MANOEL VALERIO DA SILVEIRA NETO

MONITORAMENTO E GESTÃO DA DÍVIDA TÉCNICA DE *CODE SMELL*: UMA ABORDAGEM ORIENTADA AO DESENVOLVEDOR

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Curitiba

2021

MANOEL VALERIO DA SILVEIRA NETO

MONITORAMENTO E GESTÃO DA DÍVIDA TÉCNICA DE *CODE SMELL*: UMA ABORDAGEM ORIENTADA AO DESENVOLVEDOR

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Área de concentração: Ciência da Computação

Orientador: Prof. Dra. Sheila Reinehr

Coorientador: Prof. Dra. Andreia Malucelli

Curitiba

2021

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

Curitiba, 18 de junho de 2024.

48-2024

DECLARAÇÃO

Declaro para os devidos fins, que **Manoel Valério da Silveira Neto** defendeu a dissertação de Mestrado intitulada "**MONITORAMENTO E GESTÃO DA DÍVIDA TÉCNICA DE CODE SMELL: UMA ABORDAGEM ORIENTADA AO DESENVOLVEDOR**", na área de concentração Ciência da Computação no dia 13 de outubro de 2021, no qual foi aprovado.

Declaro ainda, que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade firmo a presente declaração.

gov.br Documento assinado digitalmente
EMERSON CABRERA PARAISO
Data: 18/06/2024 10:22:23-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Edilene de Oliveira dos Santos CRB 9 / 1636

S587m 2021

Silveira Neto, Manoel Valério da
Monitoramento e gestão da dívida técnica de code smell : uma abordagem orientada ao desenvolvedor / Manoel Valério da Silveira Neto ; orientadora: Sheila Reinehr ; coorientadora: Andreia Malucelli. -- 2021
203 f. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2021
Inclui bibliografia

1. Informática. 2. Mineração de dados (Computação). 3. Engenharia de software. 4. Código fonte. 5. Code smell. I. Reinehr, Sheila dos Santos. II. Malucelli, Andreia. III. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. IV. Título

CDD 20. ed. – 004

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

“This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001".

DEDICATÓRIAS

A minha esposa Flavia, pelo apoio e incentivo nesta caminhada.

Aos meus pais José e Justine, por tudo.

AGRADECIMENTOS

A Deus, por colocar pessoas maravilhosas em meu caminho as quais me fazem acreditar no meu potencial.

A minha esposa Flavia pelo carinho e compressão nesta fase.

Aos meus pais José e Justine, pelos valores e por sempre me apoiarem.

As minhas orientadoras Sheila Reinehr e Andreia Malucelli pelos seus ensinamentos, compreensão e pela oportunidade de realizar este trabalho.

Aos professores Emerson Cabrera Paraiso e Júlio Cesar Nievola por suas valiosas considerações na minha banca de qualificação.

Aos professores Alceu de Souza Britto Jr., Altair Olivo Santin, Júlio Cesar Nievola, Eduardo Kugler Viegas por seus comentários na minha apresentação no Seminário de Pesquisa do PPGIA-PUCPR 2020, pois a partir destes comentários tive *insights* para outras oportunidades para pesquisa.

Aos colegas do grupo de pesquisa em Engenharia de *software* do PPGIA PUCPR, pelas contribuições durantes minhas apresentações.

Ao Thober Coradi Detofeno pelas orientações discussões sobre dívida técnica.

Aos amigos Alexandre Denes dos Santos e Rodrigo Gonçalves, pelas nossas conversas sobre essa pesquisa.

As empresas da qual trabalhei durante o período de mestrado, que entenderam minhas ausências para eu poder participar das aulas, aqui representadas por Douglas Dias Batista, Carlos Bonilha Junior, Rodrigo Gonçalves, Davi Ruiz, Fabio Claus Soares e Filipe Sguarizi Panceri.

A todos que de alguma forma contribuíram para a realização deste trabalho.

O começo é a metade do todo.

Platão

RESUMO

Sabe-se que a qualidade de um *software* está relacionada ao código-fonte, e a qualidade do código-fonte relacionada à experiência e ao conhecimento do desenvolvedor nas práticas de programação. Muitos desenvolvedores submetem seus códigos aos repositórios de código-fonte sem saber a qualidade do código que está sendo submetido ao controle de versões. Somente após o código ser inspecionado por alguma ferramenta de análise estática é possível verificar a existência de *code smells*, que são estruturas de código que indicam a violação de princípios de *design de código* e programação, e causam atividades futuras de manutenção por meio de refatoração. Nem sempre o desenvolvedor está alerta a esse problema. O objetivo desta pesquisa foi propor uma abordagem para que o desenvolvedor tenha acesso às tendências de geração de *code smells* e com isso possa melhorar suas habilidades referentes ao *design* de código. Esta pesquisa foi de natureza empírica experimental, utilizando mapeamento de processos e mineração de dados para propor uma abordagem para gestão de *code smell,* do ponto de vista do desenvolvedor. Os resultados apontam que por meio de dois experimentos realizados, a abordagem proposta foi suficiente para gerar conhecimento sobre *code smells* criados pelo desenvolvedor, possibilitando identificar as tendências de criação de *code smell*.

Palavras-chave: Análise estática de código. *Code smell*. Mineração de repositório de *software*.

ABSTRACT

It is known that the software quality is related to the source code, and the source code quality is associated with the experience and knowledge of the developer in programming practices. Many developers submit their code to the source code repository without knowing the quality of the code being submitted to version control. Only after the code is inspected by some static analysis tool it is possible to verify the existence of code smells, which are code structures that indicate the violation of code design and programming principles and cause future maintenance activities through refactoring. The developer is not always aware of this problem. The objective of this research is to propose an approach so that the developer has access to the trends of code smells generation and can improve his skills regarding code design. This research is of an empirical, experimental nature, which uses process mapping and data mining to propose an approach to code smell management from the developer's point of view. The results show that, through the experiments carried out, the proposed approach was sufficient to generate knowledge about code smells created by the developer, making it possible to identify trends in the creation of code smells.

Keywords: Static code analysis. Code smells. Mining software repository.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE EQUAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

AIC	Critério de informação de Akaike
API	*Application Programming Interface*
BPMN	*Business Process Model and Nation*
CD	*Continuous Delivery*
CI	*Continuous Integration*
DCcs	Dias para Correção de *code smell*
DIKW	*Data-Information-Knowledge-Wisdom*
DT	Dívida Técnica
ES	Engenharia de *software*
ESE	Engenharia de *Software* Experimental
KDD	*Knowledge Discovery in Databases*
MAE	*Mean Absolut Error*
MAPE	*Mean Absolute Percentage Error*
MFE	*Mean Forecast Error*
MSE	*Mean Squared Erro*r
MSR	*Mining Software Repositories*
OS	*Open Source*
OWASP	*Open-Source Foundation for Application Security*

RMSE	*Root Mean Squared Error*
SARIF	*Static Analysis Results Interchange Format*
SAST	*Static Application Security Testing*
SGDB	Sistema de Gerenciamento de Banco de Dados
SQALE	*Software Quality Assessment based on Lifecycle Expectations*
SQI	Índice de Qualidade SQALE
SQL	*Structured Query Language*
SWEBOK	*Software Engineering Body of Knowledge*

CAPÍTULO 1 - INTRODUÇÃO

Não insira comentários num código ruim, reescreva-o.

Brian W. Kernighan e P.J. Plaugher

The Elements of Programming, 2d., McGrawHill,1978, p.144.

A qualidade de *software* é uma das áreas de conhecimento da Engenharia de *Software* (ES) abordada no *Software Engineering Body of Knowledge* (SWEBOK). O SWEBOK está estruturado em torno dos processos de ciclo de vida de *software*, o que inclui o processo de Qualidade de *Software*. A Qualidade de *Software* é um assunto que deve estar presente em todas as práticas da ES, inclusive a qualidade permeia as diversas áreas de conhecimento do SWEBOK. O corpo de conhecimento define técnicas de controle de qualidade de *software* de dois tipos: estática e dinâmica. As técnicas dinâmicas envolvem a execução do *software* e as técnicas estáticas envolvem a análise do código-fonte, mas não a sua execução. As técnicas estáticas de controle de qualidade de *software* envolvem a execução de ferramentas de análise estática que têm por objetivo realizar a inspeção contínua da qualidade do código, com a finalidade de encontrar problemas.

Um dos problemas de código que as ferramentas de análise estática podem encontrar é o *code smell*. Fowler *et al.* (2019) definem que *code smells* são estruturas de código que indicam a violação de princípios de *design* e programação e que causam atividades futuras de manutenção por meio de refatoração. Fowler *et al*. (2019) definem refatoração como o processo de alterar um sistema de *software* de forma que não altere o comportamento externo do código, com objetivo de melhorar a estrutura interna. É uma maneira disciplinada de limpar código, que minimiza as chances de introdução de *bugs*.

Cunningham (1992) criou a metáfora da dívida técnica (DT) para explicar para os *stakeholders* que não têm conhecimento técnico de produto de *software,* sobre o que se chama, em desenvolvimento, de refatoração de *software*. Cunningham afirma que entregar o código imaturo é como entrar em dívida, pois em algum momento essa dívida deverá ser paga, em uma analogia a uma dívida financeira.

Para Kruchten *et al.* (2012) *code smells* são um dos tipos de DT de *design* que pode se encontrar em desenvolvimento de *software*. *Code smells* não são *bugs,* não são tecnicamente incorretos e não impedem o funcionamento do *software*, porém afetam a sua qualidade. *Code smell* é a DT mais básica, que pode ser facilmente evitada durante a implementação, se tratada de imediato, pois depende principalmente do conhecimento técnico do desenvolvedor.

A refatoração do código-fonte além de ser um retrabalho, também é o ato de pagar uma DT, pois demanda tempo do desenvolvedor para realizar a correção do *code smell*. Muitos desenvolvedores não têm conhecimento sobre os *code smells* que criam, possibilitando com isso que seus novos códigos levem *code smell* para a nova versão do produto. Isso causa o aumento da DT de *code smell* e do tempo de refatoração futura. Rios *et al*. (2019) apresentam resultados de uma pesquisa sobre causas e efeitos mais comuns da DT, onde, dentre as dez causas apontadas na pesquisa, uma é relacionada à falta de conhecimento e outra à pouca experiência dos desenvolvedores.

Uma das formas de identificar os *code smells* é por meio de ferramentas de análise estática. Essas ferramentas permitem realizar a análise estática do código-fonte e exibem para o desenvolvedor os *code smells* para correção. No entanto, os dados apresentados pelas ferramentas nesta análise são centrados no produto de *software*, ou seja, possibilitam saber quais são os *code smells* de um *software.* Raramente as ferramentas de análise estática quantificam os *code smells* por desenvolvedor, ou tão pouco exibem tendências ou padrões na criação de *code smell* por um desenvolvedor específico. Apesar de ferramentas como SonarQube quantificar os *code smells* por desenvolvedor, a ferramenta apenas faz no contexto de projeto/produto, ou seja, o desenvolvedor não consegue visualizar a quantidade total dos *code smells* criado por ele, independente de produto/projeto.

Se existe *code smell*, muitas vezes é pelo fato da experiência e conhecimento do desenvolvedor não serem suficientes, pois entende-se que nenhum desenvolvedor irá adicionar um *code smell* de forma proposital no código-fonte. Se o desenvolvedor tiver conhecimento antecipado, com base no seu histórico de *commits*, sobre quais *code smells* ele frequentemente cria e quais seus padrões na criação de *code smell*, ele pode aprender com seus erros e evitar criar os mesmos *code smells.*

Visando apoiar a identificação de DT, Lenarduzzi *et al*. (2019)[1] criaram um modelo de dados, associado a um *dataset*, o qual possibilita responder questões sobre DT. No entanto, apenas é fornecido o conjunto de dados para estudos experimentais e sem detalhar o processo para extração e transformação dos dados, o que dificulta a reprodução do mesmo experimento para obter outros dados de outros projetos. O *dataset* apresentado contempla o relacionamento de dados de mineração de repositório com os dados dos *softwares* SonarQube *e* Jira. Analisando o trabalho de Lenarduzzi *et al.* (2019), e os dados que as ferramentas de análise estática fornecem, é possível observar que o trabalho proposto por eles contribui com indícios para preencher a lacuna da existência de abordagem que permita aos desenvolvedores perceber as tendências e associações dos *code smells*.

Tsoukalas *et al.* (2019), em uma pesquisa sobre o estado da arte da previsão da DT, identificaram algumas questões em aberto que devem ser aprofundadas em outros estudos, como a previsão da DT. Eles consideram que os métodos de previsão da DT ainda não se encontram em um nível satisfatório de maturidade. Tsoukalas *et al.* (2019) descrevem que a previsão da DT é o principal achado da revisão de literatura, uma vez que ela pode levar ao desenvolvimento de mecanismos práticos de tomada de decisão, os quais possam auxiliar os desenvolvedores e gerentes de projeto a tomar ações proativas em relação ao reembolso da DT.

Pereira dos Reis *et al.* (2021), em uma revisão sistemática da literatura sobre detecção e visualização de *code smells*, apontaram que 80% dos estudos analisados demonstram apenas técnicas de detecção, sem fornecer técnicas de visualização. A

[1] Disponível em: https://github.com/clowee/The-Technical-Debt-Dataset. Este é um repositório público de Cloud e Web Engineering da Tampere Univerisity.

revisão demonstra que apenas três estudos propuseram soluções dedicadas para visualização de *code smell,* todos com relação ao produto de *software*. Ademais, eles abordaram que ainda há necessidade de conjuntos de dados para facilitar a replicação de experimentos de detecção e validação de códigos e ainda reportaram sobre experimentos em estudos primários, que muitas vezes não disponibilizam os fluxos de trabalho científicos correspondentes e os conjuntos de dados, não permitindo sua reprodutibilidade. Por fim, o estudo observou que as técnicas de visualização parecem ter grande potencial, especialmente em sistemas de grande porte, para ajudar os desenvolvedores a decidir se concordam com uma ocorrência de *code smell* sugerida.

Yamashita e Moonne (2013), em um *survey* para verificar se desenvolvedores se importam com *code smell*, relataram que, dos 85 profissionais de *software* de 29 países entrevistados, com 73 profissionais que respondeu a *survey*, 23 (32%) responderam que nunca ouviram falar sobre *code smell*. Dos 50 restantes (73 - 23), a grande maioria (37,50%) pertenciam tanto ao grupo que apenas "ouviu falar deles em *blogs* ou discussões, mas não tem tanta certeza do que são" ou do grupo "tenho um entendimento geral, mas não uso esses conceitos". Assim, apenas 18% dos entrevistados indicaram ter uma compreensão boa ou forte sobre *code smell* e aplicam esses conceitos em suas atividades diárias.

Tufano *et al*. (2015) apontam para a necessidade de desenvolver uma nova geração de sistemas de recomendação voltados para o planejamento adequado das atividades de refatoração de *code smells*. Essa necessidade foi observada em um estudo empírico em larga escala conduzido sobre o histórico de *commits* de 200 projetos de código aberto, no qual o objetivo foi entender quando e por que os *code smells* são introduzidos. Os resultados do estudo fornecem várias descobertas à comunidade de pesquisa, sendo que o principal resultado é que artefatos de código que possuem *code smell* como consequência de atividades de manutenção e evolução que foram realizadas, podem introduzir novos *code smells*. Este resultado aponta a necessidade de técnicas e ferramentas destinadas a avaliar o impacto das operações de refatoração no código-fonte antes de sua aplicação real. Isso quer dizer, quantificar os *code smells* antes da refatoração pode ser uma estratégia para refatoração.

Os conhecimentos quantitativos de *code smells* possibilitam o desenvolvedor saber quais *code smells* possuem maior deficiência, pois desenvolvedores diferentes podem ter uma tendência ou padrão diferente na criação do *code smell*. No entanto, apesar de esse conhecimento não garantir que o desenvolvedor continuará a criar *code smell*, permite que, com base neste novo conhecimento específico, o próprio desenvolvedor possa criar estratégias para aprender como resolver os *code smells* que adiciona no código-fonte. Além disso, à medida que o desenvolvedor aprende como resolver seus erros, é esperado que a identificação do *code smell* pela ferramenta de análise estática de código-fonte seja cada vez menos necessária, evitando assim o retrabalho envolvido na correção da DT do *code smell*, ou seja, a refatoração, visto que a causa foi tratada.

Considerando que a DT referente ao código-fonte, especialmente a de *code smell*, compromete a qualidade do código e encarece a sua manutenção, é esperado que uma abordagem que ajude o desenvolvedor a se tornar mais consciente dos problemas no seu código, evita retrabalho. Tendo em vista o problema de pesquisa abordado, os trabalhos apresentados por Tsoukalas *et al.* (2019) e Lenarduzzi *et al*. (2019) e a lacuna observada por Pereira dos Reis *et al.* (2021), evidencia-se a necessidade de abordagens e experimentos orientados a dados que tenham foco em *code smells* criados pelo desenvolvedor, e não somente *code smells* no contexto de produto. Também se faz necessária a disponibilização do processo e dos dados, a fim de permitir que interessados reproduzam um processo que responda ao problema. Logo, esse estudo colabora com o estudo de Yamashita e Moonne (2013), no sentido de quantificar os *code smells* e apresentar essa ocorrência para os desenvolvedores que não conhecem sobre o assunto.

1.1 OBJETIVOS

O objetivo geral deste trabalho é **desenvolver um processo para auxiliar o desenvolvedor a conhecer seus comportamentos em relação à dívida técnica do tipo *code smell***.

Para atingir o objetivo geral, os seguintes objetivos específicos foram necessários:

i. Desenvolver um processo que apoie o desenvolvedor na compreensão de seus comportamentos em relação ao *code smell*;

ii. Avaliar o processo em um ambiente da indústria de software.

Referente ao objetivo específico avaliar o processo em um ambiente de indústria de *software*, a avaliação do processo dar-se-á por meio da execução do processo e coleta dos dados de *code smells* obtidos de repositório de código fonte e de ferramentas de análise estática, com o objetivo de obter dados para responder o objetivo geral. As métricas necessárias para responder atender o objetivo geral e a avaliação do processo serão tratadas no Capítulo 5.

1.2 DELIMITAÇÃO DE ESCOPO

Além dos objetivos informados é necessário entender a delimitação do escopo, esclarecendo os assuntos que não serão abordados. Assim, não faz parte do escopo deste trabalho:

i. Discutir a proposição de análise de outros tipos de DT que não sejam *code smell;*
ii. Discutir a validade dos indicadores de DT das ferramentas de análise estática;
iii. Discutir métodos de classificação de *code smell;*
iv. Discutir métodos de priorização de *code smell;*
v. Discutir métodos de detecção de *code smell;*
vi. Discutir sobre as classificações de *code smell* nas ferramentas de análise estática de código.

1.3 PROCESSO DE TRABALHO

Com o objetivo de organizar o trabalho de pesquisa, foi definido um processo contendo as fases para atingir os objetivos propostos, que foram:

- Fase 1 – Preparação da Pesquisa: fase que correspondeu à delimitação da área de estudo, coleta e análise das referências bibliográficas, delimitação do tema e estabelecimento dos objetivos, questões e proposições.

- Fase 2 – Estruturação da Pesquisa: fase de elaboração de um quadro referencial teórico, com vistas a apoiar o processo de investigação e o delineamento das conclusões. Nessa fase também ocorreu a seleção do método de pesquisa e a construção do roteiro de pesquisa e do protocolo de pesquisa.
- Fase 3 – Execução da Pesquisa: fase de investigação e aplicação do método de pesquisa.
- Fase 4 - Análise dos Resultados: fase da análise dos dados de forma agregada e as conclusões da pesquisa.

1.4 ESTRUTURA DO DOCUMENTO DA DISSERTAÇÃO

Esse trabalho está estruturado da seguinte forma:

- O Capítulo 1, aqui apresentado, visa oferecer ao leitor um panorama geral sobre o contexto no qual se insere este trabalho de pesquisa;
- O Capítulo 2 aprofunda o referencial teórico inicial descrito no Capítulo 1, focando especialmente os temas relacionados à análise estática de código, repositórios de código-fonte e descoberta de conhecimento em banco de dados;
- O Capítulo 3 apresenta a estruturação da pesquisa e posicionamento metodológico;
- O Capítulo 4 apresenta o detalhamento do processo TDMINING;
- O Capítulo 5 apresenta o desenvolvimento da pesquisa;
- O Capítulo 6 apresenta as considerações.

1.5 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo apresentou a importância dos *code smells* no desenvolvimento de *software* e a importância de o desenvolvedor conhecer seus padrões na criação deste tipo de DT. Também foram apresentados os objetivos, delimitação do escopo e a justificativa deste estudo.

CAPÍTULO 2 - REVISÃO DA LITERATURA

Eu não sou um grande programador; sou apenas um bom programador com ótimos hábitos.

Kent Beck

Kent Beck in: Martin Fowler, Kent Beck, John Brant (2012) Refactoring: Improving the Design of Existing Code.

Esse capítulo apresenta a revisão da literatura desta pesquisa. Refere-se aos principais assuntos relacionados ao tema, como mineração de dados, dívida técnica e análise estática.

2.1 QUALIDADE DE SOFTWARE

A qualidade do software é o quanto um produto está em conformidade com seu projeto, com base em requisitos funcionais ou especificações. O SWEBOK (2014) inclui qualidade como uma área de conhecimento específica, mas que não deve ser considerada de forma isolada do restante do corpo de conhecimento.

A *International Organization Standardization* (ISO) e a *International Electrotechnical Comission* (IEC), organismos normalizadores e reconhecidos no setor de *software*, se uniram para editar normas conjuntas, como a ISO/IEC 25010:2011 (2011). Essa norma, parte da família de normas ISO 25000, define qualidade de *software* como: “(...) a totalidade de características de um produto de *software* que lhe confere a capacidade de satisfazer necessidades explícitas e implícitas”. Necessidades explícitas são aquelas expressas no requisito proposto. Os requisitos determinam as condições em que o produto deve ser utilizado e seus objetivos, funções e desempenho esperado. Necessidades implícitas são aquelas que não são expressas no documento de requisitos, mas são necessárias para o usuário e devem ser levadas em consideração.

A ISO/IEC 25010:2011 e a sua antecessora, a ISO/IEC 9126, fornecem um modelo de referência para a avaliação de produto de *software* com a definição de características e subcaracterísticas de qualidade de *software*, conforme apresentado na Figura 2-1. A ISO/IEC 25010 (2011) substituiu a ISO/IEC 9126, adicionando características de segurança e compatibilidade e com pequenas alterações de estrutura.

Importante citar a ISO 9126 neste trabalho, pois foi a norma base para o método *Software Quality Assessment based on Lifecycle Expectations* (*SQALE*). Cada uma das habilidades apontadas no método SQALE está associada a uma característica ou subcaracterística da norma ISO/IEC 25010:2011. O método *SQALE* será abordado na seção 2.5 desse trabalho.

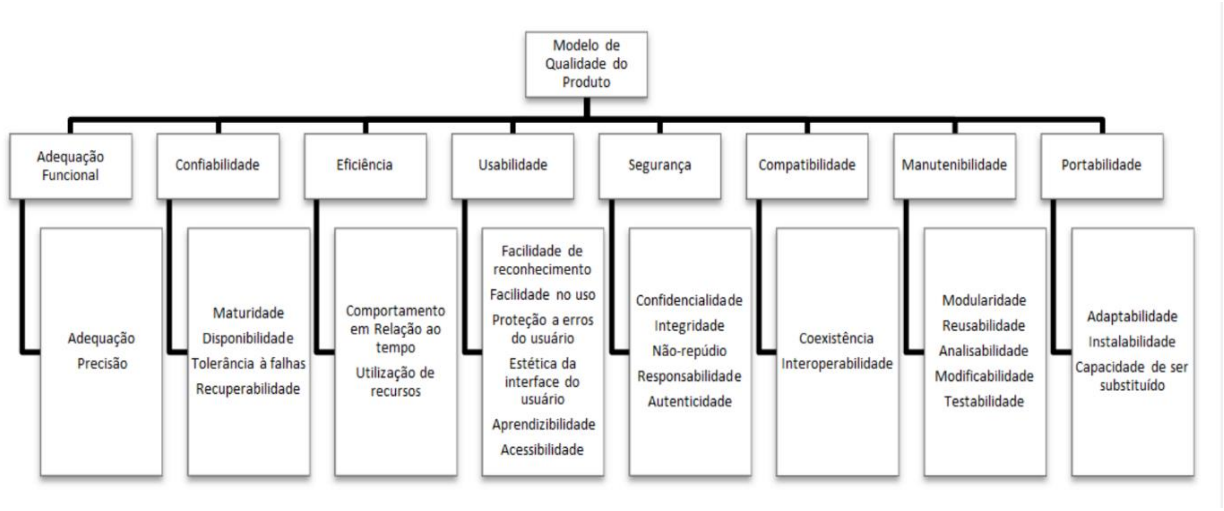


Figura 2-1. Características ISO/IEC 25010. Fonte: ISO/IEC,2011.

Um dos problemas de qualidade de *software* que existe em desenvolvimento de *software,* mas que não é tratado com detalhe nas normas ISO/IEC 9126 e ISO/IEC 25010*,* é a DT, assunto que será tratado no item 2.2 deste capítulo.

2.2 DÍVIDA TÉCNICA

Conforme dito anteriormente, a metáfora da DT foi criada por Cunningham (1992) para explicar para os *stakeholders* que não têm conhecimento técnico de produto de *software*, sobre o que chamamos em desenvolvimento de *software* de

refatoração. Cunningham afirma que entregar código imaturo é como adquirir uma dívida, pois em algum momento essa dívida deverá ser paga, em referência a uma dívida financeira. Cunningham (1992), descreve a metáfora assumindo que quando se escolhe assumir uma DT de *design* de código, ou uma arquitetura de solução ou uma abordagem que é fácil de implementar no curto prazo, essa escolha pode ter um grande impacto negativo no longo prazo. Em algum momento, o efeito da DT precisará ser tratado. No entanto, tratar o efeito, na maioria das vezes, será mais difícil e caro, dado o conjunto de alterações para evolução do *software* desde que o risco foi identificado.

Seaman e Guo (2011) apontam que não corrigir a DT pode acelerar o desenvolvimento de software no curto prazo, mas tal benefício é obtido ao custo de um trabalho extra no futuro, esse trabalho futuro é como se pagasse juros sobre a DT. Logo, se os juros da DT forem muito altos e quanto mais os juros se acumularem e quanto mais o tempo passar, torna-se difícil pagar a DT e ter controle sobre tal dívida.

Avgriou *et al.* (2016a) definem a DT como uma coleção de construções de *design* ou implementação que são convenientes no curto prazo, mas estabelecem um contexto técnico que pode tornar as mudanças futuras mais caras ou impossíveis. Essa definição ocorreu no seminário "*Managing Technical Debt, a Dagstuhl Seminar*" que reuniu pesquisadores, profissionais e fornecedores de ferramentas da indústria interessados nos fundamentos teóricos da dívida técnica e como gerenciá-la desde a medição e análise até a prevenção. Antes do seminário, os organizadores criaram um blog[2] onde os participantes puderam postar posições e iniciar discussões para facilitar a coleta de ideias sobre dívida técnica.

Seaman e Guo (2011) definem de forma mais genérica DT como uma metáfora para artefatos imaturos, incompletos e inadequados. Eles discutem também que tarefas pendentes de resolução de DT, trazem o benefício de permitir entregar o *software* mais rápido, apesar de com menor qualidade. No entanto, isso gera juros da DT, que em algum momento deverão ser pagos. Os autores ainda afirmam que

[2] Disponível em https://mtd2016dagstuhldotorg.wordpress.com/

gerenciar a DT é mais complicado que gerenciar uma dívida financeira, devido às incertezas envolvidas.

Buschmann (2011) também assemelha a DT com uma dívida financeira. Ele coloca como exemplo um caso no qual a decisão de pagar a DT naquele momento não era o ideal e, por isso, optaram apenas em pagar os juros da dívida, pois naquele cenário, as decisões comerciais tinham prioridade em relação à DT, principalmente pela dívida gerada intencionalmente. O autor ainda observa que, às vezes, corrigir a dívida pode ser pior que mantê-la.

Kruchten *et al*. (2012) concordam que a causa da DT pode estar relacionada com a pressão da entrega devido à comercialização de um produto, mas também pode estar relacionada com causas como o descuido, a falta de educação, os processos ruins, a verificação não sistemática da qualidade ou a incompetência básica. Os autores também reforçam que desenvolver e entregar rápido, sem tempo para um *design* de código adequado ou para refletir sobre a qualidade do código, leva alguns projetos ágeis a ter grandes quantidades de dívida de forma muito abrupta. Citam que a DT pode aumentar muito mais rapidamente do que em qualquer projeto cascata, observando que problemas de *design* de código têm a relação com o *code smell*.

McConnell (2013) classifica a DT como intencional e não intencional. A dívida intencional é aquela assumida deliberadamente como, por exemplo as resoluções emergenciais de problemas que são feitas da forma que é possível no momento, mas não necessariamente da melhor forma. Já a DT não intencional é aquela que é inserida sem que a equipe tenha consciência, como um código escrito por um desenvolvedor com pouca experiência em *design* de código. McConnell (2013) ainda discorre sobre o pagamento da DT em curto e longo prazo. Geralmente as de curto prazo são aquelas DT feitas para atender necessidades imediatas, enquanto as de longo prazo são aquelas que dependem de um planejamento, geralmente estando relacionadas a um problema de arquitetura.

Holvitie *et al*. (2014) apontam que em times ágeis, arquitetura, documentação, estrutura e testes são causas de DT e também que a maior parte da causa é oriunda de implementação. Alves *et al.* (2014) propuseram classificar a DT conforme sua

natureza, chegando a treze tipos. Um dos tipos de DT é a de *design*, que se refere à dívida que pode ser descoberta analisando o código-fonte, identificando o uso de práticas que violaram os princípios de um bom *design* orientado a objetos (por exemplo, classes muito grandes ou fortemente acopladas). *Code smells* são problemas de *design.*

Ampatzoglou *et al.* (2015) descrevem que uma das características mais dominantes da DT é a sua natureza interdisciplinar, pois combina elementos de engenharia de *software* e teoria financeira. Os autores apresentam uma abordagem para decisão se o custo de correção da DT é maior que o benefício de resolvê-la.

Kruchten, Nord e Ozkaya (2019) citam que a melhor forma de lidar com a DT para identificar as causas do endividamento é a conscientização. Após isso, o próximo passo é gerenciar essa dívida, o que envolve descrever tarefas relacionadas à dívida em uma lista de pendências comuns durante a liberação e planejamento de iteração, juntamente com os respectivos itens a fazer. Por fim, os autores afirmam que a DT também está relacionada à experiência dos times.

Rios *et al.* (2018) apresentam os primeiros resultados de uma pesquisa sobre as causas e efeitos mais comuns da DT e algumas, dentre as dez causas apontadas na pesquisa, é a falta de conhecimento e de experiência. Rios *et al.* (2019) apontam as principais causas e efeitos de DT em projetos ágeis, onde 21,9% dos problemas são relativos a desenvolvimento, no qual o *code smell* está relacionado a não adoção de boas práticas (7,8%) + *Bad Design* (4,3%), e efeitos do desenvolvimento, necessidade de refatoração (1,3%), baixo reuso de código (2,6%) e código ruim (1,3%).

Besker *et al.* (2017) realizaram uma pesquisa online na Web que forneceu dados quantitativos de 258 participantes e entrevistas de acompanhamento com 32 profissionais de software industrial. Apresentam como resultado da pesquisa que em média 36% de todo o tempo de desenvolvimento seja desperdiçado devido à Dívida Técnica; sendo que a maior parte do tempo é desperdiçada em entender e/ou medir a Dívida Técnica.

Stripe (2018) apresenta um estudo realizado com milhares de executivos e desenvolvedores em mais de 30 setores para examinar como as empresas empregam

o talento dos desenvolvedores. O estudo aponta que quando os engenheiros de *software* trabalham no que agrega mais valor, as empresas prosperam. Mas quando eles são alocados em posições de pouca agregação de valor, os autores estimam que as empresas perdem anualmente 300 bilhões de dólares com a produtividade de desenvolvedores. Segundo o estudo, os desenvolvedores passam mais de 17 horas semanais envolvidos com problemas de manutenção (13,5h com DT + 3,8h com código com baixa qualidade), como depuração e refatoração. Cerca de um quarto desse tempo é dedicado à correção de código de baixa qualidade, considerando uma carga horário de trabalho semanal média de 41,1h. A DT pode tornar empresas menos competitivas, visto que perdem tempo significativo em atividades para correção de DT ao invés de poder concentrar em entrega de valor e inovação.

Como já citado anteriormente, a DT é um problema de qualidade. Um dos tipos de DT que pode ser encontrado em *software* são as oriundas de implementação. O problema de implementação mais conhecido em desenvolvimento de *software* é o *code smell*, assunto que será tratado no item 2.4 deste capítulo.

2.3 REFATORAÇÃO

O termo refatoração foi apresentado por Opdyke (1992) e é uma variante de reestruturação em programação orientada a objetos, na qual sobre um sistema programado em linguagem orientada a objetos é realizada uma melhoria interna no código-fonte, sem alterar o comportamento externo.

A refatoração é uma atividade de manutenção de *software* e é geralmente considerada como qualquer mudança feita para um *software* depois que ele foi lançado para produção. A necessidade de refatorar indica a presença *code smell*. De acordo com o SWEBOK, de 40 a 60% das tarefas de manutenção são dedicados a compreender o código que está sendo mantido. Essa compreensão está diretamente associada aos problemas de *design* de código e *code smells*.

Para Fowler (2002), refatoração é o processo de alterar um sistema de software de forma que não altere o comportamento externo do código, mas que melhore sua estrutura interna. É uma maneira disciplinada de limpar código e que minimiza as

chances de introdução de *bugs*. Fowler *et al.* (2019) descrevem técnicas de refatoração sobre os tipos de *code smells* e como resolver.

Refatorar é considerado pagar uma DT e, comparado com uma dívida financeira, requer o pagamento de juros. A necessidade de refatorar o código dar-se-á devido à existência de código com baixa qualidade, muitas vezes devido à pouca experiência dos desenvolvedores, cronogramas apertados, mal entendimento de regras de negócio em requisitos de sistemas e até um processo de desenvolvimento de *software* falho. Ao refatorar um *code smell*, além de pagar uma DT, o desenvolvedor melhora a qualidade do *software*. Um exemplo de trecho de código em que foi realizada a refatoração pode ser observado na Figura 2-3, onde no trecho de código original da Figura 2-2 tem-se a declaração de uma variável não utilizada.

```
function numberOfMinutes($hours) {
  $seconds = 0;   // seconds is never used
  return hours * 60;
}
```

Figura 2-2. Exemplo de código com *code smell* antes de refatoração. Fonte: o Autor.

```
function numberOfMinutes($hours) {
  return hours * 60;
}
```

Figura 2-3. Exemplo de código sem *code smell* depois de refatoração. Fonte: o Autor.

Para Sjoberg *et al.* (2013), refatoração é uma atividade demorada e propensa a erros e deve-se investigar quando sua aplicação é benéfica. O processo de refatoração inclui também a correção das DTs de *code smell*. Por exemplo, em programação orientada a objetos, um método de uma classe com muitos condicionais ou laços de repetição ou duplicação de código.

Suryanarayana *et al*. (2015) apontam que *code smell* pode ser tanto relacionado a *design debt* como a *code debt* e, ainda, que podem ser verificados por meio de ferramenta de análise estática, pois essas ferramentas permitem identificar violações de regras de *design* de código e identificar estilo de codificação inconsistente. Os autores apontam que muitos desenvolvedores desconhecem os *code smells* que

podem se infiltrar no *design* com o tempo, e que são indicativos de má qualidade estrutural, além de contribuir para o endividamento técnico. DT, que pode ser resolvida por refatoração oportuna, mas que, no entanto, quando os desenvolvedores não têm consciência da refatoração e dos *code smells* e não realizam a refatoração, a DT se acumula com o tempo.

Tufano *et al*. (2017) realizaram um estudo empírico em larga escala com 200 projetos de código aberto e teve como objetivo entender quando e por que os *code smell* são introduzidos, o resultado da pesquisa contradizem a sabedoria comum, mostrando que a maioria dos *code smells* são introduzidas quando um artefato é criado e não como resultado de sua evolução. Ao mesmo tempo, 80% dos *code smells* sobrevivem no sistema. Além disso, entre os 20% das instâncias removidas, apenas 9% são removidos como consequência direta das operações de refatoração.

Identificar em qual código-fonte deve ser realizada a refatoração é uma atividade propensa a erros quando feita de forma manual, e que também depende da experiência do profissional que está analisando determinado código. Gerenciar os problemas de código é outra atividade que demanda tempo. Uma das formas de gerenciar, priorizar e quantificar as DTs de código é por meio do método SQALE (item 2.5 deste capítulo). A forma de automatizar a identificação de *code smell* é outro problema de código que pode ocorrer com o auxílio de ferramentas de análise estática de código (item 2.6 deste capítulo).

2.4 CODE SMELL

O termo *code smell* foi cunhado por Kent Beck, em 1990, e apresentado por Martin Fowler, em 1999, na primeira versão do livro *Refactoring*. O termo significa que *code smells* não são bugs; não são tecnicamente incorretos e não impedem o funcionamento do *software*, no entanto, afetam a qualidade do *software*.

Wake (2003) aponta que *code smells* são sinais de alerta sobre possíveis problemas no código. Nem todos os *code smells* indicam um problema, mas a maioria merece ser analisado e uma decisão para correção precisa ser tomada. Wake (2003) ainda aponta que algumas pessoas não gostam do termo *code smells* e preferem falar sobre problemas potenciais ou falhas.

Para Kruchten *et al.* (2012), *code smells* são um dos tipos de DT de *design* de código que pode se encontrar em desenvolvimento de *software*. O termo *code smell* foi abordado por Riel (1996), em um dos primeiros livros sobre problemas de código, o qual cunhou o termo “falha de *design*”, apresentando formas para melhoria de código em projetos de sistemas orientados a objetos.

Fowler *et al*. (2019) definem que *code smell*s são estruturas de código que indicam a violação de princípios de *design* e programação. Também que *code smells* indicam problemas de *software*, que causam atividades futuras de manutenção por meio de refatoração. Fowler *et al.* (2019) definem alguns tipos de *code smell*: nome misterioso, código duplicado, função longa, lista longa de parâmetros, dados globais, dados mutáveis, alteração divergente, cirurgia com rifle, inveja de recursos, agrupamento de dados, obsessão por primitivos, *switches* repetidos, laços, elemento ocioso, generalidade especulativa, campo temporário, cadeias de mensagens, intermediário, trocas escusas, classe grande, classes alternativas com interfaces diferentes, classes de dados, herança recusada e comentários.

Na lista de conceitos da *SonarSource*[3] o *code smell* é definido como um problema relacionado à manutenção no código. Deixar o código-fonte como está significa que na melhor das hipóteses, os desenvolvedores terão mais dificuldade nas alterações de código que precisarão fazer. Na pior das hipóteses, eles ficarão confusos com o estado do código e introduzirão erros adicionais ao fazerem alterações.

Martin (2014) exemplifica alguns tipos de *code smells* que podem ser, por exemplo, uma duplicação de código, um método extenso, uma classe Deus, um bloco *try-catch* calado, uma variável declarada não utilizada ou código morto.

Um exemplo de *code smell é o (Unused local variables should be removed) - RSPEC-1481*[4] que pode ser observado na Figura 2-4, onde a variável *$seconds* é declarada e inicializada não utilizada.

---

[3] Conceito de *code smell* apresentado pelo *SonarQube*: https://docs.sonarqube.org/latest/user-guide/concepts/

[4] https://rules.sonarsource.com/php/type/Code%20Smell/RSPEC-1481

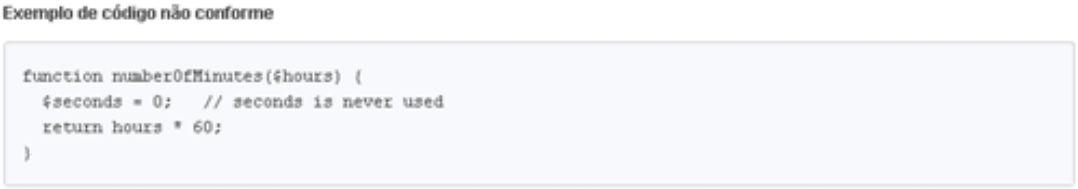


Figura 2-4. Exemplo de *code smell* variável não utilizada. Fonte: SonarSource, 2020.

Outro exemplo de *code smell* é “As estruturas de controle devem usar chaves” (RSPEC-121 [5]), conforme Figura 2-5.

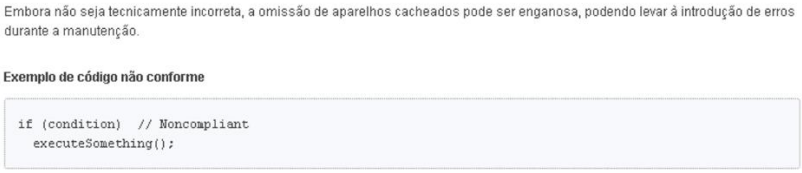


Figura 2-5. Exemplo de *code smell* “As estruturas de controle devem usar chaves”. Fonte: SonarSource, 2020.

Tanto os *code smells* apresentados por Martin (2014), quanto por Fowler *et al.* (2019), podem ser observados nas regras das muitas ferramentas de análise estática de código.

Martin (2017), Martin (2014), Martin (2011) e Fowler *et al.* (2019) ainda abordam práticas de *design* de código, código limpo e arquitetura de *software* que possibilitam ao desenvolvedor, além de melhorar o código-fonte por meio do conhecimento sobre os tipos de *code smells* e como evitá-los, também práticas de arquitetura, por meio de princípios de *design*. Martin (2017), descreve ainda sobre a diferença dos termos *design* e arquitetura:

Para Martin (2017, p.4), adaptado:

> “(..) Quais as diferenças entre os dois? (..) Nenhuma diferença (...) arquitetura é usada no contexto de algo em um nível mais alto e que independe dos detalhes dos níveis mais baixos, enquanto *design* sugere estruturas e descrições de nível mais baixo. (...) Vejo que os pequenos detalhes (nível baixo), servem de base para as descrições de nível mais alto. (...) Ambos níveis pertencem ao mesmo todo (...) que definem e moldam o sistema”.

[5] https://rules.sonarsource.com/php/type/Code%20Smell/RSPEC-121

Rios, Mendonça e Spínola (2018) apontam que DTs relacionadas a código tendem a causar efeitos que são sentidos mais rápido pela equipe de desenvolvimento. Ressaltam a importância de evitar a criação de novos *code smells*, pois à medida que os sistemas crescem (ganham mais funcionalidades, mais linhas de código, classes, arquivos de código), existe a tendência que, se não houver preocupação com o *code smell,* quanto mais o tempo passa, pior será executar a refatoração.

2.5 SQALE

Software Quality Assessment based on Lifecycle Expectations (SQALE) é um método e modelo de qualidade genérico, independente de ferramenta, que tem o objetivo de produzir indicadores para gestão da DT de *design*. A partir dessas informações, é possível saber qual a qualidade do código-fonte entregue pelos desenvolvedores. O método foi definido pela empresa Inspearit (anteriormente DNV ITGS France), sobre qualidade de software, principalmente de código-fonte.

Letouzey *et al*. (2012) afirmam que DT e qualidade de código são conceitos conectados. Eles apresentam o método SQALE, que é o método para análise da DT em código estático, o qual afirmam ser um método preciso para estimar a dívida. O SQALE tem o objetivo principal de entregar aos desenvolvedores qual a qualidade do código-fonte e identificar qual é a dívida de *design* acumulada pelo projeto. O método entrega a qualidade do código-fonte por meio de indicadores. Para Letouzey (2012), o SQALE avalia a distância da conformidade dos requisitos, considerando o custo de remediação necessário para deixar o código-fonte em conformidade. Analisar a qualidade de um código-fonte faz parte da avaliação da distância entre seu estado atual e sua meta de qualidade esperada. No método SQALE para avaliar a conformidade, utiliza-se quatro conceitos: modelo de qualidade, modelo de análise, índices e indicadores. O detalhamento destes conceitos pode ser observado no APÊNDICE A – MODELO DE QUALIDADE DO METODO SQALE.

2.6 ANÁLISE ESTÁTICA DE CÓDIGO-FONTE E SUAS FERRAMENTAS

Ferramentas de análise estática de código-fonte auxiliam na identificação da DT de *code smell* que, na maioria das vezes, estão versionados em repositórios de código.

Deming (2000), em um aspecto mais amplo, cita que realizar inspeção não melhora nem garante a qualidade, dado que a inspeção ocorre tarde demais, após o produto ou serviço já ter sido concebido. A qualidade, seja boa ou ruim, já está contida no produto. Para Deming, a inspeção pode ser útil para coletar dados sobre o processo. Especificamente, para usar esses dados para observar se um processo/produto saiu do controle e se existe uma causa especial para ser investigada. Deming (2000) ainda observa que usar esses dados para avaliar o sucesso ou fracasso de algo é uma tentativa de melhoraria contínua. Essa melhoria contínua pode ser alcançada utilizando o ciclo PDSA, introduzido por Deming, e que é semelhante ao PDCA criado por Walter A. Shewart. A diferença entre ambos os ciclos está na ação de checar x estudar. No PDCA, a sigla significa: *Plan* (Planejar) – *Do* (Fazer) – *Check* (Checar/Verificar) – *Act* (Agir). Já no PDSA ela significa: *Plan* (Planejar) – *Do* (Fazer) – *Study* – (estudar) – *Act* (Agir).

O ato de estudar do PDSA engloba uma ação muito mais abrangente do que o simples ato de checar, indo além da simples verificação do processo/produto, mas aprofundando no estudo, análise e adequação do processo para atingir a melhoria contínua. Nesse sentido, as ferramentas de análise estática de código justamente permitem a inspeção dos dados referente aos problemas de código.

Ferramentas de análise estática de código-fonte auxiliam na identificação da DT de *code smell*, *bugs* e vulnerabilidades, por meio de inspeção do código-fonte. Essas ferramentas são centradas em produto de *software*, com o objetivo principal de fornecer informações para que os desenvolvedores resolvam os problemas de código identificados na análise. A execução deste tipo de ferramenta pode ser realizada a cada nova versão de código-fonte submetido ao repositório de código.

Algumas dessas ferramentas de análise estática quantificam os problemas de código-fonte com o intuito de demonstrar o esforço total para solução dos problemas.

Existem diversas ferramentas[6] de análise estática, sendo o SonarQube uma das mais conhecidas na indústria de *software*. SonarQube, fornecida pela empresa SonarSource, é a ferramenta de análise estática de código-fonte mais comum, sendo utilizada por mais de 120 mil usuários, com suporte a mais de 25 linguagens de programação, e com mais de 6 mil empresas utilizando a versão gratuita do produto. Por outro lado, 200 mil empresas utilizam produtos da SonarSource, sendo que 58[7] das empresas consideradas no *ranking Fortune* 100 utilizam o SonarQube.

Conforme já abordado, *code smells* são considerados problemas relacionados à manutenção no código que diminuem a legibilidade e modificabilidade do código. É importante observar que o termo *code smell,* utilizado no SonarQube, não se refere exatamente ao mesmo termo comumente conhecido e definido por Fowler *et al.* (2019). No SonarQube, o termo se refere a um conjunto diferente de avisos, pois a ferramenta trata alguns dos *code smell* apontados por Fowler *et al.* (2019) como regras, sem se limitar apenas aos propostos por ele.

Linter é outra ferramenta de análise estática de código, usada também para sinalizar erros de programação, *bugs*, erros estilísticos e construções suspeita. É um componente integrado ao ambiente de desenvolvimento. *Linters* não quantificam e não identificam padrões e tendências na criação de problemas de código, apenas realizam o alerta em tempo de desenvolvimento, podendo ou não o desenvolvedor aceitar o alerta apontado pelo *linter*. É importante observar que dependendo da linguagem de programação e/ou ambiente de desenvolvimento, não existe *linter*.

O primeiro *linter* criado e abordado na literatura foi o de Johnson (1978) para linguagem de programação C. Não foram identificadas na literatura pesquisas que apontem se os desenvolvedores têm o hábito de utilizar *linters* e se seguem as recomendações desta ferramenta.

---

[6] Lista de ferramentas de análise estática: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis e https://owasp.org/www-community/Source_Code_Analysis_Tools

[7] Dados extraídos do site https://www.sonarsource.com/ . SonarSource é a empresa responsável pelo SonarQube.

Ainda sobre o SonarQube, Guaman et al. (2017) apontam o software como uma ferramenta completa para análise estática de código, pois abrange mais de 20 *scanners* de linguagem de programação e, principalmente, por atender ao método *SQALE*. Outra característica apontada é que o SonarQube possui versão gratuita.

O *code smell* faz parte do índice de manutenibilidade[8] do SQI, abordado no SQALE. Esse índice é disponibilizado no SonarQube por meio da soma do esforço de remediação do *code smell*. Tanto a informação quantitativa de *code smell* como a quantidade de esforço necessário em minutos está disponível no banco de dados do SonarQube.

Um exemplo do painel de entrada do SonarQube, representado na Figura 2-6, mostra a quantidade de *code smell* e a quantidade de dias/horas (*debt*) de esforço para solução dos problemas. Também é apresentado pelo SonarQube a quantidade de *bugs,* vulnerabilidade, cobertura de testes unitários e duplicação de código.

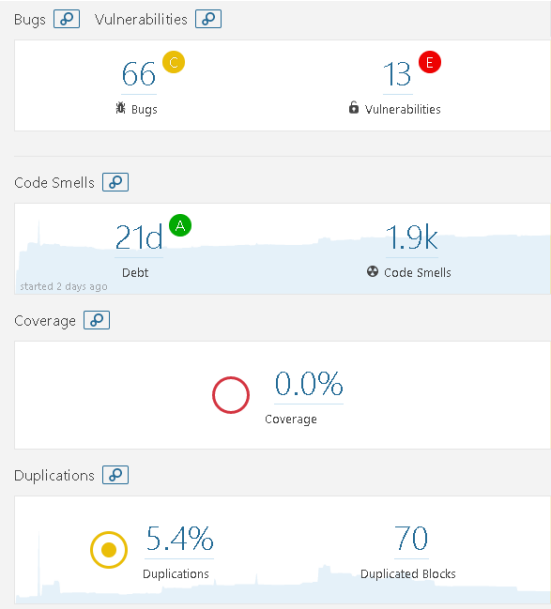


Figura 2-6. Exemplo de painel de visualização de dívida técnica no SonarQube. Fonte: o Autor.

Um exemplo de *code smell* observado no SonarQube é o " *switch statements should have default clauses*". No qual é abordado pela SonarSource na regra RSPEC-

[8] Disponível em: https://docs.sonarqube.org/latest/user-guide/metric-definitions

131[9], a mesma regra de *code smell* é atribuída pela CWE-478[10]: "*Missing Default Case in Switch Statement"* e observada na Wiki de práticas de codificação do Instituto de Engenharia de Software da Universidade Carnegie Mellon na regra MSC01-C[11] "*Strive for logical completeness*". Ou seja, as regras de *code smell* apresentadas pela ferramenta SonarQube, algumas vezes pode possuir uma descrição diferente das convenções de boas práticas de programação, no entanto, são aderentes as convenções.

A empresa *BiteGarden* disponibiliza um *plugin*[12] para o SonarQube que detalha de forma visual os dados conforme o método SQALE (Figura 2-7). Os dados apresentados por esse *plugin* são obtidos do banco de dados do SonarQube. No *site*[13] do método SQALE são informadas outras ferramentas e *plugins* que atendem o método.

As análises estáticas de código realizadas pelo SonarQube acontecem por meio de *scanners* que têm o objetivo de avaliar e inspecionar regras de qualidade de *software* e segurança por linguagem de programação. Assim, o SonarQube, além de ser uma ferramenta para análise estática da qualidade de código, também é uma ferramenta de *Static Application Security Testing (SAST),* pois tem o objetivo de analisar o código-fonte de uma aplicação a fim de encontrar problemas de segurança.

Cada um dos *scanners* do SonarQube tem suas próprias regras para determinar se um código possui problema. O objetivo do *scanner* é identificar *code smell*, mas também *bugs*, código duplicado e vulnerabilidades. Todas essas informações são persistidas ao término da análise estática, no banco de dados do SonarQube pelo *scanner* por meio de uma *Application Programming Interface* (API).

---

[9] Disponível em https://jira.sonarsource.com/browse/RSPEC-131, Acesso em: 18/10/2021.

[10] Disponível em http://cwe.mitre.org/data/definitions/478.html. Acesso em: 18/10/2021.

[11] Disponível em https://wiki.sei.cmu.edu/confluence/display/c/MSC01-C.+Strive+for+logical+completeness. Acesso em: 18/10/2021.

[12] Disponível em: https://www.bitegarden.com/sonarqube-sqale, Acesso em: 06/12/2020

[13] Disponível em: http://www.sqale.org/tools , Acesso em: 06/12/2020

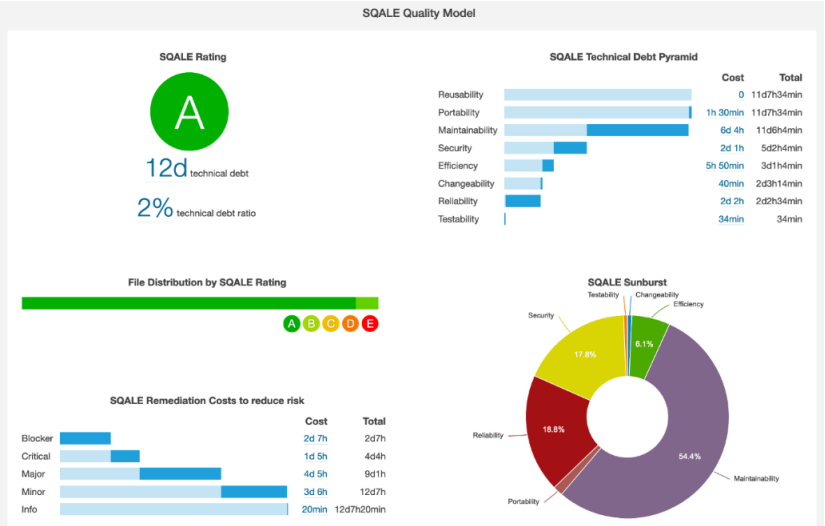


Figura 2-7. Exemplo de dashboard SQALE do plugin *Bitegarden.* Fonte: o Autor.

No entanto, tanto o SonarQube como o *plugin Bitgarden*, apresentam em sua interface gráfica apenas as informações da última versão da análise realizada por produto. Não é possível visualizar os detalhes da análise estática anterior. Os dados da análise estática realizada anteriormente são disponibilizados em banco de dados relacional por um período de 30 dias e, após esse período, os dados são excluídos, visto que o *software* SonarQube é uma ferramenta de análise estática e não um *bigdata* para análise estática de código.

Sharma *et al.* (2016) apresentam o Designite, uma ferramenta de avaliação e inspeção de qualidade de *design* de *software*. A ferramenta analisa o código em linguagem de programação C# e identifica problemas de qualidade de *software*. A ferramenta detecta um conjunto abrangente de *code smells* de arquitetura, *design* e implementação e fornece mecanismos, como análise de métricas detalhadas, matriz de estrutura de dependência, análise de tendências e mapas de distribuição de *code smells.* Designite tem por objetivo ajudar a reduzir a DT e melhorar a manutenção do *software*, sendo capaz de detectar vários *code smell* em grandes bases de código. Sharma *et al.* (2016) planejaram realizar uma extensa avaliação para computar métricas, como *recall* e precisão da ferramenta, e estender a ferramenta para oferecer suporte a recursos como análise diferencial e análise de tendências. A ferramenta permite ainda exportar os modelos de *smell* para o SonarQube usando o *plugin*

fornecido pelo próprio Designite. O Designite também tem o objetivo de analisar os problemas de código da perspectiva de produto e não do desenvolvedor.

O TeamScale, uma ferramenta de análise estática não gratuita, foi apresentado por Haas et al. (2019). Apesar do TeamScale possuir diversos tipos de relatórios e informações gráficas, não apresenta informações sobre padrões de code smell criados do ponto de vista do desenvolvedor.

Thakur *et al.* (2020) apresentam uma plataforma aberta para visualização de qualidade de código e classificação de qualidade, pois apontam que, embora repositórios de código-fonte estejam disponíveis em plataformas como o GitHub, suas informações detalhadas de qualidade de código não estão disponíveis prontamente. Os autores apontam que pesquisadores de engenharia de *software* geralmente precisam selecionar um conjunto de repositórios de alta qualidade.

Apesar da qualidade do código ser uma preocupação importante para a seleção do repositório, a falta dessas informações faz com que os pesquisadores dependam de alternativas como o número de problemas e o número de estrelas associadas a repositórios (*rating* do GITHUB, por exemplo). Outra observação de Thakur *et al.* (2020) é que os profissionais esperam formas visuais fáceis de usar para avaliar a qualidade de seus projetos durante a evolução da base de códigos, sem que seja necessário fazer um esforço considerável. Por isso, eles propõem uma plataforma aberta, a QScored, para preencher a lacuna tanto para pesquisadores quanto para profissionais.

Essa plataforma hospeda informações detalhadas de análise de qualidade de código para mais de doze mil repositórios. Contendo mais de 200 milhões de linhas de código, a plataforma calcula a pontuação de qualidade e atribui um *ranking* relativo dos repositórios hospedados com base em odores (*smells*) de arquitetura, *design* e implementação detectados, além de oferecer um conjunto abrangente de auxílios de visualização para aspectos de qualidade de código. A pontuação e classificação de qualidade computada considera *code smells* detectados em três granularidades e fornece um mecanismo conveniente para selecionar repositórios com base na qualidade do código. A limitação é que a plataforma hospeda repositórios

implementados em C# ou Java, mas a ferramenta não apresenta qualquer informação sobre padrões de *code smell* do desenvolvedor.

Alguns trabalhos empíricos já vêm sendo realizados com os dados capturados pelo SonarQube (AVGERIOU *et al*., 2016). Por exemplo, Kazman *et al.* (2015) realizaram extração de dados do SonarQube para analisar a DT de arquitetura. Já Lenarduzzi *et al.* (2019) propuseram um *dataset* que utiliza informações do banco de dados do SonarQube.

Os trabalhos de Kazman *et al.* (2015), Avgeriou *et al*. (2016) e Lenarduzzi *et al.* (2019) não apontam uma abordagem ou experimento em engenharia de software a partir dos dados do SonarQube ou outra ferramenta de análise estática que permita realizar uma análise de séries temporais e/ou regras de associação. Associações e análise temporal dos dados são de grande importância para análise estatística da qualidade de código. É por meio desses dados que é possível criar uma abordagem para auxiliar o desenvolvedor na identificação do seu comportamento em relação à criação de *code smell*.

Muitos dos trabalhos empíricos utilizam os dados da última análise estática executada pelo SonarQube. Geralmente eles comparam esses dados com dados de outros produtos. Não foram identificados na literatura experimentos e abordagens que permitam gerar conhecimento a partir de uma série histórica de análise estática de *code smells*.

2.7 REPOSITÓRIO PÚBLICO DE CÓDIGO-FONTE

O repositório de código-fonte tem o objetivo de facilitar o controle de versão de qualquer tipo de arquivo. Além das informações de data e hora da versão e autor do *commit,* os repositórios de código-fonte fornecem informações dos arquivos inseridos, excluídos e/ou alterados. Cada envio de mudança de arquivo de código-fonte para o repositório de código é, na maioria dos sistemas de controle de versão, chamado de *commit* ou *submit*. Cada envio produz uma nova versão do produto no repositório e é armazenado como uma cópia do momento.

Cada vez que um produto de *software* tem alterações de código (criar, alterar, excluir) e essas alterações são versionadas, o responsável pelo produto pode decidir

sobre como distribuir o produto, podendo gerar uma nova versão do software. Esse conjunto de alterações que origina uma versão do produto, pode conter um ou muitos *commits*. Uma nova versão é denominada *baseline*.

Os repositórios não garantem a qualidade dos produtos versionados. Os repositórios de código possuem para cada *commit,* informações básicas de arquivos inseridos, modificados, excluídos, data e hora do *commit*, autor do *commit* e diferenças da versão em relação à versão anterior.

Com a popularização de repositórios públicos de código-fonte, a comunidade de desenvolvedores ganhou uma maior visibilidade da diversidade de projetos e códigos que vêm sendo compartilhados. Códigos-fontes versionados nesses repositórios estão disponíveis para qualquer pessoa ou empresa, no entanto, em muitos repositórios ainda não há informações suficientes sobre a qualidade do produto, como cobertura de testes unitários, DT, *bugs*, vulnerabilidades e até *code smell*.

Dada a ausência de informações de qualidade de código nesses repositórios, pode subentender-se que alguns desenvolvedores, que contribuem com seus códigos, não têm conhecimento ou experiência suficiente nas melhores práticas de desenvolvimento e/ou não se preocupam com a qualidade do código-fonte, pois pode acontecer de um projeto ter um alto percentual de cobertura por teste unitário e possuir uma alta quantidade de *code smells*. Também acontece de os desenvolvedores, muitas vezes, não informar se o código-fonte foi analisado por uma ferramenta de análise estática e não apresentarem os indicadores de qualidade de código de análise estática.

O repositório de código-fonte mais comum atualmente no desenvolvimento de software é o GitHub, um serviço web gratuito que utiliza o GIT como sistema de controle de versão. Vários experimentos em engenharia de software têm considerado os projetos publicados no GitHub para diversas finalidades, tendo em vista a diversidade de linguagens de programação utilizadas, padrões de código, experiência dos desenvolvedores, qualidade de código, tamanho de projetos e/ou padrões arquiteturais.

O GitHub fornece uma API[14], a qual permite que qualquer pessoa consiga extrair informações dos repositórios, como o desenvolvedor que realizou determinado *commit*, data do *commit*, informações sobre o código-fonte versionado e sobre o *pull request*[15]. No próprio GitHub existem projetos da comunidade de código aberto que fornecem ferramentas para extração de dados do próprio controle de versão GitHub. Essas ferramentas serão apresentadas no item 2.8.

Para entender a utilização dessas ferramentas de extração de dados do GitHub é necessário entender os conceitos de descoberta de conhecimento e mineração de dados que também serão apresentados no item 2.8, devido à existência de um campo dentro da mineração de dados denominado mineração de repositórios de *software* (MSR).

2.8 KNOWLEDGE DISCOVERY IN DATABASES (KDD)

Descoberta de conhecimento em bases de dados ou KDD (do inglês: *Knowledge Discovery in Database*) é um processo que permite extrair conhecimento de informações armazenadas em um conjunto grande de dados. Entretanto, antes de introduzir o KDD é necessário entender as etapas necessárias para chegar ao DIKW (do inglês: *Data, Information, Knowledge* e *Wisdom*).

O DIKW é representado por uma Pirâmide do Conhecimento, conforme apresentado na Figura 2-8, ou seja, uma hierarquia informacional utilizada principalmente nos campos da ciência da informação e da gestão do conhecimento, onde cada camada acrescenta certos atributos sobre a anterior. Para Ackoff (1989) e GU e Zhang (2014), as camadas da pirâmide são descritas como:

- Dado: é o nível mais básico;

---

[14] *Application Programming Interface* é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por aplicativos que não pretendem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços. Uma API é um conjunto definido de mensagens de requisição e resposta HTTP, geralmente expresso nos formatos XML ou JSON.

[15] *Pull request* é quando o desenvolvedor envia uma sugestão de melhoria para o repositório para ser integrado na *branch* principal do projeto. Um *pull request* pode conter mais de um *commit*, e depende da aprovação de outro desenvolvedor que seja responsável pelo projeto para ser efetivado.

- Informação: acrescenta contexto e significado aos dados;
- Conhecimento: acrescenta a forma como usar adequadamente a informação;
- Sabedoria: acrescenta o entendimento de quando utilizá-los.

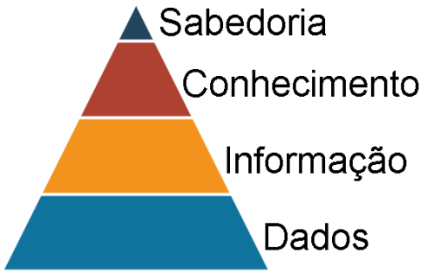


Figura 2-8. Pirâmide do Conhecimento. Fonte: GU e ZHANG (2014).

Para Ackoff (1989), um dado sozinho não tem significado total, pois ele precisa ser interpretado para que possua valor e se torne, de fato, uma informação. O conhecimento, portanto, é o conjunto e a compreensão dessas informações. Para GU e Zhang (2014), no sentido da hierarquia do DIKW, os dados são apenas fatos básicos e brutos que serão úteis apenas se deixá-los evoluir para a informação, conhecimento e sabedoria. Os autores ainda citam que os níveis de dados e informações podem usar métodos como mineração de dados, mineração de texto, mineração de *web* e ferramentas como banco de dados, armazenamento de dados e sistema de informações gerenciais para que sejam úteis. Para a forma de evoluir o dado em conhecimento pode-se usar o KDD.

Para Goldschmdt e Passos (2005), o KDD é uma área multidisciplinar que envolve estatística, inteligência computacional, aprendizagem de máquina, reconhecimento de padrões e banco de dados. Eles também citam que a busca efetiva pelo conhecimento ocorre na etapa de mineração de dados.

O termo KDD foi apresentado por Fayyad *et al*. (1996) no primeiro *workshop* de KDD, em 1989, onde foi enfatizado que o produto final do processo de descoberta em banco de dados seria o conhecimento. Fayyad *et al.* (1996) descrevem o processo de

descoberta de conhecimento em base de dados em etapas, conforme Figura 2-9, na qual é possível observar o item conhecimento como sendo a saída do processo.

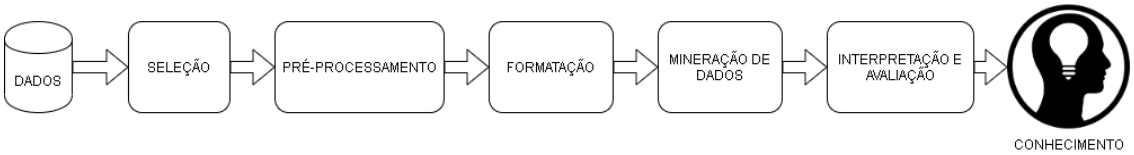


Figura 2-9. Etapas do processo de descoberta de conhecimento em base de dados. Fonte: Fayyad *et al*. (1996).

O processo de KDD consiste das seguintes etapas:

- **Seleção**: Nesta etapa é realizado um entendimento do domínio da aplicação ou problema a ser resolvido. Essa etapa entrega os dados alvo ou de interesse para a execução do processo de descoberta de conhecimento em banco de dados.
- **Pré-processamento**: Nesta etapa são realizadas ações para remover registros duplicados, o tratamento de campos em branco ou nulos, entre outras funções que tenham o objetivo de tornar os dados consistentes. Esta etapa entrega os dados processados prontos para transformação.
- **Transformação**: Esta etapa visa reduzir o número de variáveis encontradas, bem como encontrar características que melhor representem os dados, reduzindo assim as variações. Esta etapa entrega os dados formatados.
- **Mineração de dados**: Nesta etapa a finalidade é explorar o grande conjunto de dados para obter algum tipo de resultado, com o principal objetivo de gerar conhecimento e encontrar padrões.
- **Interpretação e avaliação**: Nesta etapa o objetivo é o modelo, no qual são aplicados os dados que não foram utilizados na fase de treinamento ou mineração. A validação pode ser feita por meio de métodos estatísticos e/ou passar pela avaliação dos profissionais de negócio.

2.8.1 Mineração de dados

Para Goldschmdt e Passos (2005), a Mineração de Dados é a principal etapa do processo de KDD. Neste momento ocorre a busca efetiva por conhecimentos novos e úteis a partir dos dados. Diversos autores referem-se à mineração de dados e ao processo de KDD de forma errônea como se fossem sinônimos, pois eles entendem que o conhecimento pode ser interpretado como o conjunto de padrões que podem ser formulados a partir do relacionamento entre os dados e as informações.

A execução da etapa de mineração de dados compreende a aplicação de algoritmos sobre os dados, procurando encontrar conhecimento. Estes algoritmos são fundamentados em técnicas que procuram explorar os dados de forma a produzir modelos de conhecimento. Para Goldschmdt e Passos (2005), a forma de representação de um modelo de conhecimento depende diretamente do algoritmo de mineração de dados utilizado.

Entre as tarefas de mineração de dados, têm-se: descoberta de associação, classificação, regressão, regras de associação, clusterização (*Clustering*) e sumarização, (REZENDE, 2003) e (LAROSE, 2005). As tarefas podem ser adaptadas ou compostas, originando novas. As tarefas voltadas à extração de padrões podem ser agrupadas em atividades preditivas ou descritivas. Para Rezende (2003), a escolha da tarefa deve ser feita de acordo com os objetivos para a solução a ser encontrada.

Rezende (2003) e Larose (2005) apresentam as principais atividades e tarefas de mineração de dados e suas características no sentido das funções possíveis de serem executadas para a resolução de um determinado problema.

Atividades Preditivas:

As tarefas preditivas de mineração de dados vêm com um modelo do conjunto de dados disponível que é útil para prever valores desconhecidos ou futuros de outro conjunto de dados de interesse.

a. Classificação: A classificação deriva um modelo para determinar a classe de um objeto com base em seus atributos. Uma coleção de registros estará disponível, sendo cada registro com um conjunto de atributos. Um dos

atributos será o atributo de classe. O objetivo da tarefa de classificação é atribuir um atributo de classe a um novo conjunto de registros com a maior precisão possível. A classificação consiste em predizer um valor categórico, por exemplo, se um cliente é um bom ou mal pagador, logo um valor discreto.

b. Regressão: é similar à tarefa de classificação. Utilizado para poder aprender com os dados históricos para predizer casos semelhantes futuros a partir dos dados históricos. Na regressão, o atributo que será predito consiste em um valor contínuo, como por exemplo, predizer o lucro ou perda em um empréstimo.
c. Series Temporais: é uma sequência de eventos em que o próximo evento é determinado por um ou mais dos eventos anteriores. A série temporal reflete o processo que está sendo medido e há certos componentes que afetam o comportamento de um processo. A análise de série temporal inclui métodos para analisar dados de série temporal a fim de extrair padrões, tendências, regras e estatísticas úteis. Séries temporais podem ser combinadas com predição, por exemplo, para a previsão do mercado de ações.

Atividades Descritivas

As tarefas descritivas têm como objetivo a descoberta de comportamentos intrínsecos ao conjunto de dados, considerando que estes dados não apresentam classe específica.

a. Associação: tem o objetivo de buscar por itens que ocorram de forma frequente e simultânea em transações na base de dados. Também se caracteriza quanto à existência de conjunto de atributos de registros e implica na existência de algum outro conjunto distinto.
b. Clusterização (*Clustering*): Utilizado para identificar objetos de dados semelhantes entre si com o objetivo de criar subconjuntos. A semelhança dos objetos no subconjunto pode ser decidida com base em vários fatores, como comportamento de compra, capacidade de resposta a determinadas ações, localizações geográficas e assim por diante. Essas informações do grupo serão úteis para entender melhor o subconjunto analisado.

c. Sumarização: Utilizado para resumir um conjunto de dados relevantes, o qual resulta em um conjunto menor, fornecedor de informações agregadas dos dados. Informações resumidas de alto nível podem ser úteis para uma análise detalhada do comportamento em determinado conjunto de dados.

Observando as atividades e tarefas de mineração de dados é possível que algumas delas possam servir de meio para responder a algumas perguntas do desenvolvedor em relação aos *code smells*, por exemplo:

- Sumarização: Qual o *code smell* que o desenvolvedor tem maior tendência em criar? Qual o total de tempo para remediação do *code smell* criado pelo desenvolvedor?
- Associação: Qual o *code smell* mais frequente criado pelo desenvolvedor? A criação de um determinado tipo de *code smell* está associada a outro tipo de *code smell*?
- Médias móveis e Séries Temporais: É possível verificar a tendência de criação de *code smell*?

As respostas a essas perguntas podem auxiliar no entendimento da criação dos *code smells* do desenvolvedor, no entanto, para isso é necessário entender como responder a essas perguntas. Para isso é necessário entender a aplicação dos algoritmos e técnicas em cada uma dessas tarefas de mineração de dados. Essas respostas podem ser obtidas por meio de pacotes de estatística computacional conforme citado por Tukey (1977).

Tukey (1977) cita que os pacotes de estatística computacional empregam a análise exploratória de dados com o objetivo de utilizar técnicas gráficas e quantitativas. A finalidade é obter informações ocultas na sua estrutura, descobrir variáveis importantes em suas tendências, detectar comportamentos anômalos do fenômeno, testar hipóteses, escolher modelos e determinar o número ótimo de variáveis.

Sumarização

A sumarização é o principal conceito em mineração de dados que envolve técnicas para encontrar uma descrição compacta de um conjunto de dados. Os métodos de sumarização simples, como tabular a soma, contagem, média e os desvios padrão, são frequentemente aplicados para análise exploratória e visualização de dados e automação de geração de relatório.

Por exemplo, na Tabela 2-1, são apresentados todos os *code smell* não resolvidos pelos desenvolvedores, como nomenclatura fictícia de *code smell* R1 e R2. Com base nesses dados hipotéticos, queremos saber qual desenvolvedor tem o maior tempo de remediação. Para isso, é necessário somar os tempos de remediação, agrupando por desenvolvedor, e o resultado desta soma pode ser observado na Tabela 2-2.

Tabela 2-1. Exemplo de tempo de remediação de *code smell* por desenvolvedor. Fonte: o Autor.

Code Smell	Desenvolvedor	Tempo de Remediação (min)	Data
R1	A	5	01/01/2020
R1	A	5	02/01/2020
R2	B	3	02/01/2020
R2	A	3	03/01/2020
R1	C	5	03/01/2020
R2	C	3	03/01/2020

Tabela 2-2. Exemplo de sumarização por tempo de remediação. Fonte: o Autor.

Desenvolvedor	Soma tempo de remediação (min)	Contagem de *code smell*
A	13	3
B	3	1
C	8	2

Na Tabela 2-2, percebe-se que o desenvolvedor A criou 3 code smells, com tempo total de remediação de 13 minutos. Se necessário, pode ser analisado o custo monetário do tempo de remediação dos code smells por desenvolvedor, apenas realizando a multiplicação da soma do tempo de remediação pelo seu valor de salário por minuto ou hora.

Esse tipo de análise permite tanto os desenvolvedores e/ou seus gestores tratarem estratégias para determinar como melhor gerir o custo de remediação. No entanto, é importante observar a necessidade de parametrizar os custos monetários de trabalho por desenvolvedor ou perfil (Júnior, Pleno e Sênior), e predeterminar o tempo de remediação por perfil de desenvolvedor para cada tipo de *code smell*.

Associação

A tarefa de associação para mineração de dados é o trabalho de descobrir quais atributos combinam com outros atributos. Para esse tipo de tarefa é necessário realizar dois passos de verificação: dos itens frequentes e a geração de regras (LAROSE, 2005). A tarefa da associação busca descobrir regras para quantificar o relacionamento entre dois ou mais atributos. As regras de associação têm a forma “Se antecedente, depois consequente”, juntamente com uma medida de apoio e confiança associada à regra.

O algoritmo mais conhecido para regras de associação é o Apriori, que foi apresentado por Agrawal e Srinkat (1994), com o objetivo de procurar por itens semelhantes entre registros de um subconjunto de dados. Essas semelhanças ou associações são expressas na forma de regras. Essas regras assumem uma forma de SE “L” ENTÃO “R”, que é equivalente, por exemplo, se {leite, pão} → {manteiga}, que indica que se o cliente compra leite e pão, com um determinado grau de certeza, ele também compra manteiga. Conforme exemplo, a associação é aplicada a dados categóricos, elas representam padrões existentes nas transações de um banco de dados.

Tanto Larose (2005) quanto Goldschmdt e Passos (2005) apontam o algoritmo Apriori proposto por Agrawal e Srikant (1994) como o meio mais adequado para esse tipo de tarefa de mineração. O algoritmo Apriori utiliza a busca em largura (ou busca em amplitude, também conhecido em inglês por Breadth-First Search - BFS) que, na

teoria dos grafos, é um algoritmo de busca em grafos utilizado para realizar uma busca ou travessia em um grafo e estrutura de dados do tipo árvore.

Atualmente diversos softwares de estatística computacional utilizam esse algoritmo para tarefas de regras de associação, como o Orange316 ou Weka17, criado por Witten, Frank e Hall (2011) apresentado também por Wass (2007). Esses softwares possuem também uma abordagem visual dos dados tratados.

Essa abordagem de utilizar softwares de mineração de dados e visualização de dados foi citada por Rezende (2003, p.327): “As técnicas de visualização de dados estimulam naturalmente a percepção e a inteligência humana. Logo, a visualização de dados utiliza a percepção humana como primeiro método para descobrir valores”. Além destes softwares, é possível utilizar bibliotecas, como a Apyori[18] que está disponível para a linguagem de programação Python. Esses pacotes e bibliotecas tiram a complexidade de desenvolvimento por parte do interessado, pois muitas vezes foram desenvolvidos por especialistas no assunto.

A utilização de regras de associação, por exemplo, pode ser aplicada em um histórico de *commits* (transações) e é uma das atividades de mineração de dados que pode auxiliar o desenvolvedor a entender seu comportamento em relação a *code smells*. Por exemplo, em cada *commit* que o desenvolvedor violou surge um tipo de *code smell* (RX), conforme apresentado na Tabela 2-3.

Tabela 2-3. Conjunto de dados exemplo de associação. Fonte: o Autor.

Commit	**Itens**
1	R1, R2
2	R1, R3, R4, R5
3	R2, R3, R4, R6
4	R1, R2, R3, R4

[16] Disponível em https://orange.biolab.si/

[17] Disponível em https://www.cs.waikato.ac.nz/ml/weka/

[18] Biblioteca Apyori: disponível em https://pypi.org/project/apyori/

5	R1, R2, R3, R6

Uma regra de associação é composta por um antecedente e um consequente, e ambos consistem em conjunto de itens. Por exemplo, a Equação 2-1, onde R1 e R2 são antecedentes e R3 consequente.

$$\{R1, R2\} \rightarrow \{R3\}$$

Equação 2-1. Exemplo de associação. Fonte: Agrawal e Srinkat (1994).

A medida de Suporte indica a frequência que um conjunto de itens ocorre na base de dados. Se o suporte desse conjunto é pequeno, significa que não há informações suficientes para tirar conclusões. A fórmula de suporte está representada na Equação 2-2.

$$Suporte(\{A\} \rightarrow \{B\}) = \frac{Número\ de\ transações\ com\ ambos\ A, B}{Número\ total\ de\ transações}$$

Equação 2-2. Fórmula suporte. Fonte: Agrawal e Srinkat (1994).

A medida de Confiança indica a frequência que a regra se mostrou válida. Porém, se a confiança para uma regra com um consequente muito frequente for sempre alta, considerar somente o valor da confiança pode levar a análises imprecisas. A fórmula de confiança está representada na Equação 2-3.

$$Confiança\ (\{A\} \rightarrow \{B\}) = \frac{Número\ de\ transações\ com\ ambos\ A, B}{Número\ de\ transações\ com\ A}$$

Equação 2-3. Fórmula da confiança. Fonte: Agrawal e Srinkat (1994).

Um exemplo do resultado da aplicação das fórmulas de suporte e confiança com base nos dados da

Tabela 2-3, encontra-se na Equação 2-4.

$$\{ R1, R3 \} \rightarrow \{ R4 \} = \left\{ S = \frac{2}{5} = 0.4, C = \frac{2}{3} = 0.67 \right\}$$

Equação 2-4. Resultado formula suporte e confiança. Fonte: o Autor.

A partir da tarefa de mineração de associação é possível ter indícios que com sua utilização é possível identificar quais *code smell* têm relação com os demais itens. Com isso, possibilita-se um conhecimento que pode demonstrar ao desenvolvedor se existe relação entre os tipos de *code smell* que ele adiciona ao código-fonte.

Séries temporais

Uma série temporal é um conjunto de observações de um fenômeno, ordenadas no tempo (GOLDSCHMIDT;PASSOS, 2005). A análise de uma série temporal é o processo de identificação das características, dos padrões e das propriedades importantes da série, utilizado para descrever o seu fenômeno gerador em termos gerais. Os modelos estatísticos para séries temporais utilizam o passado histórico da variável para projetar observações futuras. Por exemplo, a média de como a variável se comportará nos próximos períodos.

Como exemplos de séries temporais temos: o consumo de água de uma casa, registrado durante um ano ou mês. O objetivo da série temporal é a identificação das características, dos padrões e das propriedades importantes da série, utilizados para descrever seu fenômeno gerador. Dentre os diversos objetivos da análise de séries temporais, o maior deles é a geração de modelos voltados à previsão de valores futuros.

Goldschmidt e Passos (2005) ainda descrevem que séries temporais possuem movimentos, que podem ser:

- Movimento de ciclo: referem-se às oscilações de uma curva, que podem ou não ser periódicas. Logo, são variações que, apesar de periódicas, não são associadas automaticamente a nenhuma medida temporal.

- Movimento de tendência: é definido como um padrão de crescimento ou decrescimento da variável em um certo período.
- Movimento de sazonalidade: pode ser definido como padrões de comportamento que se repetem em determinado período. Por exemplo, o número de vendas em *e-commerce* no mês de dezembro devido ao Natal.
- Movimento de aleatoriedade: caracterizado por uma variável que se comporta de forma aleatória ao longo do tempo ao redor de uma média constante.

Um método muito comum utilizado na determinação da tendência de uma série temporal é a média móvel. O objetivo de utilização de médias móveis simples ou ponderadas é eliminar as variações cíclicas, sazonais ou aleatórias, conservando apenas o movimento de tendência.

Um exemplo de aplicabilidade de séries temporais em relação à gestão dos *code smells* criados por um desenvolvedor seria a utilização de médias móveis, como no exemplo da Figura 2-10, no qual se pode observar a evolução da média em média móvel de 14 dias (linha vermelha) x o realizado de *code smells* (linha azul). Vê-se ainda que no último, a média é 5, na média móvel 9.14, o que demonstra uma tendência de alta na criação de *code smells*. No eixo X da Figura 2-10 temos o intervalo de tempo de intervalos de 14 dias, no eixo Y da mesma figura a quantidade de *code smell*.

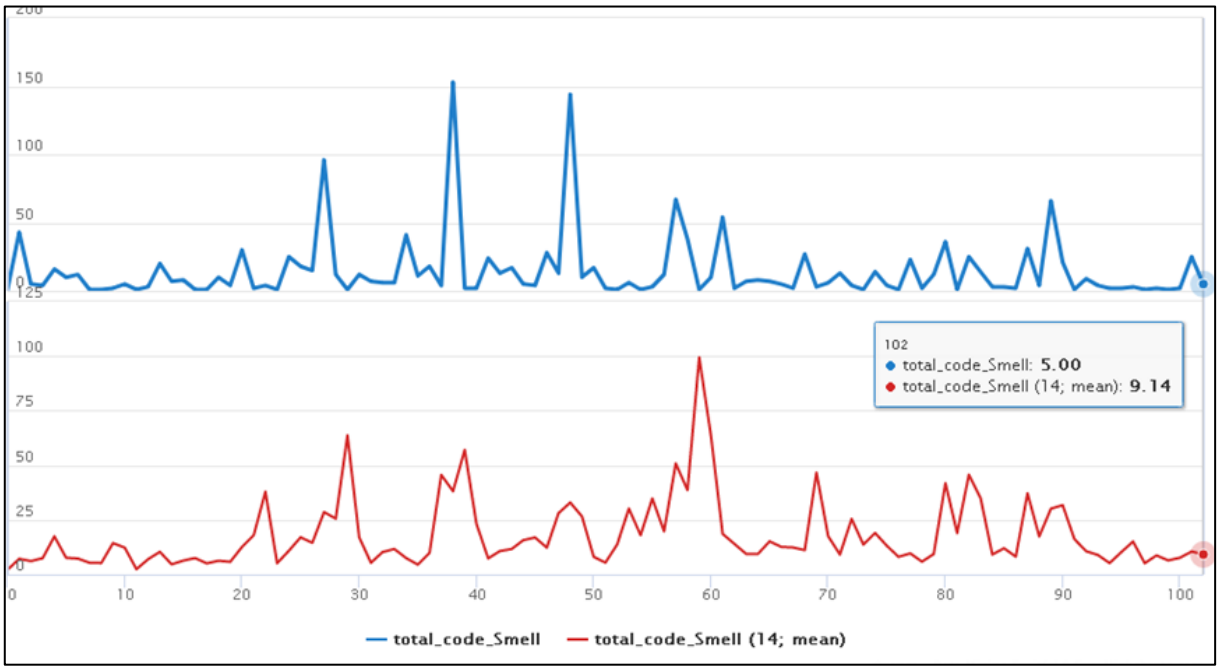


Figura 2-10. Exemplo de média móvel simples (14 dias) para criação de *code smell* pelo desenvolvedor. Fonte: o Autor.

Uma média móvel simples pode ser representada pela Equação 2-5:

$$M_\tau(t) = \sum_{i=0}^{\tau-1} \frac{x_{t-i}}{\tau}$$

Equação 2-5. Equação da média móvel simples.

Onde:

- M é a média móvel;
- x é a serie temporal;
- τ são os períodos utilizados para o cálculo da média móvel;
- t é o instante observado;
- ∑ são o somatório.

No entanto, toda a complexidade de aplicar fórmulas de médias móveis em séries temporais pode ser abstraída utilizando pacotes de estatística computacional ou bibliotecas estatísticas para linguagens de programação. Como por exemplo os softwares Orange3, IBM SPSS, Microsoft Excel e bibliotecas de estatística para linguagem de programação Python como por exemplo NumPy, Pandas e Scikit-Learn.

Outro interesse que existe em séries temporais é conseguir realizar previsões de observações no tempo. Box e Jenkins (2013) escreveram diversos trabalhos sobre a teoria de controle e análise de séries temporais. Em 1970, publicaram a primeira edição do livro *Time Series Analysis, forecasting and control.* Apresentando um método para a análise de séries temporais, esse trabalho e edições posteriores do livro, reuniram as técnicas existentes para construir modelos que descrevessem o processo gerador da série temporal, proporcionando previsões precisas de valores futuros.

Um dos modelos para previsão de série temporal abordados por Box e Jenkins é o modelo ARIMA (*AutoRegressive Integrated Moving Average*). Para construção do modelo ARIMA, Box e Jenkins sugerem três etapas, Identificação ,Estimativa e Previsão, no entanto é o modelo também trata a necessidade de verificar estacionariedade e verificação. Logo temos as seguintes etapas:

- Verificar estacionariedade ou não estacionariedade e transformar os dados, se necessário. Nesta etapa emprega se procedimentos que possam identificar a estrutura do modelo, que permitam conhecer os valores dos parâmetros d, p e q;
- Identificação de um modelo ARMA adequado;
- Estimativa dos parâmetros do modelo escolhido. Esta etapa é executada através de uso de *software* de estatística computacional;
- Verificação diagnóstica da adequação do modelo; e
- Previsão ou repetição das etapas de Identificação até Previsão.

Uma série temporal é dita estacionária quando ela se desenvolve no tempo ao redor de uma média e variância constante , refletindo alguma forma de estabilidade. A maioria das séries que encontramos apresentam algum tipo de tendência. Essa tendência pode ocorrer por períodos curtos ou longos. ARIMA é uma generalização do modelo autorregressivo de médias móveis (ARMA). Ambos os modelos ajustam os dados da série temporal para entender melhor os dados ou para prever pontos futuros

na série temporal. O modelo ARIMA é dividido em três partes, representados pelas letras p, d, q:

- AR: representado pela letra p. É a parte autorregressiva do modelo e indica que a variável de interesse em evolução é regredida em seus próprios valores anteriores. Permite incorporar o efeito de valores passados no modelo. Intuitivamente, isso seria semelhante a declarar que provavelmente estará quente amanhã se tiver feito calor nos últimos 3 dias.
- I: representado pela letra d. Indica que os valores dos dados foram substituídos pela diferença entre seus valores e os valores anteriores, logo, é o número de pontos no tempo anteriores a serem subtraídos do valor atual. Por exemplo, seria afirmar que provavelmente será a mesma temperatura amanhã se a diferença de temperatura nos últimos três dias tiver sido muito pequena.
- MA: representado pela letra q. É o número de erros de previsão defasados na equação de previsão, a parte da média móvel do modelo.

Toda a complexidade de utilização do modelo ARIMA pode ser resolvida com apoio de softwares de mineração de dados e bibliotecas para linguagem de programação. Por exemplo, o software Orange3 (ilustrado na Figura 2-11) e pacotes de estatística computacional como o Scipy[19] e Statsmodels[20] para linguagem de programação Python. No software Orange3 é necessário apenas informar a fonte de dados de série temporal e definir os valores das variáveis p, d e q.

[19] Disponível em https://www.scipy.org/

[20] Disponível em https://www.statsmodels.org/stable/index.html

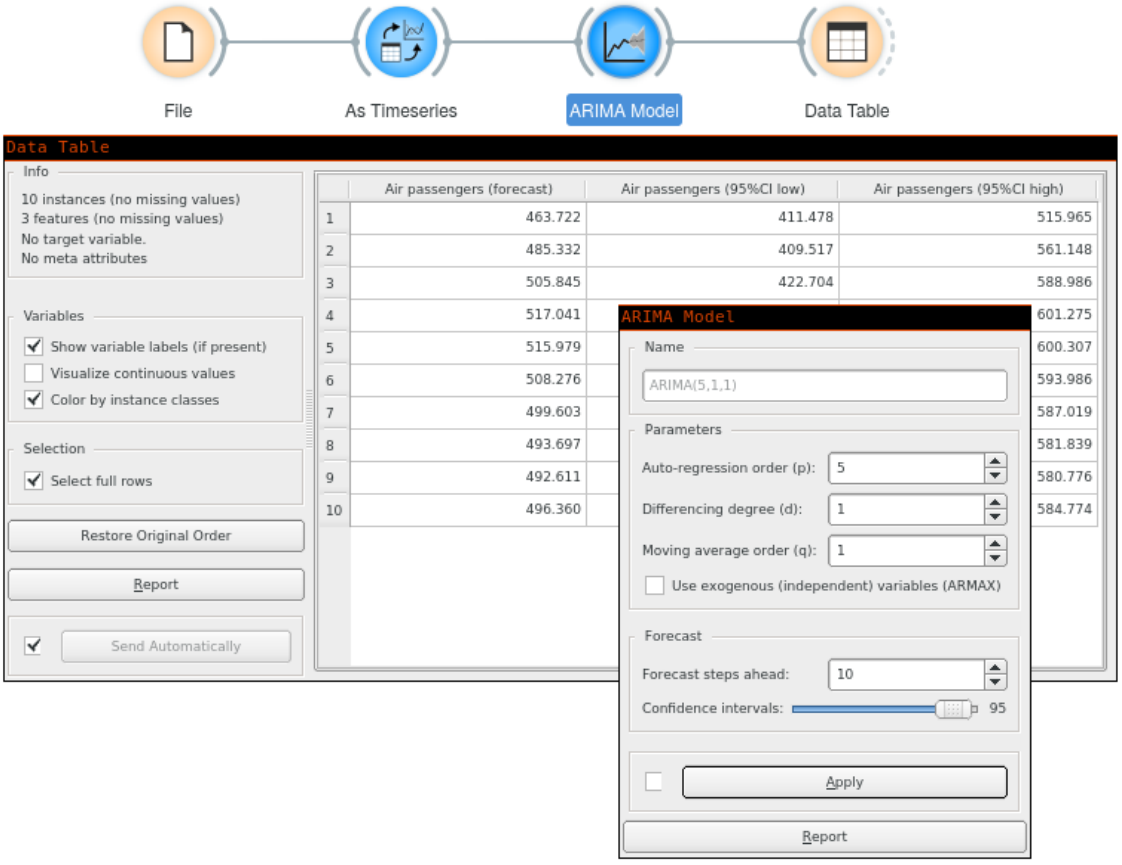


Figura 2-11. Exemplo de utilização ARIMA no software Orange3. Fonte: software Orange3.

Os valores para as variáveis p, d, q podem ser determinados com o apoio de pacotes e softwares de estatística computacional utilizando o Critério de Informação de Akaike (AIC). O AIC permite testar o quão bem o modelo se encaixa no conjunto de dados. A pontuação da AIC recompensa modelos que alcançam uma pontuação alta de ajuste e penaliza se eles se tornarem excessivamente complexos. Espera-se que o modelo com a pontuação AIC com valor mais baixo atinja um equilíbrio superior entre sua capacidade de se encaixar no conjunto de dados e sua capacidade de evitar o excesso de encaixe do conjunto de dados.

Outro aspecto que se faz necessário verificar em uma série temporal é se a série é estacionária, ou seja, quando ela se desenvolve no tempo aleatoriamente ao redor de uma média constante, refletindo alguma forma de equilíbrio estável. Bibliotecas e pacotes de estatística computacional possuem o teste de ADF (Dickey-Fuller aumentado) que permite realizar a verificação sobre se a série temporal é

estacionária. A estatística ADF, usada no teste, é um número negativo, e quanto mais negativo, mais indicativo o teste se torna de rejeitar a hipótese nula da existência de raiz unitária na série.

Para o teste de ADF, a hipótese nula do teste é que a série temporal pode ser representada por uma raiz unitária, que não é estacionária (tem alguma estrutura dependente do tempo). A hipótese alternativa (rejeitando a hipótese nula) é que a série temporal está estacionada. Então, se a hipótese nula (H0) não for rejeitada, sugere que a série temporal tem uma raiz de unidade, o que significa que não está estacionária, mas que tem alguma estrutura dependente de tempo. Já se a hipótese nula é rejeitada em favor da hipótese alternativa (H1), isso sugere que a série temporal não tem uma raiz de unidade, o que significa que está estacionária, não apresentando estrutura dependente do tempo. O resultado do teste é interpretado utilizando o valor p do teste. Um valor p abaixo de um limiar de 5% sugere que rejeitemos a hipótese nula (estacionária), caso contrário, um valor p acima do limiar sugere que não rejeitemos a hipótese nula (não estacionária).

Outro teste de estacionariedade é o teste de KPSS conforme apresentado por Kwiatkowski *et al.* (1992). As hipóteses deste teste são H0 = "a série é estacionária" e H1= "a série apresenta raiz unitária". Deve se observar que as hipóteses deste teste não são iguais aos testes de ADF para estacionariedade. A hipótese nula deste teste é igual às hipóteses alternativas do teste ADF.

Outra forma de avaliar a qualidade da previsão de um modelo ARIMA é construir uma estratégia de validação para treinar e testar o modelo. Para isso é necessário criar dois conjuntos de dados, um conjunto utilizado para treino e outro para teste. A estratégia mais comum de validação para séries temporais é a *rolling cross validation.* Esta estratégia deve ser aplicada no conjunto treino, onde é definido um conjunto inicial de observações (n), uma janela móvel (j) e um horizonte de previsão (h). Inicialmente é estimado o modelo nas n observações iniciais, em seguida é avaliada a previsão de horizonte h e estimado o modelo novamente adicionando as j próximas observações.

Por exemplo a Figura 2-12 apresenta a separação de um conjunto de dados em quatro eventos para treino e teste, onde a cada evento tem-se um incremento de

25% dos dados do conjunto. Cada evento é separado em 80% dos dados utilizados para treino e 20% para teste, essa divisão baseou-se em estudos empíricos feitos Gholamy *et. al* (2018). Ao executar cada um dos eventos de treino um dos resultados apresentado é o AIC (Critério de Informação de Akaike). Após quatro execuções teremos quatro resultados para AIC, o menor valor de AIC será utilizado para executar o último treino e teste.

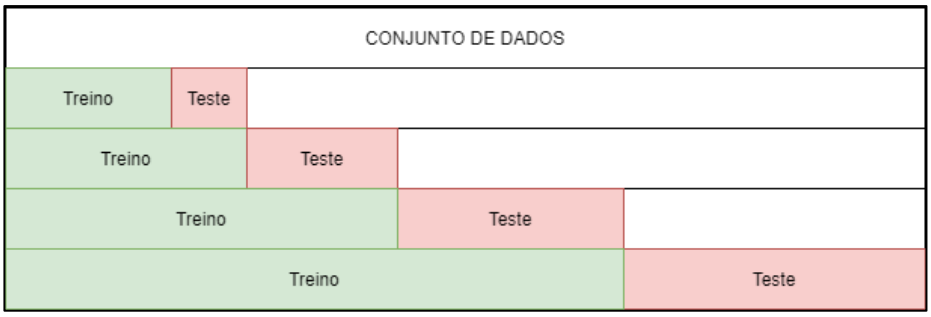


Figura 2-12. Exemplo de separação de dados para treino e teste. Fonte: o Autor.

Por exemplo, para um conjunto de dados de 5.000 registros considera-se 5 grupos (*folds*):

- Primeiro grupo: 1.000 registros, sendo 800 registros de treino e 200 de teste.
- Segundo grupo: 2.000 registros, sendo 1.600 registros de treino e 400 de teste.
- Terceiro grupo: 3.000 registros, sendo 2.400 registros de treino e 600 de teste.
- Quarto grupo: 4.000 registros, sendo 3.200 registros de treino e 800 de teste.

Cada um dos grupos após sua execução retorna como resultado o valor do AIC (Critério de Informação de Akaike) dentre os 4 resultados é selecionado o menor AIC para ser aplicado finalmente no conjunto total de dados, sendo dos 5.000 registros , 4.000 registros de treino e 1.000 registros para teste.

As medidas mais comuns para calcular a performance de modelos de previsão é a partir dos erros:

- MFE - *Mean Forecast Error* (Erro Médio da Previsão ou Viés): média dos erros da série avaliada. Os valores podem ser positivos ou negativos. Essa métrica sugere que o modelo tende a fazer previsões acima do real (erros negativos) ou abaixo do real (erros positivos). Desse modo, também pode-se dizer que o erro médio é o viés do modelo.

- MAE - *Mean Absolute Error* (Erro Médio Absoluto): muito semelhante ao erro médio da previsão mencionado acima, a diferença é o erro com valor negativo, ou seja, a previsão com valor maior que o número real é transformada em positivo e, posteriormente, a média é calculada. Essa métrica é muito usada em séries temporais, pois há casos em que o erro negativo pode zerar o positivo ou dar uma ideia de que o modelo é preciso. No caso do MAE isso não acontece, pois essa medida mostra o quanto a previsão está longe dos valores reais.

- MSE - *Mean Squared Erro*r (Erro Quadrático Médio): essa medida coloca mais peso nos erros maiores, pois cada valor individual do erro é elevado ao quadrado e posteriormente a média desses valores é calculada. Assim, essa métrica é muito sensível a outliers e coloca bastante peso nas previsões com erros mais significativos. Diferentemente do MAE e MFE, os valores do MSE estão em unidades quadráticas e não na unidade do modelo.

- RMSE - *Root Mean Squared Error* (Erro Quadrático Médio da Raiz): essa métrica é simplesmente a raiz quadrada do MSE, onde o erro volta a ter a unidade de medida do modelo, usada em séries temporais. É mais sensível a erros maiores devido ao processo de elevação ao quadrado que a originou.

- MAPE - *Mean Absolute Percentage Error* (Erro Percentual Médio Absoluto): medida usada em relatórios gerenciais, pois o erro é medido em termos percentuais e pode-se comparar o erro percentual do modelo de um produto X com o erro percentual de um produto Y. O cálculo dessa medida utiliza o

valor absoluto do erro dividido pelo valor real e, posteriormente, é calculada a média.

Uma abordagem para calcular é com o uso do MAPE, o MAPE foi utilizado por Tsoukalas *et al.* (2020) no estudo para previsão de dívida técnica. O MAPE é um dos erros mais usados na estatística por ser de fácil compreensão. Esta medida de erro é expressa em porcentagem média dos erros (em valor absoluto) cometidos na previsão da série temporal; quanto menor o valor da medida, melhor é a previsão. Visto isso esse indicador será utilizado para avaliar o modelo previsão, visto que outros estudos já apontam a utilização de avaliação de modelo de previsão utilizando o MAPE.

Regressão

Goldschmidt e Passos (2005) descrevem que a tarefa de regressão em mineração de dados compreende a busca por funções, lineares ou não, que mapeiem os registros de um banco de dados em valores reais. A tarefa de regressão é similar à tarefa de classificação, sendo restrita apenas a atributos numéricos.

A regressão é utilizada para prever comportamentos com base na associação entre duas variáveis, que geralmente possuem uma boa correlação. A regressão linear é a forma mais simples de regressão, onde a função a ser abstraída a partir dos dados é uma função linear. A equação para regressão linear simples pode ser observada na Equação 2-6.

$$Y = \alpha + \beta X$$

Equação 2-6. Equação da regressão linear simples.

Onde na Equação 2-6 as variáveis são:

- Y é a variável assumida como constante e variável aleatória e dependente;
- α e β são os coeficientes;

- X a variável independente.

2.8.2 Mineração de repositórios de *software*

Dentro da mineração de dados existe uma área denominada Mineração de Repositórios de Software (MSR - *Mining Software Repositories*). A MSR é um processo para obter evidências extraindo dados de repositórios de software. Oram e Wilson (2010) definem a MSR como fontes de dados de artefatos baseados em produtos, como código-fonte, artefatos de requisitos ou arquivos de versões.

O campo MSR tem o objetivo de analisar os dados disponíveis em repositórios de software (repositórios de controle de versão, arquivos de listas de discussão, sistemas de rastreamento de *bugs* e sistemas de rastreamento de problemas), com o intuito de descobrir informações interessantes e acionáveis sobre sistemas, projetos e engenharia de software.

Hassan (2008) observa que ao transformar esses repositórios de código em repositórios ativos, pode-se orientar processos de decisão em projetos de software modernos. O autor ainda sugere que uma possibilidade é usar alguns grandes projetos de código aberto como cobaias, para demonstrar a funcionalidade desses serviços e os benefícios das técnicas de MSR.

Na literatura é possível encontrar trabalhos sobre ferramentas para extração de dados de repositórios. As ferramentas citadas são o GHTorrent e PyDriller. O projeto GHTorrent fornece todos os dados presentes no GITHUB, ou seja, conteúdo de repositórios, *commits*, *pull request* e usuários, com o objetivo de ser uma fonte de dados para pesquisadores interessados em minerar dados do GITHUB (GOUSIOS; SPINELLIS, 2012; GOUSIOS *et al*., 2014). No entanto, nas informações presentes, não há relação com dados de qualidade de código-fonte estático.

Spadini *et al*. (2018) citam a utilização do PyDriller para mineração de repositórios de código. Os autores apontam a importância da MSR na pesquisa de engenharia de software e citam que existem poucas ferramentas para apoiar os desenvolvedores na extração de informações do repositório GIT.

Lenarduzzi *et al.* (2019) realizaram a leitura de todos *commits* de um repositório de um produto específico com o *software* PyDriller, dada a facilidade de

implementação. O objetivo foi criar um modelo de dados no qual pesquisadores poderiam se beneficiar do *dataset* de DT para ser utilizado em aprendizado de máquina estatística e estudos empíricos sobre DT.

Os trabalhos de Lenarduzzi *et al.* (2019) e Spadini *et al.* (2018) dão indícios para a utilização do PyDriller na mineração de repositórios e na extração de dados do GITHUB.

2.9 PESQUISAS RELACIONADAS

Com a finalidade de compreender os assuntos tratados nesta pesquisa, foi necessário criar a seguinte questão: **Quais são as abordagens de experimentos em engenharia de software presentes na literatura que utilizam a mineração de dados e que de alguma forma permitem ao desenvolvedor saber quais regras de *code smell* ele comete frequentemente?**

A partir desta questão de pesquisa, os critérios de inclusão (CI) e exclusão (CE) foram definidos:

- CI1 – Trabalhos que abrangem os assuntos de DT de *code smell*, que possam estar associados à mineração de dados e/ou *machine learning* e/ou previsão e/ou ciência de dados e/ou estatística e/ou GitHub e/ou SQALE e/ou séries temporais e/ou descoberta de conhecimento, relacionados à área de computação;
- CE1 – Trabalhos com ausência de resumo;
- CE2 - Trabalhos de revisão de conferência;
- CE3 – Trabalhos que não estejam em inglês;
- CE4 – Trabalhos que não sejam referentes à área de pesquisa;
- CE5 – Trabalhos duplicados;
- CE6 – Trabalhos não encontrados e/ou apresentados em base de artigos onde o *link* para o site do Publisher não disponibiliza o artigo.

Na estratégia de busca não foi considerado limitar a *string* de busca em um período específico. A pesquisas nas bases indexadas de artigos ocorreu entre 11/2019 e 05/2020.

Para responder à questão definida anteriormente, foi necessário realizar pesquisa em bibliotecas eletrônicas e bases indexadas de artigos científicos. A principal base de artigos científicos utilizada foi a Scopus, mesmo realizando a busca de artigos em outras bases como IEEEXplore, ACM, SpringLink e Elsevier. Para pesquisa foram utilizadas 6 *strings* de busca na base indexada de artigos que tem ocorrência de resultados para *string* de busca identificada no Quadro 2-1. À cada uma das *strings* de busca foi atribuído um nome.

Quadro 2-1. *String* de busca para pesquisas relacionadas. Fonte: o Autor.

NOME	**TEXTO DE BUSCA**
Principal	(“TECHNICAL DEBT" OR “CODE-SMELL*” OR “CODE SMELL*" OR "STATIC CODE”) AND (“PREDICT*” OR "FORECAST*” OR "ESTIMAT*" OR "STATISTIC*” OR “DATA SCIENCE” OR “DATA MINING” OR “MACHINE LEARNING"))
GITHUB	(“GITHUB”) AND ("TECHNICAL DEBT" OR "CODE-SMELL*" OR "CODE SMELL*" OR "SQALE")
SQALE	SQALE
GITHUB2	(“GITHUB" OR “GIT”) AND (“PREDICT*” OR "FORECAST*” OR "ESTIMAT*" OR "STATISTIC*” OR “DATA SCIENCE” OR “DATA MINING” OR “MACHINE LEARNING”) AND ("TECHNICAL DEBT" OR "CODE-SMELL*" OR "CODE SMELL*" OR "SQALE")
TIME	(“CODE-SMELL*” OR “CODE SMELL*” OR “TECHNICAL DEBT” OR “SQALE”) AND “TIME SERIES"
KDD	(“CODE-SMELL*” OR “CODE SMELL*” OR “TECHNICAL DEBT” OR “SQALE”) AND (“KDD” OR “KNOWLEDGE DISCOVERY”)

A partir das *strings* de busca aplicadas nas bases científicas, foi possível identificar 629 trabalhos. Após a aplicação dos critérios de exclusão, o resultado foi de 478 artigos. Dos 478 artigos encontrados, foram considerados 16 artigos após leitura de título e resumo. No entanto, durante o desenvolvimento da pesquisa foram identificados mais 8 artigos por meio de *snowballing*, a partir do nome de alguns

autores. O resultado dos artigos relacionados à pesquisa por *string* de busca pode ser verificado na Tabela 2-4.

Tabela 2-4. Resultado da pesquisa em base indexada de a artigos. Fonte: o Autor.

Ação	Motivo Funil	Principal	GITUB	SQALE	GITHUB2	TIME	KDD	Total
+	*String* de busca	559	25	22	14	7	2	629
-	CE1 – Trabalhos com ausência de resumo	0	0	0	0	0	0	0
-	CE3 – Trabalhos que não estejam em inglês	8	0	1	0	0	0	9
-	CE4 – Trabalhos que não sejam referentes a área de pesquisa	8	0	3	0	0	0	11
-	CE2 - Trabalho de revisão de conferência	77	4	5	2	2	1	91
-	CE6 – Trabalhos não encontrados	9	0	1	0	0	0	10
-	CE5 – Trabalhos duplicados em relação ao Principal	1	9	5	12	3	0	30
=	**SUBTOTAL**	**456**	**12**	**7**	**0**	**2**	**1**	**478**
-	Leitura do título + *Abstract* e/ou Texto	442	10	5	0	2	0	459
=	**Relacionado**	**11**	**2**	**2**	**0**	**0**	**1**	**16**
	Utilizado a partir da *string*	**16**						
+	**Obtidos a partir do *Snowballing***	**8**						
	Total	**24**						

A Tabela 2-5 apresenta os artigos obtidos após a aplicação dos critérios de inclusão e exclusão e do *snowballing*. Como se pode observar, há uma concentração de publicações em anos recentes.

Tabela 2-5. Quantidade de publicações referenciadas por ano. Fonte: o Autor.

Ano	Quantidade a partir da *string* de busca	Quantidade a partir do *snowballing*
2008		1
2012	2	1
2014	2	1
2015	3	
2017	1	
2018	1	1
2019	7	1
2020		3
Total Geral	**16**	**8**

Os tipos de publicação encontrados podem ser observados na Tabela 2-6, enquanto o idioma das publicações pode ser observado na Tabela 2-7. As 24 publicações relacionadas ao tema dessa dissertação podem ser observadas nos tópicos a seguir, referentes a cada uma das *strings* de busca e ao *snowballing*.

Tabela 2-6. Tipos de publicação. Fonte: o Autor.

Tipo de publicação	Quantidade a partir da *string* de busca	Quantidade a partir do *snowballing*
Artigo	128	8
Article in Press	3	
Capítulo de livro	7	
Conference Paper	394	
Conference Review	90	
Nota	2	
Revisão	5	
Total Geral	629	8

Tabela 2-7. Idioma das publicações. Fonte: o Autor.

Idioma	**Quantidade a partir da *string* de busca**	**Quantidade a partir do *snowballing***
Não identificado	1	
Chinês	4	
Inglês	620	8
Português	3	
Espanhol	1	
Turco	1	
Total Geral	629	8

2.9.1 *String* de busca "PRINCIPAL"

Letouzey et al. (2012) afirmam que DT e qualidade de código são conceitos conectados. Eles apresentam o método SQALE, que é o método para análise da DT em código estático, o qual afirmam ser um método preciso para estimar a DT. O trabalho apresentado por Letouzey et al. (2012) é de suma importância para essa dissertação, pois o método SQALE, apoiado por ferramentas de análise estática, permite gerenciar a DT e o tempo de remediação de code smells.

Skourletopoulos *et al.* (2014) citam a importância de prever a DT em sistemas em nuvem, pois nesse nível de serviço é necessário tomar as decisões certas dentro de um contexto de negócios. Os autores afirmam que a previsão da DT permite a criação de estratégias de retorno mais precisas e um gerenciamento ágil, assim como sugerem a utilização do COCOMO[21] para previsão da DT em ambiente de nuvem para análise de esforço e custo.

Kazman *et al.* (2015) abordam a DT de arquitetura, mostrando que, em grande escala, arquivos defeituosos raramente existem sozinhos. Eles são geralmente

[21] *Constructive Cost Model* é um modelo paramétrico que permite calcular o esforço, o custo e o prazo de um projeto através de equações matemáticas complexas que levam em consideração particularidades de cada projeto como: características do Produto, Processo, Experiência da Equipe e Plataforma de Desenvolvimento.

conectados arquitetonicamente e essas estruturas arquitetônicas exibem falhas significativas de *design*, inclusive *code smells*, que propagam erros entre os arquivos. Usando dados extraídos de artefatos de desenvolvimento, os autores foram capazes de identificar os arquivos implicados em falhas de arquitetura e sugerir refatorações com base na remoção dessas falhas. Eles realizaram a extração de dados do SonarQube e outras ferramentas para analisar a DT de arquitetura. Os autores também construíram um modelo econômico para estimar o custo de remediação e refatoração.

Apesar do trabalho de Kazman *et al*. (2015) olhar para a DT por uma perspectiva de produto x arquitetura, a abordagem é interessante para essa dissertação no que tange à preocupação de quantificar as DTs de *code smell*, e ainda o custo da remediação por desenvolvedor, visto que a abordagem utilizada por Kazman et al. (2015) tem um enfoque no produto e o custo total da remediação.

Wang *et al.* (2015) propuseram combinar o uso de engenharia de *software* baseada em pesquisa com séries temporais para recomendar estratégias de refatoração para gerenciar a DT. Adaptaram séries temporais para estimar o impacto das refatorações geradas nas próximas versões de um sistema, prevendo a evolução dos *code smells* e avaliaram a abordagem em um projeto da indústria e um *benchmark* de 4 sistemas. Os resultados confirmaram a eficiência da técnica para fornecer melhor gerenciamento de refatoração em comparação com várias técnicas de refatoração existentes. Os autores abordaram a utilização de séries temporais com ARIMA[22] para gestão da DT, no entanto, a abordagem é centrada no produto *software* e não no desenvolvedor.

Apresentado por Mendes *et al.* (2019), a ferramenta VisminerTD[23] vai ao encontro do que foi abordado por Alves *et al.* (2016) sobre a identificação e o monitoramento de DT por meio de técnicas de visualização de *software*. O VisminerTD não possui funcionalidades para análise estatística de dados de *code smell*, tampouco

[22] É um modelo de descrição probabilística de uma série temporal. Muito utilizado na modelagem e previsões de séries temporais, o modelo ARIMA foi sistematizado em 1976 pelos estatísticos George Box e Gwilym Jenkins.

[23] Disponível em: https://visminer.github.io/

está preparada para suportar a mesma variedade de linguagens de programação, quando comparada ao SonarQube.

Tsoukalas *et al.* (2019) realizaram uma pesquisa para identificar o estado da arte de métodos e ferramentas para previsão de DT. O estudo apontou que nenhum dos métodos e ferramentas apresentou um nível de maturidade desejado, além de uma ausência de estudos notáveis sobre previsão da DT, mesmo ainda existindo métricas e técnicas em potencial que ainda não tenham sido utilizadas e que possam potencialmente aumentar a integridade do conceito de estimativa de DT. Eles afirmaram que a previsão da DT pode levar ao desenvolvimento de mecanismos práticos de decisão, que poderiam ajudar desenvolvedores e gerentes de projeto em ações proativas em relação ao reembolso da DT.

Os desenvolvedores ainda são céticos quanto à precisão do tempo de remediação apresentado pelo SonarQube. No estudo de caso desenvolvido por Saarim *et al.* (2019), os resultados sugeriram que o tempo de remediação do SonarQube, comparado com o tempo real para reduzir a DT, é geralmente superestimado, e que a estimativa mais precisa diz respeito a *code smells*. Visto isso, a presente dissertação pretende colaborar para que individualmente o desenvolvedor determine o tempo de remediação para cada uma das regras de *code smell*.

Kumar *et al.* (2019) utilizaram a previsão de DT de composição serviços[24] de computação em nuvem, com utilização de séries temporais com modelo ARIMA. Por meio de um estudo de aplicação no mundo real, demonstraram que a abordagem produz uma precisão satisfatória na estimativa da DT na composição do serviço na nuvem. Esta abordagem é interessante para essa dissertação no que concerne à utilização de séries temporais e à utilização do algoritmo ARIMA, no entanto, aplicando para a dívida técnica de código.

Ebert *et al.* (2019) apresentam o *framework* Q-Rapids, que é um projeto de pesquisa e inovação da União Europeia com o objetivo de criar uma forma de explorar produtos, processos e dados de uso com ciência de dados para melhorar a qualidade

[24] Composição de serviços é uma coleção de serviços onde muitos serviços menores são combinados em um serviço maior.

do software. O Q-Rapids fornece um *framework* para ingestão de dados escalonáveis de fontes de dados e interfaces com estruturas de mineração de dados para derivar um modelo de qualidade. Essa dissertação pode contribuir em prover um produtor de dados e que possa ser utilizado na camada de "*Data Producer*" do *framework.*

Salamea e Farre (2019) apontam que as ferramentas de análise estática negligenciam outros possíveis fatores que podem afetar a qualidade do código e a avaliação da DT. A hipótese inicial é que fatores humanos associados aos desenvolvedores de software, como conhecimento de codificação, habilidades de comunicação e experiência no projeto têm algum impacto mensurável na qualidade do código. Em um estudo exploratório, os autores testaram a hipótese em dois grandes repositórios de aplicações Java usando DT como uma métrica de qualidade de código e dados que podem ser inferidos dos sistemas de controle de versão. Os dados foram obtidos no GITHUB e SonarQube. O resultado preliminar sugeriu que o nível de participação dos desenvolvedores e sua experiência no projeto têm uma correlação positiva com a quantidade de DT que eles apresentam. Encontraram ainda uma forte correlação positiva entre as linhas de código editadas e a quantidade de DT introduzida por cada desenvolvedor, o que implica que a DT produzida é proporcional às linhas de código editadas. Eles ainda pretendem avaliar o método utilizado com o *framework* Q-Rapids proposto por Ebert *et al.* (2019), mas desde já, o trabalho de Salamea e Farre (2019) contribui para essa dissertação no sentido da importância da análise da gestão da DT na visão do desenvolvedor, além de ressaltar a importância da experiência do profissional.

Lenarduzzi *et al.* (2019) como citado anteriormente, criaram um modelo de dados no qual pesquisadores podem se beneficiar do *dataset* de DT para ser utilizado em aprendizado de máquina, processamento de linguagem natural no código-fonte e estudos empíricos sobre DT. Esse modelo de dados pode ser melhorado para permitir análise de séries temporais e regras de associação em relação aos *code smells* criados por desenvolvedor. Outra possibilidade do modelo proposto por Lenarduzzi *et al.* (2019) é adicionar dimensões de dados que permitam gerir o custo de remediação de *code smells.*

2.9.2 *String* de busca “GITHUB”

Almeida Filho *et al.* (2019) analisaram 20 repositórios de projetos PL/SQL[25] no GITHUB para empiricamente estudar a prevalência de *code smells* do ponto de vista de produto. Mostraram que alguns *code smells* PL/SQL podem ocorrer em conjunto, por meio de regras de associação com o algoritmo Apriori. Demonstraram ainda, que alguns *code smells* são mais frequentes do que outros. No entanto, a estratégia não utilizou uma abordagem de engenharia de software experimental (ESE) ou descoberta de conhecimento. Porém, demonstraram a importância de utilizar métodos estatísticos para análise de dados resultantes da análise estática de código.

Anderson *et al.* (2019) apresentam o SARIF *(Static Analysis Results Interchange Format),* um padrão emergente para representar os resultados das ferramentas de análise estática. Esse padrão permite a integração entre ferramentas de análise estática e sistemas de controle de versão e, ao fazê-lo, incentiva os desenvolvedores a reduzir a DT de forma gradual e não invasiva.

O padrão SARIF originou-se na Microsoft e agora é um padrão em desenvolvimento no OASIS[26], que é um comitê técnico com membros de vários fornecedores de ferramentas de análise estática e usuários de grande escala. A especificação[27] do SARIF, informa que ele foi projetado para comunicar não apenas resultados, mas metadados sobre as ferramentas. No entanto, pelo fato de o SARIF ainda ser um padrão emergente, poucos softwares[28] de análise estática apresentam seus resultados neste formato. A Microsoft é uma das empresas que apoia o projeto SARIF e que inclusive disponibiliza um dos poucos *converters*[29] de análise estática para SARIF, um *plugin* disponibilizado pela Microsoft que deve ser instalado no IDE Visual Studio. No entanto, esse *plugin* não suporta o resultado de análise estática do

---

[25] PL/SQL (Procedural Language/Structured Query Language) é uma extensão da linguagem padrão SQL para o SGBD Oracle da Oracle Corporation.

[26] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif

[27] https://github.com/oasis-tcs/sarif-spec

[28] Microsoft SARIF Viewer - https://marketplace.visualstudio.com/items?itemName=WDGIS.MicrosoftSarifViewer

[29] https://sarifweb.azurewebsites.net/#ToolsLibraries

SonarQube, visto que o SonarQube apenas persiste seus resultados em banco de dados.

2.9.3 *String* de busca “SQALE”

Gjoshevski e Schweighofer (2015) pesquisaram a qualidade de código de aplicações móveis *Android* nativas e apontaram a confiabilidade dos indicadores coletados pela ferramenta SonarQube. Os autores procuraram identificar quais os *code smells* mais frequentes em dispositivos móveis Android na perspectiva do produto.

Guaman *et al.* (2017) abordam a utilização do SonarQube em relação ao método SQALE. Eles sugerem manter uma métrica de referência com os dados apresentados pelo SonarQube, para sugerir aos desenvolvedores a utilização de normas e não aumentar a DT, especialmente quando se trabalha em aplicações empresariais onde as equipes de trabalho são numerosas.

2.9.4 *String* de busca “KDD”

Holvitie (2014) discute a aplicação de gerenciamento de DT para implementações de software, incluindo os pontos de entrada para descoberta de conhecimento. A hipótese que o autor define diz respeito aos desafios de manutenção no contexto de implementação de software, afirmando que a DT de implementação de software pode ser capturada e mantida em tempo de implementação e que podem ser capturados por meio de ferramentas que podem estar presentes no ambiente de desenvolvimento.

Para isso, Holvitie e Leppanen (2014) apresentam a ferramenta DebtFlag para captura, rastreamento e resolução de DT. Este é um *plugin* para ambiente de desenvolvimento Eclipse que permite marcar código para relacionar o código marcado a itens de DT ao ponto do projeto em que está localizado no código, com o objetivo de evitar a propagação da DT. Durante o desenvolvimento contínuo, esses *links* (relacionamentos) são utilizados para criar caminhos de propagação para a dívida documentada. Isso permite uma apresentação atualizada e precisa de DT a ser mantida, e, ainda, que o desenvolvedor conduza um micro gerenciamento em nível de implementação.

No entanto, depende de o desenvolvedor identificar e mapear as DTs, sem utilizar uma ferramenta de análise estática para auxiliar a criar o relacionamento. Por outro lado, além de a linguagem de programação JAVA ser limitada, ela se concentra em permitir que o desenvolvedor gerencie itens da DT no nível de implementação. A importância deste trabalho em relação a essa dissertação é a abordagem de micro gerenciamento, onde o desenvolvedor, sabendo de forma prática durante o processo de implementação, pode realizar refatoração contínua a partir de evidências.

2.9.5 *SnowBalling*

Hassan (2008) observa que ao transformar esses repositórios de código em repositórios ativos, pode-se orientar processos de decisão em projetos de software modernos. Ele sugere ainda que uma possibilidade é usar alguns grandes projetos de código aberto como cobaias, para demonstrar a funcionalidade desses serviços e os benefícios das técnicas de MSR.

Gousios e Spinellis (2012) e Gousios *et al.* (2014) apresentam o projeto GHTorrent que fornece todos os dados presentes no GITHUB, ou seja, conteúdo de repositórios, *commits*, *pull request*, usuários, com o objetivo de ser uma fonte de dados para pesquisadores interessados em minerar dados do GITHUB. No entanto, das informações presentes, não há relação com dados de qualidade de código-fonte estático.

Spadini *et al.* (2018) citam a utilização do PyDriller para mineração de repositórios de código. Apontam a importância de MSR na pesquisa de engenharia de software e citam que existem poucas ferramentas para apoiar os desenvolvedores na extração de informações do repositório GIT.

Hass *et al.* (2019) apresentam, então, o TeamScale[30], uma ferramenta comercial com o objetivo de monitorar a qualidade de código-fonte. O TeamScale fornece integração com diversos ambientes de desenvolvimento e repositórios de código, mas não implementa o método SQALE. Não foi identificado na documentação da ferramenta se o produto possui funcionalidades para analisar seus dados de forma

[30] Disponível em: https://www.cqse.eu/en/teamscale/overview/

estatística. O acesso ao banco de dados das métricas e análise estática da ferramenta dá-se apenas por API Rest. Visto ao acesso a banco de dados ser restrito, para essa dissertação não será considerado este produto no que concerne à realização de extração de dados de *code smells*.

Lacerda *et al.* (2020), em uma revisão sistemática, sugeriram que ainda existem questões em aberto em relação a *code smell* e refatoração. Algumas dessas lacunas referem-se à produção de conjuntos de dados confiáveis, uso de projetos acadêmicos e industriais de grande escala, análise industrial e acadêmica em busca de inconsistências de dados, como exemplos de tendências. Outra lacuna, apontada por eles, seria a percepção da indústria, na qual é necessário avaliar se os praticantes reconhecem *code smells* e se sabem que esse código tem sintomas de problemas. Ambas as lacunas podem ser atendidas com o resultado proposto por este trabalho de dissertação, pois produz um conjunto de dados e um processo para analisar esses dados de *code smell,* que pode ser adaptado para utilização na indústria.

Lenarduzzi *et al.* (2020) realizaram um estudo de caso para identificar quanto tempo os desenvolvedores juniores levam para remover itens de DT e identificaram que os *code smells* são os itens de DT que os desenvolvedores juniores dão prioridade para refatoração, com o tempo de remediação geralmente 50% menor do que informado pelo SonarQube. Desenvolvedores nível júnior criam mais *code smells* do que outros tipos de DT, assim como realizam refatoração de *code smell* mais do que outros tipos de DT. Logo, a abordagem proposta por essa dissertação vai ao encontro deste estudo de Lenarduzzi *et al.* (2020), no sentido auxiliar os desenvolvedores menos experientes a identificar seus *code smells*, para aprender sobre suas tendências de criação de *code smell* e evitar a atividade de refatoração.

Tsoukalas et al. (2020) propuseram uma abordagem para análise preditiva de DT (Figura 2-13), adequada para projetos com um histórico relativamente longo. Segundo os autores é o primeiro estudo que investiga a viabilidade do uso de modelos de aprendizado de máquina para previsão de DT.

O estudo utilizou dados do GitHub e SonarQube de 15 aplicações em linguagem de programação JAVA e para cada aplicação utilizaram 150 *commits*, logo, 150 análises estáticas por aplicação. Apesar de utilizarem poucos *commits* por

projeto, os autores utilizaram técnicas de ML em sua abordagem. Tsoukalas *et al.* (2020), ainda constataram que o modelo ARIMA (0,1,1) pode fornecer previsões precisas do princípio da DT durante um período longo para todas as aplicações de amostra e observaram que o desempenho preditivo caiu significativamente para previsões de longo prazo.

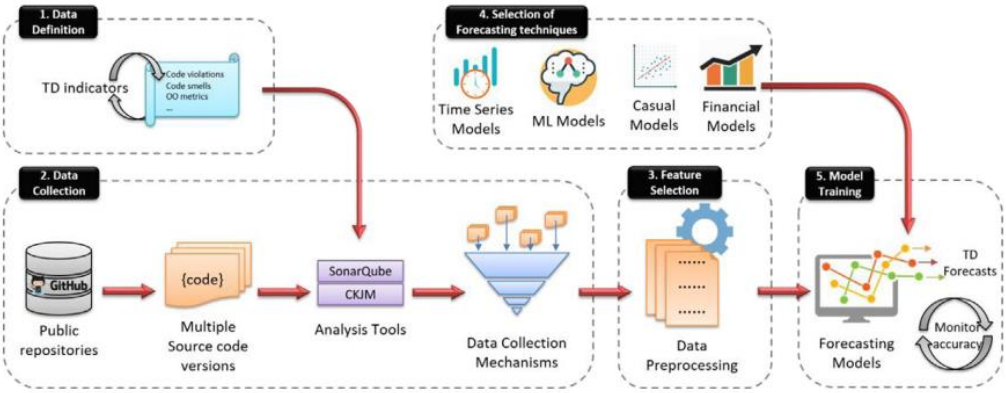


Figura 2-13. Roteiro utilizado na abordagem de ML para DT. Fonte: Tsoukalas et al. (2020).

Ebert et al. (2019), como já citado anteriormente, abordaram o *framework* Q-Rapids, conforme ilustra a Figura 2-14. Este foi identificado como o primeiro estudo que procurou abordar a utilização de ciência de dados com esta finalidade. Ebert *et al.* (2019) utilizaram o CRISP-DM (*Cross Industry Standard Process for Data Mining*), um padrão para Mineração de Dados que possui características de uma abordagem de experimento em engenharia de *software.*

Ambas as abordagens de Tsoukalas *et al.* (2020) e Ebert *et al.* (2019) são interessantes, no entanto, não apontam um modelo de dados que possa servir de referência para o desenvolvedor replicar um experimento, e os passos detalhados de ETL e mineração de dados para obter informação e/ou conhecimentos de *code smells*.

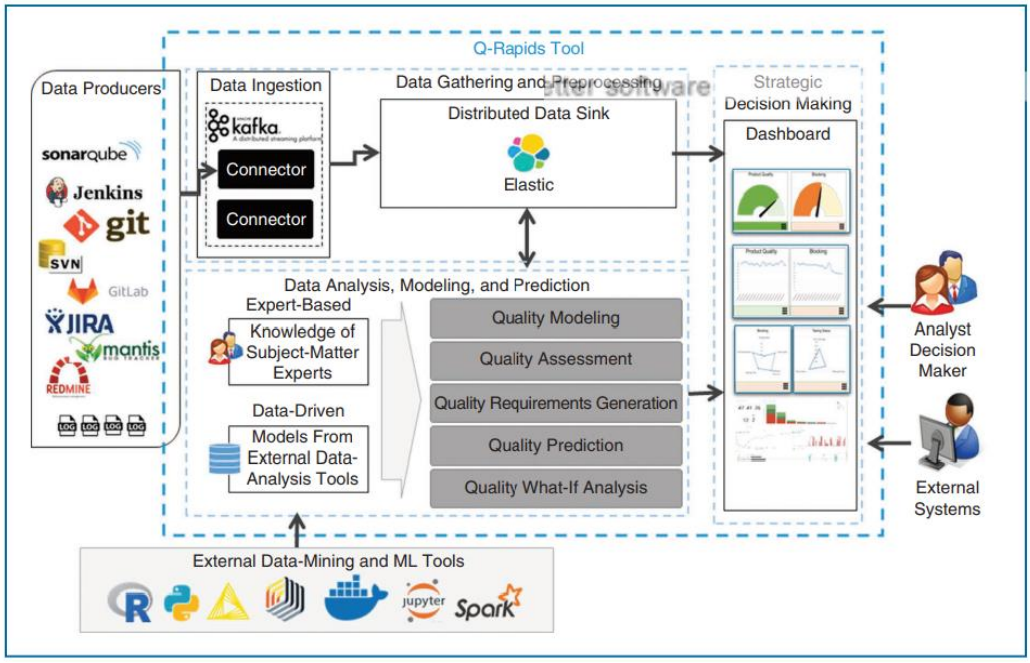


Figura 2-14. Framework Q-Rapids. Fonte: Ebert et al. (2019).

A partir das pesquisas relacionadas, é possível responder à questão inicial proposta para identificar as pesquisas relacionadas: **Quais são as abordagens de experimentos em engenharia de software presentes na literatura que utilizam a mineração de dados e que de alguma forma permitem ao desenvolvedor saber quais regras de *code smell* ele comete frequentemente?**

Conforme pôde ser observado, nenhuma das abordagens atende com precisão à necessidade do problema de pesquisa desta dissertação. No entanto, as abordagens de Ebert et al. (2019), Tsoukalas et al. (2020), Lenarduzzi et al. (2019), contribuem na identificação das etapas e processos para criar uma abordagem com tais características.

As abordagens propostas por Ebert et al. (2019) e Tsoukalas et al. (2020) apresentam etapas que envolvem a identificação dos dados, processamento, transformação, mineração de dados e visualização de dados. Essas etapas são comuns ao KDD. Já a abordagem do *dataset* proposta por Lenarduzzi et al. (2019), aponta para possíveis entidades que possam servir de referência para essa dissertação, por exemplo, entidade de projeto, desenvolvedor, commit, tipo de item da DT.

Vista a necessidade de entender quais entidades são necessárias para realizar a mineração de dados, a proposta de modelagem de dados a ser utilizada nesta dissertação foi a abordagem apresentada por Kimball (1998) referente ao esquema em StarSchema e SnowFlake. Essas abordagens são utilizadas principalmente para sistemas de apoio à decisão. A principal característica da modelagem de esquema em estrela é a presença de dados altamente redundantes, melhorando o desempenho em pesquisa de dados.

O StarSchema e SnowFlake constituem uma metodologia de modelagem de dados utilizada no desenho de um DataWarehouse[31]. No StarSchema os dados são modelados em tabelas dimensionais ligadas a uma tabela de fatos. As tabelas dimensionais contêm as características de um evento. A tabela de fatos armazena os fatos ocorridos e as chaves para as características correspondentes nas tabelas de dimensão. A tabela de fato conecta-se às demais dimensões por múltiplas junções e as tabelas de dimensões conectam-se apenas com uma junção da tabela de fatos. Já no SnowFlake, a modelagem é uma adaptação do StarSchema, onde as tabelas de dimensões podem possuir relacionamentos com outras tabelas auxiliares.

O exemplo de SnowFlake e StarSchema proposta para essa dissertação pode ser observado na Figura 2-15, na qual as dimensões podem ser observadas nos retângulos em cor azul, estruturas de dados auxiliares nos retângulos na cor branca, estrutura de dados opcional em retângulos na cor roxo, e visualizações em retângulos em cor verde. O retângulo em cor laranja representa a estrutura do fato. Os retângulos em cor verde representam as visualizações, que são os relacionamentos entre fato e dimensões com o objetivo de obter informações especificas, para aplicação de mineração de dados.

As dimensões apresentadas pelo modelo são:

- Desenvolvedor: desenvolvedor que realizou a dívida técnica;

[31] O *datawarehouse* possibilita a análise de grandes volumes de dados, coletados dos sistemas transacionais. Esse volume de dados é baseado em séries históricas que possibilitam uma melhor análise de eventos passados, oferecendo suporte às tomadas de decisões presentes e a previsão de eventos futuros.

- Regras: são as regras de *code smell*;
- Tempo: quando o ocorreu o evento do code smell;
- Ação: se o desenvolvedor adicionou ou removeu determinada code smell;
- Projeto: projeto ou repositório de código fonte do qual foi executado a análise estática.
- TD Item: Tipo da dívida técnica, *code smell*, vulnerabilidade, *bugs*. Para essa dissertação é considerado apenas o tipo de dívida técnica de *code smell*. No entanto, caso o interessado tenha interessem em avaliar outros tipos de dívida técnica, basta em suas análises alterar para o tipo do qual deseja.
- Fatos da dívida técnica, tabela que armazena grande quantidade de dados históricos de dívida técnica em função do tempo.

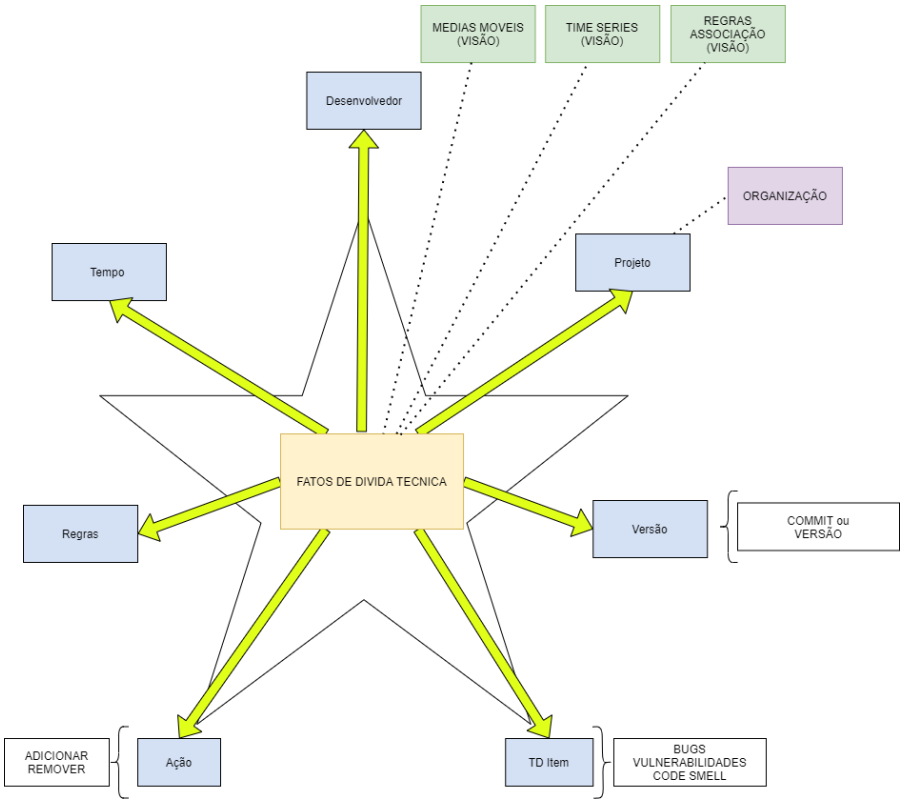


Figura 2-15. Representação das entidades para o modelo de dados *StarSchema* e *SnowFlake*.
Fonte: o Autor.

2.10 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo apresentou a revisão da literatura. Iniciou com alguns dos principais temas e trabalhos relacionados ao tema de code smell, apresentou a descoberta de conhecimento em base de dados, a mineração de dados, o método SQALE e os repositórios de código. Este capítulo analisou também a literatura sobre o tema.

CAPÍTULO 3 - ESTRUTURAÇÃO DA PESQUISA

O objetivo deste capítulo é introduzir o método de pesquisa utilizado neste trabalho e as estratégias adotadas para atingir o objetivo geral.

3.1 CARACTERIZAÇÃO DA PESQUISA

Para Basili et al. (1999), o estudo experimental é uma atividade com o propósito de descobrir algo desconhecido ou de testar uma hipótese envolvendo uma investigação de coleta de dados e de execução de uma análise para determinar o significado dos dados. Essa definição sobre o propósito de descobrir algo desconhecido se assemelha à definição do KDD, que é um processo que permite extrair conhecimento de informações armazenadas em um conjunto grande de dados, utilizando a mineração de dados como parte do processo, conforme detalhado anteriormente na seção 2.8.

Wohlin et al. (2012) apresentam métodos de experimentação voltados à engenharia de software, que consistem em ações planejadas que permitem a descoberta de conhecimento. No entanto, especificamente para descoberta de conhecimento em base de dados existe o processo proposto por Fayyad et al. (1996).

Para garantir que o experimento seja passível de replicação por outros interessados, é necessário descrever um processo que sirva de apoio para a execução do experimento quanto ao processo de dados. Processo é uma agregação de atividades e comportamentos executados por humanos ou máquinas para alcançar um ou mais resultados (KIRCHMER et al., 2019). Processo também é conceituado como a sincronização entre insumos, atividades, infraestrutura e referências necessárias para adicionar valor para o ser humano. As atividades, quando executadas, transformam insumos em algum resultado de valor para quem o executou.

Harrington (1991, p.30) cita a hierarquia do processo conforme os níveis:

- Macroprocesso: envolve mais que uma função na estrutura em uma organização, e sua operação impacta no modo como a organização funciona;
- Processo: é um conjunto de atividades sequenciais e relacionadas entre si, que tomam uma entrada com um fornecedor, acrescentam valor a esta entrada e produzem uma saída para um consumidor;
- Subprocesso: realiza um objetivo específico em apoio ao processo, contribuindo para o objetivo dele, relacionando-se com outro subprocesso ou atividades de forma lógica;
- Atividades: ocorrem dentro do processo ou subprocesso. São geralmente desempenhadas por uma pessoa ou departamento para produzir um resultado, constituindo a maior parte dos fluxogramas;
- Tarefa: parte específica do trabalho, podendo ser um único elemento e/ou um subconjunto de uma atividade. Geralmente, está relacionada a um item que desempenha uma incumbência específica.

Logo, para cumprir o objetivo geral e os objetivos específicos desta dissertação, faz-se necessário utilizar uma combinação entre a Engenharia de Software Experimental (ESE - *Experimental Software Engineering*) proposta por Wohlin et al. (2012) e uma abordagem orientada à busca de conhecimento em banco de dados, como o KDD proposto por Fayyad *et al*. (1996). O KDD faz parte da fase de execução do experimento. Cada um dos processos tem um objetivo e a junção destes processos entrega o resultado esperado.

Quanto aos objetivos de cada um dos processos usados neste trabalho:

i. Processo de negócio: processo composto de subprocessos, atividades e tarefas que permitem ao interessado replicar a abordagem proposta. O detalhamento e mapeamento desse item será tratado no Capítulo 4. A este processo deu-se o nome de TDMINING;

ii. Processo de experimento: composto de tarefas que permitem planejar a condução de um experimento controlado. Processo que será tratado no Capítulo 5;

iii. Processo de dados: composto de tarefas semiautomáticas que permitem obter os dados para a execução do experimento. Processo que será tratado em conjunto com o processo de experimento no Capítulo 5.

As junções e interseções podem ser vistas no Diagrama de Venn apresentado na Figura 3-1:

- negócio ∩ experimento: (A) Estabelecer o que é necessário para entregar resultados;
- experimento ∩ dados: (B) Implementar o plano de como executar o experimento;
- dados ∩ negócio: (C) Identificar o que se pretende obter com a execução;
- negócio ∩ experimento ∩ dados: (R) é o resultado.

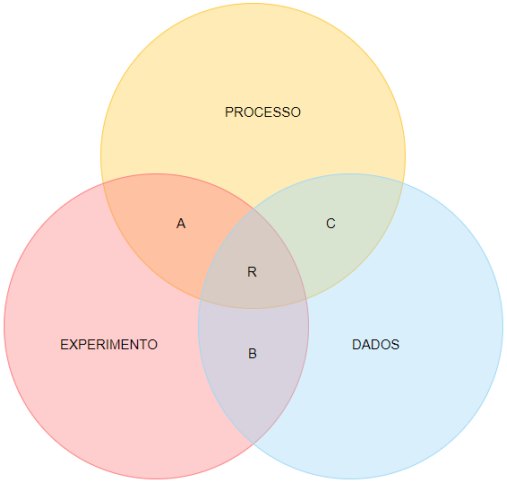


Figura 3-1. Relação entre os Processos – Identificação de interseções. Fonte: o Autor.

As interseções apresentadas na Figura 3-1 podem ser observadas na Tabela 3-1, na qual cada uma das tarefas e/ou subprocessos do processo proposto TDMINING pode estar relacionada a uma fase do experimento, e cada uma das fases do processo do experimento pode estar relacionada a uma das etapas do processo de descoberta do conhecimento.

Tabela 3-1. Relação entre Processo TDMINING x Experimento x Dados. Fonte: o Autor.

Processo Experimento	Processo de Dados	Processo de Negócio
Wohlin *et al*. (2012)	Fayyad *et al*. (1996)	TDMINING

Método de organização		
Fases	Etapas	Atividades e Subprocessos
Fase escopo		
Fase de planejamento > Seleção de indivíduos		Subprocesso 2 - ETL - Atividade 1 - Identificar Repositório de Código Fonte
Fase de Execução > Preparação		Subprocesso 1 - Infraestrutura
Fase de Execução > Execução	Seleção, Pré-processamento, Transformação	Subprocesso 2 - ETL
Fase De Análise e Interpretação	Mineração de dados, Interpretação	Subprocesso 3 - Analytics
Fase de empacotamento		Atividade 1 - Armazenar e versionar Resultado

3.2 ESTRATÉGIA DE PESQUISA

3.2.1 Processo de negócio

Kirchmer et al. (2019), no livro Guia para BPM Corpo de Conhecimento de BPM - BPM CBoK Versão 4.0, apontam que uma organização pode ter três tipos de processos: primário, de apoio/suporte e gerenciais.

Os processos primários são aqueles onde se tem contato com o cliente e se entrega valor diretamente ao cliente. Nos processos de apoio ou suporte não há um contato com cliente, não se entrega valor diretamente ao cliente. O que ocorre é o suporte aos processos primários, pois os processos de suporte entregam valor para outros processos. Por sua vez, os processos gerenciais têm por objetivo buscar a eficiência de outros processos, por meio de monitoramento, controle e ou medição.

Para Kirchmer et al. (2019, p.46):

> “O fato de processos de suporte não entregarem valor diretamente aos clientes não significa que são menos importantes. Os processos de suporte podem ser críticos e estratégicos, pois influenciam diretamente na capacidade de uma organização em executar efetivamente os processos primários”.

A abordagem proposta pode ser caracterizada como um processo de suporte, pois visa auxiliar o processo de desenvolvimento de software que, a depender da organização, pode ser tanto um processo primário como de suporte.

Von Rosing et al. (2014) apresentam a especificação do BPMN *(Business Process Model and Notation*), que é uma das formas de representar um processo a partir de uma notação gráfica, a fim de prover instrumentos para mapear, de maneira padrão, todos os processos. A especificação da notação de modelagem de processos de negócio (BPMN) é mantida pela OMG[32], por meio da versão BPMN 2.0. Desde 2014, a notação BPMN é complementada por um método de decisão de fluxograma chamado de padrão de modelo e notação de decisão, uma vez que a BPMN naturalmente não lida com fluxos de decisões.

A notação possui uma série de ícones padrões para o desenho de processos e um dos ícones mais importantes da notação BPMN é o ícone de atividade e tarefa. No entanto, apenas a representação gráfica do ícone atividade e/ou tarefa não é suficiente para entender o que é necessário para executá-la. Para isso uma das formas de detalhar as tarefas de um processo é por meio do método 5W2H.

O 5W2H é considerado uma ferramenta administrativa e de qualidade, que pode ser aplicada em várias áreas de negócio e em diferentes contextos, como no planejamento estratégico para organizar e guiar a execução de ações dentro da empresa ou até mesmo para planejar a execução de processos. O 5W2H tem como objetivo auxiliar no planejamento de ações, pois ele ajuda a esclarecer questionamentos, que são conhecidos por:

- *WHAT*: **o que** será feito? – Tem o objetivo de determinar a intenção do que se pretende realizar;
- *WHY*: **por que** será feito? – Tem o objetivo de justificar o desenvolvimento do que foi proposto;
- *WHERE*: **onde** será feito? – Tem o objetivo de definir o local de realização;
- *WHEN*: **quando** será feito? – Tem o objetivo de definir o tempo de execução – cronograma e prazos para a execução;

[32] Disponível em: https://www.omg.org/

- *WHO*: **por quem** será feito? – Tem o objetivo de definir quem ou qual área será responsável pela execução do que foi definido;
- *HOW*: **como** será feito? – Tem o objetivo de apontar os métodos ou estratégias utilizadas para a condução do que foi estabelecido;
- *HOW MUCH*: **quanto** custará? – Tem o objetivo de definir o custo e investimento necessário para a realização do que foi proposto.

Para definir por quem será realizado determinada atividade do processo ou dos subprocessos será utilizada a matriz RACI, conforme abordado no livro PMBoK - Guia para o Conjunto de Conhecimento em Gerenciamento de Projetos (PMI, 2010. p.261). A matriz de responsabilidade RACI (*Responsible, Accountable, Consulted, Informed*) é uma ferramenta para gestão de pessoas no qual a partir de uma matriz de atribuição de responsabilidades onde é possível distribuir as atividades para todos envolvidos de um projeto ou processo. Onde cada uma das letras da sigla RACI é descrita como:

- R, responsável, é quem executa a atividade.
- A, autoridade, é aquele que tem o poder de aprovar determinada atividade.
- C, consultado, é quem deve ser consultado, participando de decisões ou da execução da atividade.
- I, informado, é quem precisa ser informado sobre assuntos em relação à atividade.

Tanto a representação gráfica com BPMN como o detalhamento das tarefas com 5W2H para o processo proposto TDMINNING serão apresentados no Capítulo 4.

3.2.2 Experimento

A Engenharia de Software Experimental (ESSE) abordada por Wohlin et al. (2012) é baseada na coleta de dados, compreensão, extração da informação e abstração do conhecimento sobre quando e como os métodos e as técnicas podem e devem ser aplicados no desenvolvimento e manutenção do software, por meio da experimentação.

A ESE surgiu para apoiar o desenvolvimento de processos e produtos com o objetivo de elevar os níveis de qualidade de software, avaliando, caracterizando, predizendo e melhorando as tecnologias antes de serem inseridas na indústria. Entretanto, a pesquisa experimental é uma atividade complexa e que consome tempo, pois gera uma grande quantidade de informações e artefatos. Na engenharia de software torna-se mais complexa do que nos outros meios de produção ou engenharias, pois desenvolvedores de software não produzem sempre os mesmos produtos.

Para Basili et al. (1999), o estudo experimental é uma atividade com o propósito de descobrir algo desconhecido ou de testar uma hipótese envolvendo uma investigação de coleta de dados e de execução de uma análise para determinar o significado dos dados. O propósito de descobrir algo desconhecido é um processo que permite extrair conhecimento de informações armazenadas em um conjunto grande de dados.

Kitchenham et al. (2002) consideram como diretrizes experimentais seis tópicos básicos para condução desse tipo de pesquisa que são: contexto experimental, *design* experimental, condução do experimento e coleta de dados, análise, apresentação de resultados e interpretação de resultados.

Os seis tópicos abordados por Kitchenham et al. (2002), também são abordados por Wohlin et al. (2012), os quais definem um processo de experimentação que aborda as fases de escopo, planejamento, execução, análise e interpretação, apresentação e empacotamento, conforme Figura 3-2. Para cada fase existe uma entrada e uma saída, onde a saída é a entrada para a próxima fase.

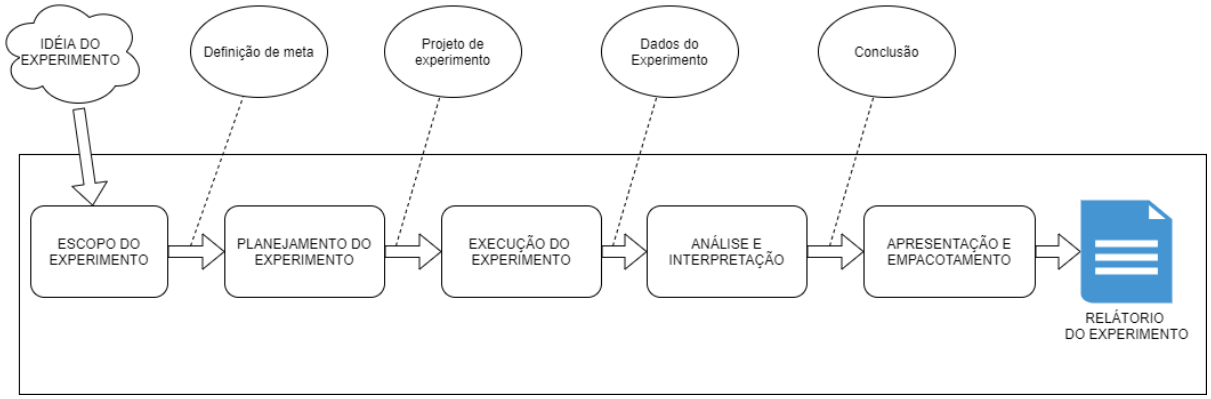


Figura 3-2. Visão do processo de experimentação. Fonte: Adaptado de Wohlin *et al.*, (2012).

O detalhamento das fases da ESE pode ser observado no APÊNDICE B.

3.3 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo apresentou a introdução da estruturação da pesquisa, suas etapas e proposta. A estratégia de pesquisa também foi introduzida, onde o detalhamento será apresentado nos Capítulos 4, e 5 . O Capítulo 4 irá apresentar o detalhamento do processo TDMINING, enquanto o Capítulo 5 irá apresentar em profundidade a utilização do experimento em engenharia de *software* combinado com a mineração de dados.

CAPÍTULO 4 - TDMINING

Este capítulo tem por objetivo apresentar o processo proposto, que denominamos de TDMINING.

4.1 PROCESSO TDMINING

O processo TDMINING é a proposta que esta dissertação apresenta para atender ao objetivo geral deste trabalho que é; **desenvolver uma abordagem para auxiliar o desenvolvedor a conhecer seus comportamentos em relação à dívida técnica do tipo *code smell***.

Para compreender o processo TDMINING é necessário conhecer os conceitos, bem como os ícones da notação BPMN, que serão utilizados na representação do processo, conforme ilustrado na Figura 4-1 e na Figura 4-2:

- *Pool* ou piscina: é um contêiner de determinado processo de negócio.
- *Lanes* ou raias: são utilizadas para definir o escopo de cada processo de negócio na forma de um diagrama, possibilitando a identificação dos responsáveis por realizar cada tarefa.
- Tarefa: Tarefa e atividade parecem ser sinônimos, mas não são. Tarefa é o trabalho prescrito, e refere-se àquilo que a pessoa deve realizar, ou seja, o que deve ser feito.
- Subprocesso: O subprocesso consiste em um conjunto de tarefas dentro de um processo. Por ser mais detalhado do que os processos, ele permite observar os fluxos de trabalho e as atividades necessárias para uma execução.
- *Gateway* ou desvio: Representa uma condição de fluxo para executar as tarefas. São classificados como:

- *Gateway* de decisão inclusivo: ícone que representa uma condição de fluxo inclusiva, em que pode haver uma combinação dos caminhos criados a partir do *gateway*, de acordo com uma informação a ser verificada. Este *gateway* é representado visualmente como o losango com um marcador de círculo dentro dele;
- *Gateway* de decisão exclusivo: ícone o qual é representado por um losango vazio ou com um marcador em forma de “x” e representa uma condição de fluxo exclusiva, em que apenas um dos caminhos criados a partir do gateway será seguido.

- Eventos
  - Evento de início: identifica onde o processo inicia e é representado por um círculo de linha simples;
  - Evento intermediário: identifica a ocorrência de eventos no decorrer do processo e é representado por um círculo de linha dupla;
  - Evento de fim: identifica onde o processo termina e é representado por um círculo de linha grossa.

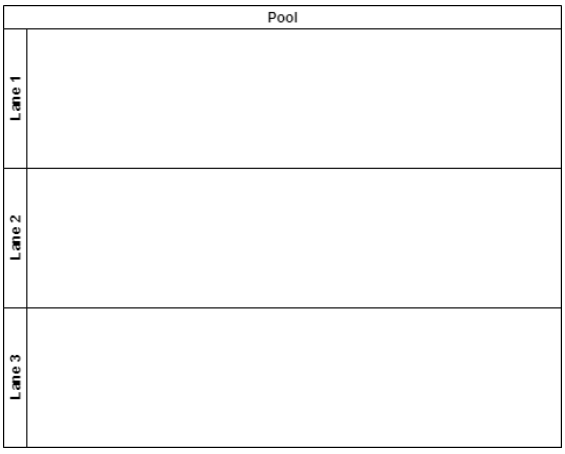


Figura 4-1. Exemplo de *Pool* e suas *Lanes.* Fonte: o Autor.

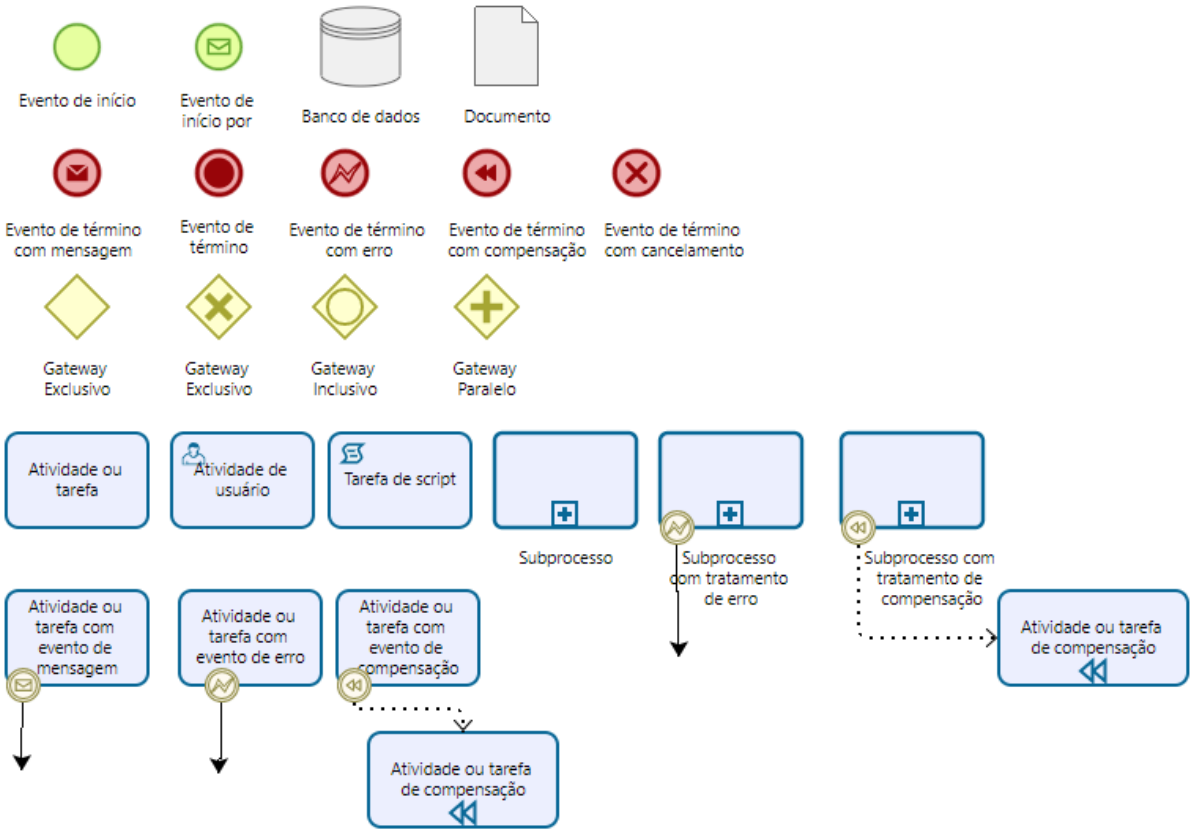


Figura 4-2. Ícones BPMN. Fonte: o Autor.

Para cada uma das atividades identificadas no processo é necessário responder as seguintes perguntas, com base no 5W2H:

- *What*? O que deve ser feito? Passos ou tarefas e descrição da ação;
- *Why*? Por que deve ser feito? Razão, justificativa;
- *Where*? Onde deve ser feito? Área, local;
- *When*? Quando deve ser feito? Data, prazo;
- *Who*? Quem deve fazer? Quem é o responsável pela ação?
- *How*? Como a ação deve ser feita? Qual o método, processo?
- *How Much*? Quanto deve custar para fazer? Custo envolvido na ação.

No entanto, as perguntas “Quando deve ser feito?” e “Quanto deve custar para fazer? o executor da atividade deve determinar. Já a pergunta “Onde deve ser feito?“ para as atividades não informada o executor da atividade deve determinar.

4.1.1 Pressupostos do processo TDMINING

Supõe se antecipadamente que o processo TDMINING será utilizado para extrair conhecimento de *code smell* de repositórios que possuem uma quantidade significativa de *commits* e desenvolvedores. Tendo em vista que o processo visa extrair conhecimentos especificamente por desenvolvedor, idealmente é esperado que os desenvolvedores tenham realizados *commits* suficiente que permitam realizar as análises, ou seja, quanto mais *commits* melhor. No entanto, não existe um número ou fórmula que determine a quantidade ideal de *commits*. Mas é esperado que o processo seja utilizado em situações em que o interessado não consiga de forma manual analisar um grande volume de dados. Podemos considerar o exemplo a seguir.

Por exemplo, o projeto opensource WordPress[33], em seus primeiros 41.285 *commits* houve a colaboração de 122 desenvolvedores. Isso significa que a média de *commits* por desenvolvedor é de 338. Onde 100 desenvolvedores realizaram menos de 338 *commits*. Ou seja, 22 desenvolvedores tem a quantidade de *commits* superior à média. Esses 22 desenvolvedores foram responsáveis por 36.079 dos 41.285 *commits*, ou seja, 87% dos commits realizados por apenas 22 desenvolvedores. Também há situação em que 16 desenvolvedores realizaram apenas 1 *commit* cada. Logo, é possível determinar para esse exemplo, que para esses 22 desenvolvedores o processo TDMINING irá gerar conhecimento sobre seus *code smells*, e 16 desenvolvedores possui dados insuficientes para análise, logo prejudicando a análise. Também é importante observar que um *commit* pode ter nenhum ou muitos *code smells*, para isso será abordado uma hipótese no experimento, item 5.1.2b.

Outra observação em relação ao processo está em que o interessando pode executar o experimento em uma quantidade ilimitada de repositório, como dito

[33] Disponível em https://github.com/WordPress/wordpress-develop

anteriormente, quanto mais *commits* melhor. Em situações em projetos opensource ou em projetos da indústria, onde desenvolvedores atuam em mais de um projeto, analisar muitos repositórios que possuem desenvolvedores em comum permite obter mais dados, logo facilitando as análises. Visto que as análises executadas não se limitam ao projeto, no modelo dimensional de dados apresentado na Figura 2-15 é possível delimitar os dados por desenvolvedor, por projeto, por regra (*code smells, bugs*, vulnerabilidade), por ação (adicionar/remover), por data e por versão (um *commit* pode ser uma versão ou um conjunto de *commits* pode ser uma versão).

4.2 Detalhamento do processo TDMINING

O processo TDMINING, representado na Figura 4-3 demonstra a raia Desenvolvimento. Embora uma empresa, possam ter uma área de infraestrutura, as atividades de infraestrutura foram consideradas em um subprocesso. No entanto, nada impede que interessados de forma autônoma executem o processo, desde que tenham acesso aos recursos necessários para sua replicação.

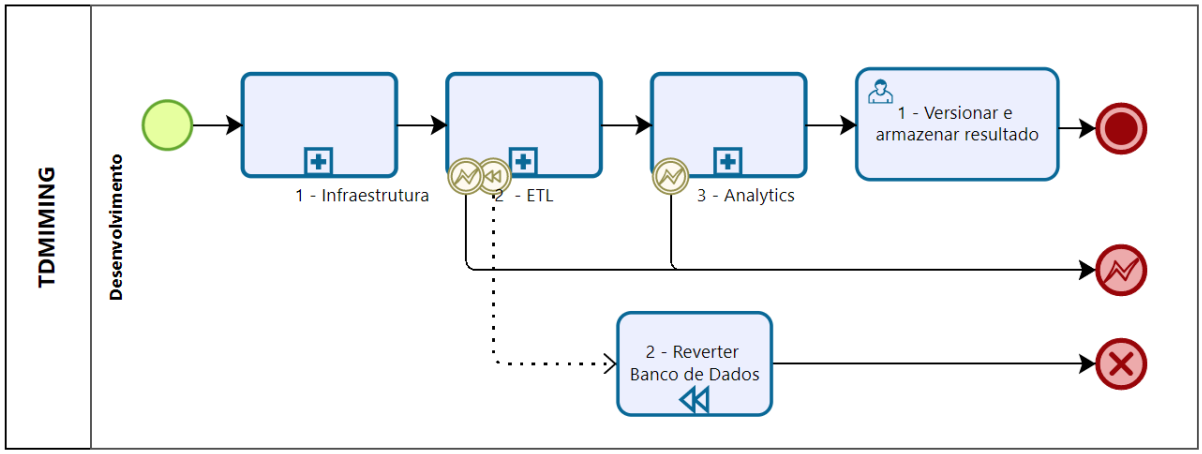


Figura 4-3. Processo TDMINING. Fonte: o Autor.

Para todas as atividades do processo TDMINING na pergunta "**quem deve fazer?"** será informado um responsável. As responsabilidades de cada uma das atividades em relação ao perfil, pode ser observada na matriz RACI no Quadro 4-1, a matriz RACI foi explicada no item 3.2.1.

Quadro 4-1: Matriz RACI do processo TDMINING. Fonte: o Autor.

	INFRAESTRUTURA	BANCO DE DADOS	INTERESSADO	ESPECIALISTA

SUBPROCESSO 1 - INFRAESTRUTURA				
ATIVIDADE 1 - SOLICITAR INFRAESTRUTURA			R/A	
ATIVIDADE 2 - PREPARAR INFRAESTRUTURA	R		A	
ATIVIDADE 3 - CONCEDER ACESSOS	R		A	
ATIVIDADE 4 - PREENCHER DOCUMENTO INFRAESTRUTURA E ACESSOS	R		A	
ATIVIDADE 5 - PREPARAR INFRAESTRUTURA DE BANCO DE DADOS	I	R	A	
SUBPROCESSO 2- ETL				
ATIVIDADE 1 - IDENTIFICAR REPOSITÓRIO DE CÓDIGO-FONTE			R/A	
ATIVIDADE 2 - PARAMETRIZAR ARQUIVO PYTHON			R/A	
ATIVIDADE 3 - PARAMETRIZAR ARQUIVO DE ETL SQL			R/A	
ATIVIDADE 4 - EXECUTAR ARQUIVO PYTHON			R/A	
ATIVIDADE 5 - EXECUTAR SCRIPT DE BANCO DE DADOS		R	R/A	
ATIVIDADE 6 - IMPORTAR DADOS			R/A	
ATIVIDADE 7 - ENTREGAR DADOS			R/A	
ATIVIDADE 8 - VALIDAR ERRO			R/A	
SUBPROCESSO 3 - ANALYTICS				
ATIVIDADE 1 - PREPARAR ARQUIVOS PARA EXECUÇÃO			R/A	
ATIVIDADE 2 - IMPORTAR ARQUIVOS JUPYTER			R/A	
ATIVIDADE 3 - EXECUTAR SCRIPTS JUPYTER			R/A	
ATIVIDADE 4 - DOCUMENTAR RESULTADOS			R/A	
ATIVIDADE 5 - VALIDAR INFORMAÇÕES			C/I	R/A
ATIVIDADE 6 - INFORMAR RESTRIÇÕES			C/I	R/A
ATIVIDADE 7 - PREENCHER RESTRIÇÕES			C/I	R/A
ATIVIDADE 8 - EXECUTAR SCRIPT DE RESTRIÇÕES			R/A	C/I
PROCESSO PRINCIPAL - TDMINING				
ATIVIDADE 1 - VERSIONAR E ARMAZENAR RESULTADO			R/A	
ATIVIDADE 2 - REVERTER BANCO DE DADOS		R	A/C/I	

O detalhamento dos conhecimentos necessários para cada um dos perfis de responsabilidade pode ser observado no Quadro 4-2, observado que para cada conhecimento é atribuído um nível de requerimento. O nível de requerimento é tratado conforme a RFC2119[34].

[34] Disponível em: https://datatracker.ietf.org/doc/html/rfc2119 e https://www.ietf.org/rfc/rfc2119.txt.

Quadro 4-2: Conhecimentos necessários por responsável. Fonte: o Autor.

RESPONSÁVEL	CONHECIMENTOS NECESSÁRIOS	PERFIL	POSSÍVEIS FUNÇÕES
INFRAESTRUTURA	Instalação de softwares (*MUST*), administração de acessos em banco de dados (*MUST*), administração de acessos em aplicações (*MUST*).	JUNIOR (*MUST NOT*) , PLENO (*SHOULD*), SENIOR (*SHOULD*)	Analista de Infraestrutura, Administradores de servidor, Técnico de Informática
BANCO DE DADOS	Administração de acessos em banco de dados (*MUST*), execução de scripts SQL (*MUST*).	JUNIOR (*MUST NOT*) , PLENO (*SHOULD*), SENIOR (*SHOULD*)	Administrator de infraestrutura de banco de dados; Administrador de dados.
INTERESSADO	Conhecimento em linguagem de programação Python (*SHOULD*), conhecimento em linguagem SQL (*SHOULD*).	JUNIOR (*SHOULD*) , PLENO (*SHOULD*), SENIOR (*SHOULD*)	Desenvolvedor, Líder Técnico; Arquiteto de *Software*, Analista de Sistemas, Analista de Qualidade de Software, Lider Técnico, Gerente de Desenvolvimento de *Software*, Engenheiro de *Software*.
ESPECIALISTA	Conhecimento sobre o produto de software (*MUST*) , conhecimento sobre qualidade de código fonte (*SHOULD*), conhecimento em linguagem de programação Python (*SHOULD*), conhecimento em linguagem SQL (*SHOULD*), conhecimento da arquitetura do software analisado (*MUST*), conhecimento sobre estatística (*MAY*), conhecimento sobre ciência de dados (*MAY*), conhecimento em boas práticas de programação (*SHOULD*)	JUNIOR (*MUST NOT*) , PLENO (*SHOULD*), SENIOR (*SHOULD*)	Desenvolvedor, Líder Técnico; Arquiteto de *Software*, Analista de Sistemas, Analista de Qualidade de *Software*, Lider Técnico, Gerente de Desenvolvimento de *Software*, Engenheiro de *Software*.

Conforme observado no Quadro 4-2, o que determina cada um dos papéis responsáveis pela execução das tarefas é o conhecimento. Se um profissional possui todos os conhecimentos necessários para os quatro papéis, logo o mesmo, pode atuar na execução das tarefas. No entanto, caso no processo não seja possível identificar um profissional que possa atuar como ESPECIALISTA, o processo pode seguir o fluxo DC1 “Não há especialista” conforme será abordado no item 4.2.3. O ônus em não ter o papel do ESPECIALISTA está em relação que possíveis *code smells* poderiam ser

desconsiderados da análise, exemplo, *code smell* aceitos por ser inerente a arquitetura.

4.2.1 SUBPROCESSO 1 – INFRAESTRUTURA

O subprocesso Infraestrutura está representado na Figura 4-4.

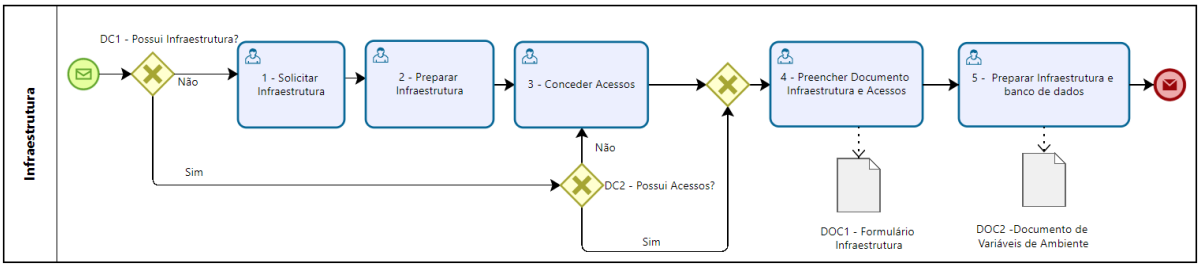


Figura 4-4. Subprocesso Preparar Infraestrutura. Fonte: o Autor.

SUBPROCESSO 1 - ATIVIDADE 1 – SOLICITAR INFRAESTRUTURA

O que deve ser feito?

- Comunicar ao responsável pela infraestrutura a necessidade de acesso a servidores de SonarQube, Jupyter Notebook e Banco de dados.

Por que deve ser feito?

- Para viabilizar a infraestrutura para executar o subprocesso de ETL e Analytics.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante envio de solicitação (e-mail, sistema de helpdesk, e/ou comunicação informal). Acesso a e-mail, sistema de helpdesk ou outro meio de comunicação para o executor.

SUBPROCESSO 1 - ATIVIDADE 2 – PREPARAR INFRAESTRUTURA

Caso DC1 – Se a resposta for "NÃO".

O que deve ser feito?

- Preparar servidor com SonarQube;
- Preparar no mesmo servidor o Jupyter Notebook;
- Preparar no mesmo servidor o banco de dados Microsoft SQL Server* (pode ser substituído por outro SGBD);
- Instalar no mesmo servidor Git e Python;
- Instalar o *SonarScanner* no servidor;
- Garantir que os recursos computacionais estão devidamente configurados e os acessos corretamente configurados.
    - Se houver qualquer erro de acesso ou infraestrutura para o SUBPROCESSO 2 – ETL e SUBPROCESSO 3 – Analytics o processo TDMINING será cancelado por ERRO.

Por que deve ser feito?

- Para conceder acessos e infraestrutura para o solicitante executar o subprocesso de ETL e Analytics.

Quem deve fazer?

- PERFIL INFRAESTRUTURA.

Como a ação deve ser feita?

- A instalação do SonarQube deve seguir os procedimentos determinados pelo fornecedor, disponível em https://docs.sonarqube.org/latest/ e https://docs.sonarqube.org/latest/requirements/hardware-recommendations/ ;
- A instalação do Jupyter Notebook deve seguir os procedimentos determinados pelo fornecedor, disponível em https://jupyter.org/install;
- A instalação do banco de dados deve considerar a recomendação do fornecedor;
- Instalação do GIT, disponível em https://git-scm.com/book/en/v2/Getting-Started-Installing-Git ;

- Instalação do Python, disponível em https://docs.python.org/3/ ;
- Instalação do SonarScanner no servidor, disponível https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/ .

SUBPROCESSO 1 - ATIVIDADE 3 – CONCEDER ACESSOS

O que deve ser feito?

- Criar *login* e senha ao servidor com SonarQube;
- Criar *login* e senha ao servidor com Jupyter Notebook;
- Criar *login* e senha ao servidor com banco de dados Microsoft SQL Server * (pode ser substituído por outro SGBD).

Por que deve ser feito?

- Para conceder acessos e infraestrutura para o solicitante executar o subprocesso de ETL e Analytics.

Quem deve fazer?

- PERFIL INFRAESTRUTURA.

Como a ação deve ser feita?

- Mediante acesso do executor aos servidores para criação de perfil e usuário de acesso aos sistemas.

SUBPROCESSO 1 - ATIVIDADE 4 – PREENCHER DOCUMENTO INFRAESTRUTURA E ACESSOS

O que deve ser feito?

- Preencher no documento DOC1 (APÊNDICE C), os acessos para solicitante, detalhando login e senha, endereço para acesso ao SonarQube, Jupyter Notebook e banco de dados.
- Enviar documento ao solicitante.

Por que deve ser feito?

- Para o solicitante ter conhecimento dos seus acessos aos servidores solicitados.

Onde deve ser feito?

- A descrição de onde será realizada no documento DOC1.

Quem deve fazer?

- PERFIL INFRAESTRUTURA.

Como a ação deve ser feita?

- Preencher o documento DOC1.
- Enviar o documento para o solicitante.

SUBPROCESSO 1 - ATIVIDADE 5 – PREPARAR INFRAESTRUTURA DE BANCO DE DADOS

O que deve ser feito?

- Aplicar o *script* CREATE_DDL.sql em banco de dados, conforme *script* no APÊNDICE D. O objetivo do *script* CREATE_DDL.sql é criar as estruturas de dados de dimensões e fatos conforme exibido no Quadro 4-3. Na Figura 4-5, a cor vermelha é definida para as tabelas que representam o *starschema* e em azul as tabelas que complementam para a estrutura de *snowflake*.
- Preenchimento do DOC2 (APÊNDICE E).

Por que deve ser feito?

- Para criar o modelo de dados para persistência de dados;
- Para preparar execução da coleta de dados.

Onde deve ser feito?

- No servidor de banco de dados descrito no documento DOC1 (APÊNDICE C)

Quem deve fazer?

- PERFIL BANCO DE DADOS.

Como a ação deve ser feita?

- Mediante o acesso ao banco de dados descrito no DOC1 (APÊNDICE C).

Quadro 4-3. Dicionário de dados TDMINING. Fonte: o Autor.

TABELA	COLUNA	TIPO	DESCRIÇÃO
DM_ACTION	DESCRIPTION	texto	Descrição da ação
DM_ACTION	ID_ACTION	número	Identificador da ação
DM_COMMITS	COMMIT_DATE	Data, hora e minuto	Data do *commit*
DM_COMMITS	COMMIT_HASH	texto	Identificador do *commit*
DM_COMMITS	ID_COMMIT	número	Identificador do *commit*
DM_COMMITS	ID_DEVELOPER	número	Identificador do desenvolvedor
DM_COMMITS	ID_PROJECT	número	Identificador do projeto
DM_COMMITS	ID_TIME	número	Identificador do tempo
DM_DEVELOPER	DEVELOPER_NAME	texto	Nome do desenvolvedor ou identificador ou e-mail
DM_DEVELOPER	ID_DEVELOPER	número	Identificador do desenvolvedor
DM_PROJECT	ID_PROJECT	número	Identificador do projeto
DM_PROJECT	PROJECT_NAME	texto	Nome do projeto
DM_RULE	ID_RULE	número	Identificador da regra
DM_RULE	ORIGIN_RULE_ID	texto	Identificador da origem da regra
DM_RULE	RULE_DESCRIPTION	texto	Descrição da regra
DM_RULE	RULE_NAME	texto	Nome da regra
DM_SEVERITY	DESCRIPTION	texto	Descrição da severidade
DM_SEVERITY	ID_SEVERITY	número	Identificador da severidade
DM_TDITEM	DESCRIPTION	texto	Descrição do tipo de dívida técnica
DM_TDITEM	ID_TDITEM	número	Identificador do tipo de item de dívida técnica
DM_TIME	DATE	Data	Data no formato dd-mm-yyyy
DM_TIME	ID_TIME	número	Identificador da data
DM_VERSION	ID_COMMIT	número	Identificador do *commit*
DM_VERSION	ID_PROJECT_VERSION	número	Identificador da versão para o projeto
DM_VERSION	ID_VERSION	número	Identificador da versão
FT_TECHNICALDEBT	EFFORT	número	Esforço para correção da dívida técnica em minutos
FT_TECHNICALDEBT	ID_ACTION	número	Identificador da ação
FT_TECHNICALDEBT	ID_DEVELOPER	número	Identificador do desenvolvedor

TABELA	COLUNA	TIPO	DESCRIÇÃO
FT_TECHNICALDEBT	ID_FTECHNICALDEBT	número	Identificador do fato de dívida técnica
FT_TECHNICALDEBT	ID_PROJECT	número	Identificador do projeto
FT_TECHNICALDEBT	ID_RULE	número	Identificador da regra
FT_TECHNICALDEBT	ID_SEVERITY	número	Identificador da severidade
FT_TECHNICALDEBT	ID_TDITEM	número	Identificador do tipo de dívida técnica
FT_TECHNICALDEBT	ID_TIME	número	Identificador da data de criação da dívida técnica
FT_TECHNICALDEBT	ID_VERSION	número	Identificador da versão do produto que foi criado a dívida técnica
FT_TECHNICALDEBT	IGNORE	número	Identificador se o dado deve ser ignorado nas análises, 1 -SIM , 0-NÃO
ORGANIZATION	ID_ORGANIZATION	número	Identificador da organização
ORGANIZATION	ORGANIZATION_NAME	texto	Nome da organização ou empresa
ORGANIZATION_PROJECT	ID_ORGANIZATION	número	Identificador da organização
ORGANIZATION_PROJECT	ID_ORGANIZATION_PROJECT	número	Identificador do projeto na organização
ORGANIZATION_PROJECT	ID_PROJECT	número	Identificador do projeto
PROJECT_VERSION	DATE_VERSION	Data, hora e minuto	Data da versão
PROJECT_VERSION	DATE_VERSION_STR	texto	Data da versão em formato texto
PROJECT_VERSION	ID_PROJECT	número	Identificador do projeto
PROJECT_VERSION	ID_PROJECT_VERSION	número	Identificador do verão do projeto
PROJECT_VERSION	VERSION_NAME	texto	Nome da versão
FT_TECHNICALDEBT_IGNORE_CONDITION	EFFORT	número	Esforço para correção da dívida técnica em minutos
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_ACTION	número	Identificador da ação
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_DEVELOPER	número	Identificador do desenvolvedor
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_FTECHNICALDEBT	número	Identificador do fato de dívida técnica

TABELA	COLUNA	TIPO	DESCRIÇÃO
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_PROJECT	número	Identificador do projeto
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_RULE	número	Identificador da regra
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_SEVERITY	número	Identificador da severidade
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_TDITEM	número	Identificador do tipo de dívida técnica
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_TIME	número	Identificador da data de criação da dívida técnica
FT_TECHNICALDEBT_IGNORE_CONDITION	ID_VERSION	número	Identificador da versão do produto que foi criado a dívida técnica

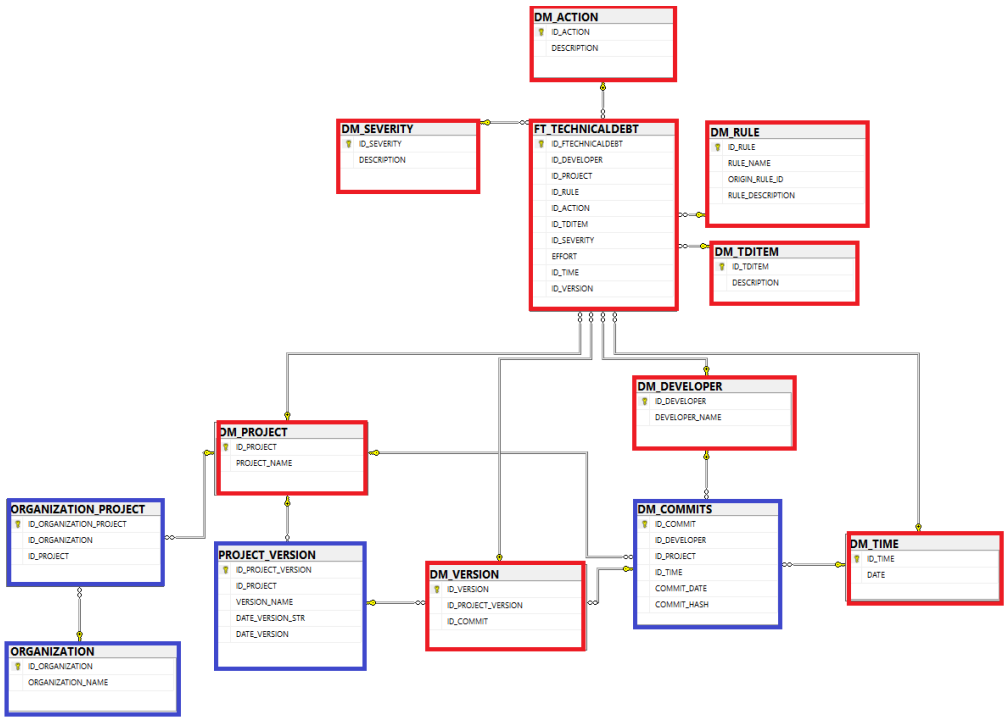


Figura 4-5. Diagrama entidade relacionamento – dimensões x fato. Fonte: o Autor.

4.2.2 SUBPROCESSO 2 – ETL

O subprocesso ETL tem o propósito entregar os dados necessários para a mineração, conforme tarefas detalhadas na Figura 4-6.

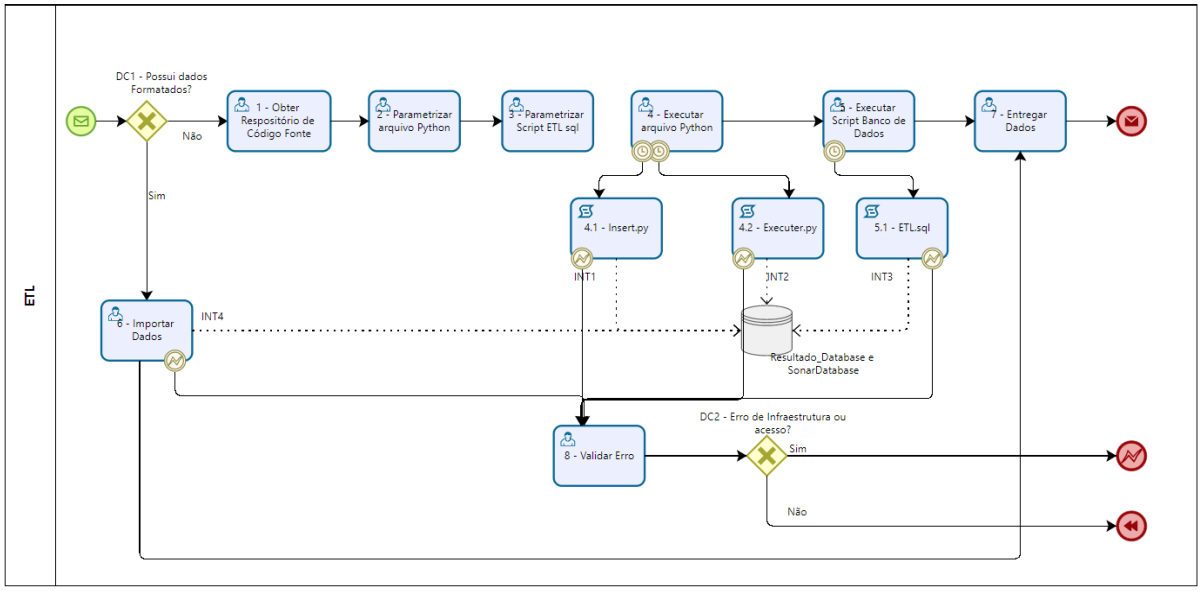


Figura 4-6. Subprocesso ETL. Fonte: o Autor.

SUBPROCESSO 2 - ATIVIDADE 1 – OBTER REPOSITÓRIO DE CÓDIGO-FONTE

O que deve ser feito?

- Identificar e obter repositório de código-fonte;
- Realizar cópia do repositório por meio de GIT CLONE.

Por que deve ser feito?

- Para obter o histórico do controle de versão do produto selecionado.

Onde deve ser feito?

- No servidor identificado no DOC1 (APÊNDICE C).

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante o acesso ao servidor descrito no DOC1 deve-se criar um diretório para realizar cópia do repositório de código-fonte. Por meio da linha de comando GIT CLONE, conforme documentação disponível em https://github.com/git-guides/git-

clone#:~:text=The%20git%20clone%20command%20is,your%20own%20machine%20by%20cloning.

SUBPROCESSO 2 - ATIVIDADE 2 – PARAMETRIZAR ARQUIVO PYTHON

O que deve ser feito?

- Alterar o arquivo config.ini informado no DOC2 (APÊNDICE E), o diretório do repositório de código e endereço do banco de dados.

Por que deve ser feito?

- Para preparar os arquivos insert.py e executer.py para execução.

Onde deve ser feito?

- No servidor informado no DOC1 (APÊNDICE C)

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante alteração de variáveis de ambiente, utilizando de editor de código-fonte/texto de preferência do executor.

Quanto deve custar para fazer?

- A definir pelo PERFIL INTERESSADO.

SUBPROCESSO 2 - ATIVIDADE 3 – PARAMETRIZAR ARQUIVO DE ETL SQL

O que deve ser feito?

- Alterar no arquivo ANALISYS_TEMP.sql o nome dos *schemas,* conforme informado no DOC2 (APÊNDICE E).

Por que deve ser feito?

- Para realizar a transformação e carga dos dados no formato do StarSchema.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante alteração de código-fonte em linguagem SQL, utilizando de editor de código-fonte /texto de preferência do executor.

Quanto deve custar para fazer?

- A definir pelo PERFIL INTERESSADO.

SUBPROCESSO 2 - ATIVIDADE 4 – EXECUTAR ARQUIVO PYTHON

O que deve ser feito?

- Executar o arquivo insert.py, aguardar a conclusão do processamento;
- Executar o arquivo executer.py, aguardar a conclusão do processamento.

Por que deve ser feito?

- Para realizar a criação dos dados para análise.

Onde deve ser feito?

- Os arquivos devem ser executados no servidor informado no DOC1 (APÊNDICE C).

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante a execução dos arquivos por linha de comando;
- Python <nome_do_arquivo.py>

SUBPROCESSO 2 - ATIVIDADE 5 – EXECUTAR *SCRIPT* DE BANCO DE DADOS

O que deve ser feito?

- Executar o arquivo ANALISYS_TEMP.sql. A tarefa é concluída quando todos os procedimentos no arquivo foram executados com sucesso.

Por que deve ser feito?

- Para realizar a transformação de dados para o modelo STARCHEMA proposto.

Quem deve fazer?

- Interessado, desenvolvedor ou pesquisador.

Como a ação deve ser feita?

- Mediante a execução do conteúdo do arquivo em interface para execução de comandos SQL.

SUBPROCESSO 2 - ATIVIDADE 6 – IMPORTAR DADOS

O que deve ser feito?

- Importar dados formatados para a base de dados. Dados formatados conforme demonstrado no Quadro 4-3. Para essa tarefa não é obrigatório que os dados que serão persistidos tenham origem do SonarQube, podendo ser originados de qualquer ferramenta de análise estática, desde que sigam o modelo de dados proposto.

Por que deve ser feito?

- Para realizar a análise dos dados.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Por meio de *script* ou *software* identificados pelo interessado.

Quanto deve custar para fazer?

- A definir pelo PERFIL INTERESSADO.

SUBPROCESSO 2 - ATIVIDADE 7 – ENTREGAR DADOS

O que deve ser feito?

- Exportar os dados das tabelas criadas para o formato CSV ou fornecer a *string* de conexão com o banco de dados.

Por que deve ser feito?

- Para permitir a execução da análise dos dados.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Os arquivos com extensão CSV podem ser extraídas por tabela, conforme IDE de preferência do executor. A *string* de conexão pode ser informada por *e-mail*, texto ou qualquer canal de comunicação de preferência do executor.

SUBPROCESSO 2 - ATIVIDADE 8 – VALIDAR ERRO

O que deve ser feito?

- Exportar os dados das tabelas criadas para o formato CSV ou fornecer a *string* de conexão com o banco de dados.

Por que deve ser feito?

- Para permitir a execução da análise dos dados.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Os arquivos com extensão CSV podem ser extraídas por tabela, conforme IDE de preferência do executor. A *string* de conexão pode ser informada por *e-mail*, texto ou qualquer canal de comunicação de preferência do executor.

4.2.3 SUBPROCESSO 3 – *ANALYTICS*

O subprocesso *Analytics,* ilustrado na Figura 4-7, tem por objetivo entregar o resultado da mineração de dados meio de gráficos e tabelas.

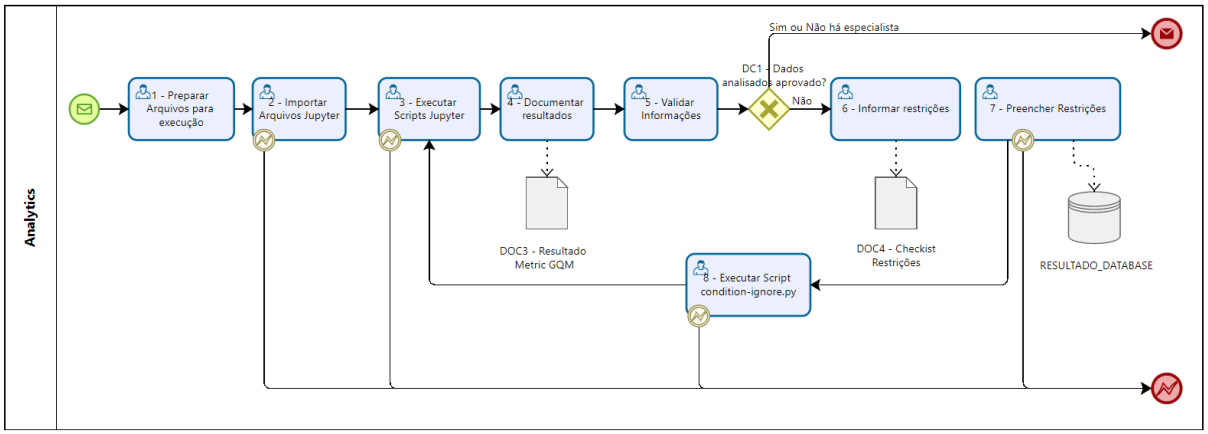


Figura 4-7. Subprocesso Analytics. Fonte: o Autor.

SUBPROCESSO 3 – ATIVIDADE 1 – PREPARAR ARQUIVOS PARA EXECUÇÃO

O que deve ser feito?

- Garantir que o arquivo config.ini está com as variáveis de ambiente conforme DOC2 (APÊNDICE E).

Por que deve ser feito?

- Garantir que os *scripts* consigam conectar ao banco de dados.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante utilização de editor de texto ou código-fonte.

SUBPROCESSO 3 – ATIVIDADE 2 – IMPORTAR ARQUIVOS JUPYTER

O que deve ser feito?

- Importar arquivos com a extensão IPYNB para a aplicação do Jupyter Notebook.

Por que deve ser feito?

- Para poder executar os arquivos no servidor Jupyter Notebook.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante a utilização da interface gráfica do Jupyter Notebook.

SUBPROCESSO 3 – ATIVIDADE 3 – EXECUTAR *SCRIPTS* JUPYTER

O que deve ser feito?

- Executar no Jupyter Notebook os *scripts:*
  - teste-normalidade.py
  - teste-hipotese.py
  - question-1-metric-1.ipynb
  - question-1-metric-2.ipynb
  - question-2-metric-1.ipynb
  - question-3-metric-1.ipynb
  - question-4-metric-1.ipynb
  - question-5-metric-1.ipynb.

Os scripts apontados estão disponibilizados na internet através do endereço https://github.com/manoelvsneto/tdmining, para cada uma das tarefas tem relação a um script Jupyter Notebook. A relação dos scripts em relação e seus objetivos podem ser observados no APÊNDICE H, onde também é possível verificar quais os tipos de informação que a execução de cada *script* retorna como resultado.

Por que deve ser feito?

- Para obter o resultado da métrica de interesse.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante a utilização da interface gráfica do Jupyter Notebook.

SUBPROCESSO 3 – ATIVIDADE 4 – DOCUMENTAR RESULTADOS

O que deve ser feito?

- Capturar o resultado da execução dos *scripts* da Tarefa 3 e documentar no arquivo DOC3.

Por que deve ser feito?

- Para tomar decisões sobre o resultado das métricas executadas.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Mediante preenchimento do documento DOC3.

SUBPROCESSO 3 – ATIVIDADE 5 – VALIDAR INFORMAÇÕES

O que deve ser feito?

- Analisar os dados apresentados no DOC3.

Por que deve ser feito?

- Para avaliar a existência de dados que devem ser desconsiderados. Pois em detrimento do conhecimento do ESPECIALISTA, pode ocorrer que determinada regra de *code smell* deva ser desconsiderada.

Quem deve fazer?

- PERFIL ESPECIALISTA.

Como a ação deve ser feita?

- Leitura e interpretação dos resultados apresentados no DOC3 .

SUBPROCESSO 3 – ATIVIDADE 6 – INFORMAR RESTRIÇÕES

O que deve ser feito?

- Preencher o documento DOC4 – Checklist de restrições com as condições que devem ser ignoradas para tratamento dos dados.

Por que deve ser feito?

- Por motivo de ser um *code smell* ser aceito no projeto, ou até mesmo ser inerente a arquitetura do sistema. Ou em condições que deva ser removida da análise um determinado desenvolvedor ou data.
    - Idealmente é esperado que não haja viés do ESPECIALISTA em relação aos dados.

Quem deve fazer?

- PERFIL ESPECIALISTA.

Como a ação deve ser feita?

- Preenchimento de formulário.

SUBPROCESSO 3 – ATIVIDADE 7 – PREENCHER RESTRIÇÕES

O que deve ser feito?

- Preencher o formulário conforme o exemplo no DOC4 – CHECKLIST DE RESTRIÇÕES.

Por que deve ser feito?

- Para ignorar dados desnecessários na análise.

Quem deve fazer?

- PERFIL ESPECIALISTA.

Como a ação deve ser feita?

- Inserção de dados em banco de dados na tabela FT_TECHNICALDEBT_IGNORE_CONDITION.

SUBPROCESSO 3 – ATIVIDADE 8 – EXECUTAR *SCRIPT* DE RESTRIÇÕES

O que deve ser feito?

- Executar o *script* condition-ignore.py.

Por que deve ser feito?

- Para ignorar os dados que devem ser desconsiderados das análises.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Por meio de execução de *script* Python em ambiente de desenvolvimento.

4.2.4 PROCESSO PRINCIPAL - ATIVIDADE 1 – VERSIONAR E ARMAZENAR RESULTADO

O que deve ser feito?

- Atribuir versão ao documento DOC3 e armazenar em sistema de controle de versão, de gestão de conteúdo ou local de conhecimento dos interessados.

Por que deve ser feito?

- Manter histórico e fonte de referência para tomada de decisão.

Quem deve fazer?

- PERFIL INTERESSADO.

Como a ação deve ser feita?

- Preenchimento do documento DOC3.
- Armazenamento do documento DOC3 em local definido pelo PERFIL INTERESSADO.

Quanto deve custar para fazer?

- A definir pelo PERFIL INTERESSADO.

4.2.5 PROCESSO PRINCIPAL - ATIVIDADE 2 – REVERTER BANCO DE DADOS

O que deve ser feito?

- Rever banco de dados TDMINING;

Por que deve ser feito?

- Para permitir que seja possível realizar o experimento após um erro.

Quem deve fazer?

- PERFIL BANCO DE DADOS.

Como a ação deve ser feita?

- Execução do *script* revert-database.py.

4.3 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo apresentou a estruturação do processo TDMINING, bem como as entradas e saídas do processo e subprocessos e o detalhamento das atividades para a execução. O próximo capítulo irá apresentar em profundidade a utilização do processo TDMINING e experimento em engenharia de *software*.

CAPÍTULO 5 - DESENVOLVIMENTO DA PESQUISA

In God we trust. All others must bring data.

W. Edwards Deming

Para Wohlin *et al.* (2012, p.16):

> "Experimentos são lançados quando queremos controle sobre a situação e queremos manipular o comportamento direta, precisa e sistematicamente. Além disso, experimentos envolvem mais de um tratamento para comparar os resultados".

Os subtópicos deste capítulo visam descrever como foi realizado o experimento em relação às etapas definidas por Wohlin et al. (2012), combinado ao processo TDMINING.

5.1 EXECUÇÃO DO PROCESSO E EXPERIMENTO

Para executar o experimento é necessário não somente entender o fluxo do processo TDMINING, mas também sua relação com as fases do experimento. O processo TDMINING procura abstrair a complexidade do experimento por meio de processos, tarefas e artefatos. A relação entre o processo TDMINING, experimento e KDD pode ser observada na Tabela 3-1 na qual as atividades e subprocessos do TDMINING são mais perceptíveis em relação às fases da execução do experimento proposto por Wohlin et al. (2012), e com as etapas do KDD.

Seguindo o fluxo do processo TDMINING, o primeiro passo é o interessado solicitar a criação de infraestrutura para execução do experimento conforme PROCESSO PRINCIPAL - ATIVIDADE 1 – Solicitar Infraestrutura. A saída desta atividade é o DOC1 (exemplo no APÊNDICE C), que respectivamente é a entrada

para o SUBPROCESSO 1 - Infraestrutura. O SUBPROCESSO 1 – Infraestrutura entrega ao solicitante os endereços (DNS e/ou IP) e acessos aos ambientes computacionais necessários para o experimento, mediante o documento DOC2, ver exemplo no APÊNDICE E.

Com as informações apresentadas no DOC2, o interessado prepara a infraestrutura de banco de dados por meio da execução da atividade SUBPROCESSO 1 - Infraestrutura - ATIVIDADE 5 – Preparar infraestrutura de banco de dados.

Com a infraestrutura pronta, seguindo a sequência do processo TDMINING a próxima etapa é identificar qual repositório irá coletar as informações, por meio da execução da SUBPROCESSO 2 – ETL. Optando em seguir o fluxo “DC1 – Possui dados formatados?” se SIM deve seguir para SUBPROCESSO 2 – ETL - ATIVIDADE 1 – Obter repositório de código-fonte SENÃO SUBPROCESSO 2 – ETL - ATIVIDADE 6 – Importar dados.

O processo TDMINING atende principalmente a execução de um experimento, a qual pode ser observada no item 5.2 e item 5.3 e o detalhamento das fases de um experimento estão descritas no APÊNDICE B.

5.1.1 Fase de escopo

O seguinte escopo foi definido, seguindo as orientações da abordagem GQM (*Goal – Question – Metric)* proposta por Basili *et al*. (1994):

Analisar: Informações obtidas da mineração de métricas *code smells.*

Com o propósito de: executar um experimento em engenharia de *software.*

Em relação a: permitir que o desenvolvedor tenha conhecimento de suas tendências na criação de *code smell.*

Do ponto de vista: do desenvolvedor profissional, da qualidade do código-fonte e da gestão de DT.

Com o resultado da abordagem do GQM, foi elaborada a questão de pesquisa conforme a Tabela 5-1.

Tabela 5-1. Questões e Métricas, usando GQM. Fonte: o Autor.

Questão 1	Qual o desenvolvedor mais cria e remove *code smells*?
Métrica 1	Sumarização. Quantidade de *code smells* criada por desenvolvedor
Métrica 2	Sumarização. Quantidade de *code smells* removida por desenvolvedor
Questão 2	Qual o *code smell* que foi mais criado pelos desenvolvedores?
Métrica 1	Sumarização. Quantidade do tipo de *code smell* mais criado pelos desenvolvedores.
Questão 3	Quais os *code smells* que ocorrem em comum pelo desenvolvedor?
Métrica 1	Regra de associação. *Code smells* frequentes, Indicador de confiança e suporte.
Questão 4	Qual a tendência de *code smell* criado?
Métrica 1	Gráfico de linhas para apresentação de médias moveis.
Questão 5	Qual a previsão de criação de *code smell*?
Métrica 1	Gráfico de linhas para apresentação do executado x previsão.

As questões e métricas propostas na Tabela 5-1 serão apresentadas no SUBPROCESSO 3 – *Analytics* do processo TDMINING. A execução do experimento também serve de meio para avaliar o processo TDMINING, visto que se ao término da execução do processo TDMINING for possível responder as questões e métricas apresentadas, logo, é subentendido que o processo atendeu seu objetivo ao chegar no término da execução.

5.1.2 Fase de Planejamento

a. Seleção de contexto

O contexto geral é relacionado à engenharia de *software* especificamente no que concerne à qualidade. Já o contexto específico é relacionado a métricas de qualidade de código-fonte, no que tange à DT de *code smell*.

b. Formulação de hipótese

Com base no exposto anteriormente, a seguinte hipótese foi formulada:

H0: não há uma influência significativa da quantidade de *commits* na quantidade de *code smells.*

HA: há uma influência significativa da quantidade de *commits* na quantidade de *code smells.*

As hipóteses H0 e HA dizem respeito à quantidade de *commits* que podem ter influência ou não na quantidade de *code smells*. Como hipótese alternativa, é esperado que haja uma relação entre o número de *commits* e *code smells*, ou seja, quanto mais *commits* um desenvolvedor realizar, maior deveria ser a quantidade de *code smells.* As hipóteses serão verificadas na fase de execução no item de teste de hipótese de cada um dos experimentos.

c. Seleção das variáveis

Para a hipótese, as variáveis selecionadas são:

- Variável dependente (resultado/efeito): *code smells*
- Variável independente (*causa*): *commits*

A variável independente é a quantidade de *commits* que o desenvolvedor realiza, pois essa é a variável sobre a qual se tem o controle. A variável dependente é a quantidade de *code smells* gerado, pois depende da quantidade de *commits* que o desenvolvedor realizou. Sem *commits* não há *code smell*, mas com *commits* pode haver ou não *code smell*.

Para testar a hipótese dos experimentos é necessário utilizar o teste de regressão não paramétrico de *Spearman,* com o objetivo de verificar a influência que a quantidade de *code smell (*variável dependente) tem na quantidade de *commits* (variável independente). Pois o teste de *Spearman* é utilizado quando a variável dependente e variável independente são quantitativas, não possui causalidade, e a distribuição não é normal.

d. Seleção de indivíduos

Para poder executar e avaliar o processo TDMINING foi necessário executar dois experimentos. Sendo que o Experimento 1 - PILOTO, com dados de repositório público será tratado no item 5.2 e no item 5.3 será tratado o Experimento 2 com dados

obtidos de um projeto da indústria. Os detalhes de ambos os experimentos serão tratados em seus respectivos itens.

Projeto de experimento

O projeto do experimento consistiu em executar e avaliar o processo TDMINING. Avaliar o processo TDMINING significa verificar se os resultados obtidos com a execução do processo atendem às necessidades propostas.

Instrumentação

Para executar os experimentos, foi necessário utilizar *softwares* para apoiar a coleta de dados, além de formulários conforme definido na atividade PROCESSO PRINCIPAL – ATIVIDADE 1 - Solicitar Infraestrutura.

Ameaças e avaliação da validade

Nos itens 5.2 e 5.3 serão tratadas com mais detalhes as ameaças e avaliação da validade de cada um dos experimentos.

5.1.3 Fase de execução

Essa fase é operacional e possuiu atividades que foram executadas de forma manual e semiautomática, conforme descrito no processo TDMINING. Atividades semiautomáticas são *scripts* que fazem parte do empacotamento do experimento. Atividades manuais são as atividades referente à alteração dos documentos solicitados no processo TDMINING. Nos itens 5.2 e 5.3 é tratado com mais detalhes a fase de execução para cada um dos experimentos.

a. Preparação

A preparação está relacionada à identificação e obtenção dos recursos computacionais necessários e à identificação dos indivíduos para o experimento. Está representado no processo TDMINING no SUBPROCESSO 1 – INFRAESTRUTURA.

Conforme apresentado anteriormente a execução do experimento está diretamente associada à execução do processo TDMINING, principalmente o SUBPROCESSO 2 - ETL.

O SUBPROCESSO 2 - ETL deve ser executado para obter os dados para análise de *code smell*, o que ocorreu para o Experimento 1. Para o Experimento 2 não se aplicou todas as atividades do SUBPROCESSO 2 - ETL, mas sim a atividade PROCESSO PRINCIPAL – ATIVIDADE 6 – Importar dados, seguindo o fluxo DC1 conforme diagrama do processo TDMINING.

Validação dos dados

Para o Experimento 1 será tratado no item 5.2.4 e para o Experimento 2 no item 5.3.5.

5.1.4 Fase de análise e interpretação

a. Estatística descritiva

Como se sabe, para corrigir o código-fonte com *code smell*, é necessário tempo, e a execução da correção/refatoração demanda esse esforço. Conforme abordado no item 2.5, o método SQALE define um índice global para o custo de remediação relativo a todas as características do modelo de qualidade. Esta medida é o Índice de Qualidade SQALE ou SQI que, para Letouzey (2016), é o principal indicador do SQALE. A medida de tempo que o SonarQube atribui para SQI ou SQALE_INDEX é de 8 horas. Como demonstrado na Figura 2-6 e Figura 2-7, que tanto para o SonarQube quanto para SQALE, a exibição do esforço é sumarizada e quantificada em número de dias é a forma de representar quantitativamente o tempo necessário de esforço para correção da DT.

Cada *code smell* pode ter um valor de esforço para correção em minutos diferente. No caso das regras utilizadas pelo SonarQube, o esforço é uma propriedade da regra que, geralmente está relacionada à severidade da regra. Ou seja, quanto mais severa a regra de *code smell,* pode ser que o tempo parametrizado de esforço seja muito superior às demais regras. Um dos indicadores que o SQALE e o SonarQube não apresentam é saber o esforço de correção por code smell à medida que novas dívidas técnicas de código veem sendo adicionadas ou removidas.

Para auxiliar os experimentos na etapa de estatística descritiva sobre como comparar o mesmo experimento após as análises executadas houve a necessidade

de propor o índice de esforço (IE) e o índice de Dias de Correção de *code smells* (DCcs).

Na Equação 7, $effort$ é o esforço em minutos para corrigir o *code smell* e $codeSmell$ é a quantidade do *code smell*. Lembrando que cada regra de *code smell* pode ter um esforço para correção diferente, por exemplo, a regra de *code smell* 'S1192 - *String literals should not be duplicated*'[35] para linguagem de programação PHP tem 2 minutos de esforço para correção conforme informado pelo SonarQube.

Conforme exemplificado anteriormente e abordado no modelo de dados *StarSchema* para o processo TDMINING na tabela fato FT_TECHNICALDEBT, cada ocorrência de *code smell* deve ter um esforço em minutos. Logo, a soma (sum) dos esforços ($effort$) dividido pela quantidade (count) de elementos de *code smell* ($codeSmell$) tem como resultado o índice de esforço (IE), logo, IE = sum(effort) / count(codeSmell). Que representa o esforço em minutos para correção de cada code smell. Logo, o benefício em aplicar a Equação 7 está em permitir do ponto de vista de gestão de dívida técnica, saber se o esforço para correção por *code smell* está em uma tendência de alta, com isso possibilidade o gestor a tomar decisões para que tal indicador venha a tornar valores satisfatórios.

Equação 7 - Índice de esforço por *code smell*. Fonte: o Autor.

$$IE = \frac{\sum_{effort=0}^{\infty} effort}{|codeSmell|}$$

A Equação 7 será aplicada para análise dos dados dos Experimentos 1 e Experimento 2.

Outra possibilidade em relação aos dados foi elaborar uma forma de ter uma equação para determinar a quantidade de tempo necessário para resolver os todos os *code smells*, baseada na quantidade de profissionais que a empresa possui para resolver esses problemas e quanto tempo diariamente esses profissionais estão disponíveis para realizar as correções. Essa equação foi denominada de DCcs, conforme apresentado na Equação 8. O questionamento surgiu em relação ao SQI do

[35] Disponível em https://rules.sonarsource.com/php/RSPEC-1192

SQALE, pois o SQI considera apenas o esforço total para correção, sem considerar a quantidade de desenvolvedores que podem realizar as correções de *code smell*. A aplicação da equação desse tipo de abordagem de calcular a quantidade de dias previstos para correção de *code smells* tem o objetivo de auxiliar os casos em que é necessário dimensionar a quantidade de profissionais para resolver o problema dos *code smells* e o esforço diário para atuar nessa correção, principalmente nas situações em que existe um prazo determinado para o lançamento de uma nova versão do produto. Logo, o benefício em utilizar a Equação 8 está justamente em apoiar a gestão da dívida técnica no sentido de ser uma alternativa para distribuir a equipe em atividades de *refactoring*, correções de code smell e novas funcionalidades.

Equação 8 - DCcs (Dias para correção de *code smell*). Fonte: o Autor

$$DCcs = \frac{\sum_{effort=0}^{\infty} effort}{qtDev\ x\ MUD}$$

Com isso, tem-se como variáveis: a soma dos esforços ($\sum_{effort=0}^{\infty} effort$) dividida pela quantidade de desenvolvedores ($qtDev$), multiplicada pelos minutos úteis para correção (MUD). Observe que a variável ($qtDev$) é a quantidade de desenvolvedores que tem por objetivo de reduzir a quantidade de *code smell* já a variável (MUD) é define por quantos minutos por dia o desenvolvedor irá atuar na correção de *code smell.* Por exemplo se considerarmos a soma de esforços para correção de todos os *code smell* ($\sum_{effort=0}^{\infty} effort$) igual a 10.000 minutos, considerarmos que temos, 10 desenvolvedores ($qtDev$) e cada um desses desenvolvedores trabalhará 3 horas por dia para corrigir todos *code smell,* (MUD) igual a 180 minutos. Logo teremos 10.000/(10*180) = 5.55. Isso significa que se todos os desenvolvedores começarem a corrigir todos os *code smells*, trabalhando 3 horas por dia nessas correções, logo, em 5.55 dias todos os *code smell* estarão resolvidos. A aplicação desta equação será utilizada no Experimento 2.

Nos itens 5.2 e 5.3 são tratados com mais detalhes a estatística descritiva, a redução de conjuntos e o teste de hipótese para cada um dos experimentos.

5.1.5 Fase de apresentação e empacotamento

a. Apresentação

A apresentação dos resultados dos experimentos por meio da execução do processo TDMINING aponta que, na abordagem adotada, é possível entender o comportamento dos desenvolvedores em relação à criação de *code smell*. Esse entendimento ocorre por meio do resultado apresentado, que visa responder a todas as questões apontadas na Tabela 5-1.

A apresentação dos resultados dos experimentos com os dados pode ser observada nos resultados obtidos na fase de execução, apresentados itens 5.2.3 e 5.3.3.

b. Empacotamento

Tanto os *scripts* para geração dos dados, os dados, como os *scripts* utilizados estão disponíveis em https://github.com/manoelvsneto/tdmining .

5.2 EXPERIMENTO 1

5.2.1 FASE DE PLANEJAMENTO – SELEÇÃO DE INDIVÍDUOS

Para poder selecionar os indivíduos para o Experimento 1 foi necessário selecionar um projeto como piloto. Foi escolhido um projeto público no GITHUB com uma quantidade significativa de *commits* e desenvolvedores contribuidores. A seleção do projeto no GITHUB aconteceu de forma empírica, sendo o projeto escolhido o Wordpress.

Justifica-se a escolha deste projeto pois em um trabalho anterior o autor e outros colaboradores analisaram a quantidade de *commits* em relação à qualidade de código, em um trabalho não publicado, denominado “Um estudo empírico da DT em projetos *open source* de comércio eletrônico disponibilizados em repositório público”, onde os autores investigaram a relação *commits* x SQI, em nível de projetos. Os testes estatísticos realizados para os projetos selecionados apontaram uma influência da quantidade de *commits* x SQI, relembrando que no método SQALE o *code smell* é parte deste índice.

Vale observar que a abordagem proposta por este experimento não se restringe somente aos indivíduos selecionados, podendo ser aplicada em qualquer outro, tanto em repositórios públicos ou privados. Ressaltando apenas que para projetos diferentes, se a quantidade de desenvolvedores, de *commits*, de *code smell* e demais métricas de qualidade de *software* forem diferentes, logo, o resultado também será diferente desta dissertação. Para este trabalho, os dados provenientes do Wordpress foram tratados como Experimento 1.

5.2.2 FASE DE PLANEJAMENTO – AMEAÇAS A AVALIAÇÃO DA VALIDADE

Uma das ameaças à validade interna é em relação ao processo de coleta de dados de repositório, pois para o Experimento 1 foi confiado ao *software PyDriller* a recuperação das informações de *commits.* No entanto, esta ameaça foi minimizada por meio da conferência, de forma empírica, se todos os *commits* que existem no GitHub do projeto Wordpress foram persistidos corretamente no banco de dados, com o nome do autor do *commit* e o identificador que representa o *commit.*

Outra ameaça à validade interna, é garantir se todos os *commits* foram processados pela ferramenta SonarQube. Esse risco foi removido e foi criada no *script* de ETL, uma fila de processamento, para que o próximo item somente pudesse ser processado se o anterior tivesse tido sucesso.

Outra ameaça são as análises estáticas realizadas pelo SonarQube. O experimentador não tem controle sobre o processamento realizado pelo *software*. O que minimiza essa ameaça é o fato de que a ferramenta SonarQube é muito bem aceita na indústria.

Quanto à validade externa, esta relaciona-se com a generalização do conjunto de dados, visto que foram selecionados apenas dois experimentos. Mesmo que o projeto do Experimento 1 seja amplamente utilizado, ele não pode representar todo os demais projetos *open source* ou da indústria, e o mesmo ocorre com o Experimento 2. Por mais que ambos os experimentos possam ser replicados não é possível afirmar sobre a qualidade de código em projetos fechados (indústria) ou até mesmo outros projetos *open source*. Também está sujeito à validade externa a aplicabilidade dos modelos estatísticos e de mineração de dados para previsão de *code smell*, pois foram

analisados apenas dois experimentos e é sempre possível que outro conjunto de dados possa exibir fenômenos diferentes.

Quanto à validade de conclusão, o processo de transformação dos dados de *code smell* descritos nesta dissertação, se baseia principalmente na saída de dados do SonarQube para calcular métricas relacionadas a *code smells* que possam atuar como indicadores do atributo de qualidade para DT. Isto significa que a abordagem proposta pode ser facilmente adaptada para analisar *bugs* e vulnerabilidades, incluindo as aplicações que são codificadas em uma linguagem de programação diferente das utilizadas no Experimento 1 e no Experimento 2. Também pode-se concluir que a abordagem proposta não necessariamente precisa ser utilizada com o SonarQube, visto que para abordagem, o principal são os dados de métricas de qualidade de código no formato proposto por esta dissertação.

A validade da construção é devida a possíveis imprecisões na identificação de métricas que atuam como indicadores *code smell*, pois para a abordagem proposta foi utilizado apenas o SonarQube. A abordagem descrita nesta dissertação não depende da seleção de ferramentas, pois poderiam ser aplicadas outras conforme a necessidade do experimentador, desde que coletadas as métricas necessárias e que a persistência ocorra conforme o modelo de dados proposto.

Os resultados apresentados dependem das medidas obtidas do SonarQube. Logo, realizar mais experimentação faz-se necessário para avaliar a correção de resultados obtidos por meio de outras ferramentas que não sejam o SonarQube. Quanto à abordagem de aplicação de regras de associação, sumarização e séries temporais presentes neste estudo, ela não depende da seleção de ferramentas e/ou *script* criados para esse experimento, pois poderia ser aplicado por outras ferramentas de mineração de dados e análise estatística.

Quanto a ameaças de confiabilidade no que diz respeito à possibilidade de replicar este experimento, por tratar-se de passos em um processo definido, é obrigatório fornecer por meio de empacotamento, tanto o conjunto de dados, como os passos necessários que foram utilizados para recuperar, transformar e analisar os dados. O TDMINING serve de guia para realizar o experimento.

5.2.3 FASE DE EXECUÇÃO

Para o Experimento 1 foram processados 41.285 *commits*, em um tempo de processamento de 102 dias. Ao todo, 122 desenvolvedores realizaram esses *commits*. Foram considerados todos os *commits* desde a criação do repositório até a data de início do experimento. O primeiro *commit* tem data de 01/04/2003 e o último foi analisado em 21/03/2020, ou seja, quase 17 anos de *commits*.

Com base nas questões e métricas definidas para o experimento, este item tem o objetivo de apresentar os resultados da execução dos *scripts* que ocorreu por meio da execução do processo TDMINING, para o Experimento 1. A discussão dos resultados será apresentada no item sobre estatística descritiva, conforme processo de ESE.

Os resultados da execução dos *scripts* apontados do SUBPROCESSO 3 - *Analytics,* atividade SUBPROCESSO 3 – ATIVIDADE 4 – EXECUTAR ARQUIVO PYTHON do processo TDMINING para o Experimento 1, podem ser observados no subitem a, b, c, d, e, f, g e h.

a. Teste Normalidade

O resultado do *script* teste-normalidade.ipynb pode ser observado na Figura 5-1 e no

Quadro 5-1. É possível observar na Figura 5-1 que dos 122 desenvolvedores tiveram uma média de *code smell* de 757. Sendo que essa média ocorre por existir desenvolvedores com quantidade de *code smell* criado superior a 5.000.

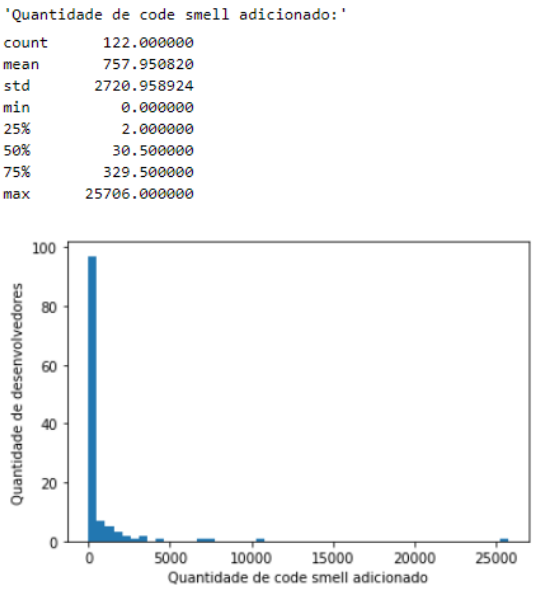


Figura 5-1. Resultado da execução do script teste-normalidade.ipynb. Fonte: o Autor.

Quadro 5-1. Resultado da execução do script teste-normalidade.ipynb. Fonte: o Autor

Shapiro-Wilk Test Statistics=0.238, p=0.000
Shapiro-Wilk Test Sample does not look Gaussian (reject H0)
Kolmogorv-Smirnov Test Statistics=0.748, p=0.000
Kolmogorv-Smirnov Test Sample does not look Gaussian (reject H0)

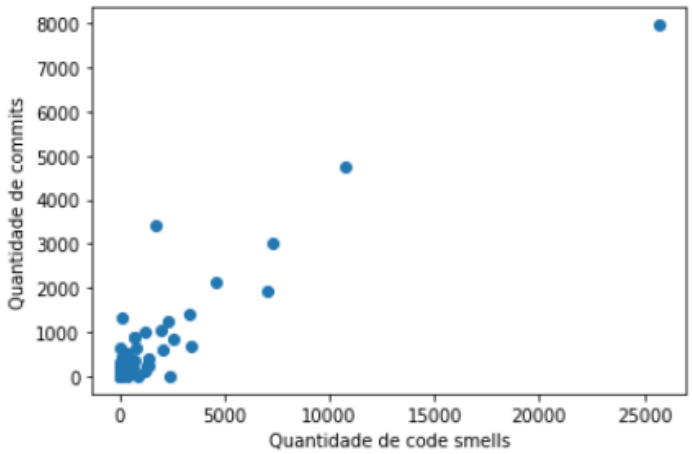
Como pode ser observado no resultado de Kolmogorov-Smirnov, tem-se p-valor(p) menor do que 0,05, podendo considerar que a distribuição da variável quantidade de *code smell* é diferente da distribuição normal. Com isso, os testes a serem realizados deverão ser não-paramétricos.

b. Teste de hipótese

Para testar a hipótese H0 foi utilizado o teste de correlação de Spearman para verificar o efeito da quantidade de *code smell* (variável dependente) x quantidade de *commits* (variável dependente). Como o resultado do teste é menor do que o nível de significância (0,05), na Figura 5-2, é possível negar a hipótese nula H0.

Portanto, é possível afirmar que há uma correlação entre a dívida técnica de *code smell* e a quantidade de *commits*. Isso significa que quanto mais commits ocorrer maior e a chance de criação de *code smell*. A Figura 5-2 mostra o gráfico de dispersão

em relação a quantidade de *code smell* e a quantidade de *commits* dos projetos selecionados. Com a figura é possível notar a relação positiva entre as duas variáveis.




```
'Spearman: Quantidade de commits x Quantidade de code smells = 0.7965628790528839'
'Spearman: p=5.377181753913002e-28'
```

Figura 5-2. Gráfico de dispersão do resultado da execução do script teste-hipotese.ipynb. Fonte: o Autor.

c. Questão 1 - Métrica 1 - Qual o desenvolvedor mais cria code smells?

A Tabela 5-2 demonstra que o DESENVOLVEDOR 110 foi o desenvolvedor que mais criou *code smell,* com 27,80%. No entanto, se observar o Índice de esforço de 6,71 minutos de esforço por code smell é menor que os 8,53 no âmbito geral do projeto. Isso significa que na média, possivelmente que os code smells criados pelo DESENVOLVEDOR 100 possuem uma menor severidade, pois quanto menor a severidade menor tende ser o esforço para correção.

Tabela 5-2. Resultado da execução do script question-1-metric-1.ipynb.Fonte: o Autor.

DESENVOLVEDOR	(A) QUANTIDADE	(B) ESFORÇO (MIN)	ÍNDICE ESFORÇO (B/A)	PERCENTUAL
110	25.706	172.513	6,71	27,80
12	10.752	96.653	8,99	11,63
13	7.276	52.342	7,19	7,87
83	7.066	42.209	5,97	7,64
112	4.582	43.584	9,51	4,96
42	3.452	47.933	13,89	3,73
81	3.369	41.852	12,42	3,64
17	2.583	38.672	14,97	2,79

36	2.388	12.140	5,08	2,58
DEMAIS 89 DEV	25.296	241.526	9,55	27,36
TOTAL	**92.470**	**789.424**	**8,53**	**100,00**

*DEMAIS DEV significa os demais 89 desenvolvedores

d. Questão 1 - Métrica 2 - Qual o desenvolvedor mais remove code smells?

A Tabela 5-3 demonstra que o DESENVOLVEDOR 110 foi o desenvolvedor que mais removeu *code smell,* com 22,63%. Dos 98 desenvolvedores que adicionaram *code smell* 92 desenvolvedores removeram 1 ou mais *code smell*. Ou seja aproximadamente 7% dos desenvolvedores não tem realizaram remoção de *code smell.*

Tabela 5-3. Resultado da execução do script question-1-metric-2.ipynb. Fonte: o Autor

DESENVOLVEDOR	QUANTIDADE	PERCENTUAL
110	14.040	22,63
42	7.874	12,69
12	6.863	11,06
13	5.307	8,55
112	4.945	7,97
83	4.453	7,18
100	1.960	3,16
81	1.739	2,80
113	1.448	2,33
DEMAIS DEV*	1.423	21,63

*DEMAIS DEV significa os demais 83 desenvolvedores

e. Questão 2 - Métrica 1 - Qual o code smell que foi mais criado pelos desenvolvedores?

A Tabela 5-4 demonstra que a regra de *code smell* ‘S1192 - *String literals should not be duplicated’* e *‘*S121 - *Control structures should use curly braces’* representam 60% das violações de regras de *code smell* executadas pelos desenvolvedores. Ou seja, de 111 tipos de *code smell,* 2 tipos de *code smell* são identificados como os mais comuns no projeto.

Tabela 5-4. Resultado da execução do script question-2-metric-1.ipynb (Todos os Desenvolvedores) – Script A. Fonte: o Autor.

REGRA DE CODE SMELL	(A) QUANTIDADE	(B) ESFORÇO (MIN)	INDICE ESFORÇO (B/A)	PERCENTUAL
S1192 - *String literals should not be duplicated*	29.211	396.140	13,56	31,59
S121 - *Control structures should use curly braces*	26.506	53.012	2,00	28,66
S1117 - *Variables should not be shadowed*	6.099	30.495	5,00	6,60
S125 - *Sections of code should not be commented out*	2.569	12.845	5,00	2,78
S1481 - *Unused local variables should be removed*	2.563	12.815	5,00	2,77
S3776 - *Cognitive Complexity of functions should not be too high*	2.151	50.194	23,33	2,33
S1142 - *Functions should not contain too many return statements*	1.637	32.740	20,00	1,77
S1827 - *Attributes deprecated in HTML5 should not be used*	1.558	7.790	5,00	1,68
S101 - *Class names should comply with a naming convention*	1.463	7.315	5,00	1,58
DEMAIS REGRAS*	18.713	186.078	9,94	20,24
TOTAL	**92.470**	**789.424**	**8,53**	**100,00**

*DEMIAS REGRAS significa as demais 102 regras.

A Tabela 5-5 demonstra que a regra de *code smell* ‘S1192 - *String literals should not be duplicated’* mais *‘*S121 - *Control structures should use curly braces’* representam 67,21% das violações de regras de code smell executadas pelos DESENVOLVEDOR 110. Ou seja, dos 84 tipos de *code smell ,* 2 tipos de *code smell* são identificados como os mais frequentemente criado pelo DESENVOLVEDOR 110. Visto isso é possível sugerir que tal desenvolvedor aprenda como resolver determinado *code smell.*

Tabela 5-5. Resultado da execução do script question-2-metric-1.ipynb (Desenvolvedor 110) - Script B. Fonte: o Autor.

REGRA DE CODE SMELL	(A) QUANTIDADE	(B) ESFORÇO (MIN)	INDICE ESFORÇO (B/A)	PERCENTUAL
S121 - *Control structures should use curly braces*	11.867	23.734	2,00	46,16
S1192 - *String literals should not be duplicated*	5.410	65.866	12,17	21,05
S125 - *Sections of code should not be commented out*	972	4.860	5,00	3,78
S3776 - *Cognitive Complexity of functions should not be too high*	659	14.355	21,78	2,56
S1142 - *Functions should not contain too many return statements*	494	9.880	20,00	1,92

S1481 - *Unused local variables should be removed*	480	2.400	5,00	1,87
S2814 - *Variables and functions should not be redeclared*	364	7.280	20,00	1,42
S1172 - *Unused function parameters should be removed*	353	1.765	5,00	1,37
S2681 - *Multiline blocks should be enclosed in curly braces*	340	1.700	5,00	1,32
DEMAIS REGRAS*	4.767	40.673	8,53	18,54
TOTAL	**25.706**	**172.513**	**6,71**	**100,00**

*DEMIAS REGRAS significa as demais 75 regras.

A Tabela 5-6 demonstra que 65,10% das violações de regras de code smell executadas pelos desenvolvedores são da severidade CRITICAL. Ou seja, se necessário definir um critério para qual code smell refatorar primeiramente, a escolha por severidade pode ser uma estratégia válida.

Tabela 5-6. Resultado da execução do script question-2-metric-1.ipynb (Todos os Desenvolvedores) – Script C. Fonte: o Autor.

SEVERIDADE	(A) QUANTIDADE	(B) ESFORÇO (MIN)	INDICE ESFORÇO (B/A)	PERCENTUAL
1-BLOCKER	1.154	5.732	4,97	1,25
2-CRITICAL	60.196	514.082	8,54	65,10
3-MAJOR	22.545	236.560	10,49	24,38
4-MINOR	7.827	33.050	4,22	8,46
5-INFO	748	0	0,00	0,81
TOTAL	**92.470**	**789.424**	**8,53**	**100,00**

A Tabela 5-7Tabela 5-6 demonstra que 71,96% das violações de regras de code smell executadas pelo DESENVOLVEDOR 110 são da severidade CRITICAL. Ou seja, se necessário definir um critério para qual code smell refatorar primeiramente, a escolha por severidade pode ser uma estratégia válida.

Tabela 5-7. Resultado da execução do script question-2-metric-1.ipynb (Desenvolvedor 110) – Script D. Fonte: o Autor.

SEVERIDADE	(A) QUANTIDADE	(B) ESFORÇO (MIN)	INDICE ESFORÇO (B/A)	PERCENTUAL
BLOCKER	380	1.794	4,72	1,48
CRITICAL	18.499	108.345	5,86	71,96
MAJOR	4.596	54.878	11,94	17,88
MINOR	1.988	7.496	3,77	7,73
INFO	243	0	0	0,95

TOTAL	**25.706**	**172.513**	**6,71**	**100,00**

f. Questão 3 - Métrica 1 - Quais os *code smells* que ocorrem em comum pelo desenvolvedor?

A Tabela 5-8 apresenta a frequência que um conjunto de itens, *code smell*, ocorre na base de dados. Por exemplo o *code smell (S1192 - String literals should not be duplicated),* ocorre em 56% dos *commits*. Já a Tabela 5-9, demonstra que a combinação da condição SE *code smell (S1192 - String literals should not be duplicated) ENTÃO code smell (S121 - Control structures should use curly braces)* ocorre em 18% dos *commits,* a medida de confiança indica a frequência que a regra se mostrou válida em 33% dos casos. Observando que tanto para a análise dos itens frequentes quanto para regra de associação foi considerado um limite mínimo de 16% para a medida de suporte.

Tabela 5-8. Itens Frequentes – Resultado da execução do script question-3-metric-1.ipynb (Desenvolvedor 110). Fonte: o Autor.

support	itemsets
0.560645	(S1192 - String literals should not be duplicated)
0.454623	(S121 - Control structures should use curly braces)
0.185666	(S121 - Control structures should use curly braces, S1192 - String literals should not be duplicated)

Tabela 5-9. Regra de Associação – Resultado da execução do script question-3-metric-1.ipynb (Desenvolvedor 110). Fonte: o Autor.

antecedents	consequents	antecedent support	consequent support	support	confidence
(S1192 - String literals should not be duplicated)	(S121 - Control structures should use curly braces)	0.560	0.454	0.185	0.331
(S121 - Control structures should use curly braces)	(S1192 - String literals should not be duplicated)	0.454	0.560	0.185	0.408

g. Questão 4 - Métrica 1 - Qual a tendência de code smell criado?

A Figura 5-3 Tabela 5-8 demonstra a média móvel da criação de *code smell*, o eixo X apresenta o período em mês/ano o eixo Y a quantidade de code smell. Para analisar a média móvel foi considerado uma janela de 12 meses para as 6 últimas observações, à medida que a média para cada mês é calculada. A linha em azul demonstra a evolução da criação de *code smell* por todos os desenvolvedores, a

linha laranja demonstra a evolução da criação de *code smell* pelo DESENVOLVEDOR 110. Conforme já citado anteriormente, o DESENVOLVEDOR 110 é o desenvolvedor que mais cria *code smell,* no entanto, é possível observar que após a saída do DESENVOLVEDOR 110 no ano de 2014 a média móvel em azul não acompanhou a média móvel em laranja, isso significa que com a saída de tal desenvolvedor o comportamento médio da criação de *code smell* se manteve. Esse comportamento vai ao encontro do gráfico apresentado na Figura 5-4, onde o índice de esforço no ano de 2014 inicia uma alta na quantidade de esforço por *code smell*. Para essa situação isso significa que a saída de um desenvolvedor do projeto, não garante que a quantidade de *code smell* poderá reduzir sendo o DESENVOLVEDOR 100 o maior contribuidor para criação de *code smell,* após a saída de tal desenvolvedor o índice de esforço para correção de *code smell* apresenta uma tendência de alta (Figura 5-4).

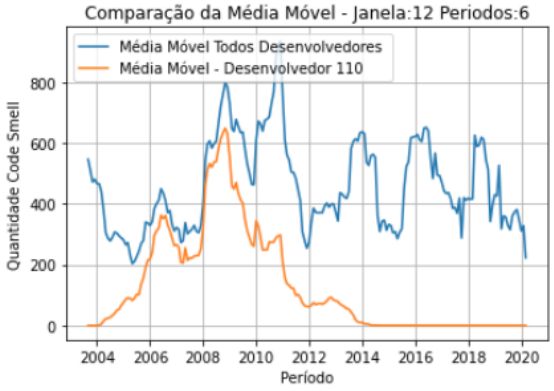


Figura 5-3. Resultado da execução do script question-4-metric-1.ipynb (Quantidade de Code Smell). Fonte: o Autor.

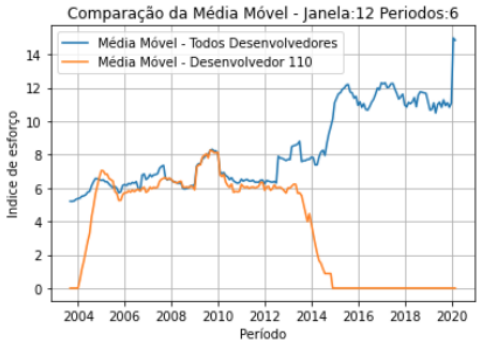


Figura 5-4. Resultado da execução do script question-4-metric-1.ipynb (Índice de Esforço). Fonte: o Autor.

h. Questão 5 - Métrica 1 - Qual a previsão de criação de code smell?

O Quadro 5-2 Tabela 5-8demonstra que após a execução dos testes de Dickey-Fuller e KPSS que a serie temporal possui um comportamento estacionário. Esse comportamento é observado na Figura 5-5, após treinar e testar o modelo é possível verificar na faixa cinza e na linha verde do gráfico o comportamento estacionário.

No gráfico exibido na Figura 5-5 contém 3.779 dias com *code smell* no período de 01/04/2003 até 14/03/2020, 6.192 dias. Ou seja, 61% dos dias do projeto possui pelo menos um *code smell*. Na Tabela 5-10 e apresentado o índice de erro MAPE, resultado aponta que em média a previsão apresentada no gráfico está incorreta em 4.11%. Outro fato que deve ser observado na Figura 5-5 é a quantidade de picos no gráfico. Esse comportamento pode ocorrer em situações de pressão para entregar uma nova versão do produto, conforme já abordado na literatura de causas de dívida técnica. Também pode ter relação com o processo de desenvolvimento de software onde o desenvolvedor também é o aprovador do código. Ou em situações em que o aprovador não tem conhecimentos suficientes para analisar um código fonte. Ou até mesmo a ausência de processo de inspeção automática e contínua de código fonte. No entanto, para análise de dados do ponto de vista do desenvolvedor devem ser mantidos, pois se algum dos dados de *code smell* for desconsiderado pode influenciar nas demais questões determinadas na Tabela 5-1, com isso não apresentando um conhecimento que reflete a realidade do desenvolvedor. Outro fato a se observar que não houve limpeza de dados destes picos, pois somente quem conhece o processo de desenvolvimento do produto pode determinar em quais condições determinadas code smells deve ser ignorado.

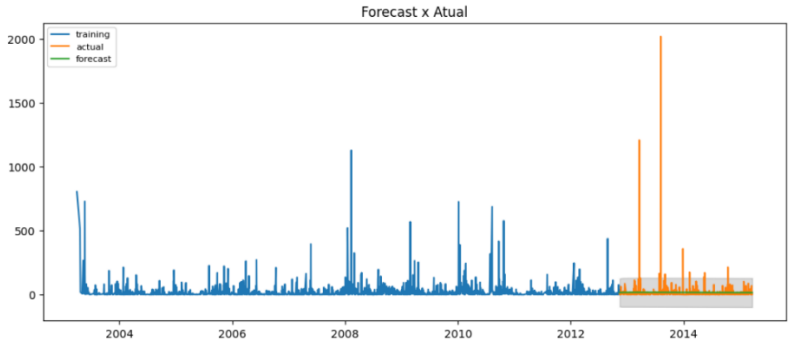


Figura 5-5. Resultado gráfico da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

Quadro 5-2. Resultado texto da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

```
Results of Dickey-Fuller Test:
Test Statistic                -42.940346
p-value                       0.000000
Lags Used                     1.000000
Number of Observations Used   3777.000000
Critical Value (1%)           -3.432083
Critical Value (5%)           -2.862306
Critical Value (10%)          -2.567178
dtype: float64
Result adf : The series is  stationary
Reject Ho - Time Series is Stationary
Results of KPSS Test:
KPSS Statistic: 0.09860594259319189
p-value: 0.1
num lags: 13
Critial Values:
   10% : 0.347
   5% : 0.463
   2.5% : 0.574
   1% : 0.739
Result kpss_test: The series is stationary
Observations: 944 Treino: 755 Teste: 189
Melhor AIC Treino: 7996.624552320086 (7, 2, 1)
Observations: 1888 Treino: 1510 Teste: 378
Melhor AIC Treino: 16415.803923992433 (7, 1, 7)
Observations: 2832 Treino: 2265 Teste: 567
Melhor AIC Treino : 24668.11718191011 (1, 1, 2)
[(7, 2, 1), (7, 1, 7), (1, 1, 2)]
AIC Selecionado: (1, 1, 2)
```

Tabela 5-10. Erros – Resultado da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

ERRO	VALOR
MAPE	4.11

5.2.4 FASE DE EXECUÇÃO – VALIDAÇÃO DOS DADOS

No Experimento 1, os dados gerados pela ferramenta SonarQube no SUBPROCESSO 2 - ETL são mantidos no banco de dados do próprio SonarQube, logo, os *scripts* utilizados no SUBPROCESSO 2-ETL não realizaram nenhuma manipulação/intervenção direta do banco de dados do SonarQube. Assim, a validação ocorreu por meio dos *scripts* informados na atividade SUBPROCESSO 2 – ATIVIDADE 5 – EXECUTAR SCRIPT DE BANCO DE DADOS que persiste os dados nas tabelas de dimensão de fato, referente ao modelo de dados proposto.

5.2.5 FASE DE ANÁLISE E INTERPRETAÇÃO – ESTATÍSTICA DESCRITIVA

Após a finalização da fase de execução do experimento foi possível iniciar a análise dos dados gerados. Os dados visam responder às questões e métricas definidas na Tabela 5-1.

Sobre o Experimento 1 (Piloto) foi possível observar que, apesar de o desenvolvedor 110 ser o maior criador de *code smells*, após a sua saída do projeto, a média móvel de criação de *code smell* apontou crescimento. A saída do desenvolvedor pode ser observada a partir do momento que não há mais *commits* com ou sem *code smell* em nome do desenvolvedor, como demonstrado na Figura 5-3 e Figura 5-4. Entretanto, é perceptível na Figura 5-5 que a previsão de criação de *code smell* se mantém em uma série estacionária, após a saída deste profissional.

No que concerne ao índice de esforço (IE) para correção de *code smell, ver* Tabela 5-2*,* os resultados demonstraram que para o desenvolvedor 110 foi de 6,71, enquanto o IE Geral é de 8,53. Ou seja, o desenvolvedor 110 apesar de ser o maior criador de *code smell*, os *code smells* criado por ele requer menos esforço para correção visto os demais desenvolvedores. Pois o para o desenvolvedor 110 o índice de esforço é 6,71 minutos de esforço por *code smell*, enquanto para os todos desenvolvedores é de 8,73 minutos de esforço por *code smell*.

Quanto à análise de regra de associação para o Experimento 1, é possível observar que para o desenvolvedor 110 existe uma possibilidade de quando ele criar o *code smell* S1192, também criar o *code smell* S121, com uma confiança de 33%.

5.2.6 FASE DE ANÁLISE E INTERPRETAÇÃO – REDUÇÃO DE CONJUNTOS

Não houve necessidade de realizar redução de conjuntos para os dados dos Experimento 1.

5.2.7 FASE DE ANÁLISE E INTERPRETAÇÃO – TESTE DE HIPOTESE

O teste de hipótese de ambos os experimentos faz parte do processo TDMINING, e, em ambos os casos, foi executado na atividade SUBPROCESSO 3 – ATIVIDADE 3 – EXECUTAR SCRIPTS JUPYTER. O objetivo foi verificar se a quantidade de *commits* tem relação com a quantidade de *code smells*.

Para testar a hipótese dos experimentos foi utilizado o teste de regressão não paramétrico de *Spearman,* a fim de verificar a influência que a quantidade de *code smell (*variável dependente) tem na quantidade de *commits* (variável independente).

Ao executar o *script* teste-hipotese.ipynb, o resultado da execução pode ser avaliado nos gráficos de dispersão dos experimentos, o qual exibe quantidade de *code smells* e a quantidade de *commits* dos desenvolvedores. No gráfico, é possível notar a relação positiva entre as duas variáveis. Como, por exemplo, o resultado do teste de hipótese do Experimento 1, onde é possível observar o valor do coeficiente de correlação (0,796) e que a variação da quantidade de *commits* influencia em 79,6% a quantidade de *code smell*.

Como o resultado da significância (p) da correlação é menor do que o nível de significância (0,05), é possível negar a hipótese nula. Portanto, é possível afirmar que no Experimento 1 há uma influência significativa da quantidade de *commits* na quantidade de *code smell*.

5.3 EXPERIMENTO 2

5.3.1 FASE DE PLANEJAMENTO – SELEÇÃO DE INDIVÍDUOS

Para este experimento os dados provenientes de uma empresa privada, foi denominado Experimento 2.

5.3.2 FASE DE PLANEJAMENTO – AMEAÇAS E VALIDAÇÃO DA VALIDADE

Uma ameaça à validade é em relação aos dados do Experimento 2. Apesar de o pesquisador receber o banco de dados do *GIT* e o banco de dados do SonarQube de uma empresa parceira, os dados do SonarQube não foram gerados de forma a identificar a cada *commit* uma relação com a análise estática gerada.

Outras ameaças a validade foram tratadas anteriormente por ser comum ao Experimento 1.

5.3.3 FASE DE EXECUÇÃO – ANÁLISE A

Conforme apresentado anteriormente a execução do experimento está diretamente associada à execução do processo TDMINING, principalmente o

SUBPROCESSO 2 - ETL. Para o Experimento 2 aplicou o SUBPROCESSO 2 – ETL o fluxo DC1 igual a “Sim”.

Para o Experimento 2 constam 7.523 *commits*, sendo o primeiro *commit* de 12/08/2019 e o último de 31/03/2021.

Os resultados da execução dos *scripts* apontados do SUBPROCESSO 3 - *Analytics,* atividade SUBPROCESSO 3 – ATIVIDADE 4 – Documentar Resultados do processo TDMINING para o Experimento 2, podem ser observados respectivamente nos subitens deste item 5.3.3.

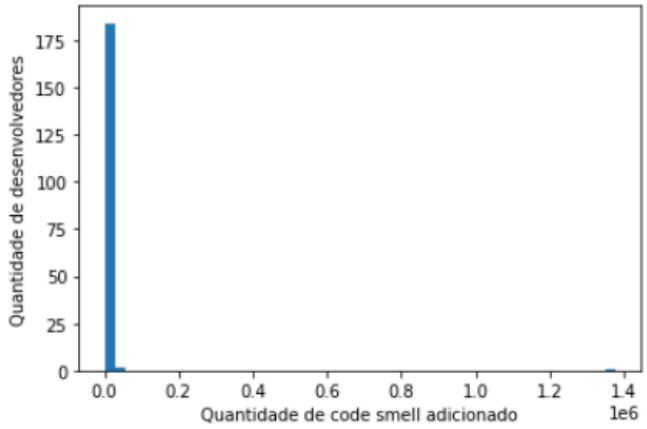
Com base nas questões e métricas definidas para o experimento, os subitens a seguir apresentam os resultados da execução dos *scripts* que ocorreu por meio da execução do processo TDMINING, para o Experimento 2 – Empresa.

Para este experimento foram realizados dois ciclos de experimentação. O primeiro ciclo (Análise A) seguiu o fluxo do processo TDMINING sem considerar as restrições apontadas pelo especialista da empresa, já o segundo ciclo (Análise B) leva em consideração as restrições apontadas pelo especialista da empresa.

a. Teste de Normalidade

O resultado do *script* teste-normalidade.ipynb pode ser observado na Figura 5-6. Os 187 desenvolvedores tiveram uma média de *code smell* de 9.925. Sendo que essa média ocorre por existir um desenvolvedor com quantidade de *code smell* de 1.376.184. No entanto, para essa condição de anomalia dos dados é necessário a interpretação do ESPECIALISTA.

```
count    1.870000e+02
mean     9.225433e+03
std      1.005805e+05
min      0.000000e+00
25%      5.950000e+01
50%      4.310000e+02
75%      1.704000e+03
max      1.376184e+06
Name: TOTAL_ADD, dtype: float64
```




```
'Shapiro-Wilk Test Statistics=0.035, p=0.000'
'Shapiro-Wilk Test Sample does not look Gaussian (reject H0)'
```

Figura 5-6. Análise A – Gráfico do teste de normalidade. Fonte: o Autor.

b. Teste de Hipótese

Para testar a hipótese H0 foi utilizado o teste de correlação de Spearman para verificar o efeito da quantidade de *code smell* (variável dependente) x quantidade de *commits* (variável dependente). Como o resultado do teste é menor do que o nível de significância (0,05), na Figura 5-7Figura 5-2, é possível negar a hipótese nula H0.

Portanto, é possível afirmar que há uma correlação entre a dívida técnica de *code smell* e a quantidade de *commits de 75%*. Isso significa que quanto mais *commits* ocorrer maior e a chance de criação de *code smell*. A Figura 5-7 mostra o gráfico de dispersão em relação a quantidade de *code smell* e a quantidade de *commits* dos projetos selecionados.

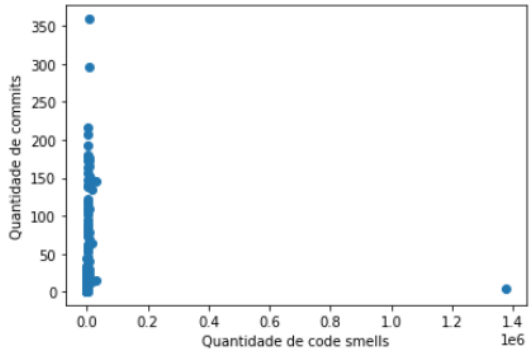


Figura 5-7. Análise A – Gráfico de dispersão teste de hipótese. Fonte: o Autor.

c. Questão 1 - Métrica 1 - Qual o desenvolvedor mais cria code smells?

A Tabela 5-11 demonstra que o DESENVOLVEDOR 63 foi o desenvolvedor que mais criou *code smell,* com 79,77%. No entanto, se observar o Índice de esforço de 6,26 minutos de esforço por *code smell* é menor que os 7,01 no âmbito geral do projeto. Isso significa que na média, possivelmente que os *code smells* criados pelo DESENVOLVEDOR 63 possuem uma menor severidade, pois quanto menor a severidade menor tende ser o esforço para correção.

Tabela 5-11. Análise A – Comparação do resultado da execução do script question-1-metric-1.ipynb. Fonte: o Autor.

DESENVOLVEDOR	(A) QTDE	(B) ESFORÇO (MIN)	INDICE ESFORÇO (B/A)	PERCENTUAL
63	1.376.184	8.612.572	6,26	79,77
130	31.211	71.092	2,28	1,81
94	28.254	365.802	12,95	1,64
131	16.392	143.448	8,75	0,95
172	15.568	226.618	14,56	0,90
153	15.203	168.578	11,09	0,88
157	13.494	202.501	15,01	0,78
166	8.977	77.550	8,64	0,52
122	8.901	47.490	5,34	0,52
DEMAIS DEV*	210.972	2.176.999	10,32	12,23
TOTAL	**1.725.156**	**12.092.650**	**7,01**	**100,00**

*Demais DEV significa os demais 176 desenvolvedores.

d. Questão 1 - Métrica 2 - Qual o desenvolvedor mais remove code smells?

A Tabela 5-12 demonstra que o DESENVOLVEDOR 63 foi o desenvolvedor que mais removeu *code smell,* com 64,18%. Coincidentemente o desenvolvedor que mais criou *code smell*.

Tabela 5-12. Análise A – Comparação do resultado da execução do script question-1-metric-2.ipynb. Fonte: o Autor.

DESENVOLVEDOR	QUANTIDADE	PERCENTUAL
63	13.462	64,18
79	1.640	7,82
134	684	3,26
172	430	2,05
131	361	1,72
94	354	1,69
65	348	1,66
40	228	1,09
DEMAIS DEV*	3.468	16,53

*Demais DEV significa os demais 107 desenvolvedores.

e. Questão 2 - Métrica 1 - Qual o *code smell* que foi mais criado pelos desenvolvedores?

A Tabela 5-13 demonstra que a regra de *code smell* 'S1808 - *Source code should comply with formatting standards* e *'*S2043 - *Superglobals should not be accessed directly'* representam 49% das violações de regras de *code smell* executadas pelos desenvolvedores, dentre os outros 155 tipos de *code smell.*

Tabela 5-13. Análise A – Resultado da execução do script question-2-metric-1.ipynb (Todos os Desenvolvedores) – Script A. Fonte: o Autor.

REGRA	(A) QUANTIDADE	(B) ESFORÇO (MIN)	ÍNDICE ESFORÇO (B/A)	PERCENTUAL
S1808 - *Source code should comply with formatting standards*	497.549	497.549	1,00	28,84
S2043 - *Superglobals should not be accessed directly*	351.608	5.274.120	15,00	20,38
S139 - *Comments should not be located at the end of lines of code*	75.914	75.914	1,00	4,40
S103 - *Lines should not be too long*	72.764	72.764	1,00	4,22

S1131 - *Lines should not end with trailing whitespaces*	66.004	66.004	1,00	3,83
S121 - *Control structures should use curly braces*	52.859	105.718	2,00	3,06
S1440 - "===" *and* "!==" *should be used instead of* "==" *and* "!="	49.538	247.690	5,00	2,87
S1827 - *Attributes deprecated in HTML5 should not be used*	38.876	194.380	5,00	2,25
InsufficientCommentDensity - Source files should have a sufficient density of comment lines	34.899	278.174	7,97	2,02
DEMAIS REGRAS*	485.145	5.280.337	10,88	28,12
TOTAL	**1.725.156**	**12.092.650**	**7,01**	**100,00**

*Demais DEV significa as demais 155 regras.

A Tabela 5-14 demonstra que a regra de *code smell* 'S1808 - *Source code should comply with formatting standards'* e *'*S2043 - *Superglobals should not be accessed directly'* representam 51% das violações de regras de code smell executadas pelos DESENVOLVEDOR 63. Ou seja, dos 151 tipos de *code smell* , 2 tipos de *code smell* são identificados como os mais frequentemente criado pelo DESENVOLVEDOR 110. Visto isso é possível sugerir que tal desenvolvedor aprenda como resolver determinado *code smell.*

Tabela 5-14. Análise A – Resultado da execução do script question-2-metric-1.ipynb (Desenvolvedor 63) – Script B. Fonte: o Autor.

REGRA	(A) QUANTIDADE	(B) ESFORÇO (MIN)	ÍNDICE ESFORÇO (B/A)	PERCENTUAL
S1808 - *Source code should comply with formatting standards*	417.926	417.926	1,00	30,37
S2043 - *Superglobals should not be accessed directly*	286.328	4.294.920	15,00	20,81
S139 - *Comments should not be located at the end of lines of code*	58.322	58.322	1,00	4,24
S103 - *Lines should not be too long*	52.520	52.520	1,00	3,82
S121 - *Control structures should use curly braces*	49.267	98.534	2,00	3,58
S1131 - *Lines should not end with trailing whitespaces*	49.016	49.016	1,00	3,56

S1440 - "===" *and* "!==" *should be used instead of* "==" *and* "!="	44.891	224.455	5,00	3,26
S1827 - *Attributes deprecated in HTML5 should not be used*	36.536	182.680	5,00	2,65
S1172 - *Unused function parameters should be removed*	31.418	157.090	5,00	2,28
DEMAIS REGRAS	349.960	3.077.109	8,79	25,43
TOTAL	**1.376.184**	**8.612.572**	**6,26**	**100,00**

A Tabela 5-15 demonstra que 76,90% das violações de regras de code smell executadas pelos desenvolvedores são da severidade MAJOR. Ou seja, se necessário definir um critério para qual *code smell* refatorar primeiramente, a escolha por severidade pode ser uma estratégia válida.

Tabela 5-15. Resultado da execução do script question-2-metric-1.ipynb (Todos os Desenvolvedores) – Script C. Fonte: o Autor.

SEVERIDADE	(A) QUANTIDADE	(B) ESFORÇO (MIN)	ÍINDICE ESFORÇO (B/A)	PERCENTUAL
1-BLOCKER	41.409	1.667.841	40,28	2,40
2-CRITICAL	142.193	473.783	3,33	8,24
3-MAJOR	1.326.629	7.922.295	5,97	76,90
4-MINOR	209.625	2016.546	9,62	12,15
5-INFO	5.300	12.185	2,30	0,31
TOTAL	**1.725.156**	**12.092.650**	**7,01**	**100,00**

A Tabela 5-16Tabela 5-6 demonstra que 80,68% das violações de regras de code smell executadas pelo DESENVOLVEDOR 63 são da severidade MAJOR. Ou seja, se necessário definir um critério para qual *code smell* refatorar primeiramente, a escolha por severidade pode ser uma estratégia válida.

Tabela 5-16. Análise A – Resultado da execução do script question-2-metric-1.ipynb (Desenvolvedor 63) – Script D. Fonte: o Autor.

SEVERIDADE	QUANTIDADE	ESFORCO	INDICE_ESFORCO	PERCENTUAL
1-BLOCKER	16.268	571.134	35,11	1,18
2-CRITICAL	115.986	382.530	3,30	8,43
3-MAJOR	1.110.364	6.550.154	5,90	80,68
4-MINOR	128.796	1.097.094	8,52	9,36
5-INFO	4.770	11.660	2,44	0,35
TOTAL	**1.376.184**	**8.612.572**	**6,26**	**100,00**

f. Questão 3 - Métrica 1 - Quais os code smells que ocorrem em comum pelo desenvolvedor?

A Tabela 5-17 apresenta a frequência que um conjunto de itens, *code smell*, ocorre na base de dados. Por exemplo o *code smell (InsufficientCommentDensity - Source files should have a sufficient density of comment lines)* ocorre em 40% dos *commits*. Já a Tabela 5-18, demonstra que a combinação da condição SE *code smell* (InsufficientCommentDensity - Source files should have a sufficient density of comment lines*) ENTÃO code smell (S2043 - Superglobals should not be accessed directly)* ocorre em 17% dos *commits,* a medida de confiança indica a frequência que a regra se mostrou válida em 41% dos casos. Observando que tanto para a análise dos itens frequentes quanto para regra de associação foi considerado um limite mínimo de 17% para a medida de suporte.

Tabela 5-17. Análise A – Itens Frequentes – Resultado da execução do script question-3-metric-1.ipynb. Fonte: o Autor.

	support	itemsets
0	0.409882	(InsufficientCommentDensity - Source files should have a sufficient density of comment lines)
1	0.340431	(S2043 - Superglobals should not be accessed directly)
2	0.308142	(S1808 - Source code should comply with formatting standards)
3	0.255672	(S103 - Lines should not be too long)
4	0.254697	(S4 - [] SQL commands should not be used in this folder.)
5	0.229923	(S113 - Files should contain an empty newline at the end)
6	0.210299	(S1131 - Lines should not end with trailing whitespaces)
7	0.184690	(S105 - Tabulation characters should not be used)
8	0.171468	(InsufficientCommentDensity - Source files should have a sufficient density of comment lines, S2043 - Superglobals should not be accessed directly)

Tabela 5-18. Análise A – Regra de Associação – Resultado da execução do script question-3-metric-1.ipynb. Fonte: o Autor.

	antecedents	consequents	antecedent support	consequent support	support	confidence
0	(InsufficientCommentDensity - Source files should have a sufficient density of comment lines)	(S2043 - Superglobals should not be accessed directly)	0.409882	0.340431	0.171468	0.418336
1	(S2043 - Superglobals should not be accessed directly)	(InsufficientCommentDensity - Source files should have a sufficient density of comment lines)	0.340431	0.409882	0.171468	0.503679

g. Questão 4 - Métrica 1 - Qual a tendência de *code smell* criado?

A Figura 5-8Tabela 5-8 demonstra a média móvel da criação de *code smell*, o eixo X apresenta o período em mês/ano o eixo Y a quantidade de *code smell*. Para

analisar a média móvel foi considerado uma janela de 12 meses para as 6 últimas observações, à medida que a média para cada mês é calculada. A linha em azul demonstra a evolução da criação de *code smell* por todos os desenvolvedores, a linha laranja demonstra a evolução da criação de *code smell* pelo DESENVOLVEDOR 63. Pode-se observar no gráfico da Figura 5-8 que a redução de criação de *code smell* pelo DESENVOLVEDOR 63 faz com que a média de criação de *code smell* em azul (Figura 5-8) tenha uma redução drástica. Já para o índice de esforço de para correção de *code smell* (Figura 5-9) em azul inicia uma tendência de alta sempre acompanhada do índice de esforço de criação de *code smell* do DESENVOLVEDOR 63 em laranja. Os dois gráficos apresentados na (Figura 5-8) e (Figura 5-9) demonstram que o comportamento de criação de *code smell* e de esforço para correção de *code smell* estão associados a criação de *code smell* do DESENVOLVEDOR 63.

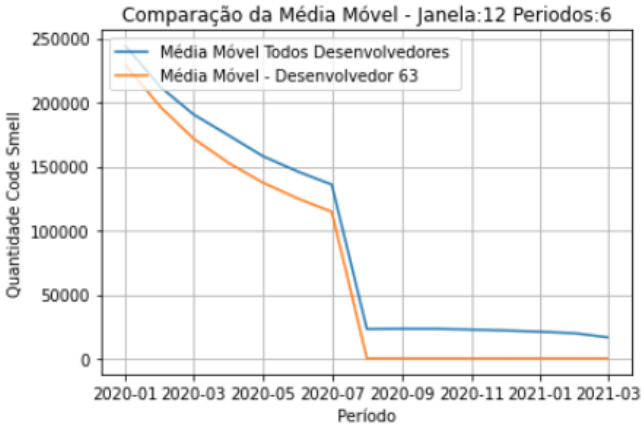


Figura 5-8. Análise A – Resultado da execução do script question-4-metric-1.ipynb (Quantidade de Code Smell). Fonte: o Autor.

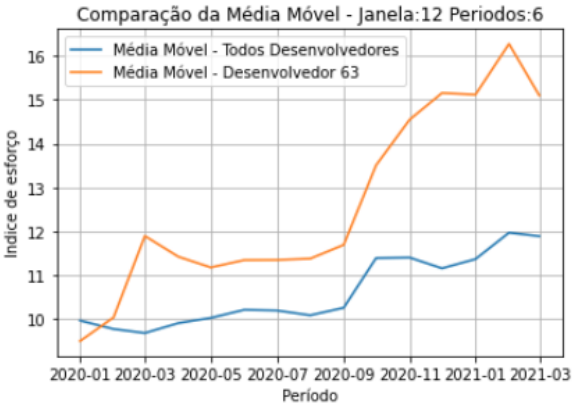


Figura 5-9. Análise A – Resultado da execução do script question-4-metric-1.ipynb (Índice de Esforço). Fonte: o Autor.

h. Questão 5 - Métrica 1 - Qual a previsão de criação de code smell?

O Quadro 5-3 Tabela 5-8demonstra que após a execução dos testes de Dickey-Fuller e KPSS que a serie temporal possui um comportamento estacionário. Esse comportamento é observado na Figura 5-10, após treinar e testar o modelo é possível verificar na faixa cinza e na linha verde do gráfico o comportamento estacionário. Ou seja, a média e a não mudam no decorrer do tempo.

No gráfico exibido na Figura 5-10 contém 465 dias com *code smell* no período de 12/08/2019 até 31/03/2021, 597 dias. Ou seja, 77% dos dias do projeto possui pelo menos um *code smell*. No Quadro 5-4 e apresentado o índice de erro MAPE, resultado aponta que em média a previsão apresentada no gráfico está incorreta em 13.80%.

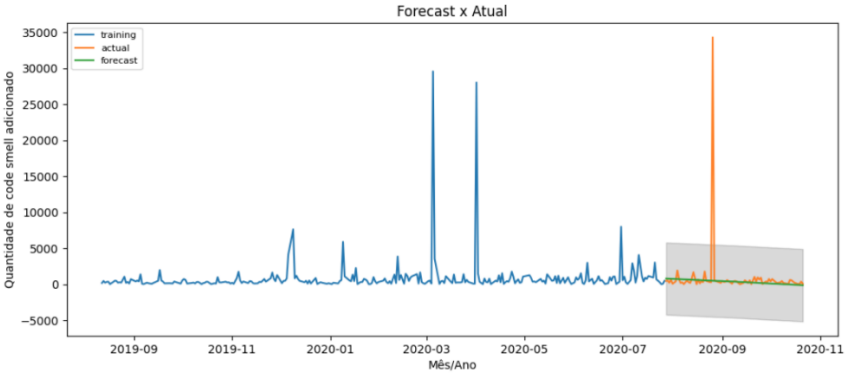


Figura 5-10. Análise A – Resultado gráfico da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

Quadro 5-3. Análise A – Saída da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

```
Results of Dickey-Fuller Test:
Test Statistic                -20.718166
p-value                       0.000000
Lags Used                     0.000000
Number of Observations Used   465.000000
Critical Value (1%)           -3.444491
Critical Value (5%)           -2.867776
Critical Value (10%)          -2.570091
dtype: float64
Result adf : The series is  stationary
Reject Ho - Time Series is Stationary'
Results of KPSS Test:
KPSS Statistic: 0.3058667331918606
p-value: 0.1
num lags: 1
Critial Values:
   10% : 0.347
   5% : 0.463
   2.5% : 0.574
   1% : 0.739
Result kpss_test: The series is  stationary
Observations: 116 Treino: 92 Teste: 24
Melhor AIC Treino : 1395.6963614937454 (0, 2, 3)
Observations: 232 Treino: 185 Teste: 47
Melhor AIC Treino : 3475.686371104389 (0, 2, 2)
Observations: 348 Treino: 278 Teste: 70
Melhor AIC Treino : 5139.955002351771 (2, 1, 3)
AIC Selecionado : (0, 2, 2)
```

Quadro 5-4. Análise A – Erros – Saída de resultado da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

NOME ERRO	VALOR
MAPE	13.80

5.3.4 FASE DE EXECUÇÃO – ANÁLISE B

Conforme apresentado anteriormente a execução do experimento está diretamente associada à execução do processo TDMINING, principalmente o SUBPROCESSO 2 - ETL. Para o Experimento 2 aplicou o SUBPROCESSO 2 – ETL o fluxo DC1 igual a “Sim”.

Com base nas questões e métricas definidas para o experimento, a seguir são apresentados os resultados da execução dos *scripts* que ocorreu por meio da

execução do processo TDMINING, para o Experimento 2 – Empresa – Análise B. Análise B segue o processo TDMINING onde há intervenção do ESPECIALISTA.

a. Teste de normalidade

O resultado do *script* teste-normalidade.ipynb pode ser observado na Quadro 5-5 e Figura 5-11. É possível observar na Figura 5-11 que dos 187 desenvolvedores tiveram uma média de *code smell* de 3.300.

Quadro 5-5. Normalidade – Resultado. Fonte: o Autor.

```
Shapiro-Wilk Test Statistics=0.305, p=0.000 - Does not look Gaussian
(reject H0)
Kolmogorv-Smirnov Test Statistics=0.913, p=0.000' - Does not look
Gaussian (reject H0)
```

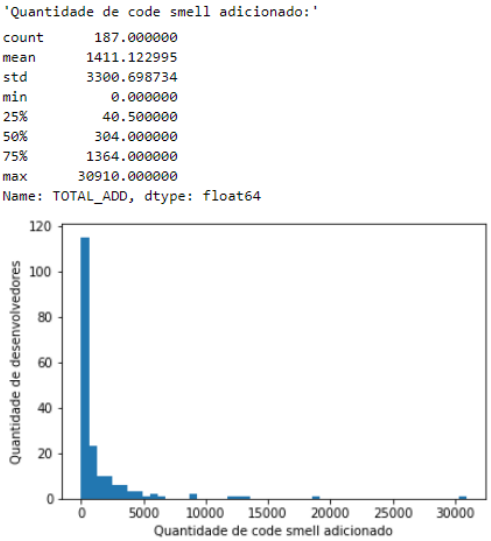


Figura 5-11. Análise B – Gráfico teste de normalidade. Fonte: o Autor.

b. Teste de hipótese

Para testar a hipótese H0 foi utilizado o teste de correlação de Spearman para verificar o efeito da quantidade de *code smell* (variável dependente) x quantidade de *commits* (variável dependente). Como o resultado do teste é menor do que o nível de significância (0,05) é possível negar a hipótese nula H0, o resultado de p-valor pode ser observado no Quadro 5-6.

Portanto, é possível afirmar que há uma correlação entre a dívida técnica de *code smell* e a quantidade de *commits*. Isso significa que quanto mais commits ocorrer maior e a chance de criação de *code smell*. A Figura 5-2 mostra o gráfico de dispersão

em relação a quantidade de *code smell* e a quantidade de *commits* dos projetos selecionados. Com a Figura 5-12 é possível notar a relação positiva entre as duas variáveis.

Quadro 5-6. Comparação do resultado do teste de hipótese. Fonte: o Autor

Spearman: Quantidade de commits x Quantidade de code smells = 0.7547164257967702 Spearman: p=2.977893744575028e-29

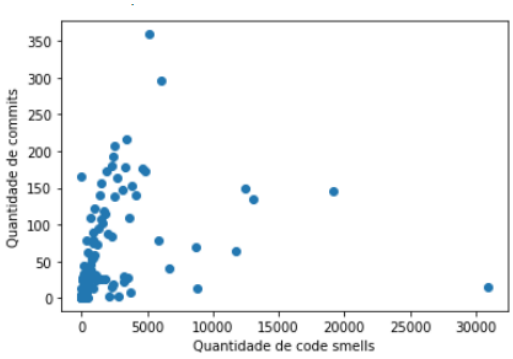


Figura 5-12. Análise B – Gráfico de dispersão do teste de hipótese. Fonte: o Autor.

c. Questão 1 métrica 1 - Qual o desenvolvedor mais cria code smells?

A Tabela 5-19 demonstra que o DESENVOLVEDOR 130 foi o desenvolvedor que mais criou *code smell,* com 11,71%. No entanto, se observar o Índice de esforço de 2,15 minutos de esforço por *code smell* é menor que os 8,68 no âmbito geral do projeto. Isso significa que na média, possivelmente que os *code smells* criados pelo DESENVOLVEDOR 130 possuem uma menor severidade, pois quanto menor a severidade menor tende ser o esforço para correção.

Tabela 5-19. Análise B – Comparação do resultado da execução do script question-1-metric-1.ipynb. Fonte: o Autor.

DESENVOLVEDOR	TOTAL	ESFORÇO	ÍNDICE_ESFORÇO	PERCENTUAL
130	30.910	66.577	2,15	11,71
94	19.101	228.481	11,96	7,23
172	13.053	188.833	14,47	4,94
157	12.418	191.555	15,43	4,70
153	11.764	130.365	11,08	4,45
131	8.832	61.811	7	3,34
122	8.713	44.670	5,13	3,30
60	6.636	68.575	10,33	2,51
74	6.020	39.501	6,56	2,28

DEMAIS 169 DEV	146.433	1.269.280	8,66	55,49
TOTAL	263.880	2.289.648	8,68	100,00

d. Questão 1 métrica 2 - Qual o desenvolvedor mais remove code smells?

A Tabela 5-20Tabela 5-12 demonstra que o DESENVOLVEDOR 134 foi o desenvolvedor que mais removeu *code smell,* com 13,92%. Dos 178 desenvolvedores que adicionaram *code smell* apenas 116 removeram 1 ou mais *code smell*. Ou seja aproximadamente 35% dos desenvolvedores não tem realizaram remoção de *code smell.*

Tabela 5-20. Análise B – Comparação do resultado da execução do script question-1-metric-2.ipynb. Fonte: o Autor.

DESENVOLVEDOR	QUANTIDADE	PERCENTUAL
134	675	13,92
94	341	7,03
65	320	6,60
172	289	5,96
40	211	4,35
113	202	4,16
153	184	3.,79
95	182	3,75
DEMAIS 108 DEV	2444	50,41

e. Questão 2 - Métrica 1 - Qual o *code smell* que foi mais criado pelos desenvolvedores?

A Tabela 5-21 demonstra que a regra de *code smell* ‘S1808 - *Source code should comply with formatting standards* e *‘InsufficientCommentDensity - Source files should have a sufficient density of comment lines’* representam 36% das violações de regras de *code smell* executadas pelos desenvolvedores, dentre os 134 tipos de *code smell.*

Tabela 5-21. Análise B – Resultado da execução do script question-2-metric-1.ipynb (Todos os Desenvolvedores) – Script A. Fonte: o Autor.

REGRA	(A) QUANTIDADE	(B) ESFORÇO (MIN)	ÍNDICE ESFORÇO (B/A)	PERCENTUAL

S1808 - *Source code should comply with formatting standards*	72.528	72.528	1,00	27,49
InsufficientCommentDensity - Source files should have a sufficient density of comment lines	24.608	123.232	5,01	9,33
S1131 - *Lines should not end with trailing whitespaces*	16.356	16.356	1,00	6,20
S139 - *Comments should not be located at the end of lines of code*	16.347	16.347	1,00	6,19
S4 - [SE] SQL *commands should not be used in this folder.*	15.898	715.410	45,00	6,02
S103 - *Lines should not be too long*	13.853	13.853	1,00	5,25
S105 - *Tabulation characters should not be used*	13.065	26.130	2,00	4,95
S113 - *Files should contain an empty newline at the end*	10.476	10.476	1,00	3,97
S3 - [SE] *Check for* $_REQUEST *use in folder* /include/*.	7.052	317.340	45,00	2,67
DEMAIS 125 REGRAS*	73.697	977.976	13,27	27,93
TOTAL	**263.880**	**2.289.648**	**8,68**	**100,00**

A Tabela 5-22 demonstra que a regra de *code smell* ‘S1808 - *Source code should comply with formatting standards ’* e *‘*S121 - *Control structures should use curly braces* representam 75% das violações de regras de *code smell* executadas pelos DESENVOLVEDOR 130. Ou seja, dos 70 tipos de *code smell* criado por este desenvolvedor, 2 tipos de *code smell* são identificados como os mais frequentemente criado pelo DESENVOLVEDOR 130. Visto isso é possível sugerir que tal desenvolvedor aprenda como resolver esses 2 tipos específicos de *code smell.*

Tabela 5-22. Análise B – Resultado da execução do script question-2-metric-1.ipynb (Desenvolvedor 130) – Script B. Fonte: o Autor.

REGRA	(A) QUANTIDADE	(B) ESFORÇO (MIN)	INDICE ESFORÇO (B/A)	PERCENTUAL
S1808 - *Source code should comply with formatting standards*	22.244	22.244	1,00	71,96
S121 - *Control structures should use curly braces*	1.116	2.232	2,00	3,61
S139 - *Comments should not be located at the end of lines of code*	914	914	1,00	2,96
S122 - *Statements should be on separate lines*	892	892	1,00	2,89

S125 - *Sections of code should not be commented out*	838	4.190	5,00	2,71
S116 - *Field names should comply with a naming convention*	580	1.160	2,00	1,88
S1766 - *More than one property should not be declared per statement*	536	1.072	2,00	1,73
S1131 - *Lines should not end with trailing whitespaces*	531	531	1,00	1,72
S3973 - *A conditionally executed single line should be denoted by indentation*	402	4.020	10,00	1,30
DEMAIS 61 REGRAS	2.857	29.322	10,26	9,24
TOTAL	**30,910**	**66.577**	**2,15**	**100,00**

A Tabela 5-23Tabela 5-15 demonstra que 53,48% das violações de regras de code smell executadas pelos desenvolvedores são da severidade MAJOR. Ou seja, se necessário definir um critério para qual *code smell* refatorar primeiramente, a escolha por severidade pode ser uma estratégia válida.

Tabela 5-23. Análise B – Resultado da execução do script question-2-metric-1.ipynb (Todos os Desenvolvedores) – Script C. Fonte: o Autor.

SEVERIDADE	QUANTIDADE	ESFORCO	INDICE_ESFORCO	PERCENTUAL
1-BLOCKER	23.818	1.037.172	43,55	9,03
2-CRITICAL	25.850	89.545	3,46	9,80
3-MAJOR	141.110	376.305	2,67	53,48
4-MINOR	73.102	786.626	10,76	27,70
5-INFO	0	0	0.00	0.00
TOTAL	**263.880**	**2.289.648**	**8,68**	**100,00**

A Tabela 5-24 demonstra que 88% das violações de regras de code smell executadas pelo DESENVOLVEDOR 130 são da severidade MAJOR. Ou seja, se necessário definir um critério para qual *code smell* refatorar primeiramente, a escolha por severidade pode ser uma estratégia válida.

Tabela 5-24. Análise B – Resultado da execução do script question-2-metric-1.ipynb (Desenvolvedor 130) – Script D. Fonte: o Autor.

SEVERIDADE	QUANTIDADE	ESFORCO	INDICE_ESFORCO	PERCENTUAL
1-BLOCKER	189	23.85	12,61	0,61
2-CRITICAL	2.708	11.739	4,33	8,76
3-MAJOR	27.206	37.994	1,39	88,01
4-MINOR	807	14.459	17,9	2,61

5-INFO	0	0	0	0
TOTAL	30.910	66.577	2,15	100

f. Questão 3 - Métrica 1 - Quais os *code smells* que ocorrem em comum pelo desenvolvedor?

A Tabela 5-25 apresenta a frequência que um conjunto de itens, *code smell*, ocorre na base de dados. Por exemplo o *code smell (S1808 - Source code should comply with formatting standards),* ocorre em 31% dos *commits*. Já a Tabela 5-26, demonstra que a combinação da condição SE *code smell (S113 - Files should contain an empty newline at the end) ENTÃO code smell (InsufficientCommentDensity - Source files should have a sufficient density of comment lines)* ocorre em 16% dos *commits,* a medida de confiança indica a frequência que a regra se mostrou válida em 67% dos casos. Observando que tanto para a análise dos itens frequentes quanto para regra de associação foi considerado um limite mínimo de 16% para a medida de suporte.

Tabela 5-25. Análise B – Itens Frequentes – Resultado da execução do script question-3-metric-1.ipynb. Fonte: o Autor.

support	itemsets
0.319690	(S1808 - Source code should comply with formatting standards)
0.212396	(S1131 - Lines should not end with trailing whitespaces)
0.429177	(InsufficientCommentDensity - Source files should have a sufficient density of comment lines)
0.241047	(S103 - Lines should not be too long)
0.239585	(S113 - Files should contain an empty newline at the end)
0.168396	(S3776 - Cognitive Complexity of functions should not be too high)
0.192370	(S105 - Tabulation characters should not be used)
0.266043	(S4 - [] SQL commands should not be used in this folder.)
0.162257	(S3 - [] Check for $_REQUEST use in folder /include/*.)
0.161088	(S113 - Files should contain an empty newline at the end, InsufficientCommentDensity - Source files should have a sufficient density of comment lines)

Tabela 5-26. Análise B – Regra de Associação – Resultado da execução do script question-3-metric-1.ipynb. Fonte: o Autor.

antecedents	consequents	antecedent support	consequent support	support	confidence
(S113 - Files should contain an empty newline at the end)	(InsufficientCommentDensity - Source files should have a sufficient density of comment lines)	0.239585	0.429177	0.161088	0.672361
(InsufficientCommentDensity - Source files should have a sufficient density of comment lines)	(S113 - Files should contain an empty newline at the end)	0.429177	0.239585	0.161088	0.375341

g. Questão 4 - Métrica 1 - Qual a tendência de code smell criado?

A Figura 5-13 demonstra a média móvel da criação de *code smell*, o eixo X apresenta o período em mês/ano o eixo Y a quantidade de *code smell*. Para analisar a média móvel foi considerado uma janela de 12 meses para as 6 últimas observações, à medida que a média para cada mês é calculada. A linha em azul demonstra a evolução da criação de *code smell* por todos os desenvolvedores, a linha laranja demonstra a evolução da criação de *code smell* pelo DESENVOLVEDOR 130. Pode-se observar no gráfico da Figura 5-13 que a média de criação de *code smell* pelo DESENVOLVEDOR 130 sempre esteve abaixo de todos os desenvolvedores, o mesmo comportamento ocorre para o índice de esforço ver Figura 5-14.

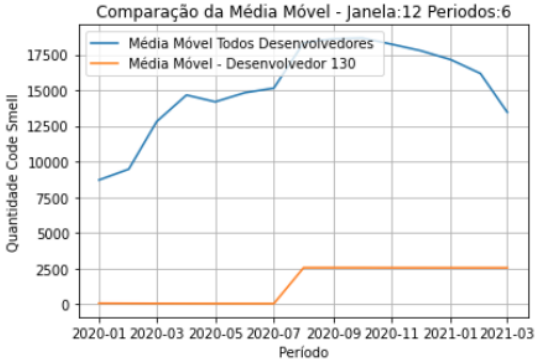


Figura 5-13: Análise B – Resultado da execução do script question-4-metric-1.ipynb (Quantidade de Code Smell). Fonte: o Autor

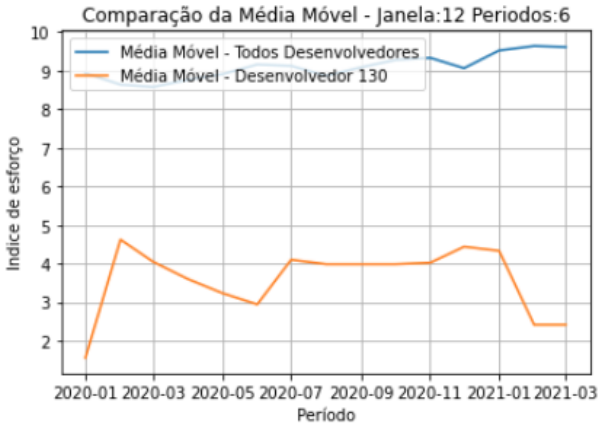


Figura 5-14: Análise B – Resultado da execução do script question-4-metric-1.ipynb (Índice de Esforço). Fonte: o Autor.

h. Questão 5 - Métrica 1 - Qual a previsão de criação de code smell?

O Quadro 5-7 Tabela 5-8demonstra que após a execução dos testes de Dickey-Fuller e KPSS que a serie temporal possui um comportamento estacionário. Esse comportamento é observado na Figura 5-15, após treinar e testar o modelo é possível verificar na faixa cinza e na linha verde do gráfico o comportamento estacionário. Ou seja, a média e a não mudam no decorrer do tempo.

No gráfico exibido na Figura 5-15, contém 464 dias com *code smell* no período de 12/08/2019 até 31/03/2021, 597 dias. Ou seja, 77% dos dias do projeto possui pelo menos um *code smell*. No Quadro 5-4 e apresentado o índice de erro MAPE, resultado aponta que em média a previsão apresentada no gráfico está incorreta em 14.51%.

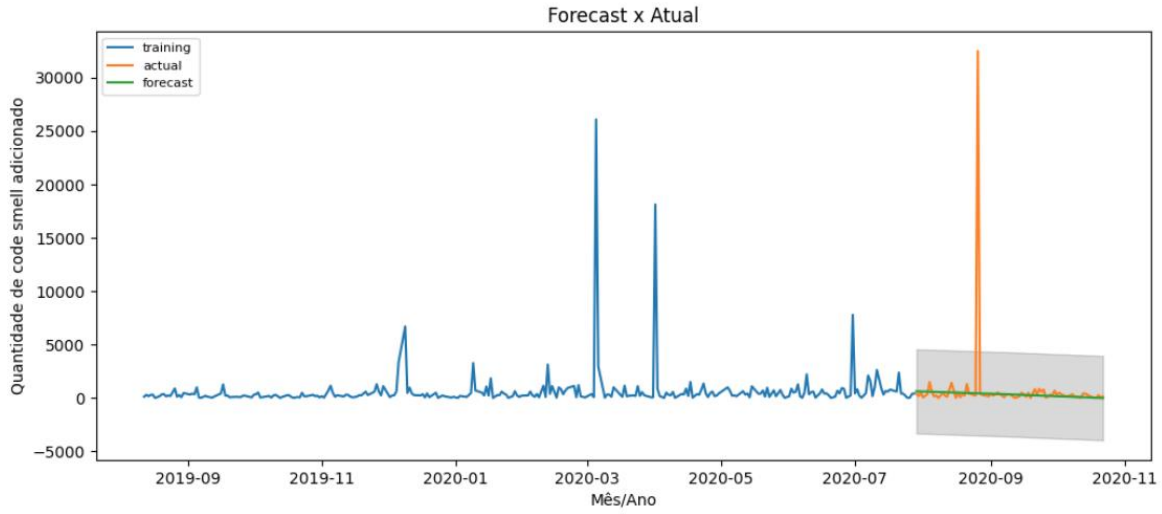


Figura 5-15. Análise B – Resultado gráfico da execução do script question-5-metric-1.ipynb. Fonte: o Autor.

Quadro 5-7. Análise B – Saída da execução do script question-5-metric-1.ipynb. Fonte: o Autor

Results of Dickey-Fuller Test:
Test Statistic -20.805280
p-value 0.000000
Lags Used 0.000000
Number of Observations Used 464.000000
Critical Value (1%) -3.444522
Critical Value (5%) -2.867789
Critical Value (10%) -2.570099
dtype: float64
Result adf : The series is stationary
Reject Ho - Time Series is Stationary
Results of KPSS Test:

```
KPSS Statistic: 0.24887380326859607
p-value: 0.1
num lags: 1
Critial Values:
   10% : 0.347
   5% : 0.463
   2.5% : 0.574
   1% : 0.739
Result kpss_test: The series is  stationary
Observations: 116 Treino: 92 Teste: 24
Melhor AIC Treino : 1349.0618418431438 (0, 2, 3)
Observations: 232 Treino: 185 Teste: 47
Melhor AIC Treino : 3385.976471654078 (0, 2, 2)
Observations: 348 Treino: 278 Teste: 70
Melhor AIC Treino : 5008.391652712416 (0, 2, 2)
AIC Selecionado : (0, 2, 2)
```

Tabela 5-27. Análise B – Saída de resultado da execução do script question-5-metric-1.ipynb. Fonte: o Autor

ERRO	VALOR
MAPE	14.51

5.3.5 FASE DE EXECUÇÃO – VALIDAÇÃO DOS DADOS

Para o Experimento 2 a validação dos dados ocorreu em conjunto com ESPECIALISTA, pois na ANÁLISE B, conforme restrições apontadas no APÊNDICE G – DOC4. *Code smells* que foram desconsiderados por serem aceitos, tendo em vista a arquitetura do produto desenvolvido. No entanto para a ANÁLISE A, não foi aplicada validação dos dados justamente para servir de meio de comparação para ANÁLISE B.

5.3.6 FASE DE ANÁLISE E INTERPRETAÇÃO – ESTATISTICA DESCRITIVA

Nesse tópico será apresentada a análise dos dados do Experimento 2, a medida de índice de esforço (IE) e o índice de Dias para Correção de *code smell* (DCcs).

Após a finalização da fase de execução do experimento foi possível iniciar a análise dos dados gerados. Os dados visam responder às questões e métricas definidas na Tabela 5-1.

Para o Experimento 2, foi necessário comparar os dados da Análise A x Análise B, vista a redução de conjuntos apontada anteriormente. O comportamento está em relação à quantidade total de *code smell* x a soma total do esforço para correção. Conforme esperado, após a redução de conjuntos a quantidade *code smell* reduziu, conforme pode ser observado na Tabela 5-28

Tabela 5-28 - Comparação da quantidade de *code smell* e esforço das análises do Experimento 2. Fonte: o Autor

Análise A			**Análise B**		
Quantidade de *code smell*	Soma de esforço (minutos)	Esforço em dias de 24h	Quantidade de *code smell*	Soma de esforço (min)	Esforço em dias de 24h
1.725.156	12.092.650	8.397	263.88	2.289.648	1.590

Logo, a redução de conjuntos de fato irá reduzir a quantidade de *code smells* e a quantidade de esforço. No entanto, surgiu a dúvida “O que é possível observar além das reduções óbvias de quantidade de *code smell* e esforço?” Para tentar responder essas perguntas foi necessário aplicar as equações de índice de esforço (IE) e o índice de Dias de Correção de *code smells* (DCcs).

Aplicando a Equação 7 para ambas as análises do Experimento 2 temos o IE como resultado para Análise A de 7,01 minutos de esforço por *code smell* e para a Análise B temos 8,68 minutos de esforço por *code smell,* conforme apresentado na Tabela 5-29.

Tabela 5-29. Comparação do índice de esforço entre as análises do Experimento 2.Fonte: o Autor

Análise A			**Análise B**		
Quantidade de *code smell*	Esforço em dias de 24h	IE	Quantidade de *code smell*	Esforço em dias de 24h	IE
1.725.156	8.397	7.01	263.880	1.590	8.68

O resultado apresentado na Tabela 5-29 demonstra que o tempo de correção por *code smell* teve um aumento, pois uma das condições apontadas pelo especialista dos dados na Análise B foi justamente desconsiderar os *code smells* com regras de severidade 'INFO', vide APÊNDICE G, pois, no contexto da empresa, esse tipo de *code smell* não deve ser contabilizado dado que a correção ocorre por meio de ferramentas automáticas para correção. Em um cenário em que a correção de *code smell* é baseada na quantidade, onde o desenvolvedor se compromete a resolver *code smells*/dia, esses resultados significam que se despenderá mais tempo no dia para resolver os problemas.

Imaginando um cenário hipotético, onde a empresa possui 10 desenvolvedores que podem atuar na correção de *code smell*, quantos dias os desenvolvedores resolveriam as correções, considerando que não entrarão novos *code smells* durante o período de correção?

Conforme já apresentado, o estudo de Stripe (2018) aponta que os desenvolvedores passam 17 horas semanais envolvidos com problemas de manutenção (13,5h com DT + 3,8h com código com baixa qualidade), como depuração e refatoração. Logo, para o caso hipotético, foram consideradas 3 horas e 40 minutos por dia com a atividade de correção por desenvolvedor.

Com isso, tem-se como variáveis a soma dos esforços ($\sum_{effort=0}^{\infty} effort$) dividida pela quantidade de desenvolvedores ($qtDev$), multiplicada pelos minutos úteis para correção (MUD) , conforme pode ser observado na Tabela 5-30. Aplicando a equação para Análise A e Análise B do Experimento 2, tem-se o resultado apresentado na Tabela 5-30.

Tabela 5-30. Resultado do cálculo dos DCcs das análises A e B do Experimento 2. Fonte: o Autor.

	Esforço	Quantidade de desenvolvedores	Minutos úteis por dias	Quantidade de minutos por dia	Quantidade de dias de 3h40min
Análise A	12.092.650	10	220	2.200	5.496
Análise B	2.289.648	10	220	2.200	1.040

Outra comparação que pode ser realizada entra a Análise A e a Análise B é que em ambas as análises, a hipótese de quanto mais *commits* mais *code smells* apresentou 75% de certeza, mesmo com a redução de conjuntos/dados. Outra comparação foi que em ambas as análises existe uma característica de quantidade de code smells/dia em uma série estacionária. No entanto, na Análise B do gráfico de previsão, observa-se um pequeno movimento de tendência de baixa. Também é possível observar que na Análise B a regra de associação, após a redução de conjuntos.

5.3.7 FASE DE ANÁLISE E INTERPRETAÇÃO – DEDUÇÃO DE CONJUNTOS

Os dados do Experimento 2 precisaram passar por um processo de limpeza da análise estática inicial do projeto, pois o SonarQube iniciou a utilização após versões de produto já distribuídas. A limpeza de dados foi necessária, pois em algumas análises estáticas realizadas não foi possível identificar o autor do *code smell*.

Ainda para o Experimento 2, foi necessário desconsiderar alguns dados, conforme orientação do especialista da empresa. Os dados ignorados do Experimento 2 seguiram a atividade SUBPROCESSO 3 – ATIVIDADE 5 – VALIDAR INFORMAÇÕES. Com o apoio do especialista, que tem o conhecimento dos dados de monitoramento da DT da empresa, foi possível definir quais dados deveriam ser ignorados. As condições apresentadas pelo especialista podem ser observadas no item APÊNDICE H – DOC4 – CHECKLIST DE RESTRIÇÕES.

5.3.8 FASE DE ANÁLISE E INTERPRETAÇÃO – TESTE DE HIPÓTESE

O teste de hipótese de ambos os experimentos faz parte do processo TDMINING, e, em ambos os casos, foi executado na atividade SUBPROCESSO 3 –

ATIVIDADE 3 – EXECUTAR SCRIPTS JUPYTER. O objetivo foi verificar se a quantidade de commits tem relação com a quantidade de code smells.

Para testar a hipótese dos experimentos foi utilizado o teste de regressão não paramétrico de *Spearman,* a fim de verificar a influência que a quantidade de *code smell (*variável dependente) tem na quantidade de *commits* (variável independente).

Ao executar o *script* teste-hipotese.ipynb, o resultado da execução pode ser avaliado nos gráficos de dispersão dos experimentos, o qual exibe quantidade de *code smells* e a quantidade de *commits* dos desenvolvedores. No gráfico, é possível notar a relação positiva entre as duas variáveis. Como, por exemplo, o resultado do teste de hipótese do Experimento 1, onde é possível observar o valor do coeficiente de correlação (0,75) e que a variação da quantidade de *commits* influencia em 75% a quantidade de *code smell*.

Como o resultado da significância (p) da correlação é menor do que o nível de significância (0,05), é possível negar a hipótese nula. Portanto, é possível afirmar que no Experimento 2 há uma influência significativa da quantidade de *commits* na quantidade de *code smell*, tanto para a Análise A quanto para Análise B.

CAPÍTULO 6 - CONSIDERAÇÕES

O capítulo anterior descreveu a execução do experimento em engenharia de software por meio do processo TDMINING, conforme descrito nos Capítulos 3, 4 e 5. Este capítulo tem o objetivo de discutir os resultados encontrados e abordar as considerações.

6.1 RESPOSTA AO OBJETIVO DE PESQUISA

A execução do processo TDMINING ocorreu para três instâncias de processo. Uma instância denominada Experimento 1, com dados obtidos de repositório público, e outras duas instâncias de experimento denominadas Experimento 2 – Análise A e Experimento 2 - Análise B. Sendo que, para o Experimento 2, os dados foram obtidos de um projeto originário da indústria.

A execução dos experimentos teve como resultado responder questões que possibilitem ao desenvolvedor entender seu comportamento em relação ao *code smell.* Esse comportamento pode ser observado nos dados apresentados ao término de cada experimento, logo, os resultados individuais de cada experimento atendem à necessidade desta pesquisa, respondendo ao objetivo geral proposto.

Sobre a execução dos experimentos, seguindo o processo definido, foi suficiente para atender o objetivo específico de criar os passos necessários para executar a abordagem proposta. Quanto à avaliação do processo como objetivo específico, também foi atendida, vista a participação de um especialista no Experimento 2.

A discussão dos resultados da execução dos experimentos ocorreu na fase de análise e interpretação, onde foram explicados os resultados por meio de estatística descritiva. A apresentação dos resultados também ocorreu na fase de apresentação e empacotamento do experimento.

6.2 TRABALHOS RELACIONADOS

Diferentemente do trabalho de Ebert *et al.* (2019), esta dissertação se baseou nos conceitos de Engenharia de Software Experimental, propondo e experimentando um processo para gerar conhecimento, com base na mineração de dados. No entanto, o estudo diferenciando-se da perspectiva que, neste caso é a percepção de DT de *code smell,* do ponto de vista de desenvolvedor e não de produto. Visto isso, o processo TDMINING pode disponibilizar uma fonte de dados para o framework Q-Rapids proposto por Ebert et al. (2019).

Quanto ao trabalho de Tsoukalas et al. (2020) sobre previsão de dívida técnica utilizado ML, a presente dissertação colaborou no sentido de criar um processo e um modelo de dados que também podem ser úteis para previsibilidade de dívida técnica. No entanto, a dissertação vai um pouco além, no sentido do modelo de dados proposto auxiliar para análises de *Business Intelligence* e outras tarefas de mineração de dados como regras de associação.

O presente estudo também vai ao encontro do *technical debt dataset*, apresentado por Lenarduzzi *et al.* (2019), pois possui características e propósitos semelhantes. Não obstante, essa pesquisa apresenta um modelo de dados dimensional que permite extrair diversas visões sobre os fatos de DT de *code smell*, além dos que já foram apresentados anteriormente.

Ademais, a presente dissertação ainda atende as necessidades apontadas por Pereira dos Reis et al. (2021), ou seja, conjuntos de dados para facilitar a replicação de experimentos de detecção e validação das técnicas de validação de códigos. A disponibilização do fluxo de trabalhos científicos correspondentes e conjuntos de dados, permite a reprodução do experimento, a apresentação e a visualização de dados de *code smell*.

6.3 VALIDADE E CONFIABILIDADE DA PESQUISA

Para validar e garantir a qualidade dos resultados deste estudo utilizou-se dos seguintes meios:

- Rigor metodológico na execução dos experimentos, por meio do processo TDMINING, combinado aos processos propostos por Wohlin et al. (2012) e Fayyad (1996);

- Rigor na criação dos *datasets* de dados, principalmente dos dados a partir dos quais o pesquisador criou o Experimento 1. Rigor na avaliação das análises estáticas realizadas e avaliação de os dados gerados no SonarQube x GIT estão contidos no *dataset* criado.
- Para o Experimento 2, houve o auxílio do especialista da empresa, visando o entendimento dos *code smells*. Isso é importante, devido à larga experiência do especialista tanto no produto, como no processo de desenvolvimento de software daquela empresa.

6.4 RELEVÂNCIA DO ESTUDO

Ter indicadores de qualidade de software é fundamental para promover melhorias tanto no processo de desenvolvimento, quanto no produto gerado, o software. Por se tratar de uma atividade essencialmente cognitiva, o papel do desenvolvedor na qualidade resultante, é fundamental. Os profissionais desenvolvedores possuem conhecimentos e experiências diferentes, além de, pela característica humana, apresentarem lacunas quanto ao conhecimento da linguagem de programação do software que está sendo desenvolvido e em relação à própria lógica de programação. Tais lacunas geralmente são percebidas quando o código-fonte do desenvolvedor é inspecionado.

Este estudo apresentou um processo que pode ser aplicado na indústria, experimentado por meio de procedimentos da engenharia de software experimental, e avaliado por especialista. Esse processo pode ser adaptado à necessidade e realidade das empresas, pois permite apresentar de forma gráfica, quantitativa e estatística os comportamentos do desenvolvedor em relação à criação de *code smells*. Os resultados dos experimentos apresentam informações suficientes para que sejam tomadas ações gerenciais de melhoria contínua, principalmente aquelas que afetam a qualidade de produto, sem deixar de quantificar as oportunidades de melhoria para quem cria o produto.

6.5 CONTRIBUIÇÕES DE PESQUISA

Como principal contribuição desse estudo destaca-se o processo TDMINING, que permite aos interessados, principalmente da indústria, ter um processo já avaliado por meio de ESE. Também um processo documentado, com artefatos

disponibilizados, que o interessado possa replicar o mesmo Experimento 1 ou aplicar em outro repositório de código.

Também como contribuição para a comunidade cientifica, os dados gerados para o Experimento 1 podem servir para outras análises relacionados à dívida técnica de *code smell*. Esses dados podem permitir novos *insights* sobre o assunto ou até mesmo a aplicação de outros testes estatísticos e *data science*.

6.6 LIMITAÇÕES

Neste estudo não foram utilizadas outras ferramentas de análise estática ou outros tipos de repositórios de código-fonte, além dos já citados durante o experimento. Essas limitações foram justificadas na fase de planejamento na avaliação da validação.

Embora tenha sido possível realizar um experimento com dados da indústria, não é possível generalizar que as conclusões sejam as mesmas para outros cenários em outras empresas. Pode-se afirmar que o processo TDMINING pode ser aplicado a outras empresas, além daquela utilizada no Experimento 2.

6.7 TRABALHOS FUTUROS

É possível criar uma ferramenta que abstraia toda a complexidade do experimento em um único produto, envolvendo fluxo contínuo de dados. Com isso, tanto a indústria como a comunidade de engenharia de *software* poderiam obter mais dados e de forma facilitada para realizar outros estudos sobre *code smell*.

Outro estudo possível, seria adaptar o *dataset* para o custo monetário de remediação da DT de *code smell*, para que gestores possam tomar decisões sob o ponto de vista financeiro.

O modelo pode ser adaptado para aceitar a parametrização da correção do esforço por regras de *code smell* ou por regra do desenvolvedor. Dado que, conforme já apontado por Lenarduzzi e Saarimaki e Taibi (2019), o tempo de remediação do SonarQube, em comparação com o tempo real para reduzir a dívida técnica, é geralmente superestimado. Assim, o ideal seria ter uma abordagem personalizada por desenvolvedor em relação ao esforço que cada um tem para corrigir determinada regra de *code smell*. Essas informações tornam o direcionamento das atividades dos

desenvolvedores para correção de *code smell*, pelos gestores, mais precisa. Com isso é possível ainda obter o custo monetário mínimo para remediação do *code smell*.

Mais uma adaptação possível seria do modelo proposto para um cenário de gamificação, onde o desenvolvedor que realizar mais correções de *code smells*, no menor tempo proposto, receberia algum tipo de recompensa.

Outra possibilidade em relação aos métodos de previsão de *code smell* com séries temporais, seria utilizar outros algoritmos e comparar os resultados, por exemplo *SARIMAX* e *Prophet*.

Por fim, outro tema muito discutido na indústria de software diz respeito às *soft skills*, ou seja, as habilidades pessoais dos profissionais. Poderia ser interessante um aprofundamento do entendimento do perfil psicológico (tipologia de Myers-Briggs, por exemplo) e de personalidade (DISC, por exemplo) dos desenvolvedores, a fim de criar uma dimensão de dados no *dataset* proposto que leve em consideração essas características pessoais. Assim, seria possível investigar se diferentes tipos de perfis têm relação com a produção ou não de *code smells.*

6.8 ÉTICA EM PESQUISA DE ESE

Durante todo o processo não houve interação com qualquer desenvolvedor que tenha realizado os *code smells* apresentados nos dados obtidos para o Experimento 1 e para o Experimento 2. A fim de proteger estes profissionais, os nomes foram substituídos por identificadores numéricos, mesmo para o caso do repositório público utilizado no Experimento 1. Dadas estas características, não houve a necessidade de submeter previamente o processo à apreciação do Comitê de Ética em Pesquisa (CEP).

Uma preocupação do pesquisador em relação ao processo TDMINING é que os dados gerados não fossem usados para fins de *rankings* de desenvolvedores e que não servissem como meio para punir ou prejudicar os profissionais, independentemente de serem provenientes de empresa ou de repositório público.

O objetivo principal do processo e dos dados gerados é tão somente permitir que os desenvolvedores conheçam seus maus hábitos de código e que possam corrigi-los, levando à melhor qualidade do software.

REFERÊNCIAS BIBLIOGRÁFICAS

ACKOFF, R. L. Fro[1] Ackoff, R.L. 1989. From data to wisdom. Journal of Applied Systems Analysis. 16, 1 (1989), 3–9.m data to wisdom. **Journal of Applied Systems Analysis**, [*s. l.*], 1989.

AGRAWAL, Rakesh; SRIKANT, Ramakrishnan. Fast Algorithms for Mining Association Rules. *In*: , 1994. **Proc. of 20th International Conference on Very Large Data Bases, {VLDB'94}**. [*S. l.: s. n.*], 1994.

ALVES, Nicolli S.R. *et al.* Towards an ontology of terms on technical debt. **Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014**, [*s. l.*], p. 1–7, 2014. Disponível em: https://doi.org/10.1109/MTD.2014.9

ALVES, Nicolli S.R. N.S.R. *et al.* Identification and management of technical debt: A systematic mapping study. **Information and Software Technology**, [*s. l.*], v. 70, p. 100–121, 2016. Disponível em: https://doi.org/10.1016/j.infsof.2015.10.008

AMPATZOGLOU, Areti Apostolos *et al.* The financial aspect of managing technical debt: A systematic literature review. **Information and Software Technology**, [*s. l.*], v. 64, n. APRIL, p. 52–73, 2015. Disponível em: https://doi.org/10.1016/j.infsof.2015.04.001

ANDERSON, P. *et al.* SARIF-enabled tooling to encourage gradual technical debt reduction. *In*: , 2019. **Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019**. [*S. l.: s. n.*], 2019. p. 71–72. Disponível em: https://doi.org/10.1109/TechDebt.2019.00024

AVGERIOU, Paris *et al.* Managing Technical Debt in Software Engineering. **Dagstuhl Reports**, [*s. l.*], v. 6, n. 4, p. 110–138, 2016a. Disponível em: https://doi.org/10.4230/DagRep.6.4.110

AVGERIOU, Paris *et al.* Reducing friction in software development. **IEEE Software**, [*s. l.*], v. 33, n. 1, p. 66–72, 2016b. Disponível em: https://doi.org/10.1109/MS.2016.13

BASILI, Victor R; CALDIERA, Gianluigi; ROMBACH, H Dieter. The goal question

metric approach. **Encyclopedia of Software Engineering**, [*s. l.*], 1994. Disponível em: https://doi.org/10.1.1.104.8626

BASILI, Victor R.; SHULL, Forrest; LANUBILE, Filippo. Building knowledge through families of experiments. **IEEE Transactions on Software Engineering**, [*s. l.*], 1999. Disponível em: https://doi.org/10.1109/32.799939

BESKER, Terese; MARTINI, Antonio; BOSCH, Jan. The pricey bill of Technical Debt - When and by whom will it be paid? **Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017**, [*s. l.*], p. 13–23, 2017. Disponível em: https://doi.org/10.1109/ICSME.2017.42

BOURQUE, Pierre; FAIRLEY, Richard E. **SWEBOK v.3 - Guide to the Software Engineering - Body of Knowledge.** [*S. l.: s. n.*], 2014. ISSN 07407459. Disponível em: https://doi.org/10.1234/12345678

BOX, George E.P.; JENKINS, Gwilym M.; REINSEL, Gregory C. **Time series analysis: Forecasting and control: Fourth edition**. [*S. l.: s. n.*], 2013. Disponível em: https://doi.org/10.1002/9781118619193

BRUCE, Peter Bruce and Andrew. **Practical Statistics for Data Scientists: 50 Essential Concepts**. [*S. l.: s. n.*], 2017.

BUSCHMANN, Frank. To pay or not to pay technical debt. **IEEE Software**, [*s. l.*], v. 28, n. 6, p. 29–31, 2011. Disponível em: https://doi.org/10.1109/MS.2011.150

CUNNINGHAM, Ward. The WyCash portfolio management system. **Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**, [*s. l.*], v. Part F1296, n. October, p. 29–30, 1992. Disponível em: https://doi.org/10.1145/157710.157715

DE ALMEIDA FILHO, Francisco Goncalves *et al.* Prevalence of bad smells in PL/SQL projects. **IEEE International Conference on Program Comprehension**, [*s. l.*], v. 2019-May, p. 116–121, 2019. Disponível em: https://doi.org/10.1109/ICPC.2019.00025

DEMING, W Edwards. **Out of the Crisis**. [*S. l.*]: The MIT Press, 2000. (MIT Press Books).v. 1*E-book*.

EBERT, C. *et al.* Data science: Technologies for better software. **IEEE Software**, [*s.*

l.], v. 36, n. 6, p. 66–72, 2019. Disponível em: https://doi.org/10.1109/MS.2019.2933681

FAYYAD, Usama; PIATETSKY-SHAPIRO, Gregory; SMYTH, Padhraic. From data mining to knowledge discovery in databases. **AI Magazine**, [*s. l.*], 1996.

FOWLER, Martin. Refactoring: Improving the design of existing code. *In*: , 2002. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [*S. l.: s. n.*], 2002. Disponível em: https://doi.org/10.1007/3-540-45672-4_31

FOWLER, Martin *et al.* Refactoring Improving the Design of Existing Code Second Edition. [*s. l.*], 2019.

GHOLAMY, Afshin; KREINOVICH, Vladik; KOSHELEVA, Olga. Why 70/30 or 80/20 Relation Between Training and Testing Sets : A Pedagogical Explanation. **Departmental Technical Reports (CS)**, [*s. l.*], p. 1–6, 2018.

GJOSHEVSKI, M.; SCHWEIGHOFER, T. Small scale analysis of source code quality with regard to native android mobile applications. *In*: , 2015. **CEUR Workshop Proceedings**. [*S. l.: s. n.*], 2015. p. 9–16.

GOLDSCHMIDT, Ronaldo; PASSOS, Emmanuel. **Data mining: um guia prático**. [*S. l.*]: Elsevier, 2005.

GOUSIOS, Georgios *et al.* Lean ghtorrent: GitHub data on demand. **11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings**, [*s. l.*], p. 384–387, 2014. Disponível em: https://doi.org/10.1145/2597073.2597126

GOUSIOS, Georgios; SPINELLIS, Diomidis. GHTorrent: Github's data from a firehose. **IEEE International Working Conference on Mining Software Repositories**, [*s. l.*], p. 12–21, 2012. Disponível em: https://doi.org/10.1109/MSR.2012.6224294

GU, Jifa; ZHANG, Lingling. Data, DIKW, big data and data science. **Procedia Computer Science**, [*s. l.*], v. 31, n. December 2014, p. 814–821, 2014. Disponível em: https://doi.org/10.1016/j.procs.2014.05.332

GUAMAN, Daniel *et al.* SonarQube as a tool to identify software metrics and technical debt in the source code through static analysis. **2017 7th International**

Workshop on Computer Science and Engineering, WCSE 2017, [*s. l.*], n. July, p. 171–175, 2017. Disponível em: https://doi.org/10.18178/wcse.2017.06.030

HAAS, Roman; NIEDERMAYR, Rainer; JUERGENS, Elmar. Teamscale: Tackle technical debt and control the quality of your software. **Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019**, [*s. l.*], p. 55–56, 2019. Disponível em: https://doi.org/10.1109/TechDebt.2019.00016

HARRINGTON, H J (H. James). **Business process improvement : the breakthrough strategy for total quality, productivity, and competitiveness**. New York ; McGraw-Hill, 1991.

HASSAN, Ahmed E. The road ahead for mining software repositories. **Proceedings of the 2008 Frontiers of Software Maintenance, FoSM 2008**, [*s. l.*], p. 48–57, 2008. Disponível em: https://doi.org/10.1109/FOSM.2008.4659248

HOLVITIE, Johannes. Software implementation knowledge management with technical debt and network analysis. **Proceedings - International Conference on Research Challenges in Information Science**, [*s. l.*], p. 1–6, 2014. Disponível em: https://doi.org/10.1109/RCIS.2014.6861083

HOLVITIE, Johannes; LEPPÄNEN, Ville; HYRYNSALMI, Sami. Technical debt and the effect of agile software development practices on it - An industry practitioner survey. **Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014**, [*s. l.*], p. 35–42, 2014. Disponível em: https://doi.org/10.1109/MTD.2014.8

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION ISO. Iso/Iec 25010:2011. **Software Process: Improvement and Practice**, [*s. l.*], 2011.

JOHNSON, S C. Lint, a C Program Checker. **Comp. Sci. Tech. Rep**, [*s. l.*], p. 78–1273, 1978.

KAZMAN, Rick *et al.* A Case Study in Locating the Architectural Roots of Technical Debt. **Proceedings - International Conference on Software Engineering**, [*s. l.*], v. 2, p. 179–188, 2015. Disponível em: https://doi.org/10.1109/ICSE.2015.146

KIMBALL, Ralph. The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses. **Architecture**, [*s. l.*], 1998.

KIRCHMER, Mathias *et al.* **BPM CBOK Version 4.0: Guide to the Business Process Management Common Body Of Knowledge**. [*S. l.: s. n.*], 2019.

KITCHENHAM, Barbara A. *et al.* Preliminary guidelines for empirical research in software engineering. **IEEE Transactions on Software Engineering**, [*s. l.*], 2002. Disponível em: https://doi.org/10.1109/TSE.2002.1027796

KRUCHTEN, Philippe; NORD, Robert L.; OZKAYA, Ipek. Technical debt: From metaphor to theory and practice. **IEEE Software**, [*s. l.*], v. 29, n. 6, p. 18–21, 2012. Disponível em: https://doi.org/10.1109/MS.2012.167

KUMAR, Satish *et al.* Identifying and Estimating Technical Debt for Service Composition in SaaS Cloud. **2019 IEEE International Conference on Web Services (ICWS)**, [*s. l.*], p. 121–125, 2019. Disponível em: https://doi.org/10.1109/icws.2019.00030

KWIATKOWSKI, Denis *et al.* Testing the null hypothesis of stationarity against the alternative of a unit root. How sure are we that economic time series have a unit root? **Journal of Econometrics**, [*s. l.*], v. 54, n. 1–3, p. 159–178, 1992. Disponível em: https://doi.org/10.1016/0304-4076(92)90104-Y

LACERDA, Guilherme *et al.* Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, [*s. l.*], v. 167, p. 110610, 2020. Disponível em: https://doi.org/10.1016/j.jss.2020.110610

LAROSE, Daniel T. **Discovering Knowledge in Data: An Introduction to Data Mining**. [*S. l.: s. n.*], 2005. Disponível em: https://doi.org/10.1002/0471687545

LENARDUZZI, Valentina *et al.* How long do junior developers take to remove technical debt items? **International Symposium on Empirical Software Engineering and Measurement**, [*s. l.*], n. September, 2020. Disponível em: https://doi.org/10.1145/3382494.3422169

LENARDUZZI, Valentina; SAARIMAKI, Nyyti; TAIBI, Davide. On the Diffuseness of Code Technical Debt in Java Projects of the Apache Ecosystem. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**, [*s. l.*], p. 98–107, 2019. Disponível em: https://doi.org/10.1109/techdebt.2019.00028

LENARDUZZI, Valentina; SAARIMÄKI, Nyyti; TAIBI, Davide. The Technical Debt

Dataset. [*s. l.*], n. May, p. 2–11, 2019. Disponível em: https://doi.org/10.1145/3345629.3345630

LETOUZEY, Jean-Louis. The SQALE Method for Managing Technical Debt Definition Document. [*s. l.*], p. 31–36, 2016. Disponível em: http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf

LETOUZEY, Jean Louis J.-L. Jean Louis. The SQALE method for evaluating technical debt. **2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings**, [*s. l.*], p. 31–36, 2012. Disponível em: https://doi.org/10.1109/MTD.2012.6225997

LETOUZEY, J.L. Jean Louis J.-L.; ILKIEWICZ, Michel. Managing technical debt with the SQALE method. **IEEE Software**, [*s. l.*], v. 29, n. 6, p. 44–51, 2012. Disponível em: https://doi.org/10.1109/MS.2012.129

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [*S. l.: s. n.*], 2017. ISSN 13563890. Disponível em: https://doi.org/10.1177/1356389011400889

MARTIN, Robert C. **Clean Code - A Handbook of Agile Software Craftmanship**. [*S. l.: s. n.*], 2014. ISSN 1098-6596. Disponível em: https://doi.org/10.1007/s13398-014-0173-7.2

MARTIN, Robert C. **The Clean Coder**. [*S. l.: s. n.*], 2011.

MCCONNELL, Steve. **Managing Technical Debt (White Paper)**. [*S. l.: s. n.*], 2013.

MENDES, Thiago S. T.S. *et al.* VisminerTD: a tool for automatic identification and interactive monitoring of the evolution of technical debt items. **Journal of the Brazilian Computer Society**, [*s. l.*], v. 25, n. 1, p. 1–28, 2019. Disponível em: https://doi.org/10.1186/s13173-018-0083-1

OPDYKE, William F. Refactoring: A program restructuring aid in designing object-oriented application frameworks. [*s. l.*], p. 206, 1992.

ORAM, Andy; WILSON, Greg. **Making Software – What Really workds, and Why We believe It**. [*S. l.: s. n.*], 2010.

PEREIRA DOS REIS, José *et al.* **Code Smells Detection and Visualization: A**

Systematic Literature Review. [*S. l.*]: Springer Netherlands, 2021. ISSN 18861784. Disponível em: https://doi.org/10.1007/s11831-021-09566-x

PROJECT MANAGEMENT INSTITUTE, Inc. **PMBOK® Guide - Fifth Edition**. [*S. l.: s. n.*], 2010. ISSN 87569728.

REZENDE, Solange Oliveira. Sistemas Inteligentes Fundamentos e Aplica{ç}{õ}es. **Sistemas Fuzzy**, [*s. l.*], 2003.

RIEL, Arthur J. **Arthur J. Riel**. [*S. l.: s. n.*], 1996.

RIOS, N. *et al.* Causes and effects of the presence of technical debt in agile software projects. *In*: , 2019. **25th Americas Conference on Information Systems, AMCIS 2019**. [*S. l.: s. n.*], 2019.

RIOS, N. *et al.* The most common causes and effects of technical debt: First results from a global family of industrial surveys. *In*: , 2018. **International Symposium on Empirical Software Engineering and Measurement**. [*S. l.: s. n.*], 2018. Disponível em: https://doi.org/10.1145/3239235.3268917

RIOS, N.; MENDONÇA NETO, M.G.D.; SPÍNOLA, R.O. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. **Information and Software Technology**, [*s. l.*], v. 102, p. 117–145, 2018. Disponível em: https://doi.org/10.1016/j.infsof.2018.05.010

SAARIM, Nyyti; BALDASSARRE, Maria Teresa; ROMANO, Simone. On the Accuracy of SonarQube Technical Debt Remediation Time. **2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**, [*s. l.*], n. August, p. 317–324, 2019. Disponível em: https://doi.org/10.1109/SEAA.2019.00055

SALAMEA, Maria Jose M.J.; FARRE, Carles. Influence of Developer Factors on Code Quality: A Data Study. **Proceedings - Companion of the 19th IEEE International Conference on Software Quality, Reliability and Security, QRS-C 2019**, [*s. l.*], p. 120–125, 2019. Disponível em: https://doi.org/10.1109/QRS-C.2019.00035

SEAMAN, Carolyn; GUO, Yuepu. **Measuring and Monitoring Technical Debt**. [*S. l.: s. n.*], 2011. ISSN 00652458. Disponível em: https://doi.org/10.1016/B978-0-12-

385512-1.00002-5

SHARMA, Tushar; MISHRA, Pratibha; TIWARI, Rohit. Designite - A software design quality assessment tool. **Proceedings - 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities, Bridge 2016**, [*s. l.*], p. 1–4, 2016. Disponível em: https://doi.org/10.1145/2896935.2896938

SJOBERG, Dag I.K. *et al.* Quantifying the effect of code smells on maintenance effort. **IEEE Transactions on Software Engineering**, [*s. l.*], v. 39, n. 8, p. 1144–1156, 2013. Disponível em: https://doi.org/10.1109/TSE.2012.89

SKOURLETOPOULOS, Georgios *et al.* Predicting and quantifying the technical debt in cloud software engineering. **2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, CAMAD 2014**, [*s. l.*], p. 36–40, 2014. Disponível em: https://doi.org/10.1109/CAMAD.2014.7033201

SPADINI, Davide; ANICHE, Maurício; BACCHELLI, Alberto. PyDriller: Python framework for mining software repositories. *In*: , 2018. **ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [*S. l.: s. n.*], 2018. Disponível em: https://doi.org/10.1145/3236024.3264598

STRIPE. Stripe - The Developer Coe ffi cient. [*s. l.*], n. September, 2018.

SURYANARAYANA, Girish; SAMARTHYAM, Ganesh; SHARMA, Tushar. **Refactoring for software design smells: managing technical debt**. [*S. l.*]: Elsevier, 2015.

THAKUR, Vishvajeet; KESSENTINI, Marouane; SHARMA, Tushar. QScored: An Open Platform for Code Quality Ranking and Visualization. **Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020**, [*s. l.*], p. 818–821, 2020. Disponível em: https://doi.org/10.1109/ICSME46990.2020.00101

TSOUKALAS, Dimitrios *et al.* Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey. **9th International Conference on Intelligent Systems 2018: Theory, Research and Innovation in Applications, IS 2018 - Proceedings**,

[*s. l.*], p. 698–705, 2019. Disponível em: https://doi.org/10.1109/IS.2018.8710521

TSOUKALAS, Dimitrios *et al.* Technical debt forecasting: An empirical study on open-source repositories. **Journal of Systems and Software**, [*s. l.*], v. 170, p. 110777, 2020. Disponível em: https://doi.org/10.1016/j.jss.2020.110777

TUFANO, M. *et al.* When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). **IEEE Transactions on Software Engineering**, [*s. l.*], v. 43, n. 11, p. 1063–1088, 2017. Disponível em: https://doi.org/10.1109/TSE.2017.2653105

TUFANO, Michele *et al.* When and why your code starts to smell bad. **Proceedings - International Conference on Software Engineering**, [*s. l.*], v. 1, p. 403–414, 2015. Disponível em: https://doi.org/10.1109/ICSE.2015.59

TUKEY, John W. **Eploratory Data Analysis**. [*S. l.: s. n.*], 1977.

VON ROSING, Mark *et al.* Business process model and notation-BPMN. **The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM**, [*s. l.*], v. 1, n. January, p. 429–453, 2014. Disponível em: https://doi.org/10.1016/B978-0-12-799959-3.00021-5

WAKE, William C. **Refactoring Workbook,**. [*S. l.: s. n.*], 2003.

WANG, H. *et al.* On the use of time series and search based software engineering for refactoring recommendation. *In*: , 2015. **7th International ACM Conference on Management of Computational and CollEctive Intelligence in Digital EcoSystems, MEDES 2015**. [*S. l.: s. n.*], 2015. p. 35–42. Disponível em: https://doi.org/10.1145/2857218.2857224

WASS, John A. Weka machine learning workbench. **Scientific Computing**, [*s. l.*], v. 24, n. 3, p. 357–361, 2007.

WITTEN, Ian H; FRANK, Eibe; HALL, Mark A. **Data Mining: Practical Machine Learning Tools and Techniques, Third Edition (The Morgan Kaufmann Series in Data Management Systems)**. [*S. l.: s. n.*], 2011. *E-book*.

WOHLIN, Claes *et al.* Experimentation in software engineering. **Experimentation in Software Engineering**, [*s. l.*], v. 9783642290, n. 7, p. 1–236, 2012. Disponível em: https://doi.org/10.1007/978-3-642-29044-2

YAMASHITA, Aiko; MOONEN, Leon. Do developers care about code smells? An exploratory survey. **Proceedings - Working Conference on Reverse Engineering, WCRE**, [*s. l.*], p. 242–251, 2013. Disponível em: https://doi.org/10.1109/WCRE.2013.6671299

APÊNDICE A - MODELO DE QUALIDADE DO METODO SQALE

Modelo de qualidade

Utilizado para organizar os requisitos não funcionais que se relacionam com a qualidade do código. Está organizado em três níveis hierárquicos: o primeiro nível é composto por características, o segundo por subcaracterísticas e o terceiro nível é composto de requisitos relacionados aos atributos internos do código-fonte, como mostrado no Quadro 6-1. Esses requisitos dependem do contexto e da linguagem de programação utilizada. Por exemplo, a característica de Manutenibilidade, na subcaracterísticas de Compreensibilidade, no requisito “Não use operadores ternários”, pode ser entendido como um *code smell*. Pois pode ser entendido que este requisito é um problema relativo à compreensão do código fonte. Os requisitos podem atender uma necessidade de teste, de segurança e até mesmo de dívida técnica de código fonte.

Quadro 6-1 - Exemplo de lista de mostra de requisitos e seu mapeamento no modelo SQALE.

CARACTERÍSTICA	SUBCARACTERÍSTICAS	REQUISITO
Manutenibilidade	Legibilidade	Não há bloco de instrução comentado
Manutenibilidade	Compreensibilidade	Não use operadores ternários.
Mutabilidade	Mudança relacionada à arquitetura	Não há dependência cíclica entre os pacotes
Confiabilidade	Confiabilidade no tratamento de exceções	O tratamento de exceções não deve capturar NPE
Confiabilidade	Confiabilidade relacionada à instrução	O código deve substituir tanto igual quanto *hashcode*
Confiabilidade	Confiabilidade relacionada aos dados	Não há comparação entre ponto flutuante
Confiabilidade	Confiabilidade relacionada à cobertura	Todos os arquivos têm teste de unidade com pelo menos 70% de cobertura de código
Testabilidade	Testabilidade de nível de teste de unidade	Não há método com complexidade ciclomática acima de 12
Testabilidade	Testabilidade de nível de teste de unidade	Não há partes clonadas de 100 tokens ou mais

Fonte: Adaptado de Letouzey (2012).

O modelo de qualidade SQALE possui nove habilidades, que são entendidas como características e estão apresentadas no Quadro 6-2, onde também é possível observar a relação com seu respectivo indicador. Letouzey (2012) cita que as

características abordadas na criação do SQALE são resultantes da norma ISO/IEC 9126 e já na última versão do SQALE alterada para versão da ISO/IEC 25010. As características ou habilidades do SQALE foram selecionadas porque dependem das propriedades internas do código. Essas propriedades podem impactar diretamente as atividades típicas do ciclo de vida de desenvolvimento de software.

Quadro 6-2: Característica SQALE x Indicador SQALE. Fonte: adaptado de Letouzey (2012).

Característica	Nome do Indicador e sigla no SQALE
Reusabilidade	Índice de reutilização: SRuI
Portabilidade	Índice de portabilidade: SPI
Manutenibilidade	Índice de manutenibilidade: SMI
Segurança	Índice de segurança: SSI
Eficiência	Índice de eficiência: SEI
Mutabilidade	Índice de mutabilidade: SCI
Usabilidade	Índice de usabilidade: SUI
Confiabilidade	Índice de confiabilidade: SRI
Testabilidade	Índice de testabilidade: STI

O *code smell* está diretamente associado às habilidades de manutenibilidade. É importante observar que tanto a ISO/IEC 9126 quanto a ISO/IEC 25010 não são específicas sobre *code smells* e DT. Ambas as normas tratam a qualidade do produto de *software* sob uma perspectiva mais ampla.

Modelo de análise de regras

Utilizado para normalizar as medidas e os controles relativos ao código e, por outro lado, as regras para agregar os valores normalizados. O método SQALE normaliza os resultados das ferramentas de análise do código-fonte, transformando-os em custos de remediação. Também define as regras para agregar os custos de remediação.

Índices

Os índices SQALE representam custos que podem ser calculados em unidade de trabalho, unidade de tempo ou unidade monetária. Em todos os casos, os valores dos índices estão em uma escala do tipo razão. Para qualquer elemento da hierarquia dos artefatos do código-fonte, o custo de remediação relacionado a uma determinada característica pode ser estimado adicionando todos os custos de remediação vinculados aos requisitos da característica.

O método SQALE define um índice global para o custo de remediação relativo a todas as características do modelo de qualidade. Esta medida é chamada de: Índice de Qualidade SQALE – SQI, que para Letouzey (2016) é o principal indicador do SQALE. O indicador SQI é a soma de outros indicadores, conforme especificado na Quadro 6-3, onde cada um dos indicadores está associado a uma das características apontadas.

Quadro 6-3: Características x indicadores SQALE. Fonte: o Autor.

Característica	Nome e Sigla do indicador no SQALE
Reusabilidade	Índice de reutilização: SRuI
Portabildiade	Índice de portabilidade: SPI
Manutenibilidade	Índice de manutenibilidade: SMI
Segurança	Índice de segurança: SSI
Eficiência	Índice de eficiência: SEI
Mutabilidade	Índice de mutabilidade: SCI
Confiabilidade	Índice de confiabilidade: SRI
Testabilidade	Índice de testabilidade: STI

Cada violação de regra tem um esforço de remediação predefinido em sua característica, uma regra pode ser um *code smell*. O SQALE calcula um índice para cada característica que é a soma de todos os esforços de remediação das suas violações da regra.

O índice SQI representa o esforço de remediação que seria necessário para corrigir os problemas detectados no componente. Uma vez que o índice de remediação representa um esforço de trabalho, a consolidação dos índices é uma simples adição de informação uniforme.

Indicadores sintetizados

O SQALE define indicadores sintetizados, com o objetivo de fornecer a representação da DT de forma visual.

Indicador de avaliação

As cores seguem uma classificação de 5 valores (A, B, C, D, E), que consiste em produzir uma medida derivada em uma escala ordinal. A classificação para um determinado artefato resulta de uma comparação do custo estimado de remediação do artefato com uma estimativa do custo de desenvolvimento deste artefato. Para isso, define-se antecipadamente uma grade de avaliação, transmitindo o grau de

aceitabilidade do custo de remediação em relação ao custo de desenvolvimento, conforme apresentado na Figura 2-9.

Avaliação	Até	Cor
A	1%	
B	2%	
C	4%	
D	8%	
E	>8%	

Figura 6-1. Grade de avaliação. Fonte: Letouzey (2012).

Por exemplo, optou-se em expressar o custo em horas e o tamanho na KSLOC[36], se o custo médio de 1 KSLOC foi estimado em 100 horas. A grade da Rating é usada da seguinte forma: um artefato que tem um custo de desenvolvimento estimado em 500 horas (porque seu tamanho é 5 KSLOC) e um Índice de Qualidade SQALE (que é o principal da DT) de 15 horas, terá uma relação de 15 horas / 500 horas = 3%. Sua classificação SQALE será "C".

Indicador radar SQALE

Letouzey (2012) apresenta o SQALE Kiviat (gráfico de radar), ilustrado na Figura 6-2, no qual quanto mais próximo ao centro do gráfico (Índice A em verde), significa que o código analisado para aquela característica está com maior nível de qualidade.

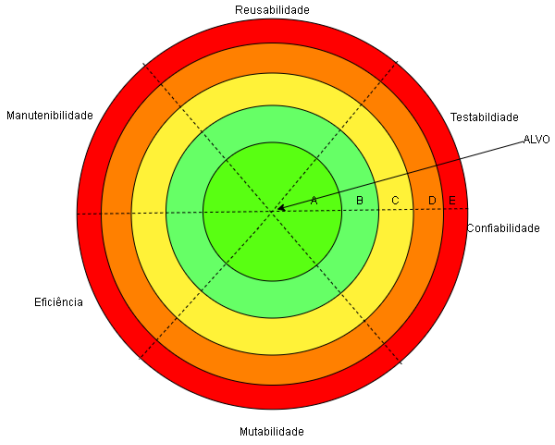


Figura 6-2. SQALE Kiviat. Fonte: Adaptado de Letouzey (2012)**.**

[36] KSLOC - (Kilo Source Lines of Code) - medida de milhares de linhas de código.

Importante citar o método SQALE nesta pesquisa pois alguns softwares de análise estática consideram a aplicação deste modelo, conforme será tratado no item 2.6

APÊNDICE B - DEFINIÇÃO DE EXPERIMENTO

O objetivo deste apêndice é apresentar as fases que compõem o experimento utilizado neste trabalho e as estratégias adotadas para atingir o objetivo geral, conforme abordado no capítulo 3.

1. Fase de escopo

Nesta fase é determinada a fundamentação do experimento e o motivo a ser conduzido. O escopo do experimento é realizado definindo seus objetivos. O modelo definido na abordagem GQM de Basili *et al.* (1994), citado por Wohlin *et al.* (2012) para elaboração de um objetivo, é baseado nas seguintes definições:

> **Objeto de estudo** é a entidade que é estudada no experimento, que pode ser produtos, processos, recursos, modelos, métricas ou teorias.
>
> **Propósito** é a definição da intenção do experimento.
>
> **Foco da Qualidade** descreve o efeito que está se tentando buscar com o experimento, como confiabilidade, eficácia, custo etc.
>
> **Perspectiva** indica o ponto de vista a partir do qual os resultados do experimento devem ser interpretados.
>
> **Contexto** define quais são os indivíduos (pessoas ou tecnologias) que estão envolvidos e quais objetos (artefatos de *software*) serão utilizados no experimento.

Para cada meta devem ser definidas uma ou várias questões. Para cada questão, uma ou várias métricas.

2. Fase de planejamento

Conforme definição de Wohlin *et al.* (2012), a fase de Escopo determina a fundamentação para o experimento, enquanto a fase de Planejamento prepara a

forma como o experimento será conduzido. A fase de Planejamento é dividida nos seguintes passos: seleção do contexto, formulação da hipótese, seleção de variáveis, seleção de indivíduos, projeto do experimento, instrumentação e avaliação da validade.

a. Seleção de contexto

Define-se o ambiente no qual o experimento será executado.

- In vitro: experimento executado sob condições controladas;
- In vivo: experimento executado sob condições reais;
- Alunos ou profissionais: caracteriza os participantes do experimento;
- Problema de sala de aula ou problema real: define o experimento que está sendo estudado;
- Específico ou geral: caracteriza se os resultados do experimento são válidos para um contexto específico ou para o domínio geral da Engenharia de *Software*.

b. Formulação de hipótese

Para Bruce (2017), os testes de hipótese ou significância permitem gerar a base para realizar a análise estatística do experimento, utilizando os dados coletados após sua execução. A hipótese nula é a que o experimentador deseja que seja rejeitada, e uma hipótese alternativa é a que deve ser contrária ao da hipótese nula. As hipóteses são duas declarações mutuamente exclusivas sobre uma população. Um teste de hipótese usa dados amostrais para determinar se deve rejeitar a hipótese nula.

Hipótese nula (H0): A hipótese nula afirma que um parâmetro da população (por exemplo, a média) é igual a um valor hipotético. A hipótese nula pode ser considerada uma alegação inicial baseada em análises anteriores ou conhecimentos especializados.

Hipótese Alternativa (H1): A hipótese alternativa afirma que um parâmetro da população é menor, maior ou diferente do valor hipotético na hipótese nula. A hipótese

alternativa é aquela que acredita que pode ser verdadeira ou espera provar ser verdadeira. A hipótese alternativa tanto pode ser unilateral como bilateral.

- Hipótese Alternativa Bilateral: Utilizada para determinar se o parâmetro da população é maior ou menor do que o valor hipótese;

- Hipótese Alternativa Unilateral: Utilizada para determinar se o parâmetro da população difere do valor da hipótese em uma direção específica. É possível especificar a direção a ser maior ou menor do que o valor hipotético.

Como o teste é baseado em probabilidades, sempre há uma possibilidade de se chegar a uma conclusão errada. Quando é realizado um teste de hipóteses, dois tipos de erros são possíveis: tipo I e tipo II, descrito na Quadro 9-1.

- Erro tipo I: Quando a hipótese nula é verdadeira e você a rejeita;

- Erro tipo II: Quando a hipótese nula é falsa e você não a rejeita.

Quadro 6-4. Tipo de Erro de Hipótese. Fonte: o Autor.

Teste de hipótese	**H0 é verdadeiro**	**H0 é falso**
Aceitar H0	Decisão Correta	**Erro tipo II** - deixa de rejeitar H0 quando ela é falsa
Rejeitar H0	**Erro tipo I** - rejeitando H0 quando ele é verdadeiro	Decisão Correta

Testes estatísticos devem ser escolhidos de acordo com as hipóteses e serão utilizados para avaliar o resultado de um experimento.

c. Seleção de variáveis

Neste passo serão definidas as variáveis dependentes e as independentes.

Variável Independente: são variáveis que influenciam, afetam ou determinam outras variáveis, sendo fator determinante, condição ou causa para determinado resultado, efeito ou consequência. Também é o fator manipulado pelo pesquisador, com o objetivo de determinar a relação do fator com o fenômeno observado, para ver que influência exerce sobre um possível resultado.

Variável dependente: são valores (fenômenos ou fatos) a serem descobertos ou explicados, em virtude de serem influenciados, determinados ou afetados pela

variável independente. Também é o fator que aparece, desaparece, aumenta ou diminui, à medida que o pesquisador modifica a variável independente. As variáveis dependentes são derivadas diretamente da hipótese do experimento.

d. Seleção de indivíduos/objetos

Nesta etapa é definida a amostra da população do experimento, podendo ser probabilística, quando são escolhidos os indivíduos/objetos, ou não probabilística, quando a escolha de indivíduos/objetos é aleatória. O tamanho da amostra também define a precisão dos resultados de um experimento. Quanto maior a variabilidade da população, uma amostra maior pode ser necessária.

e. Projeto do experimento

O projeto de experimento determina como o experimento será conduzido. Nesse passo, para obter conclusões significativas do experimento, aplica-se métodos de análise estatística nos dados coletados para interpretar os resultados. Também devem ser determinados quantos testes o experimento deve ter.

f. Instrumentação

Este passo tem como objetivo definir quais instrumentos serão escolhidos ou desenvolvidos para um experimento ser executado. Instrumentos para realizar a coleta de dados durante a execução do experimento devem ser definidos, como formulários para preenchimento ou ferramentas para captura das informações.

g. Avaliação da validade

Este passo tem o objetivo de determinar a validade dos resultados obtidos. Primeiro se observa a validade para a população de qual a amostra foi selecionada para o experimento, e se também pode ser aplicada a uma população maior. Ameaças à validade de um experimento são definidas em quatro tipos:

i. Validade da conclusão: está associada à relação entre o tratamento e o resultado. Com o objetivo de ter certeza de que existe uma relação estatística, com um determinado significado;

ii. Validade interna: quando observada uma relação entre o tratamento e o resultado, é necessário ter certeza de que se trata de uma relação

causal, e que o resultado não é um fator do qual não se tem controle ou não é medido;

iii. Validade externa: trata da validade da generalização dos resultados do experimento para a prática na indústria;

iv. Validade da construção: tem a preocupação em generalizar o resultado da experiência ao conceito ou teoria por trás da experiência. Algumas ameaças estão relacionadas ao desenho do experimento, outras ameaças a fatores sociais.

3. Fase de execução

Esta fase é operacional, os passos para esta fase do experimento são a preparação, a execução e a validação de dados.

a. Preparação

Neste passo os fatores necessários para a realização do experimento são escolhidos, consistindo basicamente em selecionar os indivíduos que participarão do experimento, assim como separar os materiais necessários para a execução.

b. Execução

Este passo segue o que foi definido na fase de planejamento. A coleta dos dados executados no experimento também é realizada neste momento.

c. Validação de dados

Este passo tem o objetivo de verificar se o experimento foi realizado corretamente e que os tratamentos foram aplicados na forma e ordem planejada, bem como se os dados foram coletados de corretamente.

4. Fase de análise e interpretação

Após os dados coletados na fase anterior é necessário realizar a análise e interpretação para gerar informações de interesse. A análise quantitativa é realizada nos três passos a seguir, estatística descritiva, redução de conjuntos e teste de hipótese.

a. Estatística descritiva

Reduzir o conjunto de dados e testar a hipótese definida para o experimento, com o objetivo de realizar o processamento numérico, utilizando métricas em um conjunto de dados para obter descrições e gráficos. Com isso é possível analisar como os dados estão distribuídos e identificar dados anormais e pontos fora da curva. Redução de conjunto

Tem o objetivo de ser aplicado caso os dados trabalhados pelos métodos estatísticos não representarem o que deveriam, e assim determinando que as conclusões obtidas foram levadas ao erro.

b. Teste de hipótese

O objetivo é validar a possibilidade de rejeitar uma hipótese nula, baseado em alguns dados estatísticos obtidos nos passos anteriores. O experimentador seleciona uma amostra dos dados e tenta rejeitar que aquelas propriedades sejam verdadeiras. Quando encerradas essas validações, descreve-se as conclusões sobre a influência, ou não, das variáveis independentes sobre as dependentes. Com isso é possível provar se há uma diferença estatística significante entre os tratamentos aplicados no experimento.

5. Fase apresentação e empacotamento

Após a conclusão de análise e interpretação é necessário apresentar os resultados e, empacotar todo o material produzido para futura replicação por outras pessoas. Códigos-fontes, procedimentos de banco de dados, artigos ou relatórios técnicos podem ser produzidos para a comunidade, ou os dados dos experimentos podem ser armazenados em uma base de dados.

Empacotar experimentos com finalidade de replicação permite que outros pesquisadores possam verificar as propriedades com as quais o experimento foi realizado, a fim de realizar o experimento novamente e validá-lo.

6. Considerações sobre o apêndice

Este apêndice apresentou a estruturação do experimento, suas etapas e proposições. A estratégia de pesquisa também foi detalhada e apresentada.

APÊNDICE C - DOC1 – FORMULÁRIO DA INFRAESTRUTURA

NÚMERO DA SOLICITAÇÃO:
SOLICITANTE:
DATA SOLICITAÇÃO:
EXECUTOR:
SERVIDOR DE BANCO DE DADOS
ENDEREÇO IP OU DNS:
USUÁRIO/SENHA:
SCHEMA/DATABASE:
SERVIDOR SONARQUBE
ENDEREÇO IP OU DNS:
USUÁRIO/SENHA:
SERVIDOR JUPYTER
ENDEREÇO IP OU DNS:
USUÁRIO/SENHA:

APÊNDICE D -*SCRIPT* DA CRIAÇÃO DO *STARSCHEMA* – DDL SQL

Script disponível em:

https://github.com/manoelvsneto/tdmining/

APÊNDICE E - DOC2 - DOCUMENTO DE VARIÁVEIS DE AMBIENTE

AS VARIÁVEIS DE AMBIENTE DEVEM SER CONFIGURADAS NO ARQUIVO CONFIG.INI

[configuration]

database_connection = string de conexão com o banco de dados RESULTADO
database_connection_sonar = string de conexão com o banco de dados SONAR
database_result = nome do banco de dados RESULTADO
database_result_sonar = nome do banco de dados RESULTADO
schema_result = nome do banco de dados RESULTADO
schema_result_sonar = nome do banco de dados RESULTADO
path_base_sonarscanner = Diretório onde está configurado o SONARSCANNER
path_git = Diretório para onde foi realizado o git clone do projeto
project_name = Nome do Projeto

QUESTION3_METRIC1_SUPORT= PERCENTUAL de suporte para considerar na questão 3.

QUESTION4_METRIC1_WINDOW= Quantidade de períodos para análise de previsão na questão 4.

QUESTION4_METRIC1_MIN_PERIODS= Quantidade de períodos considerados para previsão na questão 4.

QUESTION5_METRIC1_SUBSSETS= Quantidade de *subsets* (partições de dados) para treino do modelo da questão 5.

EXEMPLO

```
[configuration]
database_connection = Driver={SQL
Server};Server=localhost;Database=Resultado;uid=sa;pwd=1234567
database_connection_sonar = f"Driver={{SQL
Server}};Server=localhost;Database=sonar5;uid=sa;pwd=1234567"
database_result = 'Resultado'
database_result_sonar = 'sonar5'
schema_result = 'Resultado'
schema_result_sonar = 'sonar5'
path_base_sonarscanner = 'C:\\processamento_sonar\\git-1\\sonar-scanner-wordpress'
path_git = 'C:\\processamento_sonar\\git-1\\wordpress'
project_name = 'wordpress'
QUESTION3_METRIC1_SUPORT= 0.16
QUESTION4_METRIC1_WINDOW=12
QUESTION4_METRIC1_MIN_PERIODS=6
QUESTION5_METRIC1_SUBSSETS=4
```

APÊNDICE F - DOC3 – FORMULÁRIO RESULTADO DA EXECUÇÃO DOS *SCRIPTS* JUPYTER

Nome executor:	
Data Execução:	
Versão:	
Resultado:	
3.1 – Teste de Normalidade	Informar resultado da execução do *script*
3.2 – Teste de Hipótese	Informar resultado da execução do *script*
3.3 – Questão 1 – Métrica 1	Informar resultado da execução do *script*
3.4 – Questão 1 – Métrica 2	Informar resultado da execução do *script*
3.5 – Questão 2 – Métrica 1	Informar resultado da execução do *script*
3.6 – Questão 3 – Métrica 1	Informar resultado da execução do *script*
3.7 – Questão 4 – Métrica 1	Informar resultado da execução do *script*
3.8 – Questão 5 – Métrica 1	Informar resultado da execução do *script*

APÊNDICE G - DOC4 - *CHECKLIST* DE RESTRIÇÕES

DATA	DESENVOLVEDOR*	SEVERIDADE	TIPO DA DÍVIDA TECNICA	PROJETO	VERSÃO	REGRA	AÇÃO
2019-08-10							
		5 - INFO					
						5649 – (S2043) Superglobals should not be accessed directly	
	63						
	168						
	79						
	3						
	1						

*Os nomes dos desenvolvedores foram substituídos pelos seus identificadores.

APÊNDICE H - RELAÇÃO DOS SCRIPTS PYTHON COM AÇÕES DE BANCO DE DADOS

SCRIPT	OBJETIVO	APRESENTAÇÃO / ESTATISTICA	CONSULTA SQL RELACIONADO ARQUIVO QUERY.INI
insert.py	Recuperar os dados de commit de um repositório de código fonte e persistir os dados em banco de dados.	Não se aplica	insere dados na tabela COMMITS
executer.py	Processar individualmente os commits por meio de análise estática para identificar code smell em relação ao commit anterior.	Não se aplica	insere/altera/seleciona dados da tabela COMMITS, recupera dados das tabelas SNAPSHOTS, ISSUES do SonarQube.
teste-normalidade.py	Realizar teste de normalidade com os dados de code smell.	Histograma, Shapiro-Wilk, Kolmogorov-Smirnov	normalidade
teste-hipotese.py	Realizar teste de hipótese com os dados de code smell.	Gráfico de dispersão, SpearmanR	hipótese
question-1-metric-1.ipynb	Responder à questão: Qual o desenvolvedor mais cria code smells?	Gráfico de pizza, Tabela , Sumarização	query_1_1
question-1-metric-2.ipynb	Responder à questão: Qual o desenvolvedor mais remove code smells?	Gráfico de pizza, Tabela , Sumarização	query_1_2
question-2-metric-1.ipynb	Responder à questão: Qual o code smell que foi mais criado pelos desenvolvedores?	Tabela, Sumarização.	query_2_1,query_2_1_B,query_2_1_C ,query_2_1_D
question-3-metric-1.ipynb	Responder à questão: Quais os code smells que ocorrem em comum pelo desenvolvedor?	Regra de associação (APRIORI). Itens Frequentes	query_3_1
question-4-metric-1.ipynb	Responder à questão: Qual a tendência de code smell criado?	Gráfico de linha (Média Móvel (SMA)), Tabela	query_4_1, query_4_1_B
question-5-metric-1.ipynb	Responder à questão: Qual a previsão de criação de code smell?	Tabela. Histograma. Teste de estacionariedade (ADF e KPSS). Previsão para series temporais (ARIMA)	query_5_1
condition-ignore.ipynb	Ignorar registros da tabela fato. Condições informadas pelo ESPECIALISTA no DOC4 - CHECKLIST DE RESTRIÇÕES	Não se aplica.	ignore_update, ignore_select
reset-database. py	Limpar o banco de dados para uma nova carga de dados	Não se aplica	revert-database