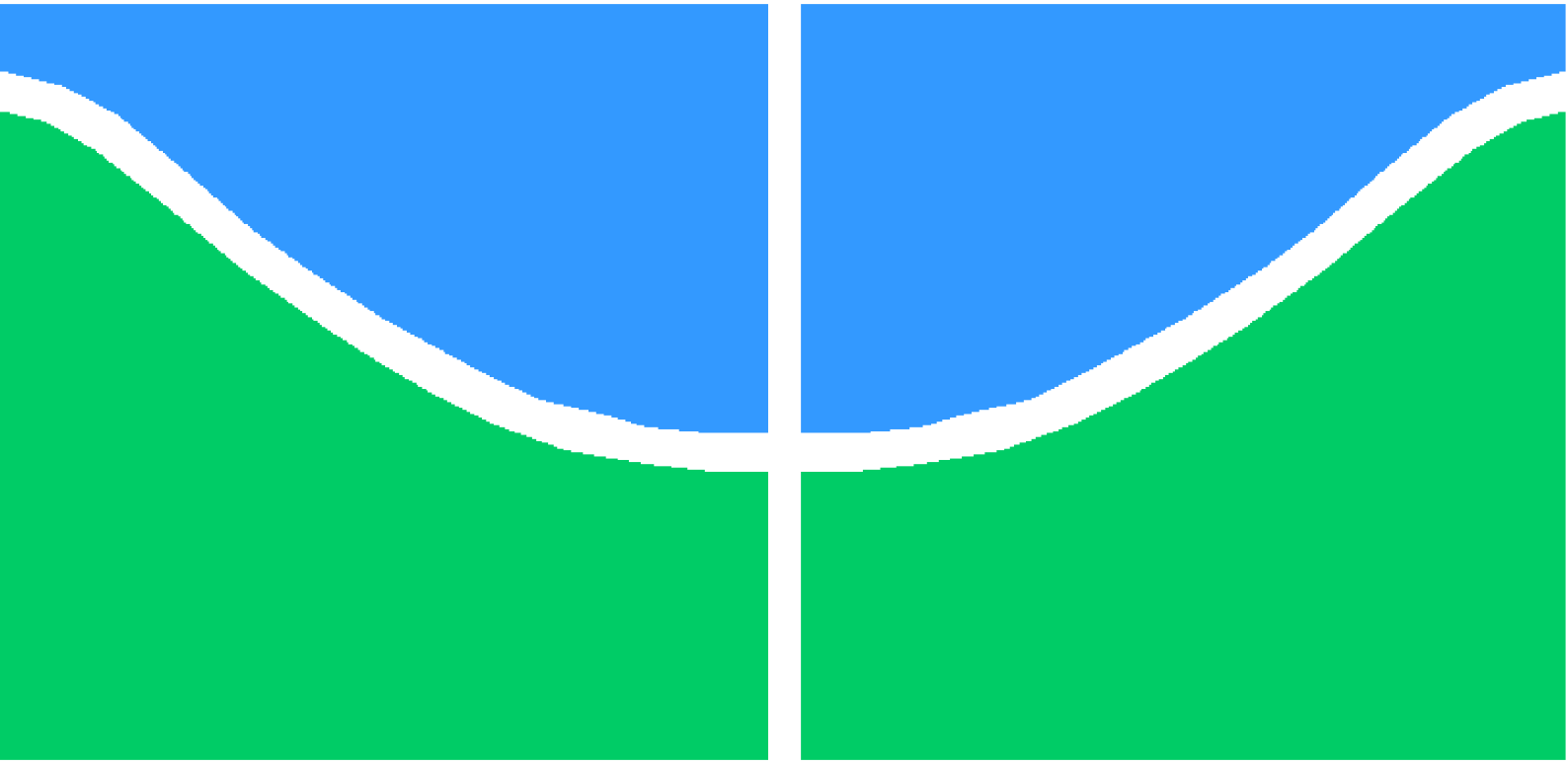

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Uma proposta de ferramenta de análise estática para avaliação de qualidade de logging para Java

Autor: Francisco Wallacy Coutinho Braz

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2022

Francisco Wallacy Coutinho Braz

Uma proposta de ferramenta de análise estática para avaliação de qualidade de logging para Java

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2022

Francisco Wallacy Coutinho Braz
Uma proposta de ferramenta de análise estática para avaliação de qualidade de logging para Java/ Francisco Wallacy Coutinho Braz. – Brasília, DF, 2022-
56 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2022.

1. logging. 2. software development. I. Prof. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Uma proposta de ferramenta de análise estática para avaliação de qualidade de logging para Java

CDU

Francisco Wallacy Coutinho Braz

Uma proposta de ferramenta de análise estática para avaliação de qualidade de logging para Java

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, :

Prof. Dr. Renato Coral Sampaio
Orientador

Prof. Dr. Fernando William Cruz
Convidado 1

Prof. Dr. Giovanni Almeida Santos
Convidado 2

Brasília, DF
2022

Este trabalho é dedicado à minha querida mãe,
cujo apoio e carinho sempre estiveram presentes na minha vida,
me motivando sempre a seguir em frente.

Agradecimentos

Aos professores, pelos ensinamentos que foram fundamentais para minha formação como pessoa e profissional.

Ao meu orientador, prof. Renato Coral Sampaio, por ter me acompanhando durante a jornada deste trabalho, sempre me direcionando no caminho certo.

Aos meus amigos, pelo suporte, e por terem compartilhado comigo tantos momentos memoráveis.

Á minha companheira, Amanda Von-Grapp, por sempre me apoiar e pelo empenho em trazer felicidade para o meu dia, além de ter agido como revisora deste trabalho.

Resumo

Logging é uma prática de software executada com o objetivo de registrar informações pertinentes sobre o *runtime* de aplicações. É obtida por meio da inserção de instruções específicas no código-fonte. Apesar de difundida, a prática raramente segue diretrizes de qualidade, que, consequentemente, se torna dependente da experiência e conhecimento prévio dos desenvolvedores praticantes. Com efeito, o mau uso da prática pode resultar em um registro de *logs* de baixo valor, que pode até mesmo ofuscar os objetivos a priori esperados. Nesse sentido, foram definidas questões de pesquisa com a finalidade de entender a ocorrência desses problemas em aplicações escritas na linguagem de programação Java. Visando responder a essas perguntas, um estudo foi realizado, e, por meio dele, alguns problemas de *logging* foram identificados. Um subconjunto desses problemas, detectáveis via análise estática, foi delimitado e partindo deles, uma ferramenta de detecção foi desenvolvida. Quatro grandes softwares escritos em Java foram selecionados e tiveram seus códigos-fontes analisados pela ferramenta, com o objetivo de validar seu uso. Os resultados mostraram que a ferramenta foi capaz de detectar os problemas elencados, apesar de haver pontos de melhoria.

Palavras-chaves: *logging*. desenvolvimento de software. qualidade de software.

Abstract

Logging is a software practice performed with the objective of recording information about the runtime of applications. It is obtained through the insertions of specific instructions in the source code. Although widespread, it rarely follows guidelines of quality, which, consequently, becomes dependent on experience and knowledge from practicing developers. Indeed, the misuse of the practice can result into a low-value log record, which can even obfuscate the goals at first expected. In this sense, research questions were defined in order to understand the occurrence of these problems in applications written in the programming language Java. In order to answer these questions, a study was carried out and, through from it, some logging problems were identified. A subset of these problems, detectable via static analysis, was delimited and based on them, a tool for detection was developed. Four great software programs written in Java were selected and had their source code analyzed by the tool, with the aim of validating its use. The results showed that the tool was able to detect the listed problems, although there are points of improvement.

Key-words: logging code. software development. software quality.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
AST	Abstract Syntax Tree
IDE	Integrated Development Environment
JCL	Apache Commons Logging
JVM	Java Virtual Machine
LI	Logging Invocation
SLF4J	Simple Logging Java Facace

Sumário

1 Introdução

Os softwares, e seus mais diversos usos, permeiam nossa atual sociedade e podem ser encontrados nos mais diversos contextos. E, diferente de anos atrás, não estão restritos apenas a computadores pessoais. Podem ser encontrados nos mais diversos tipos de dispositivos, variando de nossos relógios inteligentes (*smartwatches*) aos computadores de bordo de aviões. Um dos setores que mais se beneficiam de seu uso é o comercial. Empresas utilizam softwares com o objetivo de melhorar seus processos internos, e, em certos casos, tê-los como o principal produto.

Falhas que afetam a disponibilidade e a confiabilidade desses softwares são sempre indesejáveis, uma vez que na maioria das vezes traduzem-se em prejuízos. Como forma de buscar a diminuição dessas falhas e, consequentemente, aumentar a confiabilidade de suas soluções, empresas inserem em seus processos de desenvolvimento práticas voltadas à garantia de qualidade. Frequentemente adotada como parte principal desses processos encontra-se a prática de verificação e validação. Apesar disso, nem sempre é possível antever todos os tipos de problemas que podem vir a ocorrer com o software em uso e, como consequência, *bugs* acabam sendo descobertos. Quando a gravidade desses problemas é alta, tal como a interrupção de disponibilidade, soluções imediatas são necessárias.

Muitas vezes, a única ferramenta de diagnóstico disponível nessas situações é o registro de *logging*. Por meio desses *logs*, é possível relacionar eventos de falhas à locais específicos do código-fonte da aplicação, e, partindo deles, em ordem reversa, identificar as possíveis causas, processo chamado de *debugging* (SOMMERVILLE, 2011).

Apesar de seu uso mais conhecido ser o diagnóstico de falhas, praticantes estão obtendo valor ao consumir o conteúdo dos registros sob a ótica de outros objetivos, tais como conformidade legal, tomada estratégica de decisões, segurança e monitoramento (GHOLAMIAN; WARD, 2021).

Esses *logs* são produzidos a partir do código-fonte da aplicação, por meio de instruções que objetivam a captura de informações sobre o tempo de execução. Ainda que a prática de *logging* seja difundida, depende, em grande parte, da experiência e arbitrariedade do desenvolvedor, dado que, na indústria, não existem ainda ferramentas de suporte eficazes, apesar de haver grande esforço nesse sentido por parte dos pesquisadores (GHOLAMIAN; WARD, 2021) da área. Como resultado, acabam surgindo na prática, padrões que podem ser considerados contrários aos objetivos de *logging* desejados, diminuindo sua qualidade.

Visto que a prática de *logging* traz diversas vantagens, dentre elas, um melhor suporte ao diagnóstico de falhas, bem como, oportunidades de melhoria na qualidade da

aplicação, e, ainda que não exista entre os praticantes uma convenção sólida, é de grande necessidade a aparição de ferramentas que ofereçam suporte a esta prática, objetivando auxiliar o desenvolvedor a produzir *logs* de maior qualidade.

Neste sentido, será proposta uma ferramenta que seja capaz de identificar problemas comuns na infraestrutura de *logging* de softwares escritos na linguagem Java. Para tanto, será utilizada a técnica de análise estática, abordagem muito comum na indústria de software para garantia de qualidade.

1.1 Problematização

A prática de *logging* permite obter informações sobre o funcionamento do *runtime* de uma dada aplicação. No entanto, não tendo nem guias nem diretrizes para se basear, os praticantes acabam usando a própria arbitrariedade para tomar decisões, tais como, onde realizar o *logging*, o que registrar, e como manter o código *logging*. O primeiro, diz respeito ao local do código-fonte onde se deseja inserir a LI. O segundo, trata da escolha que se deve fazer relativo ao conteúdo propriamente dito do registro. E o último, lida com as questões que envolvem a manutenção da infraestrutura de *logging*.

Tais decisões, devem resultar em registros de *logs* que estejam bem alinhados com seus respectivos objetivos. A inconsistência nesses registros pode causar uma série de problemas, desde o custo da manutenção adicional necessário para corrigí-los, bem como o baixo valor obtido por aqueles que os consomem, que o fazem segundo algum objetivo.

Como exemplo, um dos usos mais comuns para os registros de logs é o *debugging*, que pode ser entendido, assim como descrito por Sommerville (2011), como o processo de encontrar a causa raiz de defeitos de softwares. Essa é uma forma bem aceita pelo mercado, principalmente em softwares comerciais, onde não é possível ter acesso ao ambiente do cliente, além de não ferir princípios da privacidade (YUAN; PARK; ZHOU, 2012).

Para que os registros de *logging* possam se mostrar úteis para os desenvolvedores, quando estes precisam resolver falhas de softwares, devem seguir a certos critérios de aceitabilidade. De forma geral, quando uma falha é detectada, a infraestrutura de *logging* deve:

- Registrar a ocorrência de falha;
- Registrar qual o ponto da aplicação que ocorreu a falha;
- Registrar todas as informações relevantes ao evento da falha;
- Registrar o caminho percorrido até chegar no ponto onde ocorreu a falha.

Tendo essas informações, o desenvolvedor pode, em seu próprio ambiente, tentar reproduzir a falha e, uma vez identificada a causa raiz, realizar as devidas correções no código.

Qualquer aspecto da prática de *logging*, quando esta tem o objetivo de auxiliar na resolução de falhas de software, que se distancie desses critérios, pode ser caracterizada como um problema. Da mesma forma, outros problemas, definidos para outros objetivos de *logging*, podem ser igualmente definidos.

1.2 Objetivos

1.2.1 Objetivo geral

O presente trabalho tem como objetivo desenvolver uma ferramenta de análise estática, que seja capaz de encontrar problemas na infraestrutura de *logs* de aplicações escritas na linguagem de programação Java.

1.2.2 Objetivos específicos

Para tanto, serão seguidos os seguintes objetivos:

- Definir os requisitos da ferramenta;
- Realizar *design* de arquitetura;
- Implementar a solução;
- Testar a solução implementada.

1.3 Estrutura do trabalho

O restante deste documento está estruturado da seguinte forma: no Capítulo 2, serão apresentados conceitos-chave, bem como uma revisão bibliográfica, buscando caracterizar como o tema é tratado na literatura científica; no Capítulo 3, será discorrido sobre a metodologia, descrevendo os procedimentos utilizados, a fim de alcançar o objetivo deste trabalho; no Capítulo 4, a proposta da solução é apresentada, contendo o resultado da pesquisa exploratória sobre *logging*, bem como o *design* e a arquitetura da ferramenta; e por fim, no Capítulo 5, as considerações finais são feitas, apresentando os resultados obtidos e propostas para trabalhos futuros.

2 Referencial Teórico

2.1 *Logging*

Em um sentindo mais abrangente, *logging* pode ser entendido como uma prática que tem por objetivo resgatar informações importantes durante a execução de um software, por meio da inserção de instruções específicas em seu código-fonte (FU et al., 2014). Geralmente, essas instruções, também chamadas de *Logging invocations (LI)*, consistem de uma mensagem que pode ou não ter conteúdo dinâmico, além de um nível de verbosidade, que classifica a importância do evento que está sendo registrado. A Figura 1 apresenta um exemplo de uma LI. O *logging* apresentado se encontra sob o nível de severidade *debug*, e contém uma mensagem que possui conteúdo estático e dinâmico, sendo estes representados pela string[1] "Kafka brokers"e pela variável[2] "brokerList"respectivamente.

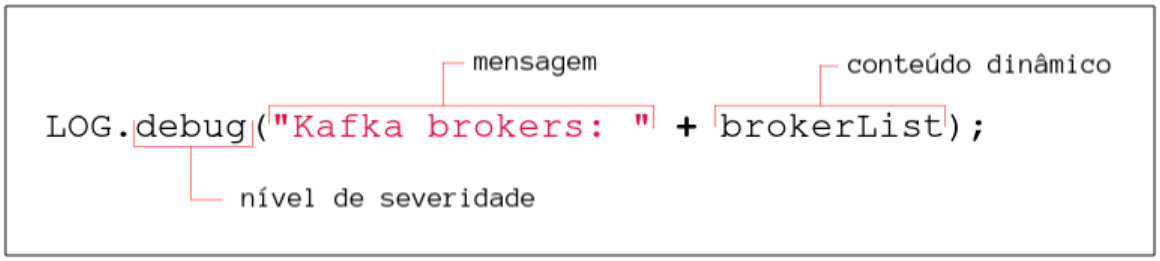

Figura 1 – Exemplo de instrução de *logging*

2.1.1 Níveis de verbosidade

O uso de níveis de verbosidade possibilita desabilitar o registro de certas mensagens em detrimento de outras (LI; SHANG; HASSAN, 2017). São posicionados de maneira hierárquica e nomeados segundo sua importância. Ao decidir por um nível específico, além da importância, é preciso levar em consideração a quantidade de informação e nível de abstração presente na mensagem. Em geral, os níveis de verbosidade mais utilizados são:

- *trace* - Reservado para *logs* cuja mensagem possua informações detalhadas sobre o fluxo de uma aplicaçao. Nível mais verboso;
- *debug* - Reservado para *logs* cuja mensagem possua informações que possam auxiliar o diagnóstico de problemas;

[1] Estrutura de dados utilizada para representar cadeia de caracteres textuais
[2] Elemento presente em linguagens de programação utilizado para armazenar conteúdo de natureza variável

- *info* - Reservado para *logs* cuja mensagem apresente informações que indiquem a ocorrência de eventos comum dentro da aplicação;
- *warn* - Reservados para *logs* cuja mensagem indique a ocorrência de situações inesperadas, que não são, porém, prejudiciais ao fluxo da aplicação;
- *error* - Reservado para *logs* cuja mensagem registre a ocorrência de eventos de falha, que são prejudiciais ao fluxo da aplicação;
- *fatal* - Reservado para *logs* cuja mensagem registre a ocorrência de um evento de falha, que colocam a aplicação em um estado de indisponibilidade. Nível menos verboso.

As bibliotecas de *logging* geralmente permitem que seus usuários possam executar a aplicação escolhendo um nível padrão de verbosidade, de forma que, apenas mensagens com prioridade igual ou maior a ele serão incluídas no registro. Desta forma, é possível decidir registrar apenas eventos de erro, configurando a aplicação para executar sob o nível homônimo de verbosidade, suprimindo, consequentemente, mensagens dos níveis *trance*, *debug*, *info* e *warn*.

Um problema que era derivado desse mecanismo é a impossibilidade de trocar o nível padrão, sem ter que reiniciar a aplicação. Porém, atualmente, *runtimes* como o de aplicações Sprint Boot, framework Java, permitem efetuar a troca dos níveis de maneira dinâmica.

2.2 Prática de *logging*

Registros de *logs* podem auxiliar no diagnóstico de problemas em produção, podendo, em muitos casos, ser o único recurso disponível (YUAN; PARK; ZHOU, 2012), (YUAN et al., 2012).

Em geral, a prática de *logging* apresenta como desafio, três principais questões: onde, o quê, e como realizar os registros de *logging* (ZHU et al., 2015). Porém, é preciso levar em consideração que possuindo muitos registros de *log*, a aplicação poderá apresentar *overhead* e um aumento significativo no esforço necessário para manutenir esses registros (LI; SHANG; HASSAN, 2017).

Cândido, Aniche e Deursen (2019), por meio de seu trabalho de mapeamento sistemático sobre o tema, classificou os estudos em três categorias: estudos empíricos, que lidam com a maneira que a prática é realizada; requerimentos para *logging*, os quais buscam avaliar se o *logging* atual alcança os objetivos propostos e, finalmente, decisões de *logging*, que lidam com técnicas que auxiliam os praticantes a fazerem melhores decisões de onde e o que logar.

A primeira pesquisa sobre o assunto foi feita por Yuan, Park e Zhou (2012). O estudo teve como base quatro softwares *open-source*. Uma de suas descobertas mostrou que a prática de *logging* é pervasiva: a cada 30 linhas de códigos, uma linha de *logging* é adicionada. Vale notar que essa observação se repetiu nos estudos posteriores. Constatou-se também, que *tickets* de *bugs*, relatórios que descrevem e mapeiam os *bugs*, contendo *logs* reduzem o tempo de resolução do problema num fator de 2.2x. Outras descobertas dizem respeito à frequência, natureza e localidade de revisões de código de *logging*. Foi mostrado ainda que, mesmo em baixa quantidade em relação ao código todo, *logs* foram atualizados em 18% de todas as revisões e que destas, 98% foram modificações no conteúdo da mensagem sendo o restante ou exclusões ou modificações.

Por meio de sua pesquisa, Yuan et al. (2012) propôs uma solução para *logging* chamada *LogEnhancer*, que busca modificar cada mensagem de *log* ao adicionar informação adicional para ajudar no processo de diagnóstico de falhas.

Já no estudo feito por Fu et al. (2014), uma pesquisa foi realizada a fim de responder as seguintes questões sobre a prática: Quais categorias de fragmentos de códigos são registrados em *logs*? Quais fatores são considerados para o *logging*? É possível automaticamente determinar onde é necessário inserir registros de *log*? Para tanto, o código-fonte de dois grandes softwares industriais foram analisados. Os resultados mostraram que, de 100 amostras de fragmentos de código que continham registros de *logs*, pode-se retirar duas grandes categorias. A primeira, situações inesperadas, é compreendida por 3 subcategorias: *logs* de verificação de assertivas, *logs* de verificação de valor de retorno e *logs* de exceção. A segunda, pontos de execução, possui *logs* que objetivam apenas fornecer informações sobre a fluxo de execução, contendo as categorias: bifurcações lógicas e pontos de observação. Outro achado mostra que cerca de metade das instruções são da primeira categoria, enquanto a outra metade, da segunda. Além disso, foi observado que apenas uma pequena porcentagem de fragmentos do código recebem instruções de *logs*, como constatado também por Yuan, Park e Zhou (2012), estando essas, em 35% dos casos, dentro de blocos *try/catch*.

A dinâmica da prática foi estudada também por Pecchia et al. (2015). No contexto de softwares críticos, os pesquisadores buscaram avaliar o processo de desenvolvimento de um software utilizado na indústria de transportes. Uma das principais observações mostra que a prática de *logging* costuma produzir registros inconsistentes.

Com o intuito de buscar novas soluções que auxiliem os desenvolvedores nas decisões pertinentes à prática de *logging*, Zhu et al. (2015) também realizaram um estudo sobre o tema. Os objetos desse estudo foram quatro softwares: dois industriais e dois *open source*. Além do já constatado fato de que a prática de *logging* é largamente difundida, foi observado também que: as decisões de *logging* podem se resumir a onde e o que logar; desenvolvedores buscam balancear o número de instruções, a fim de evitar *overhead* e

dificuldades em encontrar *logs* relevantes; que o contexto é fundamental na decisão de *logging*; e que a quantidade de problemas encontrados em um único arquivo de código-fonte, é proporcional ao número de LIs nele contida.

Realizando um estudo de replicação do trabalho feito por Yuan, Park e Zhou (2012), Chen e Jiang (2017b) buscaram responder perguntas semelhantes. O objetivo foi a caracterização da prática de *logging* em projetos *opensource* Java. Suas descobertas mostram que: projetos *server-side* possuem maior densidade de logs em comparação à aqueles de front-end; o tempo de resolução de *issues*[3] de *bugs* contendo logs foi em média maior que aqueles que não possuem. Apesar deste resultado, não é possível afirmar que *logs* dificultam a resolução de bugs[4], visto que, existem outros fatores envolvidos.

O trabalho desenvolvido por Li, Shang e Hassan (2017) buscou analisar os motivos pelos quais atualizações nos códigos *logging* são feitas. Foram encontradas 20 razões, distribuídas em quatro categorias: alteração de código contextual, melhoria de *logging*, mudanças dirigidas por dependência e resolução de problemas de *logs*.

Ao tratar o problema de como logar Chen e Jiang (2017a), buscaram descrever as características de *anti-patterns*[5] encontrados nos logs. Com esse objetivo, analisaram o histórico de revisão de três grandes softwares *open-source*. Como resultado, conseguiram delimitar seis *anti-patterns* diferentes.

No ramo do processamento de linguagem natural aplicada as descrições de logging, He et al. (2018) realizou uma pesquisa analisando as descrições de logs de 10 projetos Java e 7 projetos C#. Os resultados mostraram que existem três categorias principais de descrições de logs: descrição de condições de erro, descrição para operação do programa e descrição de semântica de código de alto nível.

Em sua tese, Hassani et al. (2018) buscou estudar os problemas relacionados ao código *logging* em dois softwares *open-source* de grande escala. Como consequência, Hassani descobriu que problemas em *logs* podem passar em média, 320 dias até que sejam reportados.

Abordando o problema do *logging*, Li e Chen (2020) buscaram desenvolver um algoritmo com o objetivo de inserir novas instruções de *logs* de forma automatizada. Para tanto, utilizaram a ideia de que é possível medir o nível de incerteza de caminhos de execução ao se inserir uma nova instrução de *log*.

Motivado por oportunidades de desenvolver uma ferramenta que também automatizasse o processo de *logging*, Li e Chen (2020) conduziram um estudo a fim de retirar diretrizes que pudessem prover sugestões de locais no código-fonte para *logging*. Para tanto, o pesquisador analisou o código-fonte de sete grandes softwares *open-source*. Entre

[3] Artefado contendo a descrição de algum problema de software
[4] Problema encontrado na execução de softwares
[5] Comportamento padrão de desenvolvimento, considerado uma má prática

seus principais achados, encontra-se a medição de que 76% dos códigos que continham *logs*, faziam parte de fragmentos que, ou buscavam gravar exceções, ou buscavam informar sobre a execução de *branchs* ou iteração de programa.

2.3 Java

Java é uma linguagem de programação orientada a objetos, de propósito geral e fortemente tipada. Sua forma de compilação é dita mista, pois o código-fonte é primeiramente compilado em *bytecode*[6] intermediário, para somente depois ser executado na plataforma de *runtime* Java, a JVM (*Java Virtual Machine*). A vantagem desse modelo se encontra na portabilidade, uma vez que, o bytecode da aplicação pode ser executado em todos os ambientes nos quais a JVM oferece suporte.

2.3.1 Tratamento de exceções

O tratamento de exceções em Java é feito por meio do uso da instrução de bloco *try/catch*. A Fig. 2 apresenta um exemplo desse tipo de instrução. Quando uma situação anormal é detectada em algum ponto da aplicação, e como consequência, uma exceção é lançada, um objeto a descrevendo é criado e repassado ao sistema de *runtime*, que passa a procurar na pilha de chamadas algum método que tenha declarado algum bloco *try/catch* que capture aquele tipo específico de exceção. Uma vez encontrado, o bloco *catch* é executado e o fluxo segue para a instrução seguinte ao *try/catch*. O *try/catch* comporta um bloco opcional, chamado de *finally*, que sempre é executado independente do que tiver ocorrido durante a execução do bloco *try*. Geralmente, esse bloco é destinado a liberar recursos alocados para as operações. A partir da versão 8 do Java SE, foi adicionado o *try-with-resources*, que permite declarar recursos que devem ser liberados ao final da operação do bloco *try*, não sendo mais necessário nesse sentido utilizar o *finally*.

```
try {
    // operações que podem gerar alguma exceção
} catch (Exception e) {
    // Código para tratar a exceção
}
```

Figura 2 – Exemplo de código try/catch

[6] conjunto de instruções da linguagem codificados em binário

2.3.2 *Logging* em Java

Nas linguagens de programação, suporte a prática de *logging* pode ser oferecida nativamente, ou por meios de bibliotecas de terceiros. Apesar de existirem diferenças, as bibliotecas costumam fornecer o mesmo conjunto de funcionalidades, sendo elas: a definição de um conjunto ordenado de níveis de verbosidade; uma interface, contendo as definições dos métodos de *logging*, pelos quais se é possível realizar a chamada das invocações de logging (LI); e a possibilidade de configuração do nível de verbosidade padrão, para que se possa controlar a quantidade de registros escritos.

O desenvolvedor que deseja realizar o registro de algum evento dentro da aplicação, precisa então:

1. Obter a referência para a interface de *logging*;
2. Definir um nível de verbosidade para o evento;
3. Definir o conteúdo da mensagem a ser escrita;
4. Realizar a invocação do método de *logging* na interface com os devidos argumentos.

```java
// Declarando a referência a interface de logging e obtendo uma instância
Logger LOG = LoggerFactory.getLogger(Application.class);

// Chamada do método de logging, caracterizando uma Logging Invocation (LI)
LOG.error("Something went wrong", e);
```

Figura 3 – Exemplo de LI em Java

A prática de *logging* na linguagem Java pode ser realizada por meio da biblioteca padrão Oracle (2021) ou por bibliotecas feitas pela comunidade. As bibliotecas da comunidade que se destacam são o Simple Logging Java Facace (SLF4J) QOS (2022), Log4j Apache (2021), e Logback QOS (2021). A Fig. 3 apresenta um exemplo de uma LI feita na biblioteca slf4j.

2.4 Análise estática

Há duas formas de implementar análises de software: estática e dinâmica. Na primeira, o único artefato examinado é o código-fonte, não sendo necessária sua execução. De acordo com Sommerville (2011) "as técnicas de análise estática podem ser usadas para verificar os modelos de especificação - e de projeto de um sistema - para encontrar erros antes que uma versão executável do sistema esteja disponível". É possível verificar

o código-fonte a fim de que seja validada a conformidade, por exemplo, com determinado padrão de estilo.

Uma técnica muito utilizada por ferramentas de análise estática é a transformação do código em um modelo abstrato conhecido como Abstract Syntax Tree (AST)[7], onde o código-fonte, serializado, é lido e transformado em uma estrutura de árvore.

[7] grafo não-direcionado, no qual dois vértices são conectados por no máximo, um caminho

3 Metodologia

A metodologia tem por finalidade explicitar os procedimentos, processos e instrumentos que deverão ser utilizados para que o objetivo deste trabalho possa ser alcançado (MARCONI; LAKATOS, 2003).

Este trabalho, fará uso de uma metodologia definida em três fases distintas e dependentes. Na primeira, será feita uma pesquisa exploratória a fim de produzir dados necessários ao desenvolvimento da ferramenta proposta. A seguir, utilizando o que foi obtido na pesquisa, uma ferramenta de análise estática será desenvolvida. Por fim, a ferramenta será testada.

3.1 Revisão bibliográfica

A pesquisa bibliográfica deste trabalho, buscou contextualizar sobre as questões pertinentes à prática de *logging*. Para tanto, foi feita uma busca na base de dados do portal de periódicos (CAPES, 2021). A *string* de busca utilizada foi: *(log or logging) AND (software OR application OR system) AND (practice OR development OR study)*.

3.2 Metodologia de pesquisa

Logo após a pesquisa bibliográfica, foi necessário realizar uma pesquisa exploratória a fim de responder as seguintes questões:

- **RQ1**: Como é possível caracterizar problemas em instruções de *logs*?

  Como este trabalho propõe uma solução que busque avaliar a qualidade dos *logs* de uma determinada aplicação, será essencial definir diretrizes necessárias a identificação de problemas.

- **RQ2**: Quais indícios contidos no código-fonte da aplicação, indicam a necessidade de inserção de instruções de *logs*?

  Concentrou-se em buscar diretrizes que pudessem ser seguidas a fim de identificar possíveis pontos onde inserir LI faltantes.

Para responder a estas questões, recorreu-se à pesquisa bibliográfica já feita, uma vez que, muitos dos estudos realizados já abordavam, em certo grau, os problemas encontrados na prática de *logging*. A pesquisa resultou então, nos requisitos funcionais da aplicação que logo foram mapeados em um documento informal de requisitos.

3.3 *Design* da ferramenta

Uma vez que os requisitos foram delineados, e todas as diretrizes de detecção de problemas de *logging* foram definidas, a ferramenta teve então sua arquitetura delimitada e cada requisito mapeado em um conjunto de tarefas. Para dar suporte ao acompanhamento das tarefas, foi utilizado a ferramenta de gestão de projeto, *Trello*. A solução foi desenvolvida usando a linguagem de programação Java em sua versão 8. Como ferramenta de apoio ao desenvolvimento, utilizou-se a *IDE* Eclipse, em sua versão 2021-06, sem adição de nenhum *plugin* a não ser aqueles já presentes por padrão.

3.4 Metodologia de avaliação

Com a ferramenta finalizada, testes foram realizados em projetos *open-source*, de acordo o seguinte processo:

1. Selecionou-se quatro projetos *open-source* escritos na linguagem de programação *java*, que fossem elegíveis para uso da ferramenta;
2. Arquivos destes projetos contendo LIs foram selecionados e casos de testes foram derivados;
3. Realizou-se a execução dos casos de teste;
4. Documentou-se o resultado dos casos de teste.

4 Proposta da ferramenta

Este capítulo tem o objetivo de apresentar a ferramenta desenvolvida ao longo do curso deste trabalho, e estará dividido da seguinte forma: Primeiramente, serão apresentados os resultados obtidos pela pesquisa exploratória sobre o tema; logo após, será exposta uma visão geral da ferramenta e considerações iniciais sobre processo de desenvolvimento; e por fim, a arquitetura será apresentada.

4.1 Exploração de Logging

A seguir são apresentados os resultados da pesquisa exploratória feita.

4.1.1 RQ1: Como é possível caracterizar problemas em instruções de *logs*?

Algumas das pesquisas realizadas sobre a prática de *logging*, tiveram, como resultado, a produção de ferramentas, com o objetivo de auxiliar os desenvolvedores a produzir códigos *logging* de maior qualidade. Algumas delas se propuseram, por diversos meios, alcançar tais melhorias ao detectar ou corrigir problemas de *logging*, e foram, portanto, utilizadas como pontos de partida. Cada ferramenta foi então analisada nesse sentido. A Tabela 1 apresenta um resumo dessas ferramentas.

Ferramenta	Descrição
Errlog	Adiciona LI em pontos faltantes
LogEnhancer	Adiciona contexto a LIs
LCAnalyzer	Busca identificar *logging* anti-patterns
Hassani	Busca *logs* faltantes e valida *logging guards* e *typos*[1]

Tabela 1 – Ferramentas de melhoria de *logging*

4.1.1.1 *ErrLog*

Errlog (YUAN et al., 2012) é uma ferramenta que executa uma análise estática no código-fonte da aplicação e indica pontos onde seria adequado, baseado em certos critérios, adicionar novas LIs. Procura-se, principalmente, por lugares onde há a verificação do retorno de funções e pontos onde fluxos alternativos são disparados, tais como, blocos *default* de instruções *switch/case*.

É preciso notar que as observações foram feitas tendo como base softwares escritos, em sua maior parte, na linguagem de programação C, que não possui recursos de tratamento de exceções. Se as mesmas regras fossem traduzidas para o Java, por exemplo, onde a maneira mais comum de sinalizar situações inesperadas ser por meio das *exceptions*, essa regra poderia ser traduzida colocando LIs em todas as localidades nas quais uma exceção é identificada, ou seja, dentro de blocos *try/catch*.

Como visto na pesquisa feita por Li e Chen (2020), foi observado que da porcentagem de LIs presentes em blocos *try/catch*, cerca de 80% são LIs que estão sob os níveis *warning* e *error*, e apenas 20% estão sob outros níveis de verbosidade. Dito isto, não há indícios que comprovem os benefícios de adicionar LIs em todos os blocos *try/catch*. Há porém, o caso onde o bloco *try/catch* não possui nenhum código para tratar a exceção capturada. Cabe então, neste caso, realizar a sugestão de, no mínimo, inserir uma LI.

Nossa solução proposta também busca detectar pontos onde se faz necessário a inserção de uma LI, porém é feita apenas em pontos onde o *overhead*, pode ser desconsiderado, como em blocos try/catch.

4.1.1.2 *LogEnhancer*

Outra ferramenta de análise estática também desenvolvida, na temática de *logging*, foi o *LogEnhancer* Yuan et al. (2012). Como funcionalidade principal, adiciona contexto as LIs com o intuito de diminuir a incerteza dos caminhos que podem levar à sua execução.

Adicionar contexto às LIs, é um ponto defendido também no artigo escrito por (SENTINELONE, 2022), onde são descritas boas práticas de *logging*. É sugerido, por exemplo, que, sempre que possível, adicionar contexto ao *logging*, para evitar que os registros não tornem-se apenas ruído, e consequente, agregando, pouco, ou, nenhum valor.

4.1.1.3 LCAnalyzer

Utilizando também de análise estática para encontrar problemas em códigos *logging*, o *LCAnalyzer* (CHEN; JIANG, 2017b) é direcionado, principalmente, para a detecção de *anti-patterns* em código logging. Apesar disso, os *anti-patterns* identificados, podem ser encontrados em toda a base do código, não estando restritos apenas a trechos contendo LIs.

4.1.1.4 Ferramenta desenvolvida por Hassani

Por fim, o estudo efetuado por Hassani et al. (2018) teve como resultado o desenvolvimento de quatro verificadores de código, cada um contendo um objetivo específico, mas todos voltados à melhoria do código *logging*. O primeiro verificador busca identificar níveis de verbosidade inconsistentes. Nele a identificação é feita ao correlacionar LIs de

contextos semelhantes, e verificar as que mais diferem no quesito da verbosidade. No segundo, é verificado a consistência relativa ao *logging* de exceções em blocos *try/catch*. O terceiro, ocupa-se em encontrar erros de digitação no conteúdo das mensagens. E por fim, o último valida a ausência de guardas de níveis[2].

4.1.2 RQ2: Quais indícios contidos no código-fonte da aplicação, indicam a necessidade de inserção de instruções de logs?

Foi verificado também, se é possível determinar, apenas por análise estática, pontos do código fonte que possam indicar a necessidade de inserção de uma LI.

4.1.2.1 Ignorar exceções capturadas

Este problema, quando identificado, indica a necessidade de inserção de código *logging*. É caracterizado quando executa-se alguma operação dentro de algum bloco *try/-catch*, mas não há absolutamente nenhuma instrução para tratar a exceção. Essa prática é muito comum quando se está executando operações que são altamente passíveis de gerar uma exceção, mas o seus resultados não são importantes do ponto de vista do fluxo da aplicação. É visto com ainda mais frequência no Java, devido às exceções checadas, onde o código cliente é obrigado a executar a operação por um bloco *try/catch*.

É considerado um problema, pois pode acabar resultando na ocultação de *bugs* ou situações não esperadas. Como nem sempre é possível determinar o real motivo dessa escolha, recomenda-se inserir LIs nesses pontos.

4.1.3 Problemas de *logging* na linguagem Java

Com a finalidade de contribuir para a melhoria da prática de *logging*, objetivando que esta possa gerar registros de maior valor. O presente trabalho, apresenta o desenvolvimento de uma ferramenta de análise estática que identifique a presença de problemas na infraestrutura de código *logging*.

Como cada linguagem de programação possui suas próprias características, e, consequentemente, seus próprios problemas, a princípio decidiu-se por escolher apenas uma, *Java*. A escolha baseou-se na contemporaneidade da linguagem; na quantidade de projetos de código aberto disponíveis; na quantidade de estudos da área de *logging* que a referenciam; na variedade de bibliotecas de suporte ao *logging* nela escritas; e na familiaridade e experiência do autor.

Cabe então, agora, descrever quais os problemas de *logging* identificados no contexto específico da linguagem. A Tabela 2 resume esses problemas. A maior parte deles

[2] Condicional que testa a verbosidade da LI com o nível de verbosidade da aplicação

podem ser classificados dentro do objetivo de *debugging* de logs.

Problema	Descrição
Bloco try/catch vazio	Ocorre quando existe a supressão da exceção capturada em algum bloco try/catch.
Exceção não repassada à LI	Ocorre quando a exceção capturada, não se encontra presente nos argumentos da LI.
Falta de verificaçao de guards	Ocorre quando a excecução de uma LI, possui uma operação custosa em termos de processamento.
Relançar exceção já logada	Ocorre quando determinada exceção, que já foi logada, é relançada.
Verbosidade da LI inconsistente	Ocorre quando a severidade da LI não está de acordo com o evento logado.
Contexto incompleto ou ausente	Ocorre quando o contexto de um determinado evento logado na LI, não está presente, ou está ausente.
Privacidade	Ocorre quando as LI acabam por realizar o logging de informações sensíveis sem o devido cuidado.

Tabela 2 – Problemas de logging

4.1.3.1 Relançar exceção já logada

Este problema, ocorre dentro de um bloco *try/catch*, quando, onde já identificada a presença de uma LI, acontece o relançamento da exceção. A justificativa para que essa caracterização seja considerada um problema de *logging*, consiste na verificação dos registros de *logging* gerados. O que pode ocorrer, é a existência de blocos try/catch em múltiplos níveis da pilha de chamadas de métodos, e esses podem, também, ter algum código de *logging*, que resulta na repetição de registros de *logs* sobre o mesmo evento.

4.1.3.2 Níveis de verbosidade inconsistentes

Como visto no trabalho de Chen e Jiang (2017b), foi observado que há uma grande tendência de se modificar a verbosidade das LIs, que acaba acontecendo em resposta a demanda de operadores e desenvolvedores. Apesar de haver diretrizes que auxiliam na escolha dos níveis de verbosidade, o conteúdo das mensagens ainda deixa margem para interpretações dúbias. No Java, porém, é muito incomum realizar o *logging* sob o nível de verbosidade de error (ERROR), quando a LI não se encontra dentro de um bloco catch. Nos casos em que acontecem, estão dentro de ramos de instruções *if/else*. Este caso pode ser considerado um problema de logging, pois está realizando a sinalização de um evento de erro, mas todo o contexto provido por uma exceção é perdido, como a pilha de chamadas de funções. Recomenda-se, nesse caso, substituir o *logging* pelo lançamento de uma exceção adequada.

4.1.3.3 Exceção não capturada

Um outro tipo de problema também encontrado em códigos de tratamento de exceção, ocorre quando a referência da exceção não é passada para a LI. A justificativa para que essa situação seja considerada um problema de *logging*, pode também ser encontrada quando visualize-se os registros de *logs* gerados. Muitas bibliotecas de *logging* do ecossistema Java, permitem que as LIs recebam, por meio de seus parâmetros, a passagem da referência da exceção capturada. Como resultado, o registro produzido possui, além de sua mensagem, a pilha de chamadas da exceção. Esse é o comportamento padrão quando uma exceção ocorre sem que haja código para tratá-la, o *runtime* do sistema acaba escrevendo a mensagem da exceção, e sua respectiva pilha de chamadsa, para o fluxo de saída de erro.

Tal situação é considerada um problema de *logging*, pois o registro escrito contém um contexto incompleto, não possuindo a pilha de chamadas que é um recurso relevante para o processo de *debugging*.

4.1.3.4 Contexto incompleto

O Problema de contexto incompleto, caracteriza-se quando a mensagem do *log* não possui todas as informações pertinentes sobre o evento logado. Pode ocorrer em qualquer nível de verbosidade, porém, só é possível identificar com eficiência, quando dentro de blocos *try/catch*, onde geralmente não há preocupação com o *overhead* provocado. A fim de sugerir quais poderiam ser as informações que deveriam estar presentes na LI, é necessário identificar quais estão causalmente ligadas ao bloco do código que pode gerar a exceção logada pela LI.

4.1.3.5 Verificação de nível de verbosidade não presente

Com o uso de múltiplos níveis de verbosidade, é preciso sempre ter cautela a fim da infraestrutura do código *logging* não acarretar um *overhead* maior do que o necessário. Podem ocorrer situações onde a simples execução de uma LI pode acabar desencadeando operações que são custosas, em termos de processamento e escrita.

Buscando cobrir os clientes nesse aspecto, muitas bibliotecas expõem em sua *API* formas de comparar o nível de verbosidade da LI com o limite do *runtime*. Dessa forma, a escrita do registro da LI, só ocorre caso seja necessário.

Diante disto, é possível identificar pontos da aplicação onde essa verificação não esteja sendo realizada. É válido destacar também, que bibliotecas mais modernas já incorporam a verificação da verbosidade das LIs por padrão.

4.1.3.6 Questões de privacidade

Uma outra questão que é possível verificar, não sendo exclusiva do Java, é a questão de privacidade do *logging*. No problema de contexto incompleto, foi falado da importância de fornecer contexto para os eventos nas LIs. No entanto, é preciso ter cautela quando tal contexto pode conter informações sensíveis dos usuários, tais como senhas ou outros dados pessoais. Nesses casos, é necessário realizar ou a criptografia ou a mascaração desses dados sensíveis, antes que possam ser escritas no registro de *logging*.

4.2 Visão geral

Objetivando buscar a implementação de registros de *log* de maior qualidade, e baseando-se nos resultados encontrados sobre a prática, foi desenvolvida uma ferramenta, escrita na linguagem Java, que busca identificar problemas de *logging* em aplicações desenvolvidas na mesma linguagem. Para tanto, utilizou-se de técnicas de análise estática, cujo escopo de verificação, limita-se apenas ao código-fonte.

Como funcionamento, a ferramenta analisa um ou mais arquivos de código fonte Java, buscando por possíveis problemas. Ao final do processo, um relatório em PDF é gerado, contendo a descrição desses problemas, bem como métricas relativas á infraestrutura de *logging* encontrada.

Durante o desenvolvimento da ferramenta foi preciso fazer uso de algumas bibliotecas externas, sendo elas:

- Spoon (PAWLAK et al., 2015): Ferramenta que foi utilizada para fazer tanto o parsing do código fonte, quando o uso do modelo AST;
- Jasper (JASPERSOFT, 2022): Ferramenta utilizada para gerar o relatório de erros;
- JfreeChart (JFREECHART, 2021): Ferramenta utilizada para gerar os gráficos que o relatório de erros faz uso.

4.3 Requisitos

O processo de desenvolvimento de uma determinando solução de software, implica à necessidade de solucionar um ou mais problemas, de um ou mais domínios. Para resolver tais problemas, é preciso, antes, identificar qual o conjunto de funcionalidades que a solução desenvolvida deve apresentar, a fim de ser capaz de solucionar tais problemas. No contexto da engenharia de software, a esse conjunto de funcionalidades esperadas, dá-se o nome de requisitos de softwares, e ao processo que dá assistência à descoberta, entendimento e documentação destes requisitos, dá-se o nome de engenharia de requisitos.

4.3.1 Requisitos funcionais

A seguir, é apresentada a lista dos requisitos funcionais identificados:

- O sistema deverá ser capaz de encontrar problemas na implementação da infraestrutura de *logging* de aplicações Java;
- O sistema deverá gerar, ao final do processo, um relatório em PDF contendo um resumo da avaliação. No relatório deverá constar, para cada problema encontrado, o nome do arquivo, a descrição do problema, e a visualização do trecho contendo o problema;
- O sistema deverá receber como parâmetros de entrada, a informação do caminho onde encontra-se o projeto que se deseja avaliar, e do caminho onde o relatório deverá ser gerado;
- O sistema deverá possuir regras que possam encontrar e caracterizar problemas que são prejudiciais à infraestrutura de *logging* e os registros por eles produzidos.

Para o escopo deste projeto, buscou-se implementar validadores para os seguintes problemas:

- Relançar exceção já logada;
- Ignorar exceções capturadas;
- Níveis de verbosidade inconsistentes;
- Exceção não presente na LI;
- Contexto incompleto.

O validador de verificação de nível de verbosidade e o validador de privacidade não foram incluídos, devido ao tamanho do escopo e devido à suas implementações serem mais complexas. Porém, estão mapeadas para trabalhos futuros.

4.3.2 Requisitos não-funcionais

Como único requisito não funcional, se faz necessário que a aplicação permita alta manutenibilidade, evidenciado, principalmente, na capacidade de permitir a extensão, seja por meio da adição de novas regras de validação, seja por meio de adição de suporte a outras bibliotecas.

4.4 Arquitetura

O projeto de arquitetura visa estabelecer uma estrutura na qual estejam bem definidos os componentes constituintes do softwares, assim como a forma como se dá a comunicação entre eles.

A escolha de uma determinada arquitetura pode ser justificada levando-se em consideração os requisitos não-funcionais, pois estão intimamente ligados. Ou, colocando em outras palavras: a arquitetura deve ser a base para a escolha de implementação dos requisitos não-funcionais (SOMMERVILLE, 2011).

Em se tratando do projeto atual, optou-se por uma arquitetura que priorize a manutenibilidade, dando maior ênfase na extensão de novas regras de validação, e na adição de suporte a outras bibliotecas de *logging*.

A figura 4 fornece uma visão de alto nível da arquitetura. Detalhes sobre cada componente serão fornecidos logo em seguida.

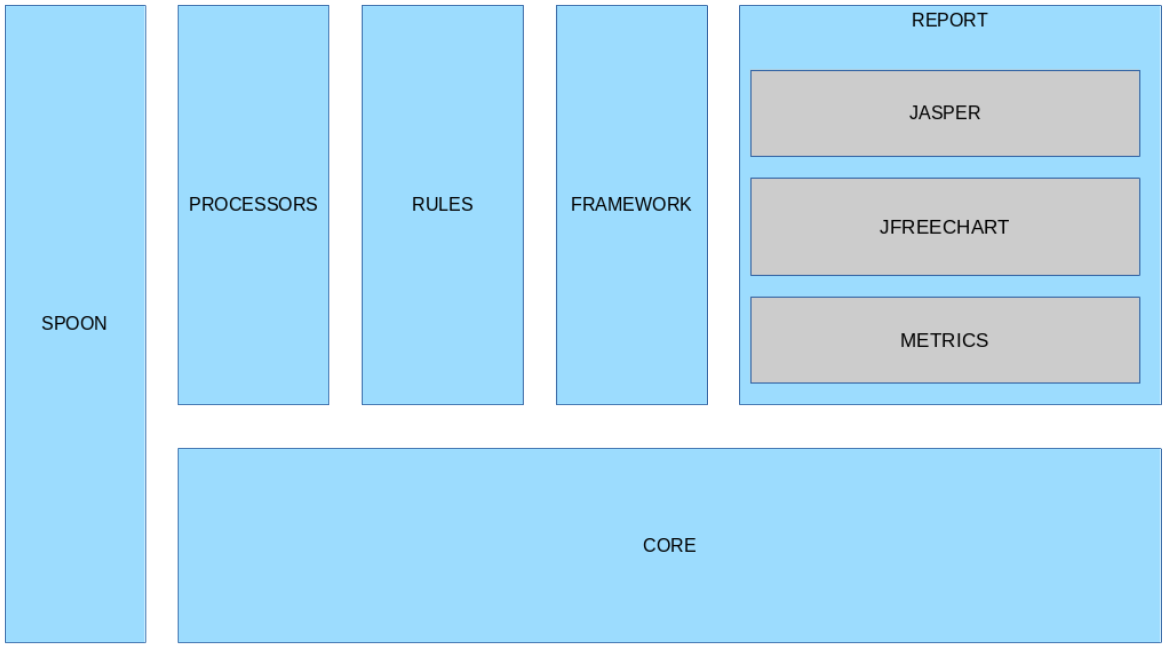

Figura 4 – Diagrama de arquitetura

4.4.1 Componentes

Os componentes que constituem a arquitetura foram distribuídos de acordo com seu papel dentro do funcionamento da aplicação. A seguir cada componente é descrito individualmente.

4.4.1.1 core

No componente core, estão definidas as principais interfaces e classes que compõem o funcionamento da aplicação. A interface *LoggingContextCharacterizer* define um único comportamento, *characterize*, que recebe um objeto representando a abstração de uma invocação *logging*, definida em alguma *LoggingLibrary*, que por fim resulta na produção de um *LoggingContext*, contendo todos as informações pertinente à invocação de uma LI.

4.4.1.2 libraries

O componente *library* é composto por todas as interfaces e classes que mapeiam as bibliotecas de *logging* que a solução oferece suporte. A interface *LoggingLibrary* define os comportamentos necessários para integração da biblioteca à arquitetura. Para tanto, é necessário definir um filtro que selecione LI, além de indicar quais regras de validação deseja-se utilizar.

4.4.1.3 rules

Neste componente encontram-se as regras de validação que cada LI do código analisado é submetido. Foram definidas as seguintes regras: *ExceptionNotLogged*, *NoRethrowAfterLogging*, *NoErrorOutsideTryCatch*, *IncompleteContext*. As regras podem ser selecionadas de forma independente da biblioteca de *logging*, uma vez que analisam diretamente o *LoggingContext*.

4.4.1.4 *report*

O componente *report* implementa as funcionalidades de geração de relatório da aplicação. Para tanto faz uso de duas bibliotecas externas, o *Jasper*, responsável por gerar o relatório em si, e o *JfreeChart*, que gera os gráficos utilizados pelo relatório.

4.4.2 Funcionamento

O funcionamento da ferramenta ocorre da seguinte maneira: primeiro é identificada a biblioteca de logging do projeto sob análise. Logo em seguida, uma instância dessa biblioteca é criada, contendo o filtro necessário para procurar todas as LI. Feito isso, passa-se para fase na qual o modelo abstrato é filtrado, buscando por todas as LI da referida biblioteca. Então, para LI encontrada no código, é criado um objeto que a descreve de acordo. Passa-se então à fase de preparação da análise, que busca selecionar as regras de validação de *logging* de acordo com os critérios da biblioteca de *logging* selecionada. Uma vez que as regras de validação estejam prontas, dá-se o início à aplicação dessas sobre os contextos de *logging*. Caso um dado contexto de *logging* não seja validada pela regra, é gerado um novo problema. Quando esta fase do processo é finalizada, tendo como

resultado a criação da abstração dos problemas de *logging*, inicia-se a fase de geração do relatório. Para tanto, é necessário realizar uma transformação na estrutura dos dados, a fim de que fiquem de acordo com a API da biblioteca de relatório. Além desses dados, são repassados para o relatório também, as métricas coletadas. O código-fonte da aplicação se encontra disponível em: (GITHUB, 2022)

5 Resultados e discussões

A fim de submeter a ferramenta desenvolvida à prática, alguns casos de testes foram desenvolvidos. Foram escolhidos quatro softwares *opensource* para servirem como alvos da análise da ferramenta. O principal critério usado para selecionar esses softwares foi a biblioteca de *logging* por eles utilizada. Foi preciso realizar um filtro inicial, levando em consideração que a presente solução suporta no momento apenas análises de softwares que façam uso do *SLF4J* como biblioteca de *logging*. Logo em seguida foi feita uma busca na plataforma *GitHub*[1], buscando por projetos Java ordenados por sua quantidade de estrelas. Dessa busca então, foram selecionados os seguintes projetos:

- Apache Flink - *Framework* para o processamento distribuído de corrente de dados;
- Apache Hadoop - Biblioteca que tem como objetivo o processamento distribuído de grandes conjuntos de dados em *clusters* de computadores;
- Apache Cassandra - Banco de dados não-relacional (NoSQL) distribuído de alta escalabilidade e disponibilidade;
- Apache Zookeeper - Serviço centralizado para manter informações de configuração e nomeação e também fornecer sincronização distribuída.

Objetivando demonstrar a aplicabilidade de cada regra de validação desenvolvida, certos trechos de códigos contendo LIs foram selecionados e submetidos a análise da ferramenta. Os resultados são apresentados a seguir.

5.1 Relançamento de exceção logada

A figura 5 apresenta um trecho de código que contém a caracterização do problema de relançamento de exceção logada. Como pode ser observado, logo após que a exceção é capturada e logada, uma nova exceção é lançada.

A ferramenta detectou o problema da forma esperada como evidenciado pela figura 6.

5.1.1 Contexto incompleto

A figura 7 apresenta um trecho de código no qual é possível visualizar uma instância do problema de contexto incompleto de *logging*. Como pode ser visto na linha 150, o

[1] Plataforma de hospedagem de códigos

```
141     private static <R> OptionalFailure<R> wrapUnchecked(String name, Supplier<R> supplier) {
142         return OptionalFailure.createFrom(
143                 () -> {
144                     try {
145                         return supplier.get();
146                     } catch (RuntimeException ex) {
147                         LOG.error("Unexpected error while handling accumulator [" + name + "]", ex);
148                         throw new FlinkException(ex);
149                     }
150                 });
151     }
```

Figura 5 – Exemplo do problema relançamento de exceção logada

AccumulatorHelper.java	147
No rethrow after logging	

```
catch (RuntimeException ex) {
   LOG.error(("Unexpected error while handling accumulator [" + name) + "]", ex);
   throw new FlinkException(ex);
}
```

Figura 6 – Resultado da verificação de relançamento de exceção

desenvolvedor inseriu uma chamada a uma LI, que, apesar de receber a referência da exceção capturada, não possui contexto relevante fora sua mensagem estática "Client delete failed", que não fornece nenhum outro detalhe. Dentro do bloco que gerou a exceção, encontram-se duas instruções, *decOutStanding*, a chamada de um método sem parâmetro, e *zk.delete(name, -1, this, null*, com *parâmetros*, *name*, -1, *this* e *null*. A fim de poder tornar a mensagem mais contextualizada, o desenvolvedor poderia, por exemplo, registrar o estado da variável *name*, pois esta, sendo utilizada como argumento, poderia ajudar na reprodução do erro caso se mostrasse necessário em algum momento.

O resultado da verificação desse mesmo trecho é apresentado na figura 8. A ferramenta foi capaz de apontar o problema, e, como pode ser notado, a mensagem contendo a descrição do problema possui a lista dos nomes das variáveis que, fazendo parte do contexto da LI, poderiam ser inseridas no contexto da mensagem da LI.

Embora o relatório produzido ofereça sugestões que poderiam ajudar a melhorar a qualidade do *log* escrito, a escolha de quais dados do contexto devem ser registrados, precisa ainda ser avaliado pelo desenvolvedor.

5.1.2 Exceção não sendo repassada à LI

Para este tipo de problema, buscou validar os pontos onde, havendo a captura de algum tipo de exceção, não existe o repasse desta para a LI, causando uma perda no contexto do evento de origem. Um exemplo é fornecido na Figura 10.

Na linha 81, a captura da exceção *e* é feita, e no bloco de tratamento, a LI é invocada. Porém, a exceção não está sendo repassa à LI. Dessa forma, faltando a referência da exceção como último argumento da LI, caracteriza-se o problema. Ao executar a ferramenta no referido arquivo do trecho, obteve-se o resultado apresentado pela Figura

```
131         public void processResult(int rc, String path, Object ctx, String name) {
132             if (rc != KeeperException.Code.OK.intValue()) {
133                 if (bang) {
134                     failed = true;
135                     LOG.error(
136                         "Create failed for 0x{} with rc:{} path:{}",
137                         Long.toHexString(zk.getSessionId()),
138                         rc,
139                         path);
140                 }
141                 decOutstanding();
142                 return;
143             }
144             try {
145                 decOutstanding();
146                 zk.delete(name, -1, this, null);
147             } catch (Exception e) {
148                 if (bang) {
149                     failed = true;
150                     LOG.error("Client delete failed", e);
151                 }
152             }
153         }
```

Figura 7 – Exemplo de problema de contexto incompleto

AsyncHammerTest.java 150

Unlogged context variables: rc, path, ctx, name

```
public void processResult(int rc, String path, Object ctx, String name) {
    if (rc != KeeperException.Code.OK.intValue()) {
        if (bang) {
            failed = true;
            LOG.error("Create failed for 0x{} with rc:{} path:{}", Long.toHexString(zk.getSessionId()), rc, path);
        }
        decOutstanding();
        return;
    }
    try {
        decOutstanding();
        zk.delete(name, -1, this, null);
    } catch (Exception e) {
        if (bang) {
            failed = true;
            LOG.error("Client delete failed", e);
        }
    }
}
```

Figura 8 – Resultado da verificação do trecho de contexto incompleto

10.

5.1.3 Verbosidade inconsistente

Um outro problema muito comum encontrado na prática, diz respeito à inconsistência dos níveis de verbosidade encontrada na infraestrutura de *logging*. A Figura 11 apresenta uma instância desse problema. O problema pode ser identificado na linha 216, onde se é realizado o *logging* de uma dada mensagem de falha, sob a verbosidade de erro, e logo após ter sido feito uma verificação no retorno de uma função, *hintsFile.exists()*

Ao se realizar o *logging* de um evento sob a verbosidade de erro, o desenvolvedor

```
67      for (Class<? extends CryptoCodec> klass : klasses) {
68        try {
69          CryptoCodec c = ReflectionUtils.newInstance(klass, conf);
70          if (c.getCipherSuite().getName().equals(cipherSuite.getName())) {
71            if (codec == null) {
72              PerformanceAdvisory.LOG.debug("Using crypto codec {}.",
73                  klass.getName());
74              codec = c;
75            }
76          } else {
77            PerformanceAdvisory.LOG.debug(
78                "Crypto codec {} doesn't meet the cipher suite {}.",
79                klass.getName(), cipherSuite.getName());
80          }
81        } catch (Exception e) {
82          PerformanceAdvisory.LOG.debug("Crypto codec {} is not available.",
83              klass.getName());
84        }
85      }
86
87      return codec;
88    }
```

Figura 9 – Trecho contendo o problema de exceção não sendo repassada a LI

CryptoCodec.java	82
Exception not being logged on catch block	

```
catch (Exception e) {
    PerformanceAdvisory.LOG.debug("Crypto codec {} is not available.", klass.getName());
}
```

Figura 10 – Resultado da verificação de exceção não sendo repassada à LI

está sinalizando que o dado evento necessita de atenção imediata. Porém, por não haver uma exceção presente, pois não há um bloco *try/catch*, são perdidas informações relevantes, como a pilha de chamadas. A sugestão nesse caso é, transformar o evento em uma exceção propriamente dita, ou também, realizar a diminuição do nível de verbosidade.

```
210     void delete(HintsDescriptor descriptor)
211     {
212         File hintsFile = descriptor.file(hintsDirectory);
213         if (hintsFile.tryDelete())
214             logger.info("Deleted hint file {}", descriptor.fileName());
215         else if (hintsFile.exists())
216             logger.error("Failed to delete hint file {}", descriptor.fileName());
217         else
218             logger.info("Already deleted hint file {}", descriptor.fileName());
219
220         //noinspection ResultOfMethodCallIgnored
221         descriptor.checksumFile(hintsDirectory).tryDelete();
222     }
```

Figura 11 – Exemplo de verbosidade inconsistente

O trecho em questão foi analisado pela ferramenta, que identificou o dado problema. A Figura 12 apresenta o trecho do relatório.

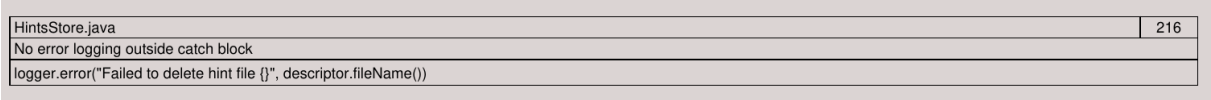

Figura 12 – Resultado da verificação de verbosidade inconsistente

5.2 Blocos *try/catch* vazios

A supressão de exceções lançadas por meio de blocos *try/catch*, apesar de muito difundida na comunidade Java, é ainda considerada má prática. A necessidade de se utilizar de blocos *try/catch* deve surgir apenas quando se tem a intenção de reagir de forma adequada a determinados problemas, tais como os que surgem quando, por exemplo, se tenta executar alguma operação de escrita ou leitura, como a tentativa de abrir um dado arquivo. As aplicações podem querer receber o caminho desses arquivos via parametrização. Logo, não há como saber de antemão, por exemplo, se o arquivo de fato existe no sistema. Essa situação é tão comum, que a linguagem Java, possui um conceito próprio pra esses tipos de exceções, as chamadas *"checked exceptions"*, que são exceções que exigem, normalmente, um tratamento imediato. Porém o que acaba acontecendo nessas situações é a supressão da exceção, por meio de um *try/catch* vazio. Um exemplo desse caso pode ser visto na figura (13)

```
317     /**
318      * Deletes the given file.
319      *
320      * <p><b>Important:</b> This method is expected to never throw an exception.
321      */
322     public static void deleteFileQuietly(Path path) {
323         try {
324             Files.deleteIfExists(path);
325         } catch (Throwable ignored) {
326         }
327     }
```

Figura 13 – Exemplo de bloco de tratamento *catch* vazio

A detecção desse problema por meio da ferramenta é simples, pois basta apenas verificar o número de instruções dentro dos blocos *catch*. Um exemplo de detecção é apresentado na figura 14

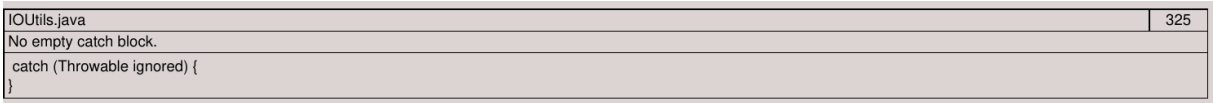

Figura 14 – Resultado da verificação de bloco *try/catch* vazio

5.3 Dados dos relatórios

Além de listar todos os problemas de *logging* detectados pela ferramenta, o relatório gerado traz também algumas métricas, que podem ser utilizadas para obter uma visão geral da atual situação do projeto dentro do contexto de *logging*. A figura 15 traz um exemplo do fragmento de um relatório, gerado para o projeto *hadoop*, contendo essas métricas. O gráfico mais à esquerda traz a distribuição dos níveis de severidade encontrado em todas às LIs presentes no projeto. Para cada nível de verbosidade é apresentado sua porcentagem e quantidade. Já o gráfico mais à direita, apresenta a distribuição em função dos problemas de *logging* encontrados. E por fim é mostrado também, duas métricas de referência, o *sloc* (*source lines of code*, que mede a quantidade de linhas gerais do projeto, e a *Logging Percentage*, que apresenta a porcentagem de linhas de código em relação ao *sloc*.

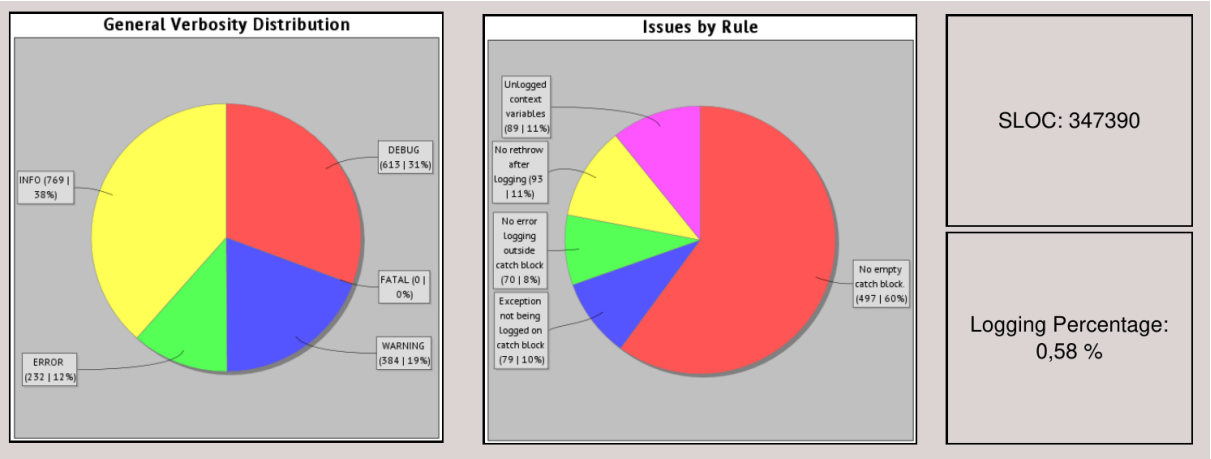

Figura 15 – Exemplo de cabeçalho do relatório

A Tabela 3 apresenta os dados referente à distribuição de *logs* de cada projeto estudado. Como é possível observar, a distribuição se manteve-se praticamente uniforme nos projetos *Hadoop*, *Flink* e *Cassandra*. Houve porém uma discrepância nos dados do *Zookeeper*, onde as LI feitas feitas sob a verbosidade *debug*, constituem mais da metade de todo projeto. Outro ponto a destacar é que, em nenhum dos projetos analisados pela ferramenta, foi encontrada uma única LI sobre o nível *fatal*.

Projeto	DEBUG	INFO	WARNING	ERROR	FATAL
Hadoop	613 (31%)	769 (38%)	384 (19%)	232 (12%)	0 (0%)
Flink	22 (20%)	27 (24%)	43 (38%)	20 (18%)	0 (0%)
Cassandra	347 (22%)	590 (38%)	357 (23%)	279 (18%)	0 (0%)
Zookeeper	282 (14%)	1063 (53%)	381 (19%)	294 (15%)	0 (0%)

Tabela 3 – Porcentagem de cada nível de verbosidade por projeto

Já em se tratando da distribuição dos problemas encontrados, como pode ser visto na Tabela 4, não houve uniformidade entre os projetos. Interessante notar também, que, o problema de não passar a exceção junto às LIs teve baixa presença em todos os projetos, e observando que o problema só pode ser detectado quando em LIs de nível *error*, que tiveram em média, 15% em cada projeto, o validador desse problema se mostra viável. Outro ponto interessante presente nos dados, é o fato que, nos projetos *Hadoop* e *Cassandra*, houve grande quantidade de problemas de exceções descartadas, sendo 60 e 94% respectivamente.

Projeto	**Relançamento de exceção**	**Contexto incompleto**	**Log de erro fora de try/-catch**	**Bloco try/-catch vazio**	**Exceção não presente no logging**
Hadoop	93 (11%)	89 (11%)	70 (8%)	497 (60%)	79 (10%)
Flink	3 (1%)	9 (3%)	9 (3%)	337 (94%)	0 (%)
Cassandra	64 (14%)	85 (24%)	127 (36%)	39 (11%)	40 (11%)
Zookeeper	42 (8%)	82 (16%)	105 (21%)	252 (50%)	25 (5%)

Tabela 4 – Porcentagem de cada problema por projeto

Por fim, notou-se também que, ao analisar os problemas encontrados, levando em consideração seus respectivos locais, percebeu-se que a regra que valida o problema de contexto incompleto, pode, com efeito, receber diversas melhorias. O que foi notado é que, muitos dos parâmetros que os métodos recebem, não estão diretamente relacionados ao bloco *try*, cujo tratamento se localiza a LI. Além disso, os argumentos de alguns métodos identificados, foram usados total, ou parcialmente, para gerar novas variáveis, e essas acabam sendo logadas. Outro falha encontrada foram os casos onde foi identificada a presença de blocos *try/catch* aninhados. Por motivos da ferramenta não ter antecipado estes casos, quaisquer problemas neles encontrados não foram detectados.

5.4 Comparação com outras ferramentas

- ErrLog - Procura por pontos na aplicação onde poderia ser inserido uma nova LI. Nossa solução também busca por esses pontos, porém é levado em consideração lugares onde aumento no *overhead* pode ser desconsiderado. É também levado em consideração a não poluição do registro.

- LogEnhancer - Adiciona informações adicionais às LIs, levando em consideração a remoção de incerteza pela variável adicionada. Nossa ferramenta também sugere a adição de contexto à LI, porém apenas em pontos onde o *overhead* pode ser desconsiderado.

- LCAnalyzer - Procura por *anti-patterns* em códigos logging. Nossa ferramenta também procura por anti-patterns na infraestrutura *logging*, porém diferente do LCAnalyzer, apenas por problemas exclusivos ao *logging*, uma vez que ferramentas de *linting* modernas podem capturar problemas gerais.

6 Considerações finais

Este trabalho buscou desenvolver uma ferramenta para identificar problemas em códigos-fonte escritos na linguagem de programação Java. Para tanto, realizou-se uma pesquisa exploratória para buscar entender como a prática é atualmente realizada, bem como, os problemas já identificados.

Os testes feitos em softwares *opensource*, apontaram que a ferramenta conseguiu identificar os problemas esperados, apesar de ainda existir espaço para melhorias, tais como, a adição de novos validadores e o aprimoramento dos já existentes. Os resultados mostraram que os projetos utilizados como teste, apontaram, por meio de suas métricas de *logging*, que a concentração da maior parte de instruções de *loggings* estão sob a verbosidade *INFO*, sendo essas, compostas por em média, 38% de todos os níveis de verbosidade; e que o problema encontrado que mais teve presença foi o bloco *try/catch vazio*, com em média, 53% de todos os problemas.

6.1 Trabalhos futuros

- Novas regras de validação: a fim de que possa agregar ainda mais valor a qualidade de logging, será preciso definir novas regras de validação;
- Adicionar suporte a IDEs: Como prática já comum na comunidade, ferramentas de avaliação de código acabam sempre integradas a IDEs;
- Adicionar suporte para outras bibliotecas, tais como a JCL (Apache Commons Logging): Como foi citado, a arquitetura da aplicação foi feita a fim de que possa permitir a fácil integração de novas bibliotecas.

Referências

APACHE. *Log4j – Apache Log4j 2*. 2021. Disponível em: <https://logging.apache.org/log4j/2.x/>. Acesso em: 2021-05-14. Citado na página 30.

CÂNDIDO, J. B.; ANICHE, M. F.; DEURSEN, A. van. Log-based software monitoring: a systematic mapping study. v. 1, p. 1–27, 2019. Disponível em: <http://arxiv.org/abs/1912.05878>. Citado na página 26.

CAPES. *CAPES Homepage*. 2021. Disponível em: <https://www-periodicos-capes-gov-br.ezl.periodicos.capes.gov.br/index.php?> Citado na página 33.

CHEN, B.; JIANG, Z. M. Characterizing and Detecting Anti-Patterns in the Logging Code. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, p. 71–81, 2017. Citado na página 28.

CHEN, B.; JIANG, Z. M. Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation. *Empirical Software Engineering*, Empirical Software Engineering, v. 22, n. 1, p. 330–374, 2017. Citado 3 vezes nas páginas 28, 36 e 38.

FU, Q. et al. Where do developers log? An empirical study on logging practices in industry. *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*, p. 24–33, 2014. Citado 2 vezes nas páginas 25 e 27.

GHOLAMIAN, S.; WARD, P. A. S. A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis. p. 1–45, 2021. Disponível em: <http://arxiv.org/abs/2110.12489>. Citado na página 21.

GITHUB. *logstaticanalyzer*. 2022. [Online; accessed 24. Sep. 2022]. Disponível em: <https://github.com/wallacybraz/logstaticanalyzer>. Citado na página 44.

HASSANI, M. et al. Studying and detecting log-related issues. *Empirical Software Engineering*, v. 23, n. 6, p. 3248–3280, 2018. Citado 2 vezes nas páginas 28 e 36.

HE, P. et al. Characterizing the natural language descriptions in software logging statements. *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, p. 178–189, 2018. Citado na página 28.

JASPERSOFT. 2022. [Online; accessed 24. Sep. 2022]. Disponível em: <https://community.jaspersoft.com/project/jaspersoft-studio>. Citado na página 40.

JFREECHART. 2021. [Online; accessed 24. Sep. 2022]. Disponível em: <https://www.jfree.org/jfreechart>. Citado na página 40.

LI, H.; SHANG, W.; HASSAN, A. E. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, Empirical Software Engineering, v. 22, n. 4, p. 1684–1716, 2017. Citado 3 vezes nas páginas 25, 26 e 28.

LI, Z.; CHEN, T.-h. P. Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks. *(SE 2020 - Research Papers) - ASE 2020*, 2020. Citado 2 vezes nas páginas 28 e 36.

MARCONI, M.; LAKATOS, E. *Fundamentos de metodologia científica*. [S.l.: s.n.], 2003. 310 p. Citado na página 33.

ORACLE. *java.util.logging (Java Platform SE 8)*. 2021. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html>. Acesso em: 2021-05-14. Citado na página 30.

PAWLAK, R. et al. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, Wiley-Blackwell, v. 46, p. 1155–1179, 2015. Disponível em: <https://hal.archives-ouvertes.fr/hal-01078532/document>. Citado na página 40.

PECCHIA, A. et al. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. *Proceedings - International Conference on Software Engineering*, IEEE, v. 2, p. 169–178, 2015. Citado na página 27.

QOS. *Logback Home*. 2021. Disponível em: <http://logback.qos.ch/>. Acesso em: 2021-05-14. Citado na página 30.

QOS. *SLF4J Manual*. 2022. [Online; accessed 24. Sep. 2022]. Disponível em: <https://www.slf4j.org/manual.html>. Citado na página 30.

SENTINELONE. Logging Best Practices: The 13 You Should Know. *Dataset*, jul. 2022. Disponível em: <https://www.dataset.com/blog/the-10-commandments-of-logging>. Citado na página 36.

SOMMERVILLE, I. *Engenharia de Software*. [S.l.]: Person Prentice Hall, 2011. 310 p. Citado 4 vezes nas páginas 21, 22, 30 e 42.

YUAN, D. et al. Be conservative: Enhancing failure diagnosis with proactive logging. *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, p. 293–306, 2012. Citado 2 vezes nas páginas 26 e 35.

YUAN, D.; PARK, S.; ZHOU, Y. Characterizing logging practices in open-source software. *Proceedings - International Conference on Software Engineering*, p. 102–112, 2012. Citado 4 vezes nas páginas 22, 26, 27 e 28.

YUAN, D. et al. Improving software diagnosability via log enhancement. *ACM SIGPLAN Notices*, v. 47, n. 4, p. 3–14, 2012. Citado 2 vezes nas páginas 27 e 36.

ZHU, J. et al. Learning to log: Helping developers make informed logging decisions. *Proceedings - International Conference on Software Engineering*, v. 1, p. 415–425, 2015. Citado 2 vezes nas páginas 26 e 27.