Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Refatorando um Sistema Open Source: Diagnóstico e Soluções com SonarQube

Larissa Vitoria Medeiros Silva

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman

São Paulo
2025

Resumo

Larissa Vitoria Medeiros Silva. **Refatorando um Sistema Open Source: Diagnóstico e Soluções com SonarQube**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

A refatoração é uma prática essencial para manter a qualidade e manutenibilidade de softwares, especialmente em projetos *open source*, onde a colaboração contínua pode gerar complexidade no código.Este trabalho apresenta um estudo de caso sobre a refatoração do projeto Squad, um sistema desenvolvido em Python/Django para monitoramento de qualidade de software. Utilizando a ferramenta SonarQube, foram identificados diversos *code smells* e problemas de manutenibilidade no código, com destaque para a função `__parse_results__()`, cuja complexidade cognitiva foi classificada como crítica. O processo de refatoração seguiu padrões propostos por Martin Fowler, aplicando técnicas como extração de funções e reorganização estrutural do código para reduzir sua complexidade. Além disso, a cobertura de testes foi analisada para garantir que as modificações não comprometessem a funcionalidade do sistema. Os resultados demonstram que a refatoração melhorou a clareza e modularidade do código, tornando-o mais compreensível e fácil de manter. Este estudo reforça a importância de técnicas de refatoração na sustentabilidade de softwares *open source* e no uso de ferramentas como SonarQube para diagnósticos precisos da qualidade do código.

Palavras-chave: Refatoração. Código Limpo. Qualidade de Software. SonarQube. Código Open Source.

Abstract

Larissa Vitoria Medeiros Silva. **Refactoring an Open Source System: Diagnosis and Solutions with SonarQube**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Refactoring is an essential practice for maintaining software quality and maintainability, especially in open-source projects, where continuous collaboration can lead to code complexity. This study presents a case study on refactoring the Squad project, a Python/Django-based system for software quality monitoring. Using the SonarQube tool, several code smells and maintainability issues were identified, particularly in the `__parse_results__()` function, whose cognitive complexity was classified as critical. The refactoring process followed patterns proposed by Martin Fowler, applying techniques such as function extraction and structural reorganization to reduce complexity. Additionally, test coverage was analyzed to ensure that modifications did not compromise system functionality. The results demonstrate that refactoring improved the code's clarity and modularity, making it more understandable and easier to maintain. This study reinforces the importance of refactoring techniques in the sustainability of open-source software and the use of tools like SonarQube for precise code quality diagnostics.

Keywords: Refactoring. Clean Code. Software Quality. SonarQube. Open Source Code.

Lista de abreviaturas

ARM	Máquina RISC Avançada (*Advanced RISC Machine*)
CI	Integração Contínua (*Continuous Integration*)
GNU	GNU Não é Unix (*GNU's Not Unix*)
LAVA	Arquitetura de Validação Automatizada da Linaro (*Linaro Automated Validation Architecture*)
YAML	Outra Linguagem de Marcação (*Yet Another Markup Language*)
HTML	Linguagem de Marcação de Hipertexto (*HyperText Markup Language*)

Lista de figuras

Lista de tabelas

Sumário

Introdução

A evolução contínua dos sistemas de software exige que a base de código seja constantemente aprimorada para garantir sua legibilidade, manutenibilidade e eficiência. Projetos de software, especialmente aqueles de código aberto, estão sujeitos a contribuições de diversos desenvolvedores com diferentes níveis de experiência, o que pode levar à introdução de inconsistências no código, crescimento desnecessário da complexidade e acúmulo de code smells e dívidas técnicas (Fowler, 2018).

A refatoração de código é uma prática essencial para mitigar esses problemas. Segundo Fowler 2018, refatoração é o processo de reestruturação do código para melhorar sua organização interna, sem modificar sua funcionalidade. Entre as ferramentas utilizadas para detectar problemas de qualidade no código, o SonarQube se destaca como uma solução *open source* amplamente adotada por desenvolvedores, mais de sete milhões, e equipes de software (SonarSource, 2025). Ele permite a análise estática do código, identificando problemas relacionados a *code smells*, complexidade, manutenibilidade, confiabilidade e segurança (SonarSource, 2024).

Diante desse contexto, este trabalho propõe uma abordagem para melhorar a qualidade de código em projetos *open source* real, utilizando o projeto Squad (https://gitlab.com/Linaro/squad/squad) como estudo de caso. O Squad é uma plataforma desenvolvida pela Linaro para análise e visualização de resultados de testes automatizados, sendo utilizado para monitorar a qualidade de softwares ao longo do tempo (Linaro, 2024). Como um software colaborativo ativo, ele está em constante evolução, o que torna sua manutenção desafiadora. Dessa forma, este trabalho busca identificar e analisar pontos críticos na base de código do Squad, propor refatorações e avaliar os impactos dessas mudanças na qualidade do código.

A proposta desse trabalho se justifica pela sua contribuição prática para a manutenção e evolução de projetos *open source*, demonstrando como técnicas de refatoração podem ser aplicadas na prática para melhorar a qualidade do código de um projeto real. Além disso, este estudo pode servir como referência para outros desenvolvedores interessados em refatorar sistemas utilizando o *SonarQube* e técnicas de refatoração conhecidas.

Objetivo

Este trabalho tem como objetivo identificar e analisar problemas de qualidade de código em um sistema *open source*, o Squad, utilizando a ferramenta *SonarQube*, e aplicar técnicas de refatoração para melhorar a manutenibilidade e legibilidade do código e, assim, facilitar

a contribuição de outros desenvolvedores. Para isso, os seguintes objetivos específicos foram definidos:

- **Diagnosticar Problemas no Código:** Utilizar o *SonarQube* para identificar áreas críticas no código, com foco em métricas como complexidade cognitiva e manutenibilidade
- **Refatorar o Sistema Squad:** Selecionar e implementar estretégias de refatoração, reduzindo a complexidade e promovendo clareza e modularidade.
- **Documentar o Processo:** Registrar as decisões tomadas durante a refatoração, incluindo a escolha dos problemas, das estratégias e a análise dos resultados.
- **Validar as Refatorações:** Validar as mudanças aplicadas por meio de testes automatizados e novas análises com o *SonarQube*.

Metodologia

A metodologia adotada neste estudo consiste em uma abordagem prática baseada em análise de código, diagnóstico com ferramentas automatizadas e aplicação de refatorações orientadas por métricas. O processo será dividido nas seguintes etapas:

1. **Preparação do ambiente:** Configuração do *SonarQube* para analisar a base de código do Squad, garantindo que a ferramenta possa fornecer métricas relevantes para a identificação de problemas.
2. **Análise inicial do código:** Execução do *SonarQube* para identificar problemas de qualidade e complexidade no código e análise das métricas fornecidas.
3. **Verificação de de testes:** Uso da ferramenta Coverage.py, para analisar a cobertura de testes, e análise da qualidade dos testes, para garantir que as refatorações possam ser validadas.
4. **Seleção dos pontos de refatoração:** Priorização dos trechos do código com maior impacto na qualidade do software e que possuem boa cobertura de testes.
5. **Aplicação de refatorações:** Implementação de técnicas de refatoração documentadas no livro *Refactoring: Improving the Design of Existing Code*, de Martin Fowler 2018.
6. **Validação das mudanças:** Reexecução do *SonarQube* e dos testes automatizados para verificar se as refatorações trouxeram melhorias sem introduzir regressões no código.

Estrutura do Trabalho

Este trabalho está estruturado em três partes principais: o primeiro capítulo apresenta o referencial teórico, abordando conceitos de refatoração de código, da qualidade de software e da ferramenta utilizada. A segunda parte detalha a aplicação prática dos conceitos discutidos na fundamentação teórica no projeto Squad, descrevendo as análises realizadas,

os pontos selecionados para refatoração e as mudanças aplicadas. Por fim, são apresentadas as conclusões e sugestões para trabalhos futuros.

Capítulo 1

Fundamentação teórica

Neste seguimento, serão abordadas as teorias dos principais tópicos analisados neste trabalho, como refatoração, code smells, *SonarQube*, qualidade de código e manutenibilidade – em específico para códigos *open source*.

1.1 Refatoração

A refatoração é um recurso que tende a melhorar o design interno de um sistema sem alterar seu comportamento externo. Seu principal objetivo é tornar o código mais flexível, legível e de fácil manutenção, conforme descrito no livro *Refactoring: Improving the Design of Existing Code*, de Matin Fowler 2018.

Essa prática consiste na aplicação de uma série de mudanças adicionais e consecutivas no código em desenvolvimento que não alteram seu comportamento, seguidas de provas afim de garantir o pleno funcionamento do software, como a execução de testes. Ao realizar essas alterações em etapas reduzidas e sempre validando-as, minimiza-se o risco de introduzir erros, permitindo que o sistema permaneça funcional durante todo o processo de refatoração.

Em suma, a refatoração é uma prática essencial para o desenvolvimento de softwares de alta qualidade, permitindo que sistemas complexos evoluam de maneira controlada e eficiente, sem comprometer sua funcionalidade ou introduzir novos problemas.

1.2 Code Smells

A refatoração é frequentemente guiada pela identificação de *code smells* (cheiros de código) no código. Utilizando Fowler 2018 como base no estudo, o autor menciona o termo *code smells* como o indicativo de uma provável instabilidade no programa. Apesar de não ser de forma obrigatória um problema ou um *bug*, expõe locais de potencial erros futuros, os quais podem ser evitados ao utilizar técnicas de refatoração documentadas, no intuito de garantir a qualidade do software. Os principais exemplos desses chamados code smells são:

- **Métodos Longos:** Quando o código é extenso, sendo formado por inúmeras linhas, apresentando uma difícil compreensão e, muito provavelmente, acúmulo de funções;
- **Classes Grandes:** Similar ao método longo, ocorre quando há classes com inúmeros métodos e propriedades, gerando a violação do Princípio da Responsabilidade única (SRP), o qual será abordado posteriormente;
- **Duplicação do código:** Quando há trechos idênticos em diferentes partes do código, sendo necessário sempre alterar em todos os locais, caso haja a necessidade de mudança;
- **Acoplamento excessivo:** Quando há uma codependência entre as classes, dificultando a manutenção individual de modo que não impacte as outras.

1.3 Qualidade do código e Manutenibilidade

Ao falar sobre a qualidade de uma base de código, não leva em conta apenas o pleno exercício de sua função, mas, também, se apresenta uma fácil leitura, entendimento e se passível a futuras alterações, principalmente por programadores que não fizeram parte do seu desenvolvimento, podendo assim, a qualidade de um código mudar ao longo da sua evolucão (Fowler, 2018).

Para definir um código de qualidade, ele deve apresentar legibilidade, ou seja, ter bons nomes para suas variáveis e classes, seus componentes devem ser independentes ao máximo, apresentando modularidade. É necessário também que o código em questão suporte ser submetido a validações, a fim de certificar a existência ou não de possíveis erros e corrigi-los sem interferir no funcionamento, ou seja, que ele possua testabilidade e, por fim, que não possua duplicações desnecessárias.

Já a manutenibilidade de um software está diretamente ligada à simplicidade com que ele pode ser alterado para solução de determinados erros, adição de novas funcionalidades ou para aperfeiçoamento do sistema. Dessa forma, Fowler 2018 afirma que quanto maior a manutenibilidade, menor o risco de instabilidades ao fazer alterações no código, se seguidos principios como:

- **Princípio da Responsabilidade Única (SRP):** Cada módulo deve ter apenas uma única função.
- **Design Limpo:** Distinção eficaz entre as camadas e suas funções, promovendo fácil entendimento e possível extensão do código.
- **Testabilidade:** Quando o software foi posto em uma quantidade suficiente de testes automatizados que garantam a redução do risco de regressão.

Ao focar em projetos *open source*, essa questão apresenta uma responsabilidade ainda maior, visto que nesses casos o código a ser desenvolvido poderá ser trabalhado por diferentes programadores em contextos totalmente diferentes, seja quanto ao nível de experiência, quanto ao estilo próprio de programação, solicitando, assim, um padrão acessível a todos.

Dessa forma, o código *open source* enfrenta alguns desafios como uma linguagem autoexplicativa, a falta de contexto no qual esse software foi desenvolvido e, até mesmo, a alternância dos desenvolvedores que trabalharão em cima de determinado projeto. Por isso, é de suma importância que uma elevada qualidade e manutenibilidade esteja presente em um sistema *open source* para que seja possível manter um código limpo e reduzir ao máximo os atritos para novos desenvolvedores, garantindo a continuidade e evolução do projeto.

Em suma, a correlação das práticas supracitadas é indispensável para que um código possua uma boa qualidade de software.

1.4 *SonarQube*

O *SonarQube* é um software de código aberto, ou seja, disponibilizado publicamente, utilizado de forma difusa para a análise constante dos códigos, identificando problemas como vulnerabilidade de segurança e *code smells* em projetos de software (SonarSource, 2024). Conforme a diretriz, a plataforma é capaz de analisar estaticamente a base do código, sem necessariamente executá-lo, avaliando sua qualidade.

O *SonarQube* comporta mais de 30 tipos de linguagens de programação, medindo a qualidade do código por meio de, por exemplo, a duplicação de linhas de código, a cobertura de testes automatizados e a complexidade ciclomática, afim de evidenciar os trechos necessários de melhoria (SonarSource, 2024).

Capítulo 2

Refatoração no Squad: Diagnóstico e Implementação

Após estudar toda a fundamentação teórica necessária para refatorar um sistema, este estudo tem como objetivo aplicar os conceitos previamente apresentados na base de código do projeto *open source* Squad.

O Squad foi desenvolvido e é mantido pela Linaro, uma organização que trabalha na otimização de software *open source* baseados na arquitetura ARM (Máquina RISC Avançada), contribuindo para diversos projetos (Linaro, 2024).

O Squad foi escrito em Python 3, utilizando Django, para o back-end e JavaScript para o front-end, e se propõe a ser um sistema de análise de qualidade de software. Ele permite coletar, organizar e apresentar resultados de testes, sendo útil para acompanhar o desempenho e a qualidade de softwares ao longo do tempo (Linaro, 2024).

Por ser um software livre ativo sob os termos da GNU Public License, o Squad está constantemente sujeito a mudanças por diferentes desenvolvedores, com diferentes níveis de conhecimento em programação, visando adições de novas funcionalidades ou correções de *bugs*. Desse modo, a refatoração do código do Squad para diminuir a presença de *code smells* e sua complexidade é essencial para que ele seja não apenas mais fácil de ser compreendido pela comunidade de desenvolvedores, como também mais fácil de ser mantido e alterado.

Este estudo de caso utiliza o Squad como exemplo prático para aplicar conceitos de refatoração, *code smells* e qualidade de código. Em vez de uma reestruturação completa, o trabalho se concentra na refatoração de trechos que possuem uma boa cobertura de testes e *code smells* que afetam a legibilidade e a manutenibilidade. A escolha do Squad reflete os desafios comuns em sistemas *open source*, onde a evolução constante demanda uma atenção especial à qualidade do código.

Para facilitar o acompanhamento do processo de refatoração das classes, extraido de Fowler (2018), suas etapas estão resumidas na tabela abaixo:

Objetivo	Passos
Identificar pontos de refatoração	• Rodar SonarQube no projeto • Identificar problemas relevantes • Conferir qualidade dos testes e cobertura
Definir estratégias de refatoração	• Entender o código • Identificar padrões de refatoração • Decidir quais refatorações podem ser aplicadas
Implementar refatoração	• Alterar o código aplicando a refatoração escolhida
Verificar funcionamento pós-refatoração	• Verificar testes • Analisar melhorias

Tabela 2.1: *Objetivos e passos do processo de refatoração*

2.1 Análise de Resultados do SonarQube e Cobertura de Testes

Para identificar *code smells* e outros problemas de qualidade no projeto Squad, foi utilizada a ferramenta SonarQube. O SonarQube foi executado em um ambiente local utilizando um MacBook M1 2020 com 8 GB de RAM. Os passos realizados para rodar a análise no projeto foram os seguintes:

1. Iniciar o contêiner do SonarQube:

```
docker run -d -p 9000:9000 -p 9092:9092 sonarqube
```

2. Acessar a interface do SonarQube no `http://localhost:9000/` pelo navegador e fazer login com o usuário padrão `admin`.
3. Criar o projeto e gerar um token para análise.
4. No diretório do projeto Squad, executar o comando do SonarScanner:

```
sonar-scanner \
      -Dsonar.projectKey={Project_Key} \
      -Dsonar.sources=. \
      -Dsonar.host.url=http://localhost:9000 \
      -Dsonar.token={Project_Token}
```

Já para realizar a análise de cobertura de testes, foi utilizada a ferramenta Coverage.py, que é uma ferramenta de medição de cobertura de testes para Python. Para isso, foi necessário rodar o seguinte comando:

```
coverage run --source='squad' manage.py test
```

2.1.1 Resultados da Análise do SonarQube

Os resultados do SonarQube revelaram um total de 700 problemas de mantenabilidade, com 357 classificados como de alta prioridade. Além disso, foram identificados 14 problemas de confiabilidade, dos quais 8 são de alta prioridade e nenhum problema de segurança. Porém, como o escopo da refatoração proposta nesse estudo de caso está limitado ao código em Python, apenas metade desses problemas, no total 358, serão analisados abaixo. Para facilitar essa análise, os problemas serão divididos em quatro categorias do próprio SonarQube: atributos de clean code, atributos de qualidade de software, gravidade e tipo de problema.

Clean Code	
Atributo	**Problemas**
Consistência	126
Intencionalidade	182
Adaptabilidade	50
Qualidade de Software	
Atributo	**Problemas**
Segurança	0
Confiabilidade	10
Manutenibilidade	348
Gravidade	
Nível	**Problemas**
Alta	58
Média	92
Baixa	208
Tipo de Problema	
Tipo	**Problemas**
Bug	10
Vulnerabilidade	0
Code Smell	348

Tabela 2.2: *Distribuição de Problemas Identificados*

De acordo com a documentação do SonarQube, *clean code* é definido como um código que segue os quatro principais atributos: consistência, intencionalidade, adaptabilidade e responsabilidade. Como nenhum problema de responsabilidade foi detectado no código Python do Squad, isso quer dizer, pela definição de responsabilidade na documentação do SonarQube, que ele respeita a regulação de licença e *copyright*, preserva dados privados sensíveis e possui linguagem respeitosa, sem uso de palavras discriminatórias e ofensivas; em outras palavras, leva em consideração obrigações éticas e legais (SonarSource, 2024).

Por outro lado, foram encontrados problemas que violam os outros atributos de *clean code*. A maioria desses problemas (182) está relacionada à falta de intencionalidade, o que indica que o trecho de código não é claro — ou seja, não é autoexplicativo e dificulta a compreensão de sua finalidade —, lógico — apresenta erros explícitos, contradições ou comandos imprevisíveis e questionáveis —, completo — não alcança seu objetivo proposto — ou eficiente — utiliza recursos desnecessários ou de forma ineficaz (SonarSource, 2024).

Outra parte dos problemas (126) está relacionada ao déficit de consistência em alguns trechos do código. A consistência refere-se à uniformidade do código, ou seja, se ele segue um padrão regular, mesmo quando múltiplos contribuidores fazem alterações, como é o caso do Squad. Para alcançar consistência, é necessário adotar uma formatação padronizada e seguir convenções, preferencialmente aquelas amplamente estabelecidas pela comunidade da linguagem utilizada (SonarSource, 2024).

Já o restante dos problemas de *clean code* (50) carecem do atributo de adaptabilidade, indicando que o código não pode ser facilmente modificado com confiança, sem causar efeitos colaterais indesejáveis, seja para alterar sua funcionalidade ou estender seu uso. Um trecho de código adaptável possui uma função específica e única, evitando o acúmulo de instruções e excesso de complexidade, não possui repetições desnecessárias, é modular — ou seja, suas partes são pequenas, claras e independentes — e é amplamente testado, permitindo alterações seguras e reduzindo o risco de introduzir *bugs* (SonarSource, 2024).

Além da divisão dos problemas encontrados por atributos de *clean code*, a análise do SonarQube também relaciona esses problemas a atributos de qualidade de software. De acordo com a documentação do SonarQube 2024, qualidade de software é alcançada através de um código limpo e é dividido em três aspectos: segurança, confiabilidade e manutenibilidade. No código Python do Squad, nenhum problema relacionado à segurança foi identificado , o que indica que o software está protegido contra vulnerabilidades que poderiam comprometer sua integridade ou expor dados sensíveis. Em relação à confiabilidade, foram encontrados apenas dez problemas, sugerindo que o software é capaz de manter seu nível de desempenho sob condições específicas por um determinado período de tempo. No entanto, o maior número de problemas (348) está relacionado ao atributo de manutenibilidade, indicando que o Squad enfrenta dificuldades em ser facilmente alterado — seja para corrigir falhas ou implementar melhorias — e em ser compreendido por desenvolvedores que não tiveram contato prévio com a base de código, o que é especialmente prejudicial para a evolução de qualquer software, mas ainda mais para projetos *open source*, devido à sua natureza colaborativa. Esses resultados destacam a necessidade de aprimorar a robustez do código para evitar falhas inesperadas e reduzir a complexidade que dificulta sua manutenção.

Outra análise oferecida pelo SonarQube é a classificação dos problemas pela sua gravidade. A gravidade de um problema é determinada com base no seu impacto na qualidade do software e é categorizada em três níveis: alto, médio e baixo. Essa métrica é fundamental para definir as refatorações que serão priorizadas nesse estudo de caso, concentrando esforços nos problemas mais críticos e com maior impacto na qualidade do código. No código Python do Squad, foram identificados 58 problemas de alta gravidade, 92 de gravidade média e 208 de gravidade baixa. Essa classificação ajuda na delimitação do escopo inicial para definir os problemas a serem resolvidos, priorizando os de alta gravidade. No

entanto, problemas de gravidade média e baixa não serão completamente desconsiderados, pois ainda podem impactar negativamente a manutenibilidade e o desempenho, embora apresentem menor urgência (SonarSource, 2024).

Por fim, o SonarQube também classifica os problemas detectados com base no seu tipo, dividindo-os em três categorias principais: *bugs*, vulnerabilidades e *code smells*. Essa classificação permite compreender a natureza dos problemas e direcionar as refatorações de acordo com o impacto conhecido de cada tipo. No código Python do Squad, foram identificados 10 *bugs*, nenhuma vulnerabilidade e 348 *code smells*. Os *bugs* correspondem a erros que podem causar falhas diretas ou comportamentos inesperados durante a execução do programa (SonarSource, s.d.). As vulnerabilidades, por sua vez, representam pontos no código que fornecem riscos de segurança e estão abertos a ataques. A ausência de vulnerabilidades na base de código faz sentido, visto que já foi constatado, anteriormente, que ela não possui problemas de segurança na análise da qualidade do software. Por outro lado, a maioria dos problemas está na categoria de *code smells*, que abrange trechos de código confusos e difíceis de manter por não seguirem boas práticas e padrões de design. Embora não causem falhas imediatas, os *code smells* são fortes indicadores de potenciais falhas, aumentam o risco de introdução de *bugs* e dificultam a manutenibilidade e clareza do código (Fowler, 2018).

Os resultados obtidos a partir da análise do SonarQube oferecem maior clareza sobre pontos cruciais a serem trabalhados no código Python do Squad. Enquanto a ausência de problemas relacionados à segurança e o baixo número de *bugs* são aspectos positivos que reforçam a confiabilidade do software, a predominância de *code smells* e problemas de manutenibilidade evidenciam a necessidade de refatoração. Além disso, os problemas classificados como de alta gravidade devem ser priorizados, pois representam os de maior impacto no sistema. Esses resultados fornecem uma base sólida para orientar o processo de refatoração, permitindo que os esforços sejam direcionados de maneira estratégica para melhorar a clareza, consistência e adaptabilidade do código, contribuindo para sua manutenibilidade e sustentabilidade a longo prazo.

2.1.2 Resultados da Análise de Cobertura de Testes

A cobertura de testes desempenha um papel crucial no processo de refatoração, pois garante que alterações no código não introduzam novos erros ou quebrem funcionalidades existentes. Testes bem estruturados permitem que os desenvolvedores realizem mudanças com confiança e segurança, identificando rapidamente problemas inseridos e validando a integridade do sistema após modificações. (Aniche, 2022)

A análise da cobertura de testes é essencial antes de qualquer refatoração (Fowler, 2018) e, nesse estudo de caso, torna-se ainda mais necessária devido à ausência de um contexto detalhado sobre as funcionalidades existentes, as decisões de implementação e a arquitetura do software. Dessa forma, a principal base de confiança para a realização das mudanças será a cobertura de testes e a análise da qualidade dos testes. Neste trabalho, serão considerados apenas os problemas apontados pelo SonarQube em trechos de código que atendem a esses dois requisitos. Nessa subseção, será analisada exclusivamente a cobertura de testes; a qualidade dos testes será avaliada durante a escolha dos pontos de refatoração, pois não faz sentido analisar todos os testes presentes na base de código.

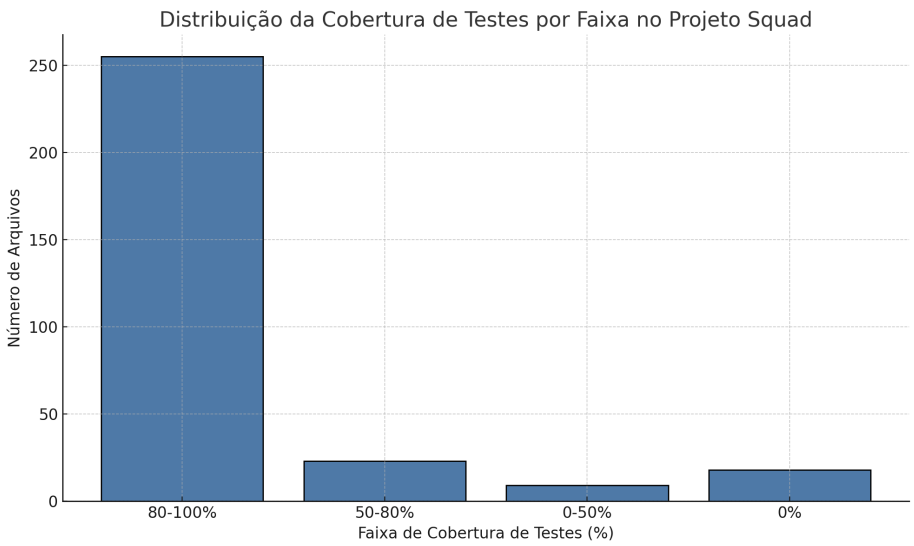

Figura 2.1: *Distribuição da Cobertura de Testes por Faixa no Projeto Squad*

A figura 2.1 apresenta a distribuição da cobertura de testes Python no projeto em diferentes faixas de cobertura, conforme medido pelo *Coverage.py*. A cobertura total obtida da base de código foi de 80%, com a maioria dos arquivos (255 de 305) atingindo entre 80-100% de cobertura. Esse resultado indica que uma parte significativa do código é amplamente testada, proporcionando uma base sólida para realizar refatorações com segurança. Contudo, ainda existem 50 arquivos com cobertura inferior a 80%, dos quais 18 não possuem nenhum tipo de teste.

Os arquivos com 0% de cobertura representam um ponto crítico no sistema, pois qualquer alteração nesses trechos apresenta um risco significativo de introduzir erros que não serão identificados por testes automatizados. Isso indica as regiões do código que não devem ser alteradas em uma refatoração sem que antes sejam adicionados testes para cobri-las. No entanto, como o objetivo desse trabalho não é aumentar a cobertura de testes, mas sim focar na refatoração em si, pontos de refatoração nesses arquivos serão desconsiderados. Por outro lado, os arquivos que possuem alguma cobertura, mas ainda inferior a 80%, fornecem um nível de segurança parcial, que pode não ser suficiente para garantir total confiança no comportamento do sistema após alterações. Assim, a inclusão desses arquivos no escopo da refatoração dependerá de sua criticidade no projeto e da qualidade dessa cobertura existente.

2.2 Seleção dos Pontos de Refatoração

2.2.1 Critérios de Seleção

A seleção dos pontos de refatoração para este estudo de caso foi fundamentada nos resultados das análises realizadas na Seção 2.1.1. Conforme discutido, os critérios principais de priorização incluíram a gravidade dos problemas, a qualidade do código em relação aos atributos *clean code* e a cobertura de testes associada aos trechos de código problemáticos. Problemas classificados como de alta gravidade, como *bugs* e *code smells* críticos, foram

priorizados, já que apresentam os maiores riscos para a sustentabilidade do sistema. Por fim, arquivos com nenhuma cobertura de testes foram descartados como candidatos à refatoração, uma vez que alterações nesses trechos sem a segurança de testes introduziriam riscos consideráveis.

Com base nesses critérios, abaixo serão detalhados os problemas e trechos selecionados para refatoração, assim como alguns candidatos que foram descartados, justificando essas escolhas com base nas métricas discutidas e nos objetivos de melhoria da manutenibilidade do código. O foco principal é facilitar futuras alterações e contribuições de outros desenvolvedores, promovendo um sistema compreensível, consistente e sustentável.

2.2.2 Pontos de Refatoração Selecionados

Os problemas selecionados para refatoração serão apresentados seguindo a estrutura abaixo, organizada de forma a facilitar a compreensão das regras violadas e das especificidades de cada ponto de refatoração:

- **Regra do SonarQube violada:** Os focos de refatoração serão agrupados com base na regra violada que classificou o trecho de código como um problema, como complexidade muito alta ou duplicação de código. Para cada regra e problema resultante, serão abordados:
  - A descrição da regra verificada pelo SonarQube, com explicação baseada na documentação oficial.
  - O problema identificado como resultado da violação da regra.
  - Os atributos de *clean code* e qualidade de software impactados, com uma análise de como o problema afeta esses atributos.
- **Especificidades únicas:** Para cada ponto de refatoração, serão detalhadas as características específicas, como:
  - Gravidade do problema detectado.
  - Explicação do trecho do código e como se relaciona com o problema identificado

Problema 1: Complexidade Cognitiva Crítica

Complexidade cognitiva é uma medida que avalia o quão difícil é entender a sequência lógica de decisões e ações de um determinado trecho de código. Códigos com alta complexidade cognitiva são difíceis de ler, entender, testar e modificar, tornando-os menos manuteníveis. O SonarQube classifica um trecho de código, geralmente uma função, como tendo uma complexidade muito alta quando seu valor excede 15 pontos, que é o limite permitido pela ferramenta (SonarSource, 2024).

A pontuação para complexidade cognitiva é calculada com base em três regras principais:

- Ignora estruturas que permitem múltiplas declarações serem expressas compactadas em apenas uma.

- Incrementa um ponto para cada vez que o código quebra o fluxo de leitura linear, como, por exemplo, ao adicionar uma estrutura de loop, uma condicional ou um `try/catch`.
- Incrementa um ponto para cada camada de aninhamento em estruturas que quebram o fluxo do código, já que quanto mais camadas são adicionadas, mais difícil se torna de manter o contexto delas em mente.

O problema causado pelo excesso de complexidade cognitiva é um *code smell* diretamente relacionado com o atributo de *clean code* conhecido como adaptabilidade, afetando principalmente o aspecto de especificidade do código, que refere-se à necessidade de que trechos de código possuam um único e conciso propósito dentro de um escopo bem definido. A ausência de adaptabilidade impacta negativamente a manutenibilidade do software, pois a alta complexidade torna o código mais difícil de ser compreendido, aumentando o esforço necessário para sua manutenção e evolução.

Complexidade Cognitiva com 70 Pontos na Função __parse_results__()

O primeiro foco de refatoração selecionado está no arquivo `lava.py`, especificamente na função `__parse_results__()` da classe `Backend`. Essa função processa os resultados de um trabalho de teste (*test job*) para extrair informações relevantes, como metadados, resultados de testes e métricas. Além disso, lida com logs associados, identifica falhas relacionadas à infraestrutura e resultados de testes, e realiza ações corretivas, como reenvio automático de trabalhos em casos específicos. Por fim, a função retorna um conjunto estruturado de dados que inclui o status do trabalho, os resultados processados, as métricas geradas e os logs refinados, fornecendo uma visão do trabalho analisado.

Mesmo em uma explicação resumida da função, é possível identificar o acúmulo de responsabilidades que ela apresenta. Essa ausência de especificidade se reflete no uso de múltiplas estruturas condicionais, loops e um bloco de tratamento de exceções `try/catch` tudo de forma aninhada, o que resulta em uma complexidade cognitiva de 70, excedendo significativamente o limite de 15 definido pelo SonarQube. Esse valor extremamente alto caracteriza um *code smell* de gravidade alta, considerado crítico para a qualidade do software, pois afeta diretamente a compreensão da implementação da função e dificulta a realização de quaisquer mudanças.

Essa função apresenta uma cobertura de testes de 97%, o que é um forte indicativo de que seria seguro refatorá-la, já que é muito provável que eventuais erros introduzidos sejam identificados por algum dos testes existentes. Contudo, além da cobertura, também foi necessário avaliar a qualidade dos testes. Ao todo, há 14 testes no arquivo `test_lava.py` que validam especificamente a função `__parse_results__()`. Esses testes cobrem diversos cenários e configurações, garantindo a confiabilidade da lógica implementada na função. A seguir, um resumo dos principais casos validados:

- `test_parse_results_metadata()` e `test_parse_results_empty_metadata()`: Esses testes verificam a correta extração dos metadados do trabalho. O primeiro garante que o `metadata` retornado é fiel à definição do trabalho, enquanto o segundo valida que um `metadata` vazio é retornado caso a definição do trabalho não contenha essa propriedade.

- `test_parse_results_metadata_with_suite_versions()`: Avalia a lógica que detecta e retorna corretamente as versões das `suite_versions` presentes nos metadados do trabalho.
- `test_parse_results_ignore_lava_suite_backend_settings()` e test_parse_results_ignore_lava_suite_project_settings(): Validam os resultados (`results`) e métricas (`metrics`) são gerados corretamente para o caso em que a variável `CI_LAVA_HANDLE_SUITE` é definida como verdadeira na configuração do backend ou projeto.
- `test_parse_results_ignore_lava_suite_empty_project_settings()`: Confirma que a lógica da função funciona corretamente quando as configurações do projeto estão vazias.
- `test_parse_results_ignore_lava_suite_project_settings_overwrites_backend()`: Garante que as configurações do projeto têm prioridade sobre as configurações do backend, retornando os resultados e métricas corretos para esse caso
- `test_parse_results_ignore_lava_boot()`: Avalia que a lógica relacionada ao "boot"do LAVA é ignorada quando sua variável não é definida como verdadeira na configuração do backend.
- `test_parse_results_ignore_infra_errors()` e `test_parse_results_dont_ignore_infra_errors()`: Verificam a lógica que decide se um erro de infraestrutura deve indicar que o trabalho não foi completado, dependendo da configuração da variável `CI_LAVA_WORK_AROUND_INFRA_ERRORS`.
- `test_parse_results_handle_lava_suite_and_ignore_lava_boot()`: Testa o caso em que o backend define nas sua configuração `CI_LAVA_HANDLE_SUITE` como verdadeira e `CI_LAVA_HANDLE_BOOT` como falsa, avaliando o impacto dessa combinação nas métricas e resultados.
- `test_parse_results()`: Garante o funcionamento para um caso geral, definindo o `log` dos resultados e o valor das métricas corretamente, conforme o esperado.
- `test_parse_results_rest()`: Testa o comportamento da função quando um dos resultados cai no bloco de exceção, verificando se o tratamento do erro é realizado adequadamente.
- `test_parse_results_clone_measurements()`: Valida um último caso em que a variável `CI_LAVA_CLONE_MEASUREMENTS` é verdadeira nas configurações do projeto, garantindo que todo teste que possui métrica também é adicionado ao retorno dos resultados.

De modo geral, o conjunto de testes do `__parse_results__()` fornecem uma validação confiável para a maioria das condições presentes na função, exceto por dois casos: (1) quando uma das chaves do `metadata` corresponde à `testsuite`, e (2) a manipulação dos logs, que não também não é diretamente testada. Apesar dessas lacunas, os testes existentes cobrem amplamente os cenários críticos, reduzindo significativamente os riscos de refatoração.

A refatoração dessa função foi priorizada como um bom foco de refatoração devido à gravidade do problema identificado, já que ela apresenta a maior complexidade dentre

todas as analisadas. Além disso, a combinação da alta cobertura com a boa qualidade dos testes permite que as alterações sejam realizadas com segurança e confiança. Apenas será necessário ter atenção especial aos casos específicos não testados e considerar a adição de novos testes após a refatoração para garantir a total confiabilidade das mudanças implementadas.

2.3 Implementação das Refatorações

Conforme descrito na Tabela 2.1, o processo de refatoração abordado nesta seção será dividido em duas etapas principais: a definição de estratégias de refatoração e a implementação das mudanças.

Na etapa de definição de estratégias, é realizada uma análise detalhada de cada trecho do código selecionado, com o objetivo de compreender suas funcionalidades, identificar padrões de refatoração adequados e decidir quais alterações poderiam ser aplicadas. Essa etapa é essencial para garantir que as mudanças planejadas sejam alinhadas com os objetivos da refatoração, que envolvem melhoria de clareza, manutenibilidade e adaptabilidade do código.

A etapa de implementação é onde as alterações no código, definidas na fase anterior, são de fato aplicadas. Elas são apresentadas de forma detalhada, mostrando tanto o estado original quanto o estado final de cada trecho modificado. Os testes existentes serão utilizados para validar as mudanças, e novos testes adicionados quando necessário, a fim de garantir a segurança da refatoração. Por fim, serão analisadas as melhorias obtidas, avaliando como elas contribuem para a qualidade do código e para os objetivos estabelecidos.

2.3.1 Refatoração do Trecho de Código com Complexidade Cognitiva Crítica

O trecho de código selecionado para refatoração devido sua complexidade é a função `__parse_results__()` da classe `Backend` no arquivo `lava.py`.

Detalhamento da função e sua complexidade

Conforme comentado anteriormente, a função `__parse_results__()` apresenta um acúmulo de responsabilidades que resulta em uma complexidade elevada, dificultando sua compreensão e manutenibilidade. Para resolver esse problema, é essencial entender detalhadamente como ela funciona e identificar os pontos responsáveis por essa complexidade. Ao fim, esse detalhamento fornecerá a base necessária para definir as estratégias de refatoração.

O seguinte trecho de código é o primeiro ponto onde ocorre uma quebra do fluxo de leitura linear da função:

```
1  def __parse_results__(self, data, test_job, raw_logs):
2  ...
3     suite_versions = {}
4     for key, value in job_metadata.items():
5        if key.endswith('__version'):
6        suite_versions[key.replace('__version', '')] = value
7     if suite_versions:
8        job_metadata['suite_versions'] = suite_versions
9  ...
```

Este bloco de código tem como objetivo identificar todas as versões de `suite_versions` presentes nos metadados do trabalho (`job_metadata`) recebido. Ele percorre cada chave presente no dicionário `job_metadata`, verifica se a chave termina com o sufixo `__version` e, caso positivo, adiciona ao dicionário `suite_versions` o valor correspondente, com a chave modificada para remover o sufixo. Por fim, se o dicionário `suite_versions` não estiver vazio, ele é adicionado aos metadados do trabalho. De modo geral, este trecho agrupa as versões de suites em um único item, refinando o metadado recebido do trabalho.

Apesar de pequeno, este trecho contribui para a complexidade da função com os seguintes elementos e suas respectivas pontuações de complexidade cognitiva calculados pelo SonarQube:

- Introdução de um loop para percorrer as chaves do dicionário: +1
- Condicional dentro do loop para verificar se termina com o sufixo `__version` (quebra de fluxo e aninhamento): +2
- Condicional para verificar se o dicionário `suite_versions` está vazio anter de adicioná-lo aos metadados: +1

Em seguida, há uma verificação para determinar se o trabalho de teste foi cancelado. Caso positivo, ele é marcado como incompleto, e o processamento dos resultados é encerrado, considerando todos os testes como incompletos e descartando qualquer resultado.

```
1  if data[status_key] == 'Canceled':
2     completed = False
```

Esta verificação inicial é simples, mas adiciona uma ramificação lógica ao fluxo da função, contribuindo com +1 ponto na complexidade cognitiva total da função. Caso o trabalho não tenha sido cancelado, o processamento entra no bloco mais complexo de toda a função, que envolve a iteração sobre os resultados dos testes, processamento de métricas e resultados, manipulação de logs e tratamento de falhas.

Iteração sobre os resultados dos testes

Após a verificação do status do trabalho, a função entra em um loop principal que percorre os resultados dos testes presentes na variável `data['results']`. Este bloco é

responsável por processar cada resultado individualmente, lidando com cada condição específica, como resultados relacionados a "boot", existência de falha ou a presença de métricas. A complexidade desse trecho está principalmente no número excessivo de ramificações lógicas e na manipulação de múltiplos tipos de dados. Para uma análise clara, o loop será dividido em subblocos de codigo com base em suas responsabilidades e estruturas lógicas.

Lógica do processamento dos resultados

O primeiro subbloco dentro do loop é responsável por processar os resultados e métricas que não pertencem à suite `lava`, a menos que o projeto esteja configurado para tratar explicitamente esses casos. O trecho de código correspondente é mostrado a seguir:

```
1   for result in data['results']:
2      if handle_lava_suite or result['suite'] != 'lava':
3         suite = result['suite'].split("_", 1)[-1]
4         res_name = "%s/%s" % (suite, result['name'])
5         res_log = None
6         if 'log_start_line' in result.keys():
7            log_url = f"{self.job_url(test_job)}#L{result['log_start_line']}"
8            res_log = f"Testcase log: <a href='{log_url}'>{log_url}</a>\n"
9            if 'log_end_line' in result.keys() and \
10                result['log_start_line'] is not None and \
11                result['log_end_line'] is not None:
12               res_log += self.__download_test_log__(raw_logs, result['
                     log_start_line'], result['log_end_line'])
13        ...
```

Este trecho verifica se o resultado deve ser processado com base na configuração da variável `handle_lava_suite` ou se a `suite` do resultado não é `lava`. Caso a condição seja verdadeira, o código define o valor da chave (`res_name`), que corresponderá ao nome do resultado do teste com base na suite e no nome do teste. Em seguida, há uma lógica específica para processar o log dos casos que o possuem. Para este tratamento:

- Caso possua apenas a linha inicial, o log resultante é apenas um link em formato HTML contendo a url do trabalho concatenada com a linha inicial, que provavelmente pode ser acessado para visualizar o conteúdo detalhado do log diretamente.
- Caso tanto a linha inicial quanto a linha final do log sejam válidas (`log_start_line` e `log_end_line`), o log final é o link HTML mais o resultado da função `__download_test_log__()`, responsável por extrai ra parte do log geral (`raw_logs`) que corresponde ao teste específico.

Apesar deste trecho tratar apenas de parte do processamento de resultados (neste caso, a lógica dos logs), ele gera um total de 6 ramificações lógicas. Esses pontos de quebra de fluxo do código contribuem significativamente para a complexidade da função com as seguintes pontuações:

- Loop que itera sobre os resultados: +2
- Condicional que verifica se o resultado deve ser processado dentro desse bloco: +3

- Condicional para verificar se há log: +4
- Condicional adicional para verificar se o log está dentro do `raw_logs`: +5

É importante ressaltar que cada ponto já possui, inicialmente, um ponto de complexidade devido ao fato de estar aninhado dentro da condicional que verifica se o trabalho foi cancelado, mencionada anteriormente. O resultado é uma excessiva complexidade cognitiva de 14 pontos em um trecho de código com apenas 12 linhas. Além disso, a função auxiliar `__download_test_log__()`, por mais que seja essencial para extrair complexidade e responsabilidades da função principal, possui um nome inadequado para sua funcionalidade, já que seu nome sugere que o log está sendo baixado de alguma fonte externa, quando, na verdade, as linhas estão sendo extraídas de um log já definido. Isso reduz ainda mais a clareza e dificulta a compreensão do código.

Apenas após o processamento dos logs é realizado de fato o processamento dos resultados em si. Essa etapa é dividida em dois casos:

1. **Resultados sem medições**: Um objeto contendo o log (`res_log`) e o valor do resultado do teste (*pass* ou *fail*) é associado ao seu nome (`res_name`) e adicionado à lista de resultados (`results`).
2. **Resultados com medições:** Inicia-se o processamento da medição.
    - Primeiro, o código tenta acessar o campo `unit` de `result`. Se a chave não estiver presente, uma exceção `KeyError` é tratada, e o código tenta recuperar a unidade de um campo alternativo (`units`). Caso nenhuma unidade esteja disponível, é usado o valor padrão `'items'`.
    - A unidade e o valor da medição (`result['measurement']`) são então associados ao nome do resultado do teste (`res_name`) e adicionados à lista de medições (`metrics`).
    - Por fim, há uma última verificação baseada na variável `clone_measurements_to_tests`. Caso ela esteja definida e seja verdadeira, o valor do resultado do teste (`result['result']`) e seu nome (`res_name`) são adicionados também à lista de resultados (`results`).

O trecho de código que realiza essas operações é apresentado a seguir:

```
1   for result in data['results']:
2      if handle_lava_suite or result['suite'] != 'lava':
3      ...
4         if result['measurement'] == 'None' or \
5               result['measurement'] is None:
6            res_value = result['result']
7            results.update({res_name: {'result': res_value, 'log': res_log}})
8         else:
9            res_value = result['measurement']
10           try:
11              unit = result['unit']
12           except KeyError:
13              unit = result.get('units', 'items')
14           metrics.update({res_name: {'value': float(res_value), 'unit': unit}})

15           if clone_measurements_to_tests:
16              res_value = result['result']
17              results.update({res_name: res_value})
```

Esse bloco, que já está dentro da terceira camada de aninhamento, adiciona diversos fatores à complexidade cognitiva da função. Os pontos mais relevantes que contribuem para essa complexidade são os seguintes:

- Ramificação para tratamento de resultados sem medições: +4
- Identificação de exceção lançada dentro dos casos que possuem medição: +5
- Condicional para verificar `clone_measurements_to_tests`: +5

Caso o resultado não se enquadre nas condições da primeira verificação do loop, mas cumpra os requisitos para ser considerado um resultado de teste de boot artificial, o seguinte subbloco é executado:

```
1   for result in data['results']:
2   ...
3      elif result['name'] == 'auto-login-action' and handle_lava_boot:
4         boot = "boot-%s" % test_job.name
5         res_name = "%s/%s" % (boot, definition['device_type'])
6         res_time_name = "%s/time-%s" % (boot, definition['device_type'])
7         if 'testsuite' in job_metadata.keys():
8            res_name = "%s-%s" % (res_name, job_metadata['testsuite'])
9         try:
10           unit = result['unit']
11        except KeyError:
12           unit = result.get('units', 'items')
13        results.update({res_name: result['result']})
14        metrics.update({res_time_name: {'value': float(result['measurement']),
              'unit': unit}})
15  ...
```

Para esses casos, o resultado é adicionado a ambas as listas de processamento: resultados

(`results`) e métricas (`metrics`). Para cada uma delas, ele é tratado de forma específica:

- **Lista de Resultados (`results`):**
  - O nome do resultado (`res_name`) é construído unindo prefiro `"boot"`, nome do trabalho de teste (`test_job.name`), tipo de dispositivo (`definition['device_type']`) e, caso o metadado do trabalho (`job_metadata` contenha a chave `testsuite`, o valor dessa chave também é incluído.
  - O valor associado ao nome do resultado é o próprio resultado do teste (`result['result']`).
- **Lista de Métricas (`metrics`):**
  - O nome da chave da métrica (`res_time_name`) é semelhante ao nome do resultado, mas a chave `testsuite` não é incluída em nenhum caso e o sufixo `"time"` também é adicionado.
  - O valor associado ao nome inclui a métrica medida (`result['measurement']`), convertida para `float`, e a unidade da métrica (`result['unit']`), obtida com o mesmo tratamento de unidade citado anteriormente.

Por fim, o último bloco dentro do loop é responsável por processar resultados da suite lava com o nome *job* que falharam.

```
1   for result in data['results']:
2   ...
3      if result['suite'] == 'lava' and \
4            result['name'] == 'job' and \
5            result['result'] == 'fail':
6         metadata = result['metadata']
7         if isinstance(metadata, str):
8            metadata = yaml.safe_load(metadata)
9         test_job.failure = str(metadata)
10        test_job.save()
11        error_type = metadata.get('error_type', None)
12        if error_type in ['Infrastructure', 'Job', 'Lava']:
13           if not ignore_infra_errors:
14              completed = False
15        if error_type in ['Infrastructure', 'Job', 'Test']:
16           self.__resubmit_job__(test_job, metadata)
```

Esse bloco adiciona uma nova camada de lógica ao loop principal, que contém mais três ramificações responsáveis pelo tratamento de cada caso específico:

- Caso o metadado do resultado seja uma string, ele é convertido para o formato YAML antes de ser salvo no `test_job.failure`.
- Se o tipo do erro for classificado como `'Infrastructure'`, `'Job'` ou `'Lava'` e a variável `ignore_infra_errors` não for definida ou for falsa, o tarbalho é marcado como incompleto.

- Caso o tipo do erro seja classifido como `'Infrastructure'`, `'Job'` ou `'Test'`, o trabalho é reenviado automaticamente para o serviço do backend utilizando o método `__resubmit_job__`.

Após a conclusão da iteração por todos os resultados, o loop é finalizado, e o processamento dos resultados é finalizado. A função retorna então um conjunto de informações estruturadas, incluindo:

- O status do trabalho.
- A informação se o trabalho foi completado.
- Os metadados processados.
- Os resultados e métricas coletados e tratados.
- O log geral processado pelo método `__parse_log__`.

Definição de estratégias

A análise detalhada do código revelou os pontos que contêm excesso de lógica e ramificações que tornam a função difícil de compreender e manter. Esses problemas evidenciam a necessidade de refatoração, que será orientada pelas estratégias propostas no livro Refactoring: Improving the Design of Existing Code, de Martin Fowler (segunda edição). O processo de seleção das estratégias foi baseado em todos os passos anteriores desse estudo:

- **Identificação dos Problemas:** Os problemas foram inicialmente detectados e priorizados com o auxílio do SonarQube, considerando métricas como atributos de *clean code* e qualidade de software, tipos de problemas e gravidade.
- **Entendimento do Código e dos Problemas**: A análise detalhada da função problemática selecionada (`__parse_results__()`), permitiu identificar as causas dos problemas reportados pelo SonarQube e entender como as características do código contribuíam para esses problemas.

Nesta seção, utilizando o catálogo de estratégias descritas por Martin Fowler, serão escolhidas aquelas cujas motivações e objetivos se alinham aos problemas identificados. Dessa forma, as estratégias de refatoração escolhidas são motivadas pelos problemas detectados e serão aplicadas com base nas necessidades específicas de cada trecho de código analisado.

1. **Extrair Função (*Extract Function*):** Essa estratégia será utilizada para dividir a função `__parse_results__()` em funções menores e mais específicas, cada uma responsável por uma parte do processamento dos resultados. Isso reduzirá a complexidade da função principal, aumentar a modularidade e reduzir o acúmulo de responsabilidade existente, facilitando a compreensão e manutenção do código.
2. **Combinar Funções em Classes (*Combine Functions into Classes*):** Essa estratégia será aplicada para agrupar as funções resultantes da estratégia anterior, que são relacionadas em uma nova classe, de forma a organizar melhor o código e agrupar

responsabilidades semelhantes. Isso permitirá uma melhor estruturação do código e facilitará a compreensão e manutenção do sistema.

3. **Decompor Condicional (*Decompose Conditional*):** Essa estratégia será utilizada para simplificar as condicionais complexas presentes na função, dividindo-as em partes menores e mais simples. Isso também reduzirá a complexidade da função, tornando-a mais legível.

Implementação das Mudanças

A primeira estratégia de refatoração aplicada foi a de extrair função. A função `__parse_results__()` foi extraída nas seguintes funções:

- **Função para extrair os dados relacionados aos metadados do trabalho**

```
1   def __extract_job_metadata__(self, definition):
2      job_metadata = definition.get('metadata', {})
3
4      suite_versions = {
5         key.replace('__version', ''): value
6         for key, value in job_metadata.items()
7         if key.endswith('__version')
8      }
9
10     if suite_versions:
11        job_metadata['suite_versions'] = suite_versions
12
13     return job_metadata
```

- **Função para pegar configurações do projeto**

```
1   def __get_project_settings__(self, test_job):
2      settings = self.__resolve_settings__(test_job)
3      handle_lava_suite = settings.get('CI_LAVA_HANDLE_SUITE', False)
4      handle_lava_boot = settings.get('CI_LAVA_HANDLE_BOOT', False)
5      clone_measurements_to_tests = settings.get('CI_LAVA_CLONE_MEASUREMENTS',
           False)
6      ignore_infra_errors = settings.get('CI_LAVA_WORK_AROUND_INFRA_ERRORS',
           False)
7
8      return {
9         'handle_lava_suite': handle_lava_suite,
10        'handle_lava_boot': handle_lava_boot,
11        'clone_measurements_to_tests': clone_measurements_to_tests,
12        'ignore_infra_errors': ignore_infra_errors
13     }
```

- **Função para processar o log do teste**

```
1   def __get_log__(self, result, test_job, raw_logs):
2      log = None
3      if 'log_start_line' in result.keys():
4         log_url = f"{self.job_url(test_job)}#L{result['log_start_line']}"
5         log = f"Testcase log: <a href='{log_url}'>{log_url}</a>\n"
6         if 'log_end_line' in result.keys() and \
7               result['log_start_line'] is not None and \
8               result['log_end_line'] is not None:
9            log += self.__download_test_log__(raw_logs, result['log_start_line'],
                  result['log_end_line'])
10
11     return log
```

- **Função para processar resultados de boot**

```
1   def _parse_boot_result(self, result, definition, job_metadata):
2      result_data = {}
3      metric_data = {}
4
5      boot = "boot-%s" % self.test_job_name
6      res_name = "%s/%s" % (boot, definition['device_type'])
7      res_time_name = "%s/time-%s" % (boot, definition['device_type'])
8      if 'testsuite' in job_metadata.keys():
9         res_name = "%s-%s" % (res_name, job_metadata['testsuite'])
10     try:
11        unit = result['unit']
12     except KeyError:
13        unit = result.get('units', 'items')
14     result_data[res_name] = result['result']
15     metric_data[res_time_name] = {'value': float(result['measurement']), 'unit
          ': unit}
16
17     return result_data, metric_data
```

- **Função para processar resultado padrão**

```
1   def _parse_result_data(self, result, res_log):
2      result_data = {}
3      metric_data = {}
4
5      suite = result['suite'].split("_", 1)[-1]
6      res_name = "%s/%s" % (suite, result['name'])
7      if result['measurement'] == 'None' or result['measurement'] is None:
8         res_value = result['result']
9         result_data[res_name] = {'result': res_value, 'log': res_log}
10     else:
11        res_value = result['measurement']
12        try:
13           unit = result['unit']
14        except KeyError:
15           unit = result.get('units', 'items')
16        metric_data[res_name] = {'value': float(res_value), 'unit': unit}
17        if self.project_settings['clone_measurements_to_tests']:
18           res_value = result['result']
19           result_data[res_name] = res_value
20
21     return result_data, metric_data
```

- **Função para realizar o devido processamento dependendo do tipo do resultado**

```
1   def parse_result(self, result, definition, job_metadata, res_log):
2      if self.project_settings.handle_lava_suite or result['suite'] != 'lava':
3         return self._parse_result_data(result, res_log)
4
5      if 'login-action' in result['name'] and self.project_settings['
           handle_lava_boot']:
6         return self._parse_boot_result(result, definition, job_metadata)
7
8      return {}, {}
```

- **Função para tratar erros em testes**

```
1   def __handle_lava_job_failure_result__(self, result, test_job,
        ignore_infra_errors, completed):
2      metadata = result['metadata']
3      if isinstance(metadata, str):
4         metadata = yaml.safe_load(metadata)
5      test_job.failure = str(metadata)
6      test_job.save()
7      error_type = metadata.get('error_type', None)
8
9      if error_type in ['Infrastructure', 'Job', 'Lava'] and not
           ignore_infra_errors:
10        completed = False
11     if error_type in ['Infrastructure', 'Job', 'Test']:
12        self.__resubmit_job__(test_job, metadata)
13
14     return test_job, completed
```

Após extrair as funções, é possível perceber que `__parse_boot_result__()`, `__parse_result_data__()` e `__parse_result__()` possuem comportamentos que se relacionam com um mesmo objetivo: processar os resultados dos testes. Dessa forma, elas podem ser combinadas em uma nova classe dedicada a esse objetivo, implementando uma segunda estratégia de refatoração: combinar funções em classes.

A classe criada é *JobResultParser*, que possui como atributo o nome do teste (*test_job_name*) e as configurações do projeto (*project_settings*), definidos na inicialização da classe.

```
1   class JobResultParser:
2      def __init__(self, test_job_name, project_settings):
3         self.test_job_name = test_job_name
4         self.project_settings = project_settings
5
6      def parse_result(self, result, definition, job_metadata, res_log):
7         ...
8
9      def _parse_boot_result(self, result, definition, job_metadata):
10        ...
11
12     def _parse_result_data(self, result, res_log):
13        ...
```

O passo seguinte, e último, foi aplicar a estratégia de decompor as condicionais da função `__parse_result__()` em dois métodos diferentes: *_is_suite_processable* e *_is_boot_login_action*.

```
class JobResultParser:
   def parse_result(self, result, definition, job_metadata, res_log):
      if self._is_suite_processable(result):
         return self._parse_result_data(result, res_log)

      if self._is_boot_login_action(result):
         # add artificial 'boot' test result for each test job
         # by default the boot test is named after the device_type
         return self._parse_boot_result(result, definition, job_metadata)

      return {}, {}

   ...

   def _is_boot_login_action(self, result):
      return 'login-action' in result['name'] and self.project_settings['
          handle_lava_boot']

   def _is_suite_processable(self, result):
      return self.project_settings['handle_lava_suite'] or result['suite'] !=
           'lava'
```

2.4 Validação e Resultados

Após a implementação das refatorações discutidas na Seção 2.2, foi realizada uma validação para garantir que as mudanças não introduziram regressões no sistema e que houve melhorias nos atributos de qualidade previamente identificados. Essa validação foi conduzida através da execução dos testes automatizados, análise de métricas geradas pelo SonarQube e comparação dos resultados antes e depois das refatorações.

2.4.1 Execução dos Testes Automatizados

Para verificar se a refatoração manteve o comportamento esperado do sistema, os testes automatizados existentes foram executados antes e após as modificações. A Tabela 2.3 apresenta um resumo dos resultados obtidos:

Métrica	Antes da Refatoração	Após a Refatoração
Total de testes executados	890	890
Testes bem-sucedidos	886	886
Testes falhos	4	4
Cobertura de código no arquivo alterado (%)	70%	72%

Tabela 2.3: *Comparação dos resultados dos testes antes e depois da refatoração*

Os resultados mostram que todos os testes continuaram passando corretamente após a refatoração, garantindo que não houve quebra de funcionalidade. Além disso, a cobertura de código permaneceu estável, indicando que os testes existentes foram suficientes para validar as alterações.

2.4.2 Análise das Métricas do SonarQube

Para avaliar os impactos das refatorações na qualidade do código, foi realizada uma nova análise utilizando o SonarQube para comparar as métricas obtidas especificamente no trecho do código refatorado antes e depois das modificações. A Tabela 2.4 apresenta a comparação das principais métricas de qualidade analisadas.

Métrica	Antes da Refatoração	Após a Refatoração
Complexidade Cognitiva	70	<15
Número de linhas da função	92	30

Tabela 2.4: *Comparação das métricas no trecho refatorado antes e depois da refatoração*

A complexidade cognitiva foi reduzida de 70 para menos de 15 pontos, enquanto o número de linhas da função foi reduzido de 92 para 30. Esses resultados sugerem que as refatorações aplicadas foram eficazes em melhorar a qualidade do código.

2.4.3 Discussão dos Resultados

A análise dos resultados demonstra que as refatorações implementadas tiveram um impacto positivo na qualidade do código sem comprometer a funcionalidade do sistema. A redução na complexidade sugere que as modificações facilitarão a manutenção futura e a compreensão do código por novos desenvolvedores.

Entretanto, algumas limitações foram identificadas:

- Apesar da redução de complexidade, algumas funções ainda possuem um nível elevado de aninhamento e podem exigir refatorações adicionais.
- Algumas refatorações exigiram modificações mais extensas, o que pode ter introduzido uma curva de aprendizado para novos contribuidores do projeto.

Capítulo 3

Conclusão

A evolução contínua dos sistemas de *software* torna essencial a adoção de práticas que garantam a qualidade do código, como a refatoração. Neste trabalho, foi analisada a importância da refatoração como um processo fundamental para a melhoria da manutenibilidade, legibilidade e adaptabilidade de sistemas de código aberto. A fundamentação teórica abordou conceitos como *code smells*, métricas de qualidade de código e o uso do SonarQube como ferramenta de apoio na identificação de problemas na base de código.

Para validar os conceitos discutidos, foi conduzido um estudo de caso no sistema *open source* Squad, utilizando o SonarQube para detectar *code smells* e métricas de qualidade de código. A partir da análise das métricas, foi identificado que a função `__parse_results__()` apresentava um nível elevado de complexidade cognitiva, dificultando sua compreensão e manutenção. Como solução, aplicamos técnicas de refatoração baseadas no livro de Fowler 2018, como extrair funções, combinar funções em classes e decompor condicional, ,resultando em um código mais modular e compreensível.

Os resultados da refatoração foram validados por meio de testes automatizados e novas análises no SonarQube. A comparação das métricas antes e depois das modificações demonstrou uma redução na complexidade cognitiva da função analisada, além da eliminação de *code smells* e melhoria na organização do código. Os testes automatizados garantiram que as mudanças não afetaram o comportamento esperado do sistema.

Além de confirmar a eficácia da refatoração para melhorar a qualidade do código, este estudo demonstrou que o uso do SonarQube possibilita contribuições para sistemas *open source* mesmo sem um conhecimento aprofundado da base de código. A ferramenta permitiu identificar trechos críticos que necessitavam de melhorias e ofereceu métricas objetivas para embasar as decisões de refatoração, tornando o processo mais estruturado e assertivo. Dessa forma, o SonarQube se mostrou um recurso valioso para desenvolvedores que desejam colaborar com projetos *open source*, facilitando a navegação e compreensão do código existente.

Apesar das melhorias alcançadas, algumas limitações foram identificadas. Primeiramente, a cobertura de testes, embora suficiente para validar a refatoração, pode ser aprimorada para garantir refatorações em partes do código que estão sem cobertura. Além

disso, algumas funções do Squad ainda apresentam um nível elevado de aninhamento e podem se beneficiar de refatorações futuras.

Dessa forma, como trabalhos futuros, sugere-se:

- Expandir a análise para outras funções críticas do Squad, aplicando novas refatorações e comparando seus impactos;
- Aprimorar a cobertura de testes automatizados para garantir uma validação mais abrangente do sistema;

Com base nos resultados obtidos, conclui-se que a refatoração é um processo essencial para a sustentabilidade de sistemas *open source*, promovendo um código mais claro, organizado e de fácil manutenção. A aplicação de técnicas estruturadas, aliada ao uso de ferramentas como SonarQube, possibilita identificar e corrigir problemas de qualidade, tornando o desenvolvimento de software mais eficiente e menos suscetível a falhas.

Referências

[Aniche 2022] Mauricio Aniche. *Effective Software Testing: A developer's guide*. 1ª ed. Manning, 2022 (citado na pg. 13).

[Fowler 2018] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2ª ed. Addison-Wesley, 2018 (citado nas pgs. 1, 2, 5, 6, 13, 31).

[Linaro 2024] Linaro. *SQUAD — Software Quality Dashboard*. Acessado em: 29 de janeiro de 2025. 2024. url: https://squad.readthedocs.io/en/latest/ (citado nas pgs. 1, 9).

[SonarSource 2024] SonarSource. *SonarQube Server 10.8 Documentation*. Acessado em: 29 de janeiro de 2025. 2024. url: https://docs.sonarsource.com/sonarqube-server/10.8/ (citado nas pgs. 1, 7, 11–13, 15).

[SonarSource 2025] SonarSource. *About SonarSource*. 2025. url: https://www.sonarsource.com/company/about/ (acesso em 29/01/2025) (citado na pg. 1).

[SonarSource s.d.] SonarSource. *What are software bugs?* Acessado em: 29 de janeiro de 2025. url: https://www.sonarsource.com/learn/software-bugs/#:~:text=Software%20bugs%20are%20faults%2C%20flaws,debugging%20tools%2C%20and%20developer%20cooperation (citado na pg. 13).