso indiscriminado do comando **System.out.print()** com o intuito de depurar o código em execução.

Tabela 5.2: Diretrizes - Trabalho 1 e 2 da Turma de 2019 (Produção Própria)

(a) Diretrizes - Trabalho 1

#	D	CR	%
1	D18	StackTrace	7,63%
2	D16	Loggers	7,20%
3	D13	Atributos e Variáveis	6,64%
4	D15	Entrada/Saída	6,56%
5	D12	Atribuição	6,39%

(b) Diretrizes - Trabalho 2

#	D	CR	%
1	D13	Atributos e Variáveis	11,94%
2	D3	Classes	11,65%
3	D20	Bloco de Instrução	8,81%
4	D14	Escopo	7,68%
5	D6	Garbage Collection	7,16%

Podemos confirmar na Tabela 5.2 que de fato ocorreram alterações nas diretrizes conforme o aluno adquiriu experiência ao longo do curso.

Para exemplificar como as diretrizes são formadas, ao clicar em um Conceito Relacionado, a ferramenta lista cada uma das regras que foram violadas e a quantidade de violações por regra. Para o exemplo do Conceito Relacionado *StackTrace*, o primeiro da Tabela 5.2a, foram somadas as violações de 4 diferentes regras, incluindo a regra *"Standard outputs should not be used directly to log anything"*, que conforme pode ser observado na Tabela 5.1a, foi a regra com maior incidência, contribuindo para a posição do conceito na diretriz gerada.

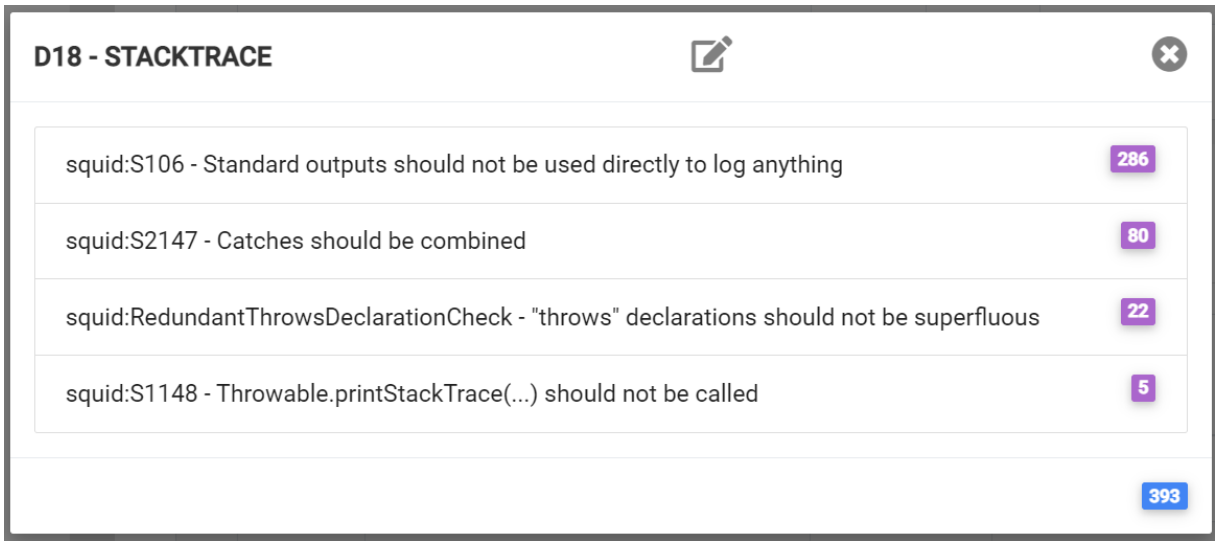
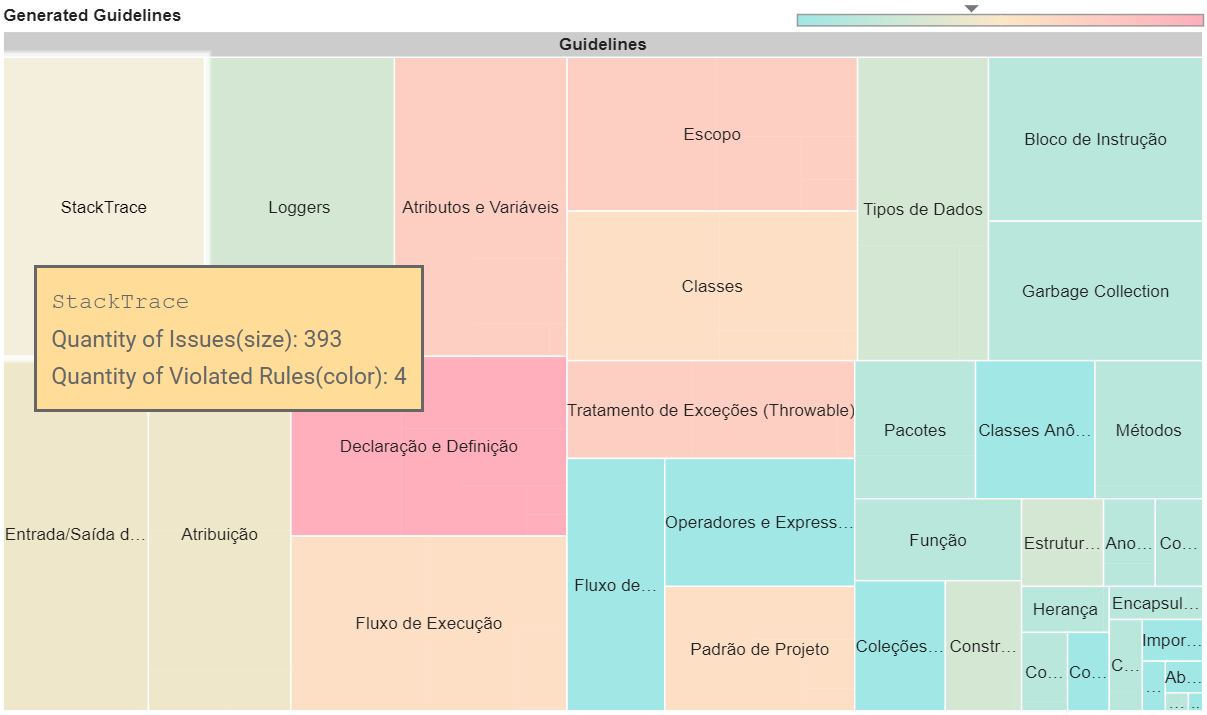
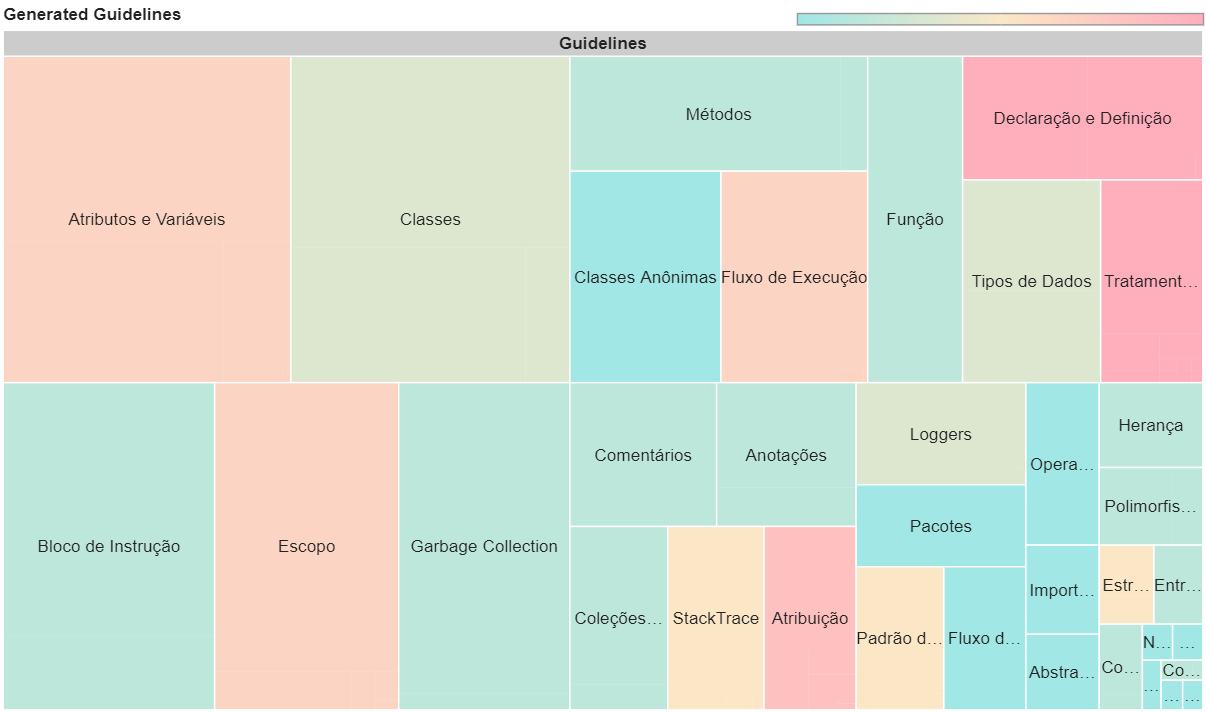


Figura 5.2: Conceitos Relacionados com a Disciplina de POO (Produção Própria)

Os conceitos relacionados a correta utilização da tecnologia (programação em Java), tiveram menos ocorrência do T1 para o T2, indicando que os alunos adquiriram habilidades na utilização da mesma.



(a) Mapa das Diretrizes baseadas no Trabalho 1



(b) Mapa das Diretrizes baseadas no Trabalho 2

Figura 5.3: Mapa das Diretrizes Propostas para o primeiro e segundo trimestre da turma de 2019 (Produção Própria)

5.2 Estudo de Caso 2

Em uma segunda análise, comparamos a submissão dos trabalhos 1 e 2 da turma de 2018. Para conseguir demonstrar mais informações as próximas diretrizes serão geradas e os dados tabelados de acordo com o resultado obtido.

Na Tabela 5.3, são listadas as regras com maior número de violações para o T1 e T2 da turma de 2018. Assim como na turma de 2019, houve uma considerável diminuição nas violações relacionadas com a regra *"Standard outputs should not be used directly to log anything"*, que conforme indicado no estudo anterior, demonstra que o aluno diminuiu a utilização da saída para o console.

Tabela 5.3: Regras mais Violadas - Trabalho 1 e 2 da Turma de 2018 (Produção Própria)

(a) Regras - Trabalho 1

#	Regra	%	Total
1	Private fields only used as local variables in methods should become local variables	14,47%	351
2	Standard outputs should not be used directly to log anything	13,11%	318
3	Anonymous inner classes containing only one method should become lambdas	9,57%	232
4	Unused method parameters should be removed	8,37%	203
5	Catches should be combined	6,02%	146

(b) Regras - Trabalho 2

#	Regra	%	Total
1	Private fields only used as local variables in methods should become local variables	19,66%	915
2	Anonymous inner classes containing only one method should become lambdas	11,13%	518
3	Unused method parameters should be removed	10,50%	489
4	Field names should comply with a naming convention	5,82%	271
5	Package names should comply with a naming convention	4,94%	229

Tabela 5.4: Diretrizes - Trabalho 1 e 2 da Turma de 2018 (Produção Própria)

(a) Diretrizes - Trabalho 1

#	D	CR	%
1	D13	Atributos e Variáveis	9,26%
2	D14	Classes	9,15%
3	D21	StackTrace	7,66%
4	D7	Escopo	7,42%
5	D12	Bloco de Instrução	7,23%

(b) Diretrizes - Trabalho 2

#	D	CR	%
1	D13	Atributos e Variáveis	13,99%
2	D3	Classes	11,94%
3	D20	Bloco de Instrução	9,34%
4	D14	Escopo	8,25%
5	D6	Garbage Collection	7,34%

5.3 Considerações Finais

Durante a realização dos estudos, foi possível perceber que a análise baseada nas diretrizes proporcionou um ganho considerável de tempo ao professor no que diz respeito ao entendimento das dificuldades enfrentadas pelos alunos.

Tal percepção pode ser comprovada após relato do docente especialista na disciplina. Segundo o mesmo "A análise com base nas diretrizes proporciona um ganho considerável de tempo, além de um panorama mais focado na realidade dos alunos e da disciplina."

Ao comparar as tabelas de regras com as tabelas de diretrizes, fica evidente que tomar decisões que guiem o ensino baseadas nas diretrizes fornecidas tendem a ser vantajosas. A elucidação de um conceito relacionado tende a fazer com que o aluno diminua as violações cometidas a diversas regras, o que é considerado um código de qualidade.

A comparação entre turmas permitiu observar que seus alunos entram com um perfil variado, mas no final da disciplina cometem os mesmos tipos de violações, ou seja, há uma evolução dos aluno convergindo para um mesmo patamar de conhecimento. Isso é um indicativo de que a maneira de conduzir a disciplina consegue equiparar alunos com diferentes habilidades iniciais.

CAPÍTULO

6

Conclusão

As constantes mudanças no mercado de tecnologia da informação fazem com que empresas que demandem de profissionais da área, busquem profissionais cada vez mais capacitados. Pesquisas mostram que o recém-formado nem sempre está preparado para adentrar ao mercado de trabalho, encontrando dificuldades para se estabelecer no mercado.

Ao analisar as competências requeridas, fica evidente a busca das empresas por profissionais que saibam produzir código de qualidade. Para a indústria, um programador é considerado experiente se possuir domínio da escrita de código com qualidade, de maneira a contribuir para o desenvolvimento do projeto.

Durante a graduação, o aluno é exposto a diversos conceitos de qualidade, bem como aplicações desses conceitos, com objetivo de torná-lo capaz de desenvolver código com qualidade. Logo quando ingressa na universidade, são demonstrados ao aluno os primeiros conceitos de programação. O aluno aprende a programar assim que inicia o curso, mas só nos anos posteriores são demonstrados os conceitos relacionados com a qualidade. Na prática, o aluno aprende a desenvolver soluções que para determinada entrada, produzam uma saída esperada. São considerados apenas aspectos relacionados ao código em execução, sendo desconsiderados aspectos referentes a qualidade interna do código produzido.

As ementas dos cursos se baseiam nos *Currículos de Referência* para os cursos de Computação no Brasil. Estes, por sua vez, sumarizam as *Diretrizes Curriculares Nacionais* – um conjunto de orientações que definem e delimitam a área de atuação dos cursos de computação. As diretrizes definem o que é esperado do egresso, sem detalhar quais conteúdos e disciplinas devem ser ofertados ao aluno durante a sua formação. Ficam definidos nos currículos,

em caráter de recomendações, os conteúdos considerados importantes para a formação do profissional, em conformidade com essas diretrizes.

É importante destacar que em nenhum dos currículos analisados neste estudo, estão presentes recomendações para que o ensino de programação seja conduzido concomitantemente com o ensino de qualidade interna de código. Não sendo também identificadas recomendações contrárias a essa abordagem.

Ao analisar as matrizes curriculares e ementas das disciplinas de diversos cursos de Bacharelado em Ciência da Computação das principais instituições de ensino superior do país, são identificas evidências do emprego das recomendações propostas nos currículos de referências em todas as instituições analisadas. A similaridade entre as matrizes curriculares e ementas de todas as instituições analisadas, apontam para a adoção dos currículos de referência na definição do projeto pedagógico de ensino das mesmas.

Não estando prevista nos currículos de referência, também não são identificadas nas instituições propostas de ensino de programação associadas ao ensino de qualidade. Nesse sentido, supõem-se que em instituições que se tem algum indício desse foco em qualidade interna, seja pelo docente responsável, ou alguma iniciativa participar de um grupo de ensino.

Na academia o aluno é incentivado a programar para cumprir os requisitos obrigatórios das disciplinas. Após essas disciplinas, as soluções costumam ser descartadas, fazendo com que o aluno seja incentivado a optar sempre pela maneira mais fácil de resolver o problema e não a mais correta. O aluno aprende a programar sem conhecer padrões e convenções voltados a qualidade, produzindo códigos de baixa qualidade interna e sem entender os problemas inerentes ao código de baixa qualidade.

Este projeto define uma proposta para prover melhorias no processo de ensino de programação, fornecendo diretrizes ao professor para conduzir o ensino de programação baseado nas faltas cometidas pelos alunos. Essa abordagem torna possível direcionar o conteúdo ministrado pelo professor às dificuldades encontradas pelos alunos.

Quanto mais cedo o aluno compreender os conceitos de qualidade, mais tempo ele terá para desenvolver essas competências tão importante para a industria, criando o hábito de desenvolver corretamente e dominando o entendimento dos conceitos inerentes ao processo de software.

6.1 Consolidação da Proposta

A proposta inicial parte de uma necessidade identificada pelo próprio autor que, junto com seu orientador, em um estudo anterior (de Andrade Gomes et al., 2017), identificam a dificuldade de alunos em avaliar a qualidade interna do código-fonte daquilo que é produzindo. Tendo o conhecimento empírico

das dificuldades relatadas pelos colegas em ingressar no mercado de trabalho, optou-se por identificar se e como as situações se relacionam, afim de propor melhorias para o processo de ensino-aprendizagem, beneficiando professores e alunos.

Para a implementação da proposta, optou-se pela utilização de ferramentas que possuem a sua empregabilidade validada em ambiente acadêmico, como a ferramenta SonarQube™ e a ferramenta *Teacher Mate*, sendo esta última idealizada, desenvolvida e mantida por participantes do grupo de pesquisa dos autores, o que permitiu o desenvolvimento de um novo módulo estendendo as funcionalidades da mesma e adequando-a a proposta deste trabalho.

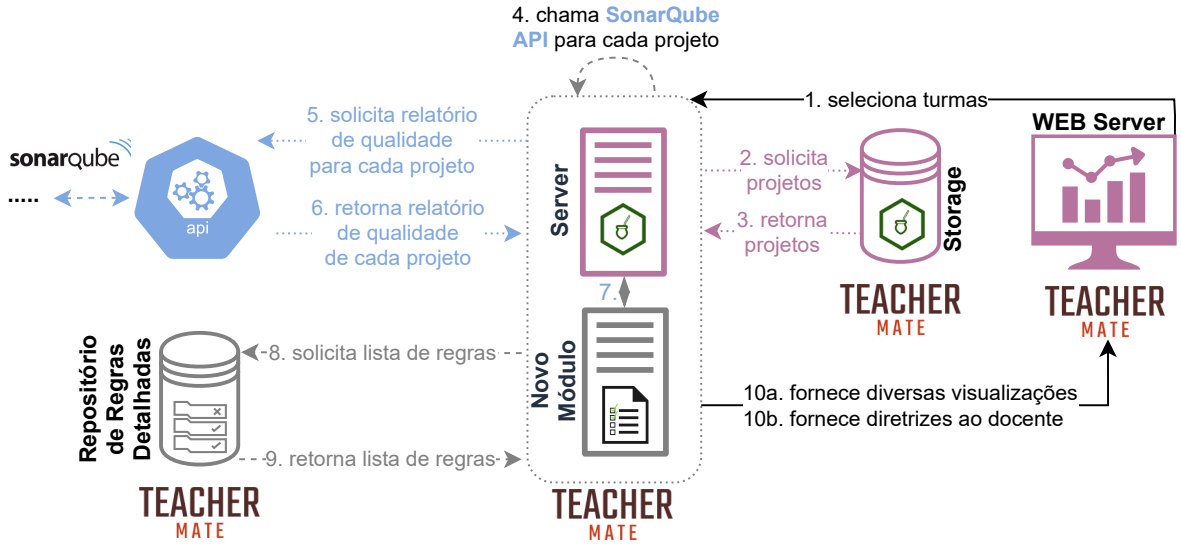


Figura 6.1: Arquitetura da Ferramenta após Evolução (Produção Própria)

A Figura 6.1 descreve a arquitetura da solução implementada, a partir da arquitetura descrita na Seção 4.2.2 e demonstrada na Figura 4.2. Sendo os passos de 1 à 6 explicados na seção citada, com a nova implementação, os passos seguintes podem ser descritos da seguinte maneira:

7. O Servidor Teacher Mate passa a trocar informações com o Novo Módulo (TM-NM), sendo o repositório local compartilhado entre os módulos.
8. O TM-NM solicita ao Repositório de Regras Detalhas a lista de regras detalhadas de acordo com as violações identificadas.
9. O repositório fornece a lista de regras detalhas para que a computação possa ser feita no TM-NM e as diretrizes geradas.
10. Fornecimento dos resultados:
    (a) O Servidor TM fornece as diversas visualizações, conforme demonstrado na seção 3.5.3.2.
    (b) O TM-NM fornece as diretrizes ao docente conforme explicado no Capítulo 4.

6.2 Aspectos a Ressaltar

A seguir são destacados os pontos fortes e fragilidades deste projeto. Esses aspectos são baseados na visão dos autores e revisores do artigo aprovado.

Como pontos fortes, pode ser destacado os seguintes pontos:

- Um ponto de destaque deste projeto é a quantidade de projetos avaliados. Um quantitativo de 180 projetos e mais de 60 mil trechos de código violados possibilita diversos tipos de análises, sendo possível a utilização deste repositório para outros estudos.
- Outro ponto é a adoção de ferramentas já consolidadas e validadas no escopo do ensino, facilitando a adoção das mesmas no projeto. A adoção de revisão aos pares nos artefatos produzidos possibilita o aumento da confiabilidade do que foi produzido.
- Outro ponto forte é a disponibilização de todos os artefatos em repositórios abertos, permitindo não só a adoção, como também a colaboração por parte da comunidade.

Como fragilidades, podem ser destacados os seguintes pontos:

- A primeira fragilidade deste trabalho é a falta de validação por parte dos alunos. Com a situação atual, a condução de um estudo piloto, em um ambiente controlado, que envolvesse os alunos, excederia o cronograma proposto. O autor não previu o cenário atual em seu cronograma.
- Este mesmo cenário, impactou no curso da disciplina para o ano de 2020, sendo as atividades produzidas pelos alunos adaptadas ao novo contexto, onde tornou-se impossível a submissão ao repositório para efeitos de comparação com os demais anos. Uma nova adequação transcorre para o presente ano, sendo a proposta de trabalho para 2021, diferente da do ano anterior e totalmente diferente do conjunto de dados utilizados.
- Um terceiro ponto fraco é a validação em pares realizada nas seções 4.4.1 e 4.4.3, as quais a validação em pares ocorreu entre o próprio autor e o único docente especialista da disciplina no Campus.

6.3 Contribuições e Estudos Futuros

São contribuições do projeto – a definição de uma abordagem para a utilização do código-fonte como subsídio para o ensino de qualidade, a implementação de uma ferramenta que apoia a abordagem definida, fornecendo ao professor um conjunto de diretrizes para apoiar o processo de ensino-aprendizagem.

Na perspectiva do aluno, a análise das violações permite fornecer ao aluno informações sobre a evolução das suas habilidades como programador.

O Repositório de Faltas Cometidas e o Repositório de Diretrizes desenvolvidos para apoiar a metodologia podem ser utilizados em conjunto ou separados como bases de conhecimento, sendo possível a utilização dos mesmos para fins de pesquisa, de estudo e de apoio ferramental.

Todos os artefatos produzidos para apoiar a abordagem proposta são de código aberto, sendo submetidos para um repositório de código-fonte, permitindo a adoção do projeto pela comunidade acadêmica.

A proposta de uma intervenção em sala de aula, permite que os alunos participantes possam se beneficiar da pesquisa, conseguindo desenvolver suas competências e consequentemente, aprimorar a qualidade interna de seus códigos.

Outra contribuição obtida é a publicação deste trabalho em uma das principais conferências da área da computação com a temática voltada para a Educação (*IEEE Frontiers in Education Conference*).

Referências

ACKERMAN, A. F.; BUCHWALD, L. S.; LEWSKI, F. H. Software inspections: an effective verification process. *IEEE software*, v. 6, n. 3, p. 31–36, 1989.

AL-OBTHANI, F.; AMEEN, A. Towards customized smart government quality model. *International Journal of Software Engineering & Applications*, v. 9, n. 2, p. 41–50, 2018.

ALLAMANIS, M.; BARR, E. T.; BIRD, C.; SUTTON, C. Learning natural coding conventions. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, p. 281–293.

DE ANDRADE GOMES, P. H.; GARCIA, R. E.; SPADON, G.; ELER, D. M.; OLIVETE, C.; CORREIA, R. C. M. Teaching software quality via source code inspection tool. In: *2017 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2017, p. 1–8.

ANDRY, J. F.; SUROSO, J.; BERNANDA, D. Improving quality of smes information system solution with iso 9126. *Journal of Theoretical and Applied Information Technology*, v. 96, n. 14, p. 4610–4620, 2018.

DE ARAÚJO, D. M. N.; ELER, D. M.; GARCIA, R. E. Teacher mate: A support tool for teaching code quality. In: *17th International Conference on Information Technology–New Generations (ITNG 2020)*, Springer, 2020, p. 407–413.

BAGGEN, R.; CORREIA, J. P.; SCHILL, K.; VISSER, J. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, v. 20, n. 2, p. 287–307, 2012.

BAILEY, J.; MITCHELL, R. B. Industry perceptions of the competencies needed by computer programmers: technical, business, and soft skills. *Journal of Computer Information Systems*, v. 47, n. 2, p. 28–33, 2006.

BAILEY, J. L.; STEFANIAK, G. Industry perceptions of the knowledge, skills, and abilities needed by computer programmers. In: *Proceedings of the 2001 ACM SIGCPR conference on Computer personnel research*, ACM, 2001, p. 93–99.

BANKER, R. D.; DAVIS, G. B.; SLAUGHTER, S. A. Software development practices, software complexity, and software maintenance performance: A field study. *Management science*, v. 44, n. 4, p. 433–450, 1998.

BAXTER, I. D.; PIDGEON, C.; MEHLICH, M. Dms®: Program transformations for practical scalable software evolution. In: *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, 2004, p. 625–634.

BLOCH, J. *Effective java*. Addison-Wesley Professional, 2017.

BOOCH, G.; GRADY *Object-oriented analysis and design with applications*. Benjamin/Cummings Pub. Co, 589 p., 1994.

BOURQUE, P.; FAIRLEY, R. E.; ET AL. *Guide to the software engineering body of knowledge (swebok (r)): Version 3.0*. IEEE Computer Society Press, 2014.

BOWYER, J.; HUGHES, J. Assessing undergraduate experience of continuous integration and test-driven development. In: *Proceedings of the 28th international conference on Software engineering*, ACM, 2006, p. 691–694.

ASSOCIATION FOR COMPUTING MACHINERY, J. T. F. O. C. C.; SOCIETY, I. C. *Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science*. New York, NY, USA: ACM, 999133, 2013.

ASSOCIATION FOR COMPUTING MACHINERY, J. T. F. O. C. C.; SOCIETY, I. C. *Acm curricula recommendations for computer science 2020 – paradigms for global computing education*. New York, NY, USA: Association for Computing Machinery, 2020.
Disponível em https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf

CROSBY, P. B. *Quality is free : the art of making quality certain*. New American Library, 1980.

DEHAGHANI, S. M. H.; HAJRAHIMI, N. Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, v. 21, n. 1, p. 63, 2013.

DEITEL, H. *Java: Como programar*. Pearson Education, 2016.

DIETZ, L. W.; MANNER, J.; HARRER, S.; LENHARD, J. Teaching clean code. In: *2018 Combined Workshops of the German Software Engineering Conference, SE-WS 2018, 6 March 2018*, CEUR-WS, 2018, p. 24–27.

DJOUAB, R.; BARI, M. An iso 9126 based quality model for the e-learning systems. *International journal of information and education technology*, v. 6, n. 5, p. 370, 2016.

DUVALL, P. M. *Continuous Integration*. Pearson Education India, 2007.

EDER, S.; JUNKER, M.; JÜRGENS, E.; HAUPTMANN, B.; VAAS, R.; PROMMER, K. How much does unused code matter for maintenance? In: *2012 34th International Conference on Software Engineering (ICSE)*, 2012, p. 1102–1111.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, v. 15, n. 2.3, p. 182–211, 1986.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

FOWLER, M.; FOEMMEL, M. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, v. 122, p. 14, 2006.

FOWLER, M. Is High Quality Software Worth the Cost? 2019. Disponível em https://martinfowler.com/articles/is-quality-worth-cost.html

GARCIA, R. E.; CORREIA, R. C. M.; OLIVETE, C.; BRANDI, A. C.; PRATES, J. M. Teaching and learning software project management: A hands-on approach. In: *Frontiers in Education Conference (FIE), 2015 IEEE*, IEEE, 2015, p. 1–7.

GAROUSI, V. Applying peer reviews in software engineering education: An experiment and lessons learned. *IEEE Transactions on Education*, v. 53, n. 2, p. 182–193, 2010.

GILB, T.; GRAHAM, D.; FINZI, S. *Software inspection*. Addison-Wesley Longman Publishing Co., Inc., 1993.

GLASS, R. L. Frequently forgotten fundamental facts about software engineering. *IEEE software*, v. 18, n. 3, p. 112–111, 2001.

HERVÉ, B. Software Qualimetry at Schneider Electric: a field background. 2011.

HUNT, A.; THOMAS, D.; CUNNINGHAM, W. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

HUO, M.; VERNER, J.; ZHU, L.; BABAR, M. A. Software Quality and Agile Methods. *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, p. 520–525, 2004.

IEEE Ieee standard for software reviews and audits. 2008.

ISO/IEC-25010 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. 2011.
Disponível em https://www.iso.org/standard/35733.html

ISO/IEC-9126-1 ISO/IEC 9126: Qualidade de produto-Parte 1: Modelo de qualidade. *Rio de Janeiro: ABNT*, p. 1–21, 2003.
Disponível em https://www.iso.org/standard/22749.html

ISO/IEC-9126-3 ISO/IEC 9126: Qualidade de produto-Parte 3: Métricas internas. *Rio de Janeiro: ABNT*, p. 1–21, 2003.
Disponível em https://www.iso.org/standard/22749.html

KNUTH, D. E. *The art of computer programming: sorting and searching*, v. 3. Pearson Education, 1997.

KOLLER, H. G. *Effects of clean code on understandability: An experiment and analysis*. Dissertação de Mestrado, University of Oslo, 2016.

KRUSCHE, S.; SEITZ, A. Artemis: An automatic assessment management system for interactive learning. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ACM, 2018, p. 284–289.

LAITENBERGER, O.; DEBAUD, J.-M. An encompassing life cycle centric survey of software inspection. *Journal of systems and software*, v. 50, n. 1, p. 5–31, 2000.

LETOUZEY, J.-L. The SQALE Method Definition Document. 2012a.
Disponível em http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf

LETOUZEY, J.-L. The SQALE Method for Evaluating Technical Debt. 2012b.

LEWIS, W. E. *Software testing and continuous quality improvement*. Auerbach publications, 2004.

LEWIS, W. E. *Software testing and continuous quality improvement*. CRC press, 2016.

LEWIS, W. E.; DOBBS, D.; VEERAPILLAI, G. *Software testing and continuous quality improvement*. CRC Press, 655 p., 2009.

LUFTMAN, J.; LYYTINEN, K.; BEN ZVI, T. Enhancing the measurement of information technology (it) business alignment and its influence on company performance. *Journal of Information Technology*, v. 32, n. 1, p. 26–46, 2017.

MARI, M.; ET AL. The impact of maintainability on component-based software systems. In: *null*, IEEE, 2003, p. 25.

MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

MATTURRO, G. Soft skills in software engineering: A study of its demand by software companies in uruguay. In: *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, IEEE, 2013, p. 133–136.

MCCONNELL, S. *Code complete*. Pearson Education, 2004.

MELVILLE, N.; KRAEMER, K.; GURBAXANI, V. Review: Information technology and organizational performance: An integrative model of it business value. *MIS Q.*, v. 28, n. 2, p. 283–322, 2004.

MILLER, A. A hundred days of continuous integration. *Proceedings - Agile 2008 Conference*, p. 289–293, 2008.

MYERS, G. J.; BADGETT, T.; THOMAS, T. M.; SANDLER, C. The Art of Software Testing, Second Edition. 2004.

NBR/ISO-12207 12207: 2017 – systems and software engineering – software life cycle processes. 2017.

NBR/ISO-9000 9000: 2015–Sistemas de gestão da qualidade–Fundamentos e vocabulário. 2015.

PARNAS, D. L.; LAWFORD, M. The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering*, v. 29, n. 8, p. 674–676, 2003.

PÉREZ, Y. F.; CORONA, C. C.; VERDEGAY, J. L. Modelling the interrelation among software quality criteria using computational intelligence techniques. *International Journal of Computational Intelligence Systems*, v. 11, n. 1, p. 1170–1178, 2018.

PRAUSE, C. R.; AUGUSTIN, S. An Approach for Continuous Inspection of Source Code. In: *WoSQ'08*, New York, New York, USA: ACM Press, 2008, p. 17–22.

PRESSMAN, R.; MAXIM, B. *Engenharia de software-8ª edição*. McGraw Hill Brasil, 2016.

ROCHA, M. D. G. B.; NICOLETTI, M. D. C.; FABBRI, S. C. P. F.; BARROS, E.; FRERY, A. C. *Currículo de Referência da SBC para Cursos de Graduação em Bacharelado em Ciência da Computação e Engenharia de Computação.* São Paulo: Sociedade Brasileira de Computação (SBC), 16 p., 2005.

ROCHIMAH, S.; RAHMANI, H. I.; YUHANA, U. L. Usability characteristic evaluation on administration module of academic information system using iso/iec 9126 quality model. In: *2015 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, IEEE, 2015, p. 363–368.

SACRISTÁN, J. G.; GÓMEZ, Á. I. P.; RODRÍGUEZ, J. B. M.; SANTOMÉ, J. T.; RASCO, F. A.; MÉNDEZ, J. M. Á. *Educar por competências: O que há de novo?* Artmed Editora, 2016.

SALLEH, N.; MENDES, E.; GRUNDY, J. Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review. *IEEE Transactions on Software Engineering*, v. 37, n. 4, p. 509–525, 2011.

SAMI, M.; MALEK, M.; BONDI, U.; REGAZZONI, F. Embedded systems education: job market expectations. *ACM SIGBED Review*, v. 14, n. 1, p. 22–28, 2017.

SCALLON, G. *Avaliação da aprendizagem numa abordagem por competências*. PUCPRess, 2017.

SCATALON, L. P.; CARVER, J. C.; GARCIA, R. E.; BARBOSA, E. F. Software testing in introductory programming courses: A systematic mapping study. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, New York, NY, USA: ACM, 2019, p. 421–427 (*SIGCSE '19*, v.1).

SIMUNIC, T.; BENINI, L.; DE MICHELI, G.; HANS, M. Source code optimization and profiling of energy consumption in embedded systems. In: *Proceedings 13th International Symposium on System Synthesis*, 2000, p. 193–198.

SMIT, M.; GERGEL, B.; HOOVER, H. J.; STROULIA, E. Maintainability and source code conventions: An analysis of open source projects. *University of Alberta, Department of Computing Science, Tech. Rep. TR11*, v. 6, 2011.

SONARSOURCE S.A. Architecture and Integration - SonarQube Documentation - SonarQube. 2019.
Disponível em http://docs.sonarqube.org/display/SONAR/Architecture+and+Integration

SQUORE Techinal Debt - SQUORE. 2016.

SULLIVAN, K. J.; GRISWOLD, W. G.; CAI, Y.; HALLEN, B. The structure and value of modularity in software design. In: *ACM SIGSOFT Software Engineering Notes*, ACM, 2001, p. 99–108.

SUWAWI, D. D. J.; DARWIYANTO, E.; ROCHMANI, M. Evaluation of academic website using iso/iec 9126. In: *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, IEEE, 2015, p. 222–227.

TONELLA, P.; LEMMA ABEBE, S. Code quality from the programmer's perspective. *PoS*, p. 001, 2008.

ZORZO, A. F.; NUNES, D.; MATOS, E. S.; STEINMACHER, I.; LEITE, J. C.; ARAUJO, R.; CORREIA, R. C. M.; MARTINS, S. *Referenciais de Formação para os Cursos de Graduação em Computação 2017*. São Paulo: Sociedade Brasileira de Computação (SBC), 153 p., 2017.

Apêndice

APÊNDICE

A

Disciplinas presentes no Currículo de Referência de 2005

A seguir, são listadas as matérias relacionadas aos cursos de Computação, presentes no Currículo de Referência de 2005 (Rocha et al., 2005).

- **Fundamentos da Computação**: compreende o núcleo de matérias que envolvem a parte científica e as técnicas fundamentais à formação sólida dos egressos dos diversos cursos de computação;

  F1. Análise de Algoritmos

  F2. Algoritmos e Estrutura de Dados

  F3. Arquitetura e Organização de Computadores

  F4. Circuitos Digitais

  F5. Fundamentos de Sistemas

  F6. Linguagens de Programação

  F7. Linguagens Formais, Autômatos e Computabilidade

  F8. Organização de Arquivos e dados

  F9. Sistemas Operacionais

  F10. Teoria dos Grafos

- **Tecnologia da Computação**: compreende o núcleo de matérias que representam um conjunto de conhecimento agregado e consolidado que capacitam o aluno para a elaboração de solução de problemas nos diversos domínios de aplicação.

  T1. Análise de Desempenho

T2. Bancos de Dados

T3. Circuitos Integrados

T4. Compiladores

T5. Computação Gráfica

T6. Automação e Controle

T7. Engenharia de Software

T8. Inteligência Artificial

T9. Interação Humano-Computador

T10. Matemática Computacional

T11. Métodos Formais

T12. Modelagem e Simulação

T13. Processamento Digital de Sinais

T14. Processamento de Imagens

T15. Programação Paralela

T16. Redes de Computadores

T17. Segurança e Auditoria de Sistemas

T18. Sistemas Digitais

T19. Sistemas Distribuídos

T20. Sistemas Embarcados

T21. Sistemas Multimídia

T22. Tolerância a Falhas

T23. Telecomunicações

APÊNDICE

B

Conteúdo Programático da Disciplina de POO

1. **Fundamentos da Programação Orientada a Objetos**
   → Aspectos históricos das linguagens orientadas a objetos; principais conceitos (classes e objetos, encapsulamento, herança, polimorfismo) relacionados ao paradigma de orientação a objetos e sua implementação através das linguagens de programação;

2. **Conceitos Básicos de Linguagem OO**
   → Variáveis; tipos, operadores e expressões; conversões padrão; estruturas de controle; entrada/saída de dados; declarações e definições; funções; tratamento de exceções;

3. **Classes e Objetos**
   → Encapsulamento; classe, objeto; atributos e comportamentos; construtores e destrutores; sobrecarga;

4. **Herança e Polimorfismo**
   → Classes derivadas, classes abstratas; herança simples; herança múltipla, e polimorfismo; sobreposição;

5. **Controle de acesso**
   → Especificadores de acesso: público, privado e protegido;

6. **Gabaritos**
   → Implementação usando gabaritos;

7. **Biblioteca de Classes e Coleções**
   → Introdução; contêineres; iteradores; classes e algoritmos

APÊNDICE C

Lista de Itens de Conformidade com as Regras Mapeadas

Tabela C.1: Relação entre as regras e conformidades

Conformidade	**Regra**
RSPEC-2055	A superclasse não serializável de uma classe *Serializable* deve ter um construtor sem argumento
RSPEC-1161	Anotação @Override deve ser utilizada para sobrescrever e implementar métodos
MITRE - CWE-478	As instruções *switch* devem terminar com cláusulas *default*
CERT - ERR02-J	As saídas padrão não devem ser usadas diretamente para registrar nada
RSPEC-4165	Atribuições não devem ser redundantes
MITRE - CWE-493	Atributos da Classe não devem ter acessibilidade pública
MITRE - CWE-607	Atributos mutáveis não devem ser *public static*
RSPEC-1450	Atributos privados usados apenas como variáveis locais nos métodos devem se tornar variáveis locais
RSPEC-2147	*Catches* precisam ser combinados
RSPEC-1604	Classes internas anônimas contendo apenas um método devem se tornar *lambdas*
RSPEC-2118	Classes não serializáveis não devem ser persistidas
MITRE - CWE-500	Contextos *public static* devem ser constantes

RSPEC-1905	Conversões redundantes não devem ser utilizadas
RSPEC-1192	Código-fonte devem possuir uma quantidade suficiente de linhas comentadas
RSPEC-1319	Declarações devem utilizar as Interfaces das coleções como *List* ao invés da implementação *LinkedList*
RSPEC-1130	Exceções *lançadas* não devem ser supérfluas
RSPEC-1128	Importações não utilizadas devem ser removidas
RSPEC-1125	Literais booleanos não devem ser redundantes
RSPEC-2696	Métodos instanciados não devem escrever em contextos estáticos
DCL52-J	Múltiplas variáveis não devem ser declaradas na mesma linha
Oracle	Nome dos métodos devem estar de acordo com convenções de nomes
RSPEC-120	Nome dos pacotes devem ser nomeados de acordo com as convenções de nomenclatura
RSPEC-117	Nomes de variáveis devem obedecer a uma convenção de nomenclatura
RSPEC-1220	O pacote sem nome padrão não deve ser usado
RSPEC-1197	Os designadores de arranjos [ ] devem estar na declaração do tipo e não da variável
RSPEC-117	Os nomes de variáveis locais e de parâmetros de métodos devem obedecer a uma convenção de nomenclatura
MITRE - CWE-754	Os valores de retorno não devem ser ignorados quando contêm o código de status da operação
CERT - MSC12-C	Parâmetros não utilizados devem ser removidos
MITRE - CWE-476	Ponteiros nulos não devem ser referenciados
MITRE - CWE-546	Rastrear o uso das anotações *TODO*
MITRE - CWE-476	Recursos devem ser encerrados
MISRA C:2012, 4.4	Seções de código não devem ser comentadas
RSPEC-1192	Strings literais não devem ser duplicadas
RSPEC-1643	Strings não devem ser concatenadas utilizando '+' em um laço de repetição
MITRE - CWE-489	Throwable.printStackTrace(...) não deve ser chamado

RSPEC-110	Árvore de herança de classes não deve ser muito profunda

APÊNDICE

D

Lista de Competências associadas as Regras

A seguir são apresentadas as Competências relacionadas com as Regras de acordo com as Diretrizes Nacionais da Computação, discutidas na Seção 3.2 e os Currículos de Referência Rocha et al. (2005) e Zorzo et al. (2017).

A processo de associação entre a Lista de Habilidades Adquiridas (Competências) definidas na DCN e RF-CC-17 com a Lista de Regras é descrito na Seção 4.4.2.

Tabela D.1: Competências do Eixo de Desenvolvimento presentes nos Referenciais de Formação para o Curso de Ciência da Computação de acordo com Zorzo et al. (2017).

#	Competência	Regra
07	Arquitetura e Projeto	A superclasse não serializável de uma classe 'Serializable' deve ter um construtor sem argumento
		Atributos da Classe não devem ter acessibilidade pública
		Nome dos pacotes devem ser nomeados de acordo com as convenções de nomenclatura
		Nomes de variáveis devem obedecer a uma convenção de nomenclatura
		Os nomes de variáveis locais e de parâmetros de métodos devem obedecer a uma convenção de nomenclatura
		Parâmetros não utilizados devem ser removidos
		Árvore de herança de classes não deve ser muito profunda

CONTINUAÇÃO – Competências do Eixo de Desenvolvimento presentes nos Referenciais de Formação para o Curso de Ciência da Computação de acordo com Zorzo et al. (2017).

#	Competência	Regra
15	Desenvolvimento de Sistemas	As instruções 'switch' devem terminar com cláusulas 'default'
		Atribuições não devem ser redundantes
		Atributos mutáveis não devem ser 'public static'
		Atributos privados usados apenas como variáveis locais nos métodos devem se tornar variáveis locais
		Contextos 'public static' devem ser constantes
		Código-fonte devem possuir uma quantidade suficiente de linhas comentadas
		Exceções 'lançadas' não devem ser supérfluas
		Importações não utilizadas devem ser removidas
		Literais booleanos não devem ser redundantes
		Múltiplas variáveis não devem ser declaradas na mesma linha
		Nome dos métodos devem estar de acordo com convenções de nomes
		Os designadores de arranjos '[ ]' devem estar na declaração do tipo e não da variável
		Ponteiros nulos não devem ser referenciados
		Seções de código não devem ser comentadas
		Strings literais não devem ser duplicadas
14	Utilização de Tecnologia	Anotação @Override deve ser utilizada para sobrescrever e implementar métodos
		As saídas padrão não devem ser usadas diretamente para registrar nada
		Catches precisam ser combinados
		Classes internas anônimas contendo apenas um método devem se tornar lambdas
		Classes não serializáveis não devem ser persistidas
		Conversões redundantes não devem ser utilizadas
		Declarações devem utilizar as Interfaces das coleções como 'List' ao invés da implementação 'LinkedList'

CONTINUAÇÃO – Competências do Eixo de Desenvolvimento presentes nos Referenciais de Formação para o Curso de Ciência da Computação de acordo com Zorzo et al. (2017).

#	Competência	Regra
		Métodos instanciados não devem escrever em contextos estáticos
		O pacote sem nome padrão não deve ser usado
		Os valores de retorno não devem ser ignorados quando contêm o código de status da operação
		Rastrear o uso das anotações 'TODO'
		Recursos devem ser encerrados
		Strings não devem ser concatenadas utilizando '+' em um laço de repetição
		Throwable.printStackTrace(...) não deve ser chamado

APÊNDICE

E

Lista de Regras com Conceitos Relacionados

A seguir são apresentados os conceitos relacionados com a disciplina de Programação Orientada a Objetos, e as devidas vinculações com as regras selecionadas.

Esta lista é similar a que foi enviada para o professor na validação dos conceitos relacionados.

Tabela E.1: Relação entre Regras e Conceitos Relacionados

#	**Regra**	**Conceito**
3	Anotação *@Override* deve ser utilizada para sobrescrever e implementar métodos	Abstração
		Anotações
		Polimorfismo
4	Contextos *public static* devem ser constantes	Contexto Estático (static)
		Declaração e Definição
		Escopo
		Padrão de Projeto
3	As instruções *switch* devem terminar com cláusulas *default*	Estruturas de Controle
		Fluxo de Execução
		Tratamento de Exceções
2	Exceções *lançadas* não devem ser supérfluas	StackTrace
		Tratamento de Exceções

CONTINUAÇÃO – Relação entre Regras e Conceitos Relacionados

#	**Regra**	**Conceito**
2	Classes internas anônimas contendo apenas um método devem se tornar lambdas	Classes
		Classes Anônimas
3	Os designadores de arranjos [ ] devem estar na declaração do tipo e não da variável	Coleções (Arranjos)
		Declaração e Definição
		Tipos de Dados
2	Atribuições não devem ser redundantes	Atribuição
		Atributos e Variáveis
1	Literais booleanos não devem ser redundantes	Estruturas de Controle
5	*Catches* precisam ser combinados	Bloco de Instrução
		Fluxo de Execução
		Loggers
		StackTrace
		Tratamento de Exceções
5	Atributos da Classe não devem ter acessibilidade pública	Construtores e Destrutores
		Controles de Acesso
		Encapsulamento
		Escopo
		Padrão de Projeto
5	Declarações devem utilizar as Interfaces das coleções como *List* ao invés da implementação *LinkedList*	Atribuição
		Coleções (Arranjos)
		Declaração e Definição
		Herança
		Polimorfismo
2	Nomes de variáveis devem obedecer a uma convenção de nomenclatura	Atributos e Variáveis
		Declaração e Definição
3	Árvore de herança de classes não deve ser muito profunda	Classes
		Herança
		Padrão de Projeto

CONTINUAÇÃO – Relação entre Regras e Conceitos Relacionados

#	**Regra**	**Conceito**
2	Métodos instanciados não devem escrever em contextos *static*	Atribuição
		Contexto Estático (static)
3	Os nomes de variáveis locais e de parâmetros de métodos devem obedecer a uma convenção de nomenclatura	Atributos e Variáveis
		Declaração e Definição
		Escopo
1	Nome dos métodos devem estar de acordo com convenções de nomes	Métodos
2	Múltiplas variáveis não devem ser declaradas na mesma linha	Atribuição
		Declaração e Definição
5	Atributos mutáveis não devem ser *public static*	Atributos e Variáveis
		Controles de Acesso
		Declaração e Definição
		Encapsulamento
		Escopo
3	Classes não serializáveis não devem ser persistidas	Classes
		Entrada/Saída de Dados
		Serialização
5	Ponteiros nulos não devem ser referenciados	Atribuição
		Estruturas de Controle
		Null Pointers
		Objetos
		Tratamento de Exceções
1	Nome dos pacotes devem ser nomeados de acordo com as convenções de nomenclatura	Pacotes
5	Atributos privados usados apenas como variáveis locais nos métodos devem se tornar variáveis locais	Atributos e Variáveis
		Bloco de Instrução
		Classes
		Escopo
		Garbage Collection

CONTINUAÇÃO – Relação entre Regras e Conceitos Relacionados

#	**Regra**	**Conceito**
3	Conversões redundantes não devem ser utilizadas	Conversões (Cast)
		Declaração e Definição
		Tipos de Dados
5	Recursos devem ser encerrados	Construtores e Destrutores
		Entrada/Saída de Dados
		Fluxo de Execução
		Garbage Collection
		Tratamento de Exceções
4	Os valores de retorno não devem ser ignorados quando contêm o código de status da operação	Estruturas de Controle
		Fluxo de Execução
		Função
		Tratamento de Exceções
1	Seções de código não devem ser comentadas	Comentários
1	Código-fonte devem possuir uma quantidade suficiente de linhas comentadas	Comentários
3	As saídas padrão não devem ser usadas diretamente para registrar nada	Entrada/Saída de Dados
		Loggers
		StackTrace
2	*Strings* literais não devem ser duplicadas	Escopo
		Padrão de Projeto
5	*Strings* não devem ser concatenadas utilizando '+' em um laço de repetição	Atribuição
		Fluxo de Execução
		Fluxo de Repetição
		Operadores e Expressões
		Tipos de Dados
2	O pacote sem nome padrão não deve ser usado	Pacotes
		Padrão de Projeto
1	Importações não utilizadas devem ser removidas	Importação

CONTINUAÇÃO – Relação entre Regras e Conceitos Relacionados

#	**Regra**	**Conceito**
2	Rastrear o uso das anotações *TODO*	Anotações
		Comentários
3	*Throwable.printStackTrace()* não deve ser chamado	Loggers
		StackTrace
		Tratamento de Exceções
7	A superclasse não serializável de uma classe *Serializable* deve ter um construtor sem argumento	Classes
		Construtores e Destrutores
		Declaração e Definição
		Entrada/Saída de Dados
		Herança
		Objetos
		Serialização
3	Parâmetros não utilizados devem ser removidos	Atributos e Variáveis
		Função
		Métodos

APÊNDICE

F

Lista de Regras Mapeadas

A seguir são apresentadas as Regras Avaliadas, conforme descrito na Seção 4.3.1. São demonstradas a definição, o problema conceitual, os conceitos relacionados, a solução proposta pelo autor, a conformidade da regra e a competência associada.

Por uma questão de organização estrutural, foram omitidos os exemplos de trecho de código em Conformidade/Não-Conformidade. Os mesmos podem ser vistos a partir das referências disponíveis no Anexo C.

R1 - Resources should be closed

Recursos devem ser encerrados

Definição

Conexões, streams, Arquivos e classes que implementam a interface Closable, devem ser encerrados após o uso.

Problema Conceitual

A falta de encerramento dos recursos implica em vazamento de recursos.

Conceitos Relacionados

Garbage Collector, Fluxo de Execução, Construtores e Destrutores, Entrada/Saída de Dados, Tratamento de Exceções

Solução Proposta

Utilizar o bloco finally ou o padrão try-with-resources.

Conformidade

MITRE, CWE-459 - Incomplete Cleanup

Competência

Utilização de Tecnologia

R2 - Class variable fields should not have public accessibility

Atributos da Classe não devem ter acessibilidade pública

Definição

atributos não devem ter acessibilidade pública, pois não respeitam o princípio de Encapsulamento.

Problema Conceitual

os valores estão sujeitos a alterações a partir de qualquer parte do código, podendo acarretar em valores inesperados devido à dificuldade em implementar regras de utilização e validação.

Conceitos Relacionados

Encapsulamento, Padrão de Projeto, Construtores e Destrutores, Escopo

Solução Proposta

a utilização de atributos privados e definição de métodos de acesso previnem modificações não autorizadas.

Conformidade

MITRE, CWE-493 – Critical Public Variable Without Final Modifier

Competência

Arquitetura e Projeto

R3 - Assignments should not be redundant

Atribuições não devem ser redundantes

Definição

a propriedade transitiva garante que se a == b e b == c, então a == c.

Problema Conceitual

não faz sentido atribuir a == c e/ou vice-versa, pois já são equivalentes.

Conceitos Relacionados

Atribuição, Atributos e Variáveis

Solução Proposta

Remover o trecho redundante.

Conformidade

RSPEC-4165

Competência

Desenvolvimento de Sistemas

R4 - 'public static' fields should be constant

Contextos "public static" devem ser constantes

Definição

o modificador static altera o escopo de um método, atributo ou classe(interna). O uso desse modificador indica que o contexto pertence a Classe e não a uma instância da mesma, sendo o mesmo para todos os Objetos.

Problema Conceitual

qualquer modificação nesse contexto repercute todas as instâncias, sendo importante a cautela em sua implementação.

Conceitos Relacionados

Padrão de Projeto, Declaração e Definição, Contexto Estático, Escopo

Solução Proposta

adicionar o modificador final sempre que não forem desejadas modificações no contexto.

Conformidade

MITRE, CWE-500 - Public Static Field Not Marked Final

Competência

Desenvolvimento de Sistemas

R5 - Standard outputs should not be used directly to log anything

As saídas padrão não devem ser usadas diretamente para registrar nada

Definição

O uso de saídas padrão para registrar os logs é altamente desaconselhável.

Problema Conceitual

Registros de logs precisam ser facilmente recuperáveis, devem possuir uma estrutura uniforme e devem restringir o acesso, quando necessário.

Conceitos Relacionados

Entrada/Saída de Dados, Loggers, StackTrace

Solução Proposta

Substituir as saídas padrão para Loggers

Conformidade

CERT, ERR02-J. - Prevent exceptions while logging data

Competência

Utilização de Tecnologia

R6 - Throwable.printStackTrace(...) should not be called

Throwable.printStackTrace(...) não deve ser chamado

Definição

a utilização deste recurso imprime a pilha de execução de algum fluxo.

Problema Conceitual

por padrão, esse fluxo pode expor informações sensíveis

Conceitos Relacionados

Tratamento de Exceções, Loggers, StackTrace

Solução Proposta

é recomendada a utilização de Loggers para a definição de mensagens de erro.

Conformidade

OWASP Top 10 2017 Category A3 - Sensitive Data Exposure, MITRE, CWE-489 - Leftover Debug Code

Competência

Utilização de Tecnologia

R7 - Mutable fields should not be 'public static'

Atributos mutáveis não devem ser "public static"

Definição

Não existem razões para objetos mutáveis possuírem visibilidade pública.

Problema Conceitual

O acesso direto a esses atributos pode gerar comportamentos inesperados.

Conceitos Relacionados

Atributos e Variáveis, Controles de Acesso, Declaração e Definição, Encapsulamento, Escopo

Solução Proposta

Membros estáticos mutáveis de classes e enumerações acessadas diretamente, e não através de getters e setters, devem ser protegidos na medida do possível. Tais variáveis devem ser movidas para classes e sua visibilidade reduzida.

Conformidade

MITRE, CWE-607 - Public Static Final Field References Mutable Object, CERT, OBJ01-J. - Limit accessibility of fields, CERT, OBJ13-J. - Ensure that references to mutable objects are not exposed

Competência

Desenvolvimento de Sistemas

R8 - Strings should not be concatenated using '+' in a loop

Strings não devem ser concatenadas utilizando '+' em um laço de repetição

Definição

a utilização do operador de soma para a concatenação de Strings, embora permitida, é desaconselhada.

Problema Conceitual

por se tratarem de objetos imutáveis, para concatenar Strings é necessário transformá-la em um objeto intermediário (arranjo de caracteres), sendo necessário percorrer o arranjo para adicionar os novos caracteres e depois converter novamente em String, sendo essa operação custosa.

Conceitos Relacionados

Atribuição, Fluxo de Execução, Fluxo de Repetição, Operadores e Expressões, Tipos de Dados

Solução Proposta

recomenda-se a utilização da classe StringBuilder que fornece métodos para manipulação correta de Strings.

Conformidade

RSPEC-1643

Competência

Utilização de Tecnologia

R9 - Private fields only used as local variables in methods should become local variables

"Atributos privados usados apenas como variáveis locais nos métodos devem se tornar variáveis locais"

Definição

quando um atributo privado é utilizado para atribuição e leitura de valores apenas dentro de métodos, esse atributo não pertence à classe.

Problema Conceitual

mesmo sendo um atributo privado, atributos definidos dentro das classes possuem visibilidade para todos os métodos da classe, podendo acarretar em um acesso inesperado.

Conceitos Relacionados

Atributos e Variáveis, Bloco de Instrução, Classes, Escopo, Garbage Collection

Solução Proposta

mover os atributos para os métodos.

Conformidade

RSPEC-1450

Competência

Desenvolvimento de Sistemas

R10 - Multiple variables should not be declared on the same line

Múltiplas variáveis não devem ser declaradas na mesma linha

Definição

É difícil ler várias variáveis em uma mesma linha.

Problema Conceitual

A dificuldade na identificação pode gerar dúvidas e confusão no desenvolvedor.

Conceitos Relacionados

Declaração e Definição, Atribuição

Solução Proposta

declarar uma variavel por linha

Conformidade

CERT, DCL52-J. - Do not declare more than one variable per declaration

Competência

Desenvolvimento de Sistemas

R11 - Anonymous inner classes containing only one method should become lambdas

Classes internas anônimas contendo apenas um método devem se tornar lambdas

Definição

Diversas Linguagens de Programação (Java>=8, Python, C#, JavaScript, C++) permitem a utilização de funções anônimas. Ao usar expressões lambda, é possível escrever funções locais que podem ser passadas como argumentos ou retornadas como o valor de chamadas de função.

Problema Conceitual

Classes Internas aos métodos podem gerar problemas de legibilidade.

Conceitos Relacionados

Classes, Classes Anônimas

Solução Proposta

Substituir as classes internas aos métodos por expressões lambda.

Conformidade

RSPEC-1604

Competência

Utilização de Tecnologia

R12 - Unused method parameters should be removed

Parâmetros não utilizados devem ser removidos

Definição

Quaisquer que sejam os valores passados para esses parâmetros, o comportamento será o mesmo.

Problema Conceitual

Código que não tem efeito ou nunca é executado (ou seja, inacessível) normalmente é o resultado de um erro de codificação e pode causar comportamento inesperado.

Conceitos Relacionados

Atributos e Variáveis, Função, Métodos

Solução Proposta

Remover os parâmetros não utilizados.

Conformidade

CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

Competência

Arquitetura e Projeto

R13 - Catches should be combined

Catches precisam ser combinados

Definição

o comando catch permite capturar uma ou mais exceções.

Problema Conceitual

Trechos de códigos repetidos geram problemas de legibilidade e a duplicação impacta diretamente na manutenibilidade.

Conceitos Relacionados

Bloco de Instrução, Fluxo de Execução, Loggers, StackTrace, Tratamento de Exceções

Solução Proposta

quando vários blocos catch tiverem o mesmo código, eles deverão ser combinados para melhor legibilidade.

Conformidade

RSPEC-2147

Competência

Utilização de Tecnologia

R14 - Array designators '( )' should be on the type, not the variable

Os designadores de arranjos "[ ]" devem estar na declaração do tipo e não da variável

Definição

Os designadores de matriz sempre devem estar localizados no tipo para melhor legibilidade do código.

Problema Conceitual

Desenvolvedores devem examinar o tipo e o nome da variável para saber se uma variável é ou não um arranjo.

Conceitos Relacionados

Coleções (Arranjos), Declaração e Definição, Tipos de Dados

Solução Proposta

Utilizar os designadores na declaração de tipo.

Conformidade

RSPEC-1197

Competência

Desenvolvimento de Sistemas

R15 - Package names should comply with a naming convention

Nome dos pacotes devem ser nomeados de acordo com as convenções de nomenclatura

Definição

Quando diversos programadores manipulam os mesmos repositórios de código, é importante que convenções de nomenclaturas sejam utilizadas.

Problema Conceitual

Convenções de nomes são importantes para rápida identificação e entendimento do código. A maioria das IDEs cria nomes genéricos para os pacotes, dificultando a organização dos arquivos.

Conceitos Relacionados

Pacotes

Solução Proposta

Nomear pacotes de acordo com as convenções de nomes de pacotes (`^[a-z_]+(\.[a-z_][a-z0-9_]*)*$`). Não se esquecendo de utilizar nomes que identificam o conteúdo dos arquivos presentes.

Conformidade

RSPEC-120

Competência

Arquitetura e Projeto

R16 - String literals should not be duplicated

Strings literais não devem ser duplicadas

Definição

Strings literais muitas vezes são utilizadas como constantes.

Problema Conceitual

A duplicação impacta diretamente na manutenibilidade, tornando o processo de refatoração propenso a erros.

Conceitos Relacionados

Escopo, Padrão de Projeto

Solução Proposta

Definir uma constante “static final” para ser utilizada.

Conformidade

RSPEC-1192

Competência

Desenvolvimento de Sistemas

R17 - Return values should not be ignored when they contain the operation status code

Os valores de retorno não devem ser ignorados quando contêm o código de status da operação

Definição

Quando o valor de retorno de uma chamada de função contém o código de status da operação, esse valor deve ser testado para garantir que a operação foi concluída com êxito.

Problema Conceitual

A não validação do retorno da função colabora para a incidência de comportamentos inesperados.

Conceitos Relacionados

Estruturas de Controle, Fluxo de Execução, Função, Tratamento de Exceções

Solução Proposta

validar as funções de retorno antes de continuar o fluxo de execução do programa.

Conformidade

MISRA C:2012, 17.7 - The value returned by a function having non-void return type shall be used, MITRE, CWE-754 - Improper Check for Unusual Exceptional Conditions

Competência

Utilização de Tecnologia

R18 - The non-serializable super class of a 'Serializable' class should have a no-argument constructor

A superclasse não serializável de uma classe 'Serializable' deve ter um construtor sem argumento

Definição

Quando um objeto serializável possui um ancestral não serializável em sua cadeia de herança, a "desserialização" de objetos (reinstanciando o objeto do arquivo) inicia na primeira classe não serializável e prossegue pela cadeia, adicionando as propriedades de cada classe filho subsequente, até que o objeto final seja instanciado.

Problema Conceitual

O ancestral não serializável de uma classe Serializable deve ter um construtor no-arg, caso contrário, a classe é serializável, mas não "desserializável".

Conceitos Relacionados

Classes, Construtores e Destrutores, Declaração e Definição, Herança, Entrada/Saída de Dados, Objetos, Serialização

Solução Proposta

Criar um Construtor sem argumentos.

Conformidade

RSPEC-2055

Competência

Arquitetura e Projeto

R19 - Local variable and method parameter names should comply with a naming convention

Os nomes de variáveis locais e de parâmetros de métodos devem obedecer a uma convenção de nomenclatura

Definição

Quando diversos programadores manipulam os mesmos repositórios de código, é importante que convenções sejam utilizadas.

Problema Conceitual

Convenções de nomes são importantes para rápida identificação e entendimento do código.

Conceitos Relacionados

Atributos e Variáveis, Declaração e Definição, Escopo

Solução Proposta

Nomear variáveis locais e parâmetros de acordo com as convenções de nomes de pacotes (`^[a-z][a-zA-Z0-9]*$`).

Conformidade

RSPEC-117

Competência

Arquitetura e Projeto

R20 - Field names should comply with a naming convention

Nomes de variáveis devem obedecer a uma convenção de nomenclatura

Definição

Quando diversos programadores manipulam os mesmos repositórios de código, é importante que convenções sejam utilizadas.

Problema Conceitual

Convenções de nomes são importantes para rápida identificação e entendimento do código.

Conceitos Relacionados

Atributos e Variáveis, Declaração e Definição

Solução Proposta

Nomear variáveis de acordo com as convenções de nomes de pacotes (`^[a-z][a-zA-Z0-9]*$`).

Conformidade

RSPEC-117

Competência

Arquitetura e Projeto

R21 - The default unnamed package should not be used

O pacote sem nome padrão não deve ser usado

Definição

Os pacotes sem nome são fornecidos pela plataforma Java principalmente por conveniência ao desenvolver aplicativos pequenos ou temporários ou ao iniciar o desenvolvimento.

Problema Conceitual

A organização dos arquivos é fundamental no desenvolvimento de sistemas legados.

Conceitos Relacionados

Pacotes

Solução Proposta

Não utilizar os pacotes padrões.

Conformidade

RSPEC-1220

Competência

Utilização de Tecnologia

R22 - Null pointers should not be dereferenced

Ponteiros nulos não devem ser desreferenciados

Definição

uma referência a null não deve ser acessada.

Problema Conceitual

ao acessar uma referência null uma Exceção é lançada. Na melhor das hipóteses o programa é encerrado abruptamente. Em casos mais graves, informações sensíveis podem ser expostas.

Conceitos Relacionados

Atribuição, Estruturas de Controle, Null Pointers, Objetos, Tratamento de Exceções

Solução Proposta

Não acessar objetos desreferenciados.

Conformidade

MITRE, CWE-476 - NULL Pointer Dereference

Competência

Desenvolvimento de Sistemas

R23 - 'switch' statements should end with 'default' clauses

As instruções 'switch' devem terminar com cláusulas 'default'

Definição

Instruções "switch" podem definir uma série de comportamentos conforme a entrada correspondente.

Problema Conceitual

Uma entrada desconhecida pode gerar um comportamento inesperado no programa.

Conceitos Relacionados

Estruturas de Controle, Fluxo de Execução, Tratamento de Exceções

Solução Proposta

A cláusula deve tomar as medidas apropriadas ou conter um comentário adequado sobre o motivo pelo qual nenhuma ação foi tomada.

Conformidade

MITRE, CWE-478 - Missing Default Case in Switch Statement, CERT, MSC01-C. - Strive for logical completeness

Competência

Desenvolvimento de Sistemas

R24 - Inheritance tree of classes should not be too deep

Árvore de herança de classes não deve ser muito profunda

Definição

Herança é um dos conceitos fundamentais da Orientação a Objetos, mas abusar de sua utilização pode trazer problemas.

Problema Conceitual

Uma árvore de herança muito profunda torna o código complexo e insustentável.

Conceitos Relacionados

Herança, Classes, Padrão de Projeto

Solução Proposta

Convém refatorar o código-fonte a fim de identificar trechos nos quais a utilização da herança não é necessária, podendo ser substituída pela Composição, por exemplo.

Conformidade

RSPEC-110

Competência

Arquitetura e Projeto

R25 - Non-serializable classes should not be written

Classes não serializáveis não devem ser persistidas

Definição

Nada em uma classe não serializável pode ser gravado em arquivo, e a tentativa de serializar essa classe resultará em uma exceção.

Problema Conceitual

Somente uma classe que implementa Serializable ou uma que estende essa classe pode ser serializada (ou desserializada) com êxito.

Conceitos Relacionados

Classes, Entrada/Saída de Dados, Serialização

Solução Proposta

Implementar Serializable na classe que se deseja persistir.

Conformidade

RSPEC-2118

Competência

Utilização de Tecnologia

R26 - Redundant casts should not be used

Conversões redundantes não devem ser utilizadas

Definição

Conversões não devem ser redundantes.

Problema Conceitual

Expressões desnecessárias de conversão tornam o código mais difícil de ler e entender.

Conceitos Relacionados

Conversões (Cast), Declaração e Definição, Tipos de Dados

Solução Proposta

Remover as conversões desnecessárias.

Conformidade

RSPEC-1905

Competência

Utilização de Tecnologia

R27 - 'throws' declarations should not be superfluous

Declarações "lançadas" não devem ser supérfluas

Definição

Utilizadas para indicar um possível problema no fluxo de execução, essas declarações não devem ser genéricas.

Problema Conceitual

O lançamento de exceções genéricas ou incorretas dificulta a manipulação dos estados de exceção.

Conceitos Relacionados

StackTrace, Tratamento de Exceções

Solução Proposta

Utilizar os estados de exceção que descrevem corretamente a interrupção no fluxo.

Conformidade

RSPEC-1130

Competência

Desenvolvimento de Sistemas

R28 - Source files should not have any duplicated blocks

Código-fonte não deve ter blocos duplicados

Definição

Essa violação ocorre quando o desenvolvedor duplica um bloco inteiro de código-fonte, normalmente uma função ou procedimento.

Problema Conceitual

A duplicação de código gera dificuldade na manutenção e na legibilidade.

Conceitos Relacionados

Padrão de Projeto

Solução Proposta

Identificação do trecho duplicado e refatoração da solução.

Conformidade

RSPEC-1192

Competência

Desenvolvimento de Sistemas

R29 - Track uses of 'TODO' tags

Rastrear o uso das marcações "TODO"

Definição

Marcações TODO normalmente indicam a necessidade de revisar trechos de código.

Problema Conceitual

Caso o trecho de código já tenha sido revisado, pode acabar gerando uma confusão no programador.

Conceitos Relacionados

Anotações

Solução Proposta

Revisar os trechos de código e remover marcações desnecessárias.

Conformidade

MITRE, CWE-546 - Suspicious Comment

Competência

Uso de Tecnologia

R30 - Instance methods should not write to 'static' fields

Métodos instanciados não devem escrever em contextos "static"

Definição

Idealmente, os contextos estáticos são atualizados apenas a partir de métodos estáticos sincronizados.

Problema Conceitual

A atualização correta de um contexto estático a partir de um método não estático é difícil de corrigir e pode facilmente levar a erros se houver várias instâncias de classe e / ou vários threads em execução.

Conceitos Relacionados

Contexto Estático, Atribuição

Solução Proposta

Utilizar métodos sincronizados.

Conformidade

RSPEC-2696

Competência

Utilização de Tecnologia

R31 - Boolean literals should not be redundant

Literais booleanos não devem ser redundantes

Definição

Testes de expressões booleanas não precisam ser comparados com as primitivas de igual valor.

Problema Conceitual

Comparar o retorno das funções ou valores das variáveis com as primitivas pode prejudicar a legibilidade.

Conceitos Relacionados

Estruturas de Controle

Solução Proposta

Remover a redundância

Conformidade

RSPEC-1125

Competência

Desenvolvimento de Sistemas

R32 - Useless imports should be removed

Importações não utilizadas devem ser removidas

Definição

Para a utilização de bibliotecas, módulos e coleções, é necessário referenciar esses artefatos através de comandos de importação.

Problema Conceitual

Importações não utilizadas podem gerar confusão nos desenvolvedores, que não sabem quais artefatos estão em utilização ou não.

Conceitos Relacionados

Importação

Solução Proposta

remover as importações não utilizadas.

Conformidade

RSPEC-1128

Competência

Desenvolvimento de Sistemas

R33 - Multiple variables should not be declared on the same line

Múltiplas variáveis não devem ser declaradas na mesma linha

Definição

É difícil ler várias variáveis em uma mesma linha.

Problema Conceitual

A dificuldade na identificação pode gerar dúvidas e confusão no desenvolvedor.

Conceitos Relacionados

Atribuição, Declaração e Definição

Solução Proposta

Declarar um atributo(ou variável) por linha.

Conformidade

DCL52-J

Competência

Desenvolvimento de Sistemas

R34 - Method names should comply with a name convention

Nome dos métodos devem estar de acordo com convenções de nomes

Definição

Em repositórios de código compartilhado, a adoção de convenções para nomear métodos, classes, arquivos e atributos é fundamental.

Problema Conceitual

A não adoção de convenções de nomes prejudica a legibilidade do código e consequentemente sua manutenibilidade.

Conceitos Relacionados

Métodos

Solução Proposta

Adequar o nome do método ao padrão de expressão regular `^[a-z][a-zA-Z0-9]*$:.`

Conformidade

Oracle

Competência

Desenvolvimento de Sistemas

R35 - '@override' should be used on overriding and implementing methods

Anotação @Override deve ser utilizada para sobrescrever e implementar métodos

Definição

A utilização da anotação @Override se faz necessária em duas situações: 1) permite ao compilador emitir um alerta caso não houver alteração no método sobrescrito. 2) melhoria na legibilidade do código, destacando que aquele trecho de código está sobrescrevendo algo.

Problema Conceitual

A não utilização de anotações pode impactar na legibilidade do código.

Conceitos Relacionados

Anotações, Polimorfismo, Abstração

Solução Proposta

Incluir a anotação @Override em cima do método que estiver sendo sobrescrito ou implementado (no caso de métodos abstratos).

Conformidade

RSPEC-1161

Competência

Utilização de Tecnologia

R36 - Sections of code should not be 'commented out'

Seções de código não devem ser comentadas

Definição

Trechos de código não utilizados devem ser removidos do repositório e nunca comentados.

Problema Conceitual

Alguns programadores comentam grandes trechos de códigos quando não possuem certeza da sua usabilidade ou aplicabilidade futura. Esse código

deve ser removido, buscando sempre deixar o repositório limpo.

Conceitos Relacionados

Comentários

Solução Proposta

Remover trechos de códigos comentados.

Conformidade

MISRA C:2012, Dir. 4.4

Competência

Desenvolvimento de Sistemas

R37 - Declarations should use java collection interfaces such as 'List' rather than specific implementation classes such as 'LinkedList'

Declarações devem utilizar as Interfaces das coleções como 'List' ao invés da implementação 'LinkedList'

Definição

As implementações (HashSet, ArrayList, LinkedList) da interface (Set, List) devem ser utilizadas na hora de instanciar uma coleção, mas o resultado deve ser armazenado em uma variável do tipo da interface (Set, List).

Problema Conceitual

O objetivo da API de coleções Java é fornecer uma hierarquia bem definida de interfaces para ocultar detalhes de implementação.

Conceitos Relacionados

Herança, Declaração e Definição, Coleções (Arranjos), Polimorfismo

Solução Proposta

EX: List<Classe> catalogo = new ArrayList<Classe>();

Conformidade

RSPEC-1319

Competência

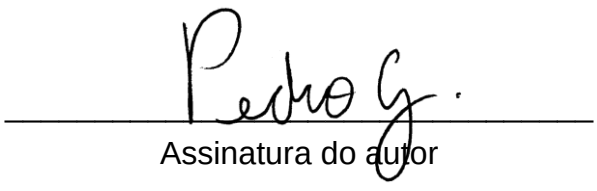
Utilização de Tecnologia

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, 22 / 06 / 2021



Assinatura do autor