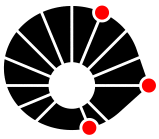

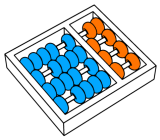

Explorando *Technical Debt* em *code samples*: Uma Análise Detalhada das Implicações para a Manutenibilidade do Código

Rafael Alves *Thaina Oliveira* *Bruno Cafeo*

Relatório Técnico - IC-PFG-24-20
Projeto Final de Graduação
2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Explorando Technical Debt em *code samples*: Uma Análise Detalhada das Implicações para a Manutenibilidade do Código

Rafael Santa Rosa Alves Thaina Milene de Oliveira Bruno Cafeo

Resumo

Contexto: Este trabalho investiga a detecção e a análise de *code smells* e refatorações em projetos de *code samples*, abordando a forma como essas práticas impactam a qualidade do código ao longo do tempo. A presença de *code smells* – características de código que podem indicar problemas de design e comprometer a manutenibilidade – é diretamente relacionada à qualidade de um *software* e à sua capacidade de evolução. Refatorações, por sua vez, são modificações no código que não alteram seu comportamento externo, mas visam melhorar sua estrutura interna, facilitando a manutenção e reduzindo a possibilidade de bugs futuros. **Objetivo:** O objetivo desta pesquisa é compreender a presença, a evolução e a eficácia de *code smells* e refatorações em repositórios de *code samples*. Utiliza-se a abordagem *Go*al Question Metric (GQM) para estruturar o estudo, definindo metas específicas e criando métricas para quantificar o impacto das refatorações na qualidade do código, bem como para analisar como *code smells* mudam ao longo do tempo em resposta a essas refatorações. **Método:** Foram selecionados seis ecossistemas de *software* representativos e amplamente utilizados, como *spring-guides*, *spring-cloud-samples*, *googlesamples*, entre outros, aplicando critérios como número mínimo de contribuidores, LOC entre 500 e 100.000 e atividade recente nos repositórios. A coleta de dados utiliza a API do GitHub para identificação dos repositórios e o *SonarQube* (versão 10.7) junto ao *SonarScanner* (versão 6.2.1) para análise de *code smells* e para identificar refatorações, foi utilizado o *RefactoringMiner*. **Resultados:** Os resultados indicam que *code smells* são frequentes em *code samples*, mas a sua resolução nem sempre é priorizada. Refatorações, por sua vez, apresentam padrões cíclicos e a sua frequência está relacionada com fatores como complexidade do código e colaboração. **Conclusão:** O estudo demonstra a importância de ferramentas e processos para gestão de code smells e refatorações em *code samples*. É crucial promover a qualidade do código e conscientizar os desenvolvedores sobre a importância dessas práticas para garantir a eficácia e a longevidade dos projetos.

1 Introdução

No desenvolvimento de *software*, a identificação de *code smells* e a refatoração de código são práticas fundamentais para a manutenção e melhoria da qualidade de projetos ao longo do tempo. A refatoração consiste em reestruturar o código existente sem modificar seu comportamento externo, com o objetivo de torná-lo mais legível, de fácil manutenção e extensível. Já *code smells* são indícios de que o código pode estar mal projetado ou implementado de

forma inadequada, comprometendo a sua manutenção e dificultando a evolução do *software*. A presença desses sinais não significa erros de execução, mas representa a possibilidade de uma **dívida técnica**, ou seja, uma escolha que, embora possa economizar tempo no curto prazo, acarreta custos mais elevados no futuro.

Para desenvolvedores, *code samples* desempenham um papel essencial para seu aprendizado, principalmente quando disponibilizados por grandes organizações, como *Google*, *Microsoft* e *Amazon*. Esses exemplos, amplamente utilizados para demonstrar o uso de funcionalidades específicas de plataformas, como *Android*, *AWS* e *Azure*, são frequentemente consultados para fins educacionais. No entanto, mesmo os *code samples*, que servem como guias para desenvolvedores, podem conter *code smells* e se beneficiar de refatorações para assegurar sua relevância e qualidade.

Este projeto busca explorar a relação entre a presença de *code smells* e as refatorações realizadas ao longo do tempo em *code samples*. A investigação será conduzida por meio de um estudo longitudinal, com foco em identificar os tipos de *code smells* mais comuns nesses exemplos, acompanhar sua evolução e analisar as refatorações aplicadas. Nessa pesquisa categorizamos e analisamos os *code smells* e as refatorações aplicadas nos *code samples*. A partir desses dados, buscamos entender como as práticas de refatoração podem contribuir para mitigar ou agravar problemas associados aos *code smells* identificados, proporcionando uma visão integrada dos desafios de qualidade presentes nesses projetos de *software*.

2 Fundamentação Teórica

2.1 Ecossistema de *software*

Um ecossistema de *software* é a interação de *software* e atores em torno de uma infraestrutura tecnológica comum, resultando em um conjunto de contribuições que influenciam direta ou indiretamente o ecossistema. A atividade de cada ator é motivada pela criação de valor, que pode ser monetário, como taxas por serviços, ou intangível, como reconhecimento de habilidades ou aquisição de experiência. Essas atividades geralmente geram contribuições, que podem ser técnicas, como componentes, produtos ou sistemas de *software*, ou não técnicas, como dados, marketing, administração ou *Go*vernança.

A infraestrutura tecnológica compartilhada, como uma plataforma, protocolos ou um conjunto de padrões, sustenta as interações entre diferentes componentes de *software* e entre atores variados, como desenvolvedores, usuários, empresas e comunidades. Essas interações são fundamentais para o sucesso e a evolução do ecossistema, já que os atores contribuem com recursos como código, documentação, suporte, serviços e negócios, impulsionando o crescimento contínuo do ecossistema e criando valor de maneira colaborativa.

Os ecossistemas de *software* podem ser classificados com base na tecnologia que compartilham. Entre os tipos mais comuns estão as plataformas, onde as contribuições geralmente são voltadas para o desenvolvimento ou extensão de um *software*, como no caso do *Android* e da *Eclipse Foundation*. Já em ecossistemas baseados em protocolos, a interação entre *softwares* ocorre por meio de um protocolo específico, exigindo que as contribuições sigam rigorosamente as especificações técnicas, como acontece com o *Ruby*. Nos ecossistemas baseados em padrões, a interação ocorre pela conformidade com um ou mais padrões

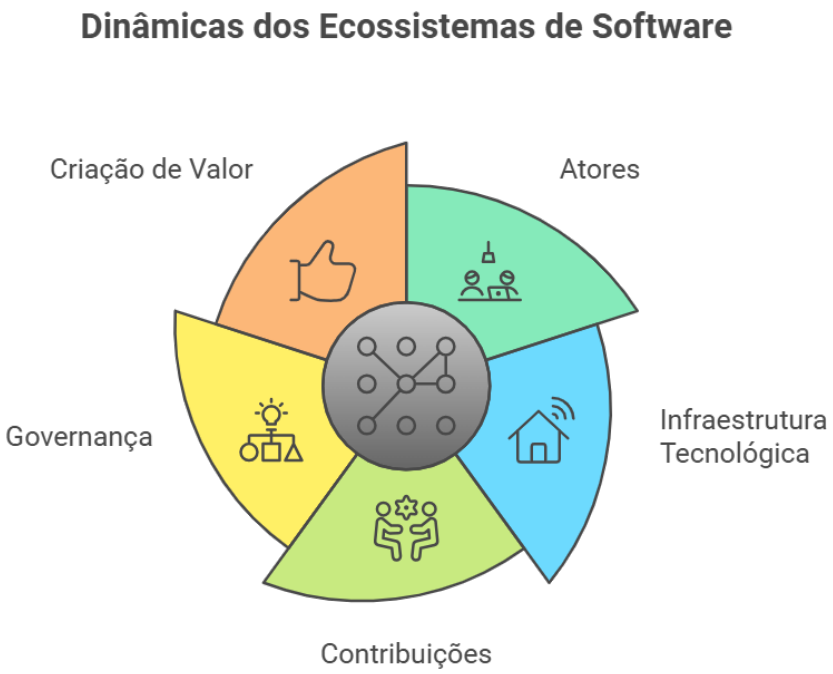

Figura 1: Dinâmica dos Ecossistema de *Software*.

específicos, oferecendo um controle mais rigoroso. Além da conformidade técnica, as contribuições nesse tipo de ecossistema podem abranger outras áreas, com a aderência aos padrões sendo frequentemente avaliada por terceiros, como no *Open Design Alliance*. Em ecossistemas baseados em infraestrutura, a interação ocorre principalmente durante o desenvolvimento, com as contribuições usando uma infraestrutura tecnológica comum, mas mantendo independência e abrangendo diferentes domínios, como é típico em infraestruturas ou ferramentas de desenvolvimento, como *Gnome* ou *Apache Foundation*.

A *Go*vernança tem um papel fundamental na definição do comportamento e na evolução dos ecossistemas de *software* [1]. O orquestrador, que pode ser um único ator ou um grupo, é responsável por promover a interação entre os participantes e o *software*, conduzindo o ecossistema em diferentes estilos de gestão, como monarquia, federal, coletivo ou anarquia, dependendo de como o poder de decisão é distribuído.

Os ecossistemas de *software* tornaram-se essenciais para o desenvolvimento moderno devido a fatores como a reutilização de recursos, que facilita o reaproveitamento de código, componentes e bibliotecas, permitindo que os desenvolvedores construam sobre o trabalho uns dos outros. Ao oferecer uma base de componentes e infraestrutura compartilhada, os ecossistemas aumentam a produtividade e reduzem o tempo necessário para lançar produtos. A natureza colaborativa dos ecossistemas também estimula a inovação, proporcionando um

ambiente onde novas ideias podem ser testadas e ampliando a base de conhecimento comum. Além disso, essa reutilização de recursos e o ganho de produtividade resultam em uma maior eficiência de custos, reduzindo o investimento necessário para o desenvolvimento e aumentando o retorno. Em síntese, os ecossistemas de *software* promovem um ambiente dinâmico para o desenvolvimento, estimulando colaboração, inovação e crescimento dentro da comunidade de *software*.

2.2 *Code Samples*

Os *code samples*, são amostras de código que têm como objetivo facilitar e acelerar o processo de aprendizagem dos recursos oferecidos pelos *frameworks*. Normalmente, eles são disponibilizados por projetos e organizações de *software* ao redor do mundo, como *Android*, *Spring*, *Google Maps*, *Twitter* e *Microsoft*. Esses *code samples* de *frameworks* podem mostrar desde o uso de funcionalidades básicas até recursos mais avançados. Por exemplo, o *framework Spring Boot* fornece *code samples* para ajudar desenvolvedores iniciantes a construir serviços web *RESTful* e a proteger aplicações web. Por sua praticidade, muitos desenvolvedores acabam copiando e colando esses exemplos diretamente em suas bases de código e os levam à produção. Idealmente, os *code samples* devem seguir boas práticas de desenvolvimento, como serem simples, ou seja, fáceis de entender e usar; pequenos, para que permaneçam concisos e focados em uma funcionalidade específica; autocontidos, contendo todos os arquivos necessários para sua execução; fáceis de entender, com documentação adequada e código limpo; seguros, seguindo recomendações de segurança; e eficientes, com um bom desempenho.

Os *code samples* não são projetos estáticos; eles evoluem como qualquer sistema de *software*. Frequentemente são atualizados para acompanhar novas versões de *framework*, mantendo-se relevantes para os usuários. Essas atualizações geralmente envolvem mudanças em arquivos de código-fonte e de configuração, como a atualização da documentação, ajustes em arquivos de configuração e migração para novas versões de *framework*. No entanto, nem todos os *code samples* recebem a mesma atenção, resultando em diferentes níveis de popularidade, atividade e engajamento da comunidade.

Desenvolvedores muitas vezes enfrentam dificuldades ao tentar modificar *code samples*, seja para adicionar novos recursos ou realizar pequenas alterações. Análises de postagens no *Stack Overflow* e problemas no *GitHub* relacionados a *code samples* identificaram alguns desafios recorrentes: problemas de importação, dificuldades de execução devido a erros em tempo de execução, e obstáculos na modificação ou aprimoramento dos exemplos [2].

Algumas implicações para a criação e uso de *code samples* ajudam a orientar as melhores práticas de manutenção e utilização. Em termos de simplicidade e tamanho, os *code samples* devem ser diretos e pequenos para facilitar a reutilização. Ambientes de trabalho também são recomendados para simplificar o uso, juntamente com ferramentas de automação para melhorar a qualidade. Como os *code samples* não são estáticos, é essencial que sejam atualizados ao longo do tempo.

Em resumo, os *code samples* são recursos valiosos para desenvolvedores, oferecendo exemplos práticos que aceleram o aprendizado. Para que esses exemplos sejam realmente eficazes, é essencial que criadores e usuários compreendam suas características, desafios e

as melhores práticas associadas à sua manutenção e uso. Seguindo essas recomendações, os *code samples* podem se tornar ferramentas ainda mais úteis no desenvolvimento de *software* moderno.

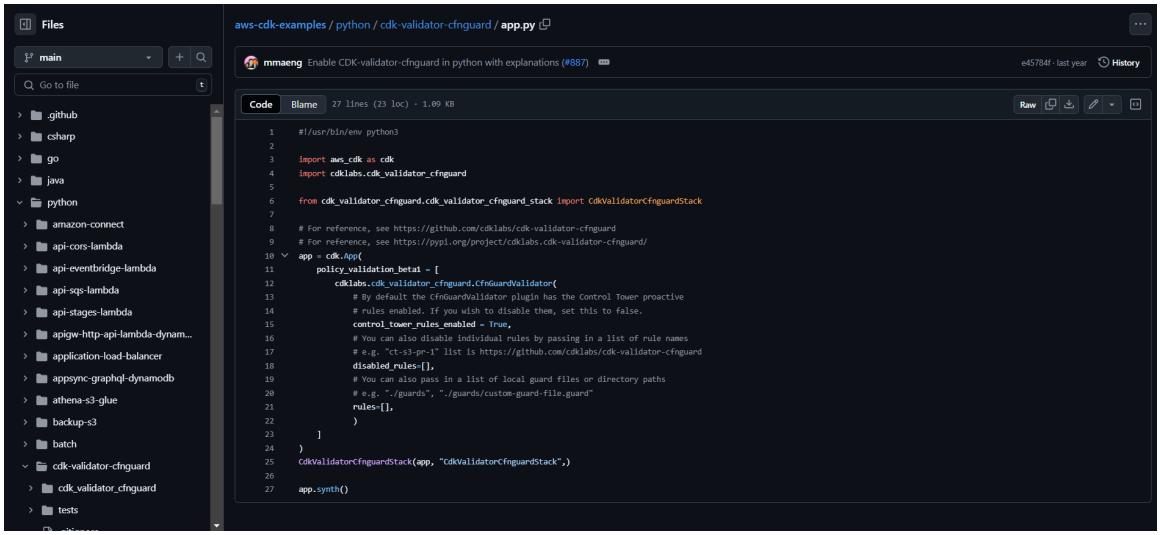

Figura 2: Exemplo de Code Sample do repositório *aws-samples/aws-cdk-examples*

2.3 Débito Técnico

Débito técnico é um conceito que se refere à construção de *software* com deficiências durante o processo de desenvolvimento. O custo de corrigir essas deficiências aumenta com o tempo, de forma semelhante ao pagamento de juros em uma dívida financeira. Quanto mais tempo as falhas permanecem no *software*, mais difícil e caro se torna repará-las, assim como o aumento dos juros em uma dívida monetária quando seu pagamento é postergado. Por isso, é importante medir e gerenciar o débito técnico para evitar ou reduzir o custo desses "juros" [2].

O débito técnico pode surgir no *software* por várias razões, como prazos apertados, falta de conhecimento ou experiência, código mal escrito, ausência de testes e documentação inadequada. No curto prazo, ele pode permitir uma entrega mais rápida ao cliente, mas, a longo prazo, pode gerar consequências negativas, como atrasos, perdas financeiras, fracasso em projetos de desenvolvimento, desmotivação das equipes e degradação da imagem da organização.

Há diferentes tipos de débito técnico, relacionados a artefatos e recursos envolvidos durante o desenvolvimento de *software*: débito de código (código mal escrito, complexo ou duplicado), débito de arquitetura (arquitetura inadequada ou mal concebida), débito de design (design mal planejado), débito de infraestrutura (infraestrutura inadequada ou mal configurada), débito de documentação (documentação inexistente, incompleta ou desatualizada), débito de testes (falta de testes ou testes inadequados), débito de requisitos (requisitos mal definidos ou incompletos), débito de *build* (processo de *build* inadequado ou

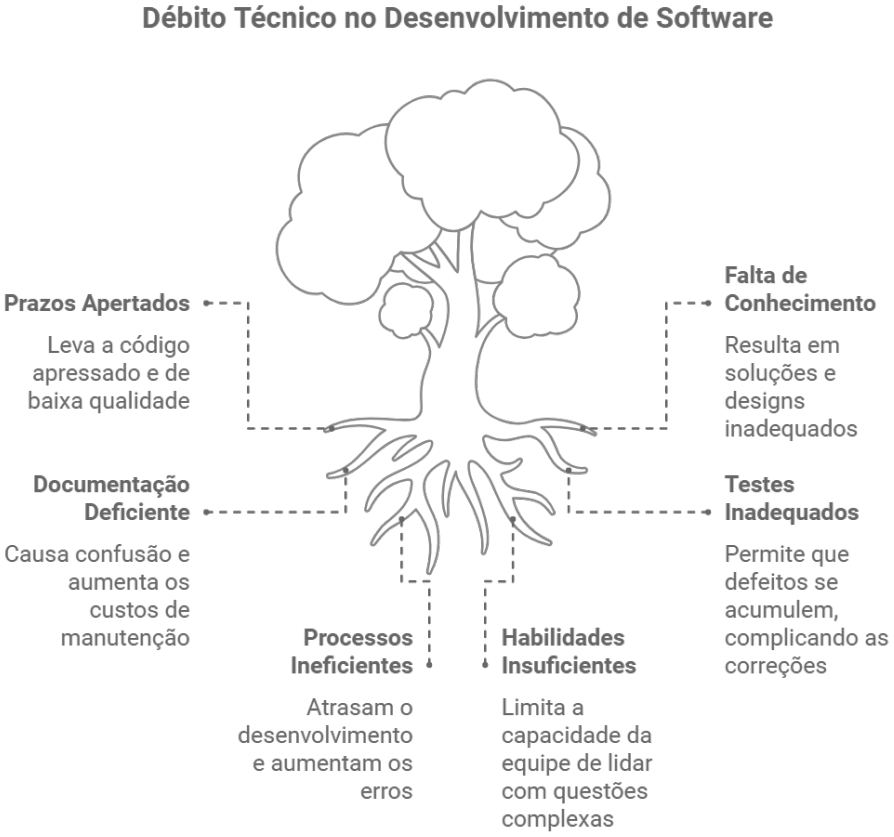

Figura 3: Causas do débito técnico

ineficiente), débito de defeitos (número elevado de defeitos no *software*), débito de pessoal (falta de competências ou experiência na equipe de desenvolvimento), débito de processo (processo de desenvolvimento inadequado ou ineficiente) e débito de uso de serviços web (uso inadequado ou ineficiente de serviços web) [3].

A medição do débito técnico é complexa, pois envolve múltiplos fatores. É preferível que a medição seja automatizada, para não sobrecarregar a equipe de desenvolvimento. Os tipos de débito técnico mais fáceis de medir e para os quais há mais estratégias de gestão são o débito de arquitetura, design e código.

2.4 Code Smells

Code Smells, ou maus cheiros no código, são sinais de problemas de design no código-fonte de um *software*. Eles não representam bugs que impeçam o funcionamento, mas indicam que o código pode ser difícil de entender, manter e evoluir.

Os *code smells* podem surgir por várias razões. A pressão por prazos, por exemplo, pode levar a decisões de design apressadas, enquanto desenvolvedores com menos experiência podem ter dificuldade em aplicar boas práticas. Padrões de design mal implementados também

podem gerar acoplamento excessivo e complexidade desnecessária. Além disso, mudanças frequentes no código sem atenção ao design e a falta de cuidado dos desenvolvedores — como a repetição de código e métodos muito extensos — também contribuem para o surgimento desses problemas.

A presença de *code smells* torna a manutenção do *software* mais difícil, aumenta o risco de introdução de bugs e, a longo prazo, eleva os custos de desenvolvimento, pois a manutenção e correção de problemas demandam mais tempo.

A refatoração é uma solução importante para mitigar os *code smells*. Esse processo permite reorganizar o código-fonte sem alterar seu comportamento, eliminando *code smells* e tornando o código mais legível e fácil de manter. Em projetos grandes, onde muitos *code smells* podem ser encontrados, é inviável corrigir todos de uma vez, então priorizar os problemas mais críticos é fundamental. Para isso, os desenvolvedores devem avaliar o impacto dos *code smells* na qualidade do *software*, a gravidade de cada problema, o esforço necessário para a refatoração e os objetivos específicos do projeto.

Para detectar *code smells*, diversas ferramentas estão disponíveis, cada uma com características que atendem a diferentes tipos de projeto e às necessidades da equipe de desenvolvimento. A escolha da ferramenta ideal facilita a identificação dos pontos críticos e a priorização das refatorações. Em resumo, *code smells* são sinais de problemas de design que, se não tratados, podem comprometer a qualidade, manutenibilidade e evolução do *software*. A refatoração é uma técnica essencial para remover esses maus cheiros e aprimorar o código. Em cenários com muitos *code smells*, a priorização das refatorações ajuda a focar nos problemas mais impactantes, otimizando o uso do tempo e dos recursos da equipe. [4]

Exemplo de *Code Smell* em *Python*

Código com *code smell (dead code)*

```
def calcular_media(notas):
    if not notas:  # Evita erro caso a lista de notas seja vazia
        return "Lista de notas está vazia"
    soma = sum(notas)
    media = soma / len(notas)
    valor_extra = 100  # Variável que nunca é utilizada
    if media >= 5:
        return "Aprovado"
    else:
        return "Reprovado"
```

No código acima, a variável `valor_extra` é declarada, mas nunca utilizada no fluxo do programa. Isso configura um *code smell* do tipo *Dead Code*, pois a variável não tem efeito algum sobre o comportamento da função. O valor atribuído a ela é calculado, mas não é utilizado em nenhuma operação subsequente.

2.5 Refatoração

A refatoração é o processo de reorganizar o código existente sem modificar seu comportamento externo. O objetivo é aprimorar a qualidade interna do *software*, tornando-o mais fácil de entender, manter e evoluir. A prática da refatoração é fundamental para a gestão da dívida técnica, pois, ao reorganizar o código, os desenvolvedores conseguem corrigir deficiências e melhorar a qualidade geral do *software*.

Entre os principais benefícios da refatoração estão a melhoria da legibilidade, a redução da complexidade, a facilidade de manutenção, a melhoria da extensibilidade e o aumento da reusabilidade do código. Em geral, isso resulta em um *software* mais simples de entender, com menos probabilidade de conter erros, mais fácil de modificar e capaz de integrar novas funcionalidades de forma mais fluída.

Para realizar uma refatoração eficiente, é importante que o processo ocorra continuamente ao longo do desenvolvimento. Sinais que indicam a necessidade de refatoração incluem a duplicação de código, a presença de métodos longos e complexos, classes que acumulam várias funções e código mal documentado. A refatoração cuidadosa deve ocorrer em pequenos passos para evitar a introdução de novos erros, e é fundamental testar o código frequentemente para assegurar que o comportamento do *software* permanece inalterado. Utilizar ferramentas de refatoração disponíveis em IDEs modernas ajuda a automatizar o processo, tornando-o mais rápido e seguro.

Ao seguir essas boas práticas, o código torna-se mais robusto, fácil de manter e de alta qualidade, beneficiando tanto a equipe de desenvolvimento quanto o ciclo de vida do *software*.

Exemplo de Refatoração em *Python* (correção do exemplo de *code smell*)

Refatoração do Código

```
def calcular_media(notas):
    if not notas:  # Evita erro caso a lista de notas seja vazia
        return "Lista de notas está vazia"
    soma = sum(notas)
    nota_media = soma / len(notas)

    aprovado = nota_media >= 5  # Variável booleana para tornar a lógica mais clara

    if aprovado:
        return "Aprovado"
    else:
        return "Reprovado"
```

A refatoração proposta resolve o problema do `Dead Code` e melhora a legibilidade e a robustez do código. A variável `valor_extra` foi removida, o nome da variável `media` foi

alterado para `nota_media`.

A variável `nota_media` tem um nome mais claro e a variável booleana `aprovado` facilita a leitura e manutenção da lógica de aprovação.

2.6 *SonarQube*

O *SonarQube* é uma ferramenta de análise estática amplamente usada para detectar e gerenciar *code smells*, com o objetivo de melhorar a qualidade do código. Ele analisa o código-fonte em busca de violações de boas práticas de programação, sinalizando possíveis problemas que podem comprometer a manutenibilidade e a qualidade do *software* a longo prazo. Ao identificar esses problemas cedo, os desenvolvedores podem corrigi-los rapidamente, prevenindo impactos negativos futuros[5].

O funcionamento do *SonarQube* baseia-se no modelo *SQALE (Software Quality Assessment based on Lifecycle Expectations)*, que estabelece um conjunto de regras específicas para cada linguagem de programação. Cada uma dessas regras representa uma boa prática, e qualquer violação detectada é classificada como um *code smell*. Além de identificar o problema, o *SonarQube* também oferece um custo de remediação, que é uma estimativa do tempo necessário para corrigir o *code smell*, calculada com base em dados históricos e na complexidade do problema[3].

Para ajudar no gerenciamento de *code smells*, o *SonarQube* fornece métricas importantes, entre elas a "dívida técnica", que representa o tempo total estimado para corrigir todos os *code smells* de um projeto. Essa métrica dá uma visão geral do esforço necessário para melhorar a qualidade do código. Outra métrica relevante é a "relação de dívida técnica", que compara o custo de remediação dos *code smells* com o custo total de desenvolvimento do *software*, ajudando a entender o impacto dos problemas detectados. Apesar de suas vantagens, o *SonarQube* apresenta algumas limitações. A estimativa de tempo para correção de *code smells* nem sempre é precisa, podendo superestimar o tempo real necessário para algumas correções. Além disso, nem todos os problemas detectados são críticos ou relevantes para o projeto, e a configuração padrão da ferramenta pode ser genérica, exigindo personalização para atender melhor às necessidades específicas de cada projeto. Outro ponto a ser considerado é que o *SonarQube* foca na análise do código-fonte, deixando de lado outros tipos de dívida técnica, como as relacionadas à infraestrutura ou aos requisitos.

Em conclusão, o *SonarQube* é uma ferramenta poderosa para detectar e gerenciar *code smells*, mas é essencial ter em mente suas limitações. Complementar a análise automatizada do *SonarQube* com revisões manuais por especialistas pode garantir uma análise mais precisa e uma priorização mais eficaz dos problemas detectados, contribuindo para a qualidade e a sustentabilidade do projeto de *software*.

2.7 *RefactoringMiner*

O *RefactoringMiner* é uma ferramenta de mineração de refatoração que se destaca pela capacidade de analisar o histórico de *commits* de um repositório de código e identificar refatorações aplicadas, com alta precisão e velocidade de execução. Ele é especialmente útil em estudos empíricos sobre evolução de *software*, migração de API e revisão de código,

permitindo que os desenvolvedores e pesquisadores entendam como o código evolui ao longo do tempo.

O processo de reconhecimento de refatorações pelo *RefactoringMiner* ocorre em duas fases principais. Na primeira fase, chamada de Correspondência de Código-Fonte, a ferramenta realiza o mapeamento de instruções entre diferentes versões do código usando algoritmos sofisticados. A correspondência precisa entre as instruções é crucial para identificar as mudanças realizadas nas refatorações. Além disso, a ferramenta utiliza técnicas de abstração para lidar com mudanças nas refatorações, simplificando as instruções e melhorando a comparação entre as versões do código.

Na segunda fase, Detecção de Refatorações, o *RefactoringMiner* aplica um conjunto de regras de detecção específicas para identificar diferentes tipos de refatorações, como renomeação de variáveis ou extração de métodos. Essas regras são aplicadas de forma hierárquica, começando pelas refatorações simples e progredindo para aquelas que envolvem movimentação de código. A ferramenta também é capaz de detectar refatorações aninhadas, ou seja, refatorações que ocorrem dentro de outra refatoração, o que aumenta a precisão da análise.

O *RefactoringMiner* suporta a detecção de 40 tipos de refatorações, abrangendo uma ampla gama de mudanças, desde as mais simples até as mais complexas. Isso torna a ferramenta útil em muitos contextos, garantindo uma análise abrangente das mudanças no código.

Entre as vantagens do *RefactoringMiner*, destacam-se sua alta precisão e velocidade de execução, que permitem que ele seja aplicado em projetos de grande escala. Além disso, sua abordagem independente de limiares de similaridade torna a ferramenta mais robusta, diferenciando-a de outras que dependem de ajustes manuais. O suporte a refatorações aninhadas também é uma vantagem significativa, pois permite capturar mudanças complexas durante a evolução do código.

No entanto, o *RefactoringMiner* tem algumas limitações. Uma delas é sua especificidade de linguagem, pois atualmente a ferramenta suporta apenas a linguagem *Java*. Estender sua aplicação para outras linguagens apresenta desafios técnicos. Além disso, a ferramenta analisa apenas os arquivos modificados em um *commit*, o que pode resultar em detecção incorreta do tipo de refatoração em alguns casos, devido à falta de contexto completo.

Em conclusão, o *RefactoringMiner* é uma ferramenta poderosa para a detecção de refatorações, especialmente em projetos *Java*, oferecendo alta precisão e eficiência. Apesar de suas limitações, seus benefícios superam as desvantagens na maioria dos casos, tornando-a uma ferramenta essencial para desenvolvedores e pesquisadores interessados na evolução do código e na análise de refatorações.[6]

3 Metodologia

3.1 Estruturação dos Objetivos e Perguntas (Abordagem GQM)

A metodologia adotada para estruturar a investigação deste trabalho baseia-se na abordagem GQM (Goal Question Metric) [7], que facilita a definição de objetivos de maneira sistemática, desdobrando-os em perguntas e métricas específicas para obtenção de dados

quantitativos e qualitativos. A seguir, detalhamos os objetivos, perguntas e métricas deste trabalho:

- **Objetivo G1**: Investigar a evolução dos *code smells* em repositórios de *code samples*.
  - **Pergunta P1.1**: Quantos *code smells* são encontrados na média?
    - Métrica M1.1.1: Média dos números de *code smells* detectados.
  - **Pergunta P1.2**: Quanto tempo demora para o *code smell* ser resolvido?
    - Métrica M1.2.1: Tempo de vida do *code smell*.
  - **Pergunta P1.3**: Quais tipos de *code smells* são encontrados?
    - Métrica M1.3.1: Tipo de *code smells* encontrados.
    - Métrica M1.3.2: Severidade do *code smell*.
  - **Pergunta P1.4**: Quem introduz um *code smell* no código?
    - Métrica M1.4.1: Usuário do *commit*.
  - **Pergunta P1.5**: Como diferentes organizações lidam com *code smells*?
    - Métrica M1.5.1: Usuário do *commit*.
    - Métrica M1.5.2: Quantidades de *code smells* resolvidos (issues fechadas).
  - **Pergunta P1.6**: Há alguma relação entre *code smells* e a complexidade do código?
    - Métrica M1.6.1: Tipo de *code smells* encontrados
    - Métrica M1.6.2: Complexidade do código
    - Métrica M1.6.3: Números de linhas do código,
    - Métrica M1.6.4: Números de arquivos
  - **Pergunta P1.7**: Quem corrigi os *code smells*?
    - Métrica M1.7.1: Usuário do *commit*
  - **Pergunta P1.8**: Existe alguma relação entre *code smells* e a popularidade do repositório?
    - Métrica M1.8.1: Número de *code smells*
    - Métrica M1.8.2: Métricas do repositório (estrelas, forks, contribuidores, contribuições)
  - **Pergunta P1.9**: Qual a relação entre o tempo do projeto e a vida dos *code smells*?
    - Métrica M1.9.1: Tipo de *code smells* encontrados
    - Métrica M1.9.2: Tempo de vida do *code smells*
    - Métrica M1.9.3: Quantidades de *code smells* resolvidos (issues fechadas)
    - Métrica M1.9.4: Número de *code smells* detectados

        * Métrica M1.9.5: Métricas do repositório (estrelas, forks, contribuidores, contribuições)
        * Métrica M1.9.6: Tempo do projeto

- **Objetivo G2**: Investigar a evolução das refatorações em repositórios de *code samples*.
    - **Pergunta P2.1**: Quais são os tipos de refatorações que são aplicadas?
        * Métrica M2.1.1: Tipos de refatoração.
    - **Pergunta P2.2**: O número de refatorações aumenta ou diminui com o tempo?
        * Métrica M2.2.1: Data do *commit*.
        * Métrica M2.2.2: Número de refatorações.
    - **Pergunta P2.3**: Existe relação entre as refatorações e os *code smells* encontrados?
        * Métrica M2.3.1: *commit* da refatoração.
        * Métrica M2.3.2: *commit* dos *code smells* resolvidos (issues fechadas)
    - **Pergunta P2.4**: Como diferentes organizações lidam com refatorações?
        * Métrica M2.4.1: Usuário do *commit*.
        * Métrica M2.4.2: Número de refatorações.
    - **Pergunta P2.5**: Como diferentes organizações lidam com *code smells*?
        * Métrica M2.5.1: Usuário do *commit*.
        * Métrica M2.5.2: Quantidades de *code smells* resolvidos (issues fechadas).
    - **Pergunta P2.6**: Tem alguma relação entre refatoração e complexidade do código?
        * Métrica M2.6.1: Quantidade de refatoração
        * Métrica M2.6.2: Complexidade do código
        * Métrica M2.6.3: Números de linhas do código,
        * Métrica M2.6.4: Números de arquivos
    - **Pergunta P2.7**: Quem faz as refatorações?
        * Métrica M2.7.1: Usuário do *commit*
    - **Pergunta P2.8**: Existe alguma relação entre refatorações e a popularidade do repositório?
        * Métrica M2.8.1: Número de refatorações
        * Métrica M2.8.2: Métricas do repositório (estrelas, forks, contribuidores, contribuições)
    - **Pergunta P2.9**: Qual é a relação entre o tempo do projeto e a refatoração?
        * Métrica M2.9.1: Tipo de refatoração encontrados

* Métrica M2.9.2: Quantidades de refatoração
* Métrica M2.9.3: Métricas do repositório (estrelas, forks, contribuidores, contribuições)
* Métrica M2.9.4: Tempo do projeto

3.2 Coleta e Análise de Dados

Para responder às perguntas propostas na pesquisa, realizamos inicialmente uma seleção de seis ecossistemas de *software* amplamente utilizados, sendo eles: *spring-guides*, *spring-cloud-samples*, *googlesamples*, *googlearchive*, *Azure-Samples* e *aws-samples*. Esses ecossistemas foram escolhidos com base em sua representatividade e amplitude, contemplando *code samples* empregados em diversos setores e plataformas. A partir dessa seleção, listamos os *code samples* de cada ecossistema, visando uma amostra abrangente para a análise.

Com essa lista de *code samples* em mãos, utilizamos a API do *GitHub* para coletar dados detalhados sobre os repositórios correspondentes. Esta coleta incluiu uma série de informações importantes, como a linguagem de programação predominante, métricas de atividade (número de estrelas, *forks*), além de dados sobre contribuidores e histórico de *commits*, entre outros. Esses detalhes nos permitiram traçar um perfil completo de cada repositório, gerando um conjunto de dados robusto para a etapa de análise. Após a coleta inicial, estabelecemos quatro critérios de inclusão para a seleção dos projetos a serem analisados, buscando assim restringir a amostra a repositórios com características que garantam relevância e continuidade. Os critérios definidos foram: (1) o projeto deveria possuir mais de cinco contribuidores ativos, de forma a garantir que não se trata de um repositório com desenvolvimento limitado; (2) deveria conter entre 500 e 100 000 linhas de código (LOC), assegurando uma complexidade mínima necessária sem incluir projetos massivamente grandes que poderiam enviesar a análise; (3) o repositório não deveria estar arquivado, indicando que o projeto ainda está em andamento ou tem manutenção ativa; e (4) deveria ter recebido atualizações nos últimos doze meses, garantindo uma análise de projetos recentes. Ao final dessa filtragem, obtivemos uma lista de projetos em linguagens como *Python*, *JavaScript*, *Java*, *C#*, *TypeScript*, *Kotlin*, *PHP* e *Go*. Vale ressaltar que a seleção foi finalizada em 28 de agosto de 2024, e quaisquer mudanças ocorridas nos projetos após essa data foram desconsideradas para fins de padronização na análise.

Para a análise de *Code Smells*, utilizamos a versão 10.7 do *SonarQube* juntamente com a versão 6.2.1 do *SonarScanner*. O *SonarQube* foi originalmente desenvolvido para análise contínua de código, ou seja, ele foi projetado para integrar-se ao fluxo de desenvolvimento e fornecer feedback contínuo sobre a qualidade do código à medida que os desenvolvedores fazem alterações. Isso significa que sua utilização para a análise de *commits* passados exigiu adaptações significativas.

Desenvolvemos, então, um script em *Python* para clonar os repositórios selecionados e percorrer os *commits* de cada projeto, utilizando o *SonarScanner* para integrar diretamente com uma instância local do *SonarQube*. A análise foi realizada de forma minuciosa em cada *commit*, porém, devido à natureza contínua do *SonarQube*, foi necessário ajustar suas configurações e comportamento para que ele pudesse realizar a análise de código já existente e não apenas de novos *commits*.

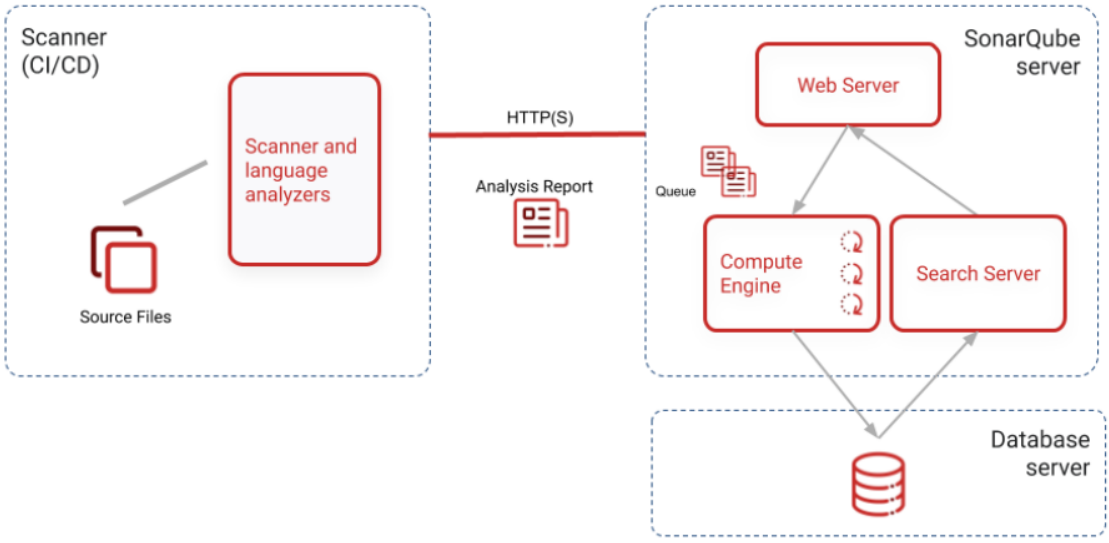

Figura 4: Componentes do *SonarQube* e *SonarScanner*

Alguns projetos exigiram compilação antes de serem analisados, sendo esses, os projetos de *Java*, *Android* e *C#*, o que tornou a análise inviável para esses repositórios, uma vez que não podíamos garantir que o processo de *build* fosse realizado corretamente em todos os casos, portanto, removemos os repositórios em que a linguagem principal fosse um desses.

Para otimizar o tratamento e exportação dos dados, implementamos um *scraper* dedicado que extraiu as informações diretamente do servidor *SonarQube* e as converteu para o formato CSV. Isso facilitou a manipulação e análise dos resultados em ferramentas estatísticas e de visualização, tornando o processo de extração de dados mais eficiente e acessível para análises posteriores.

Além disso, foi realizada uma análise paralela, nos projetos com *Java*, utilizando o *RefactoringMiner* (versão 3.0.0). Adotamos uma abordagem similar à análise anterior, implementando um script em *Python* para automatizar o processo de clonagem dos repositórios e executar o *RefactoringMiner*. Esta ferramenta percorreu automaticamente os *commits* de cada projeto, identificando refatorações de forma precisa e ágil, o que nos permitiu eliminar a necessidade de análise manual *commit* a *commit*.

Os *scripts* utilizados no projeto podem ser encontrados nos seguintes repositórios:

- Coleta e análise dos repositórios: `https://github.com/Cafeo-Group/git-infer`
- Coleta e análise dos *code smells*: `https://github.com/Cafeo-Group/SonarQube-analysis`
- Coleta e análise das refatorações: `https://github.com/Cafeo-Group/RefactoringMiner-analysis`

4 Resultados

4.1 Análise dos repositórios

Na etapa inicial do projeto, dedicada à coleta e análise dos repositórios, foram examinados um total de 9 320 repositórios, distribuídos da seguinte forma: 6 920 do *aws-samples*, 2 619 do *Azure-Samples*, 240 do *googlearchive*, 69 do *googlesamples*, 27 do *spring-cloud-samples* e 74 do *spring-guides*.

Com base no critério de análise de linhas de código, elaboramos o gráfico da Figura 5, que apresenta a porcentagem das 7 linguagens de programação mais utilizadas nesses projetos.

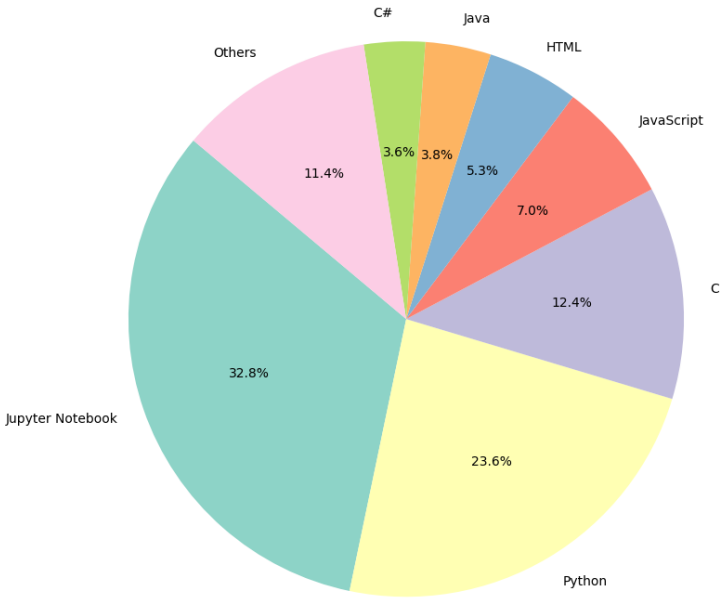

Figura 5: Porcentagem de Linguagens Utilizadas.

As métricas utilizadas para a filtragem dos repositórios, conforme descrito na Seção 3.2, foram analisadas individualmente para cada repositório. A Tabela 1 apresenta um resumo estatístico dessas métricas, incluindo informações como a média, os valores mínimo e máximo, e as medidas de tendência central (mediana e 25%, 75% percentis).

Selecionamos, então, um total de 373 repositórios, dos quais 132 pertencem ao *aws-samples*, 152 ao *Azure-Samples*, 53 ao *spring-guides*, 5 ao *spring-cloud-samples* e 1 ao *googlesamples*. A Tabela 2 apresenta um resumo consolidado desses dados.

Como observado, apesar da Tabela 2 apresentar médias menores para métricas como *forks*, estrelas e tamanho dos repositórios, ela demonstra uma maior estabilidade nas métricas. Isso fica evidente pela menor variação entre os percentis e o valor máximo, em comparação com a Tabela 1, que apresenta uma distribuição mais dispersa, especialmente no que diz respeito ao tamanho e ao número de estrelas e *forks*. Em resumo, enquanto os repositórios da Tabela 1 têm uma diversidade maior em termos de atividade e tamanho, os repositórios

Métrica	Média	25%	Mediana	75%	Máximo
Número de linhas	8 935,18	64	408	2 560	1 965 261
Estrelas	37,45	2	6	19	9 846
Forks	30,29	1	4	13	8 314
Issues Abertas	2,90	0	1	2	509
Issues Totais	20,24	1	3	12	9 505
Issues Fechadas	17,34	0	2	8	9 370
Pull Requests Fechados	14,02	0	2	7	9 351
Pull Requests Abertos	1,47	0	0	1	138
Pull Requests Mesclados	9,76	0	1	5	3 205
Commits	48,95	7	16	36	10 404
Contribuidores	4,37	2	3	4	296

Tabela 1: Resumo Estatístico das Métricas dos Repositórios

Métrica	Média	25%	Mediana	75%	Máximo
Número de linhas	158,04	3,69	28,21	196,50	982,00
Estrelas	60,02	4,00	17,00	57,00	949,00
Forks	44,89	4,00	11,00	38,00	666,00
Issues Abertas	3,25	0,00	2,00	4,00	29,00
Issues Totais	38,37	12,00	21,00	44,50	1 063,00
Issues Fechadas	35,12	10,00	18,00	40,50	1 061,00
Pull Requests Fechados	28,91	8,00	14,00	28,00	1 058,00
Pull Requests Abertos	1,80	0,00	1,00	2,00	27,00
Pull Requests Mesclados	20,82	3,50	11,00	21,00	448,00
Commits	92,95	31,00	47,00	131,00	989,00
Contribuidores	9,20	5,00	7,00	11,00	30,00

Tabela 2: Resumo Estatístico das Métricas dos Repositórios Filtrados

da Tabela 2 exibem um comportamento mais equilibrado e consistente, o que possibilita uma análise mais clara e focada na identificação de *code smells* ao reduzir a influência de extremos nos dados.

4.2 Análise dos dados gerados pelo *SonarQube*

Com os repositórios selecionados, foi possível coletar algumas métricas necessárias para responder às questões do GQM. Para isso, analisamos exclusivamente as *issues* classificadas como *Code Smell*. Dos 373 projetos inicialmente considerados, o *SonarQube* conseguiu identificar *issues* em 50 projetos. Durante o processo, foram descartados os repositórios que utilizavam *Java*, Kotlin e C#, o que eliminou 194 projetos. Assim, dos 179 repositórios analisados, apenas 50 apresentaram *code smells* detectados pelo *SonarQube*, conforme resumido na Tabela 3.

Ecossistema	Linguagem	Quantidade de Repositórios
Azure-Samples	*JavaScript*	3
Azure-Samples	Jupyter Notebook	1
Azure-Samples	*Python*	6
aws-samples	*Go*	1
aws-samples	*JavaScript*	21
aws-samples	Jupyter Notebook	2
aws-samples	*Python*	16

Tabela 3: Número de repositórios observados por ecossistema.

4.2.1 Métricas Coletadas

- **Média dos números de *code smells* detectados:**

  A métrica foi obtida utilizando uma média simples. A Tabela 4 apresenta as médias dos *code smells* em diferentes dimensões relevantes aos repositórios analisados.

Média	Média
code smells por 1000 linhas de código	5,38
code smells a cada 100 dias de atividade	55,90
code smells por contribuidor	28,90
code smells por estrelas	8,22
code smells por forks	12,92
code smells por issues fechadas	8,47
code smells por *commit*	1,45

Tabela 4: Médias de *code smells* observadas.

O gráfico da Figura 6 apresenta a relação entre o tempo de atividade dos projetos analisados (em dias) e o número de *code smells* detectados. A linha de regressão linear indica uma tendência positiva, sugerindo que, em média, projetos com maior tempo de duração acumulam mais *code smells*. Essa relação pode estar associada ao aumento da complexidade e do tamanho do código ao longo do tempo, especialmente em projetos que não passam por processos regulares de refatoração.

Apesar dessa tendência, observa-se uma alta dispersão nos dados. Muitos projetos apresentam um número reduzido de *code smells* mesmo com maior tempo de atividade, enquanto outros apresentam valores significativamente altos. Esses *outliers*, como o caso de um projeto com mais de 3500 *code smells*, podem indicar situações específicas, como falta de manutenção contínua ou padrões de código menos rigorosos.

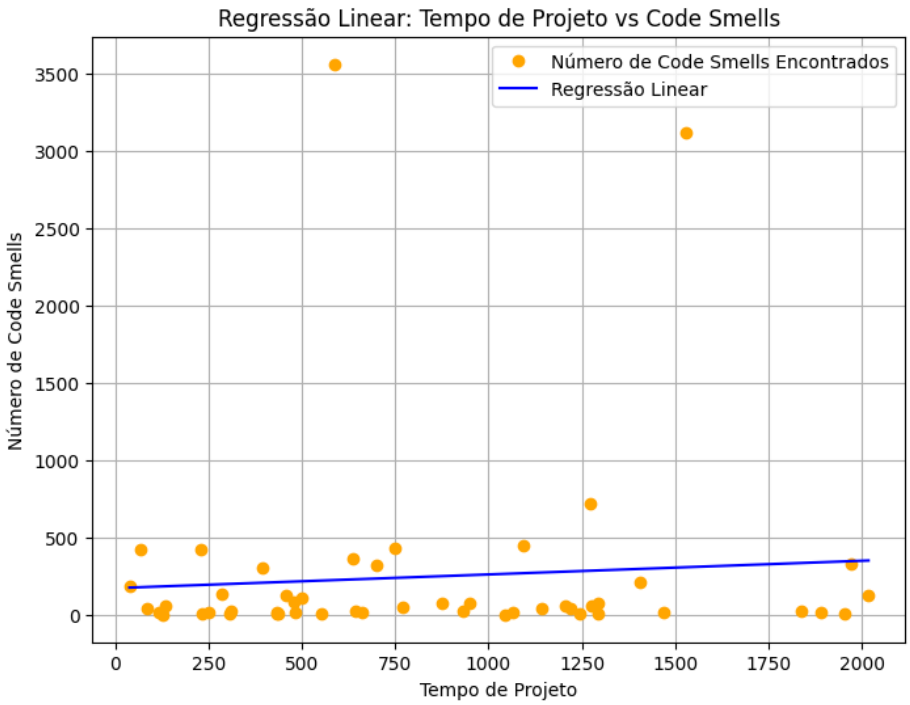

Figura 6: Regressão Linear: Tempo de Projeto vs *code smells*.

- **Tipos de *code smell* detectados**

  As Figuras 7 e 8 apresentam a distribuição dos tipos de *code smells* detectados, categorizados de acordo com a tecnologia dos projetos analisados. Cada segmento nos gráficos representa um tipo específico de *code smell*, identificado e classificado com base nas regras predefinidas do *SonarQube*.

  O significado de cada regra estão listadas na página de regras da Sonar [8]. Abaixo estão os *code smells* mais frequentes detectados, organizados por relevância no total dos dados analisados:

  – **S117: Nomes de variáveis e parâmetros devem seguir convenções de nomenclatura**
  Este foi o *code smell* mais detectado, aparecendo com alta frequência em diversas tecnologias, como *Python*, *Go* e *Azure Resource Manager*. Seguir convenções de nomenclatura é fundamental para a legibilidade e consistência do código, especialmente em equipes de desenvolvimento, onde a colaboração exige padrões claros.

  – **S6570: Uso de aspas duplas para prevenir *globbing* e divisão de palavras**
  Este problema foi predominante em arquivos *Docker*, destacando a importância

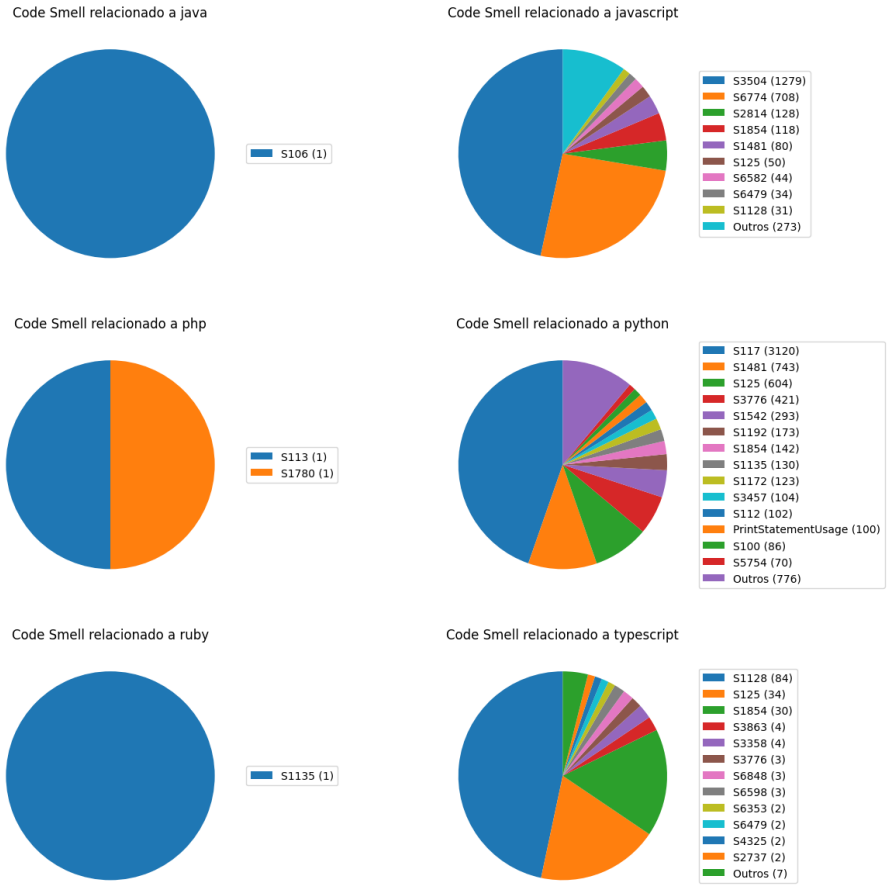

Figura 7: Tipos de *code smell* Encontrados.

de práticas adequadas ao lidar com arquivos de configuração para evitar comportamentos inesperados e vulnerabilidades.

– **S1481: Variáveis locais não utilizadas devem ser removidas**
Detectado especialmente em *Python* e *JavaScript*, esse *code smell* indica o acúmulo de código morto, que reduz a clareza e pode gerar confusão para futuros desenvolvedores.

– **S3504: Variáveis devem ser declaradas com `let` ou `const`**
Frequente em *JavaScript*, esse problema enfatiza a necessidade de utilizar declarações modernas para evitar erros relacionados ao escopo e melhorar a previsibilidade do comportamento das variáveis.

– **S6587: O cache deve ser limpo após a instalação de pacotes**
Presente em arquivos *Docker*, este *code smell* reflete a preocupação com o uso eficiente de recursos, reduzindo o tamanho de imagens geradas e garantindo que elas não carreguem dados desnecessários.

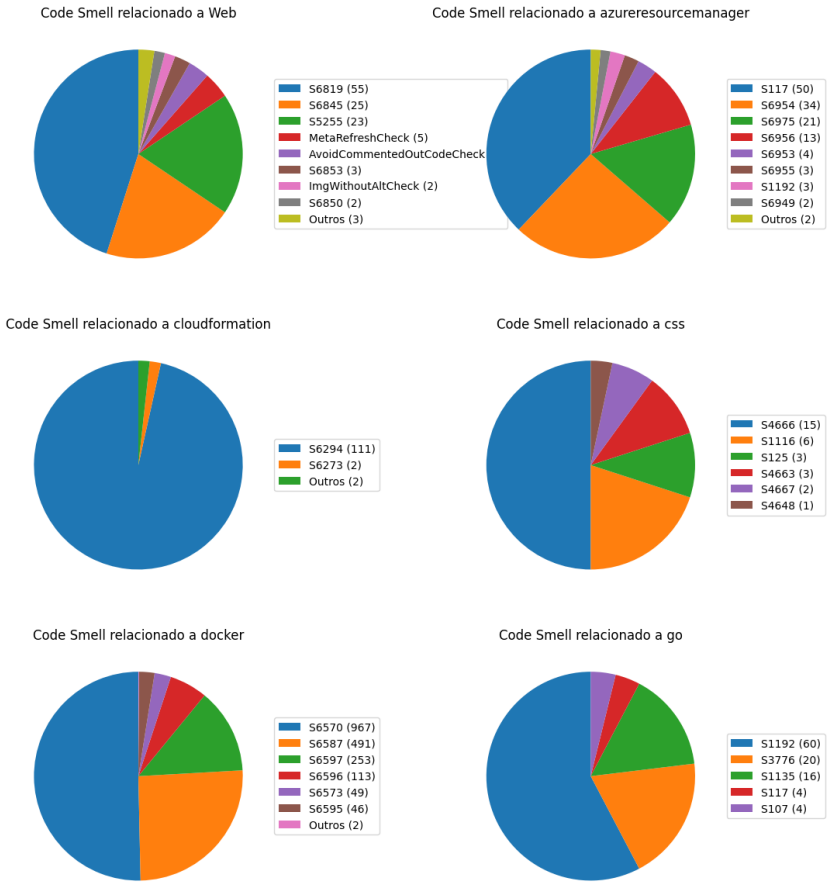

Figura 8: Tipos de *code smell* Encontrados.

- **S3776: A complexidade cognitiva das funções não deve ser alta**
  Um problema recorrente em linguagens como *Python*, *Go* e *TypeScript*, este *code smell* aponta para a necessidade de refatorar funções excessivamente complexas, dividindo-as em partes menores e mais fáceis de compreender.
- **S6845: Violações em elementos ou estruturas específicas de acessibilidade**
  Este problema, detectado em projetos Web, ressalta a importância de práticas inclusivas no desenvolvimento de interfaces, garantindo que o conteúdo seja acessível a todos os usuários.

Esses *code smells* mais recorrentes refletem tanto problemas técnicos quanto de boas práticas no desenvolvimento.

- **Severidade do *code smell*.**

  A Figura 9 apresenta a distribuição dos *code smells* detectados nos projetos analisados, categorizados de acordo com a severidade atribuída pelo *SonarQube*. Essa

classificação reflete o impacto potencial de cada *code smell* na qualidade e manutenção do código, indo de questões informativas (menor impacto) a problemas críticos que podem comprometer funcionalidades importantes.

De acordo com o gráfico, os *code smells* classificados como *Major* representam a maior proporção, com 44,2% do total, seguidos pelos *Minor*, que correspondem a 36,7%. Esses dois níveis de severidade destacam problemas significativos, mas geralmente não bloqueantes, que afetam a legibilidade e manutenção do código.

Os *Critical* compõem 17%, indicando questões que exigem atenção prioritária por seu potencial impacto na funcionalidade ou segurança do sistema. Por outro lado, os *Blocker*, que são os problemas mais graves, aparecem em apenas 1% dos casos, sugerindo que os projetos mantêm, em geral, uma boa estabilidade em relação a falhas críticas.

Os *Info* representam 1,2%, refletindo sugestões ou informações que não impactam diretamente o funcionamento do código, mas que podem contribuir para melhorias futuras.

Essa distribuição reforça a importância de priorizar a resolução de problemas classificados como *Critical* e *Major*, que juntos representam mais de 60% dos *code smells* identificados. Já os problemas *Minor* e *Info*, embora menos graves, podem ser abordados como parte de um esforço contínuo de refatoração para manter a qualidade do código ao longo do tempo.

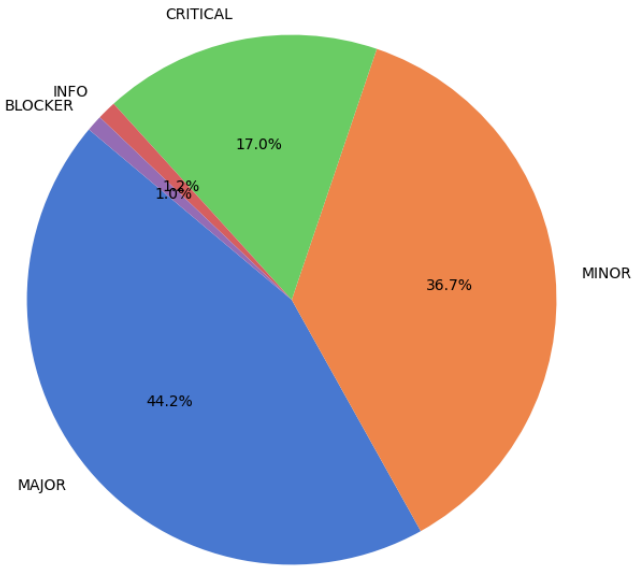

Figura 9: Distribuição de Severidade.

- **Tempo de vida do *code smell*.**

A Figura 10 apresenta o histograma da distribuição do tempo de vida dos *code smells* detectados nos repositórios analisados, medido em dias. O tempo de vida corresponde ao intervalo entre a introdução de um *code smell* no código e sua resolução. Observa-se que a maioria dos *code smells* possui um tempo de vida relativamente curto, com a maior frequência concentrada no intervalo inicial (0 a 250 dias), representando mais de 7 000 ocorrências. Isso sugere que, em muitos casos, os *code smells* são corrigidos rapidamente após sua introdução, o que pode estar associado a práticas de manutenção contínua ou a ciclos curtos de desenvolvimento. No entanto, há uma diminuição significativa na frequência à medida que o tempo de vida aumenta, com poucas ocorrências ultrapassando 500 dias. Esse padrão indica que a persistência de *code smells* no código é menos comum em períodos prolongados, mas pode refletir uma menor atenção ou prioridade dada a problemas mais antigos. Vale destacar a presença de picos secundários em períodos específicos, como entre 1 000 e 1 250 dias. Esses picos podem estar relacionados a características particulares de alguns projetos, como refatorações pontuais em códigos acumulados ou períodos de desenvolvimento intensivo.

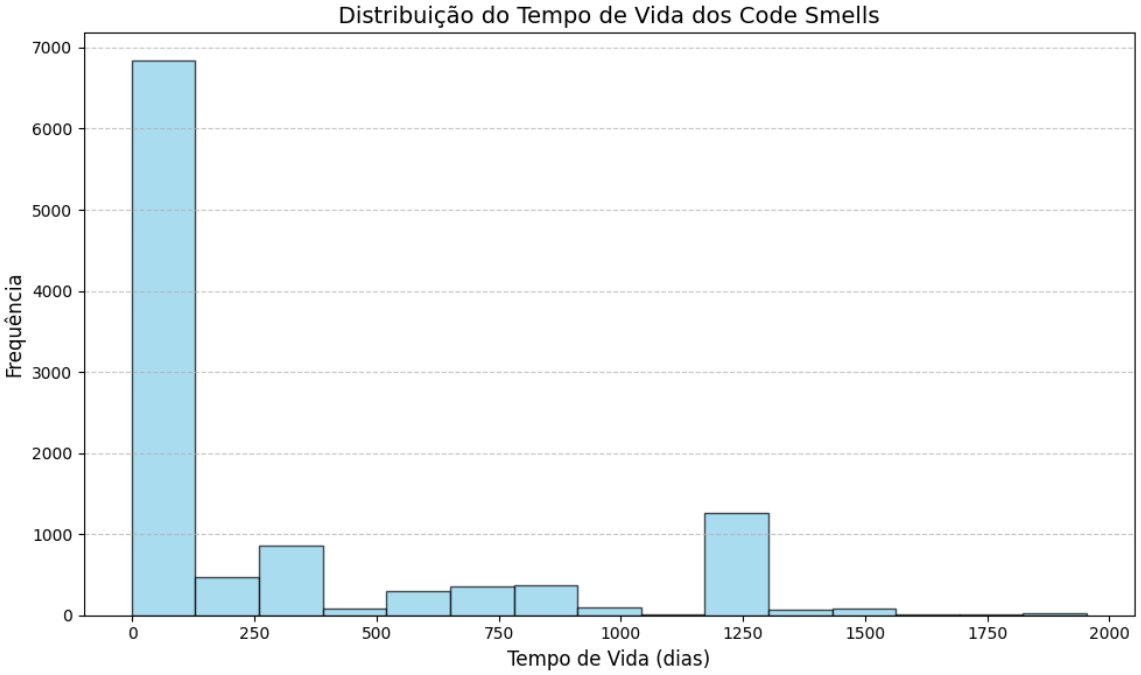

Figura 10: Histograma de tempo de vida dos *code smells*.

A Figura 11 mostra um gráfico de violino representando a distribuição do tempo de correção dos *code smells* agrupados por severidade (*Minor*, *Critical*, *Blocker*, *Major*, *Info*). O tempo de correção refere-se ao intervalo entre a detecção e a resolução do *code smell*. Analisando o gráfico:

- ***Minor*:** Os *code smells* classificados como *Minor* apresentam uma ampla va-

riação no tempo de correção, com a mediana concentrada em torno de 1 000 dias. No entanto, há uma dispersão significativa, com casos resolvidos rapidamente e outros persistindo por até mais de 2 000 dias.

- ***Critical:*** Apesar de sua maior gravidade, os *code smells Critical* possuem uma distribuição semelhante à de *Minor*, indicando que muitas vezes esses problemas não são tratados de forma prioritária. A mediana também está próxima de 1 000 dias.
- ***Blocker*:** Os *code smells Blocker*, que representam a severidade mais alta, têm um tempo de correção muito mais concentrado, indicando que são tratados de forma rápida e eficiente. Isso reflete a importância dada a problemas críticos no código.
- ***Major:*** Assim como os *Minor*, os *code smells Major* apresentam uma dispersão ampla, com casos corrigidos rapidamente e outros permanecendo por longos períodos. A mediana está em torno de 750 – 1 000 dias.
- ***Info*:** Os *code smells Info* possuem uma distribuição mais restrita e tempos de correção geralmente menores, indicando que problemas considerados mais Informativos ou de baixa prioridade são corrigidos de forma mais previsível.

Comparando com o histograma apresentado na Figura 10, observa-se uma consistência nos padrões de resolução. A concentração de *code smells* corrigidos em períodos curtos (até 250 dias) no histograma reflete principalmente os casos *Blocker* e alguns *Info*, que possuem distribuições mais estreitas e tempos médios de correção menores. Por outro lado, a dispersão observada em *Minor* e *Major* explica a longa cauda de frequências no histograma, indicando que *code smells* de severidade intermediária tendem a permanecer mais tempo no código antes de serem resolvidos.

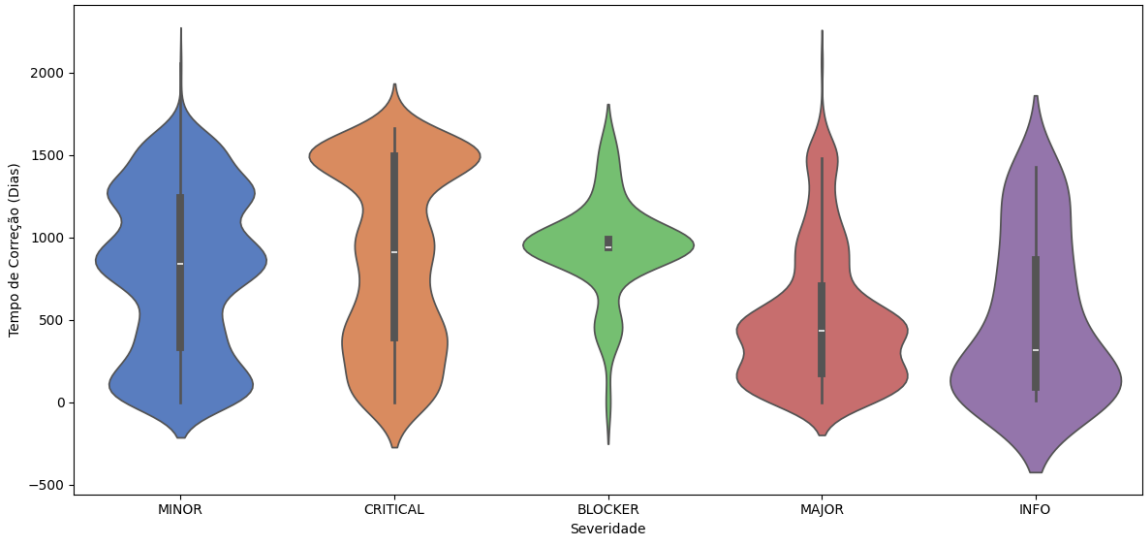

Figura 11: Tempo de correção médio por severidade.

- **Usuários que introduzem e resolvem o *code smell*.**

  Os dados das tabelas 5 e 6 revelam como as contribuições de usuários que introduzem e removem *code smells* se distribuem entre as categorias *amazon*, *microsoft* e *outros*. Para isso, categorizamos os e-mails de acordo com os domínios: os que contêm *amazon* foram agrupados como *amazon*, os que contêm *microsoft* como *microsoft*, e todos os demais foram classificados como *outros*.

  Nos repositórios da *AWS*, a categoria *amazon* domina amplamente tanto na introdução quanto na remoção de *code smells*, refletindo uma concentração significativa de contribuições dentro da própria organização. Já nos repositórios da *Azure*, observa-se uma distribuição mais balanceada: a categoria *outros* apresenta uma participação quase equivalente à da *microsoft*, indicando uma maior diversidade e colaboração externa.

  Os gráficos das Figuras 12 e 13 complementam essa análise ao ilustrar a distribuição de usuários que abrem e fecham issues, sem distinguir diretamente entre *AWS* e *Azure*. Eles evidenciam que, em geral, a contribuição está concentrada em um pequeno grupo de usuários. Esse padrão é mais acentuado no gráfico de abertura de issues, onde poucos usuários respondem pela maior parte das atividades. Em contrapartida, a distribuição de usuários que fecham issues é mais uniforme, sugerindo uma maior participação coletiva nesse aspecto.

  Em resumo, enquanto a *AWS* apresenta um modelo centralizado e dominado pela *amazon*, a *Azure* demonstra uma abordagem mais distribuída, com maior equilíbrio entre contribuições internas e externas. Isso reflete diferenças significativas nas dinâmicas de colaboração entre os dois ecossistemas.

email	**Azure-Samples**	**aws-samples**
amazon	0	9 024
microsoft	844	0
outros	839	1 598

Tabela 5: Usuários que introduzem os *code smells*

email	**Azure-Samples**	**aws-samples**
amazon	0	8 329
microsoft	354	0
outros	763	1 382

Tabela 6: Usuários que resolvem os *code smells*

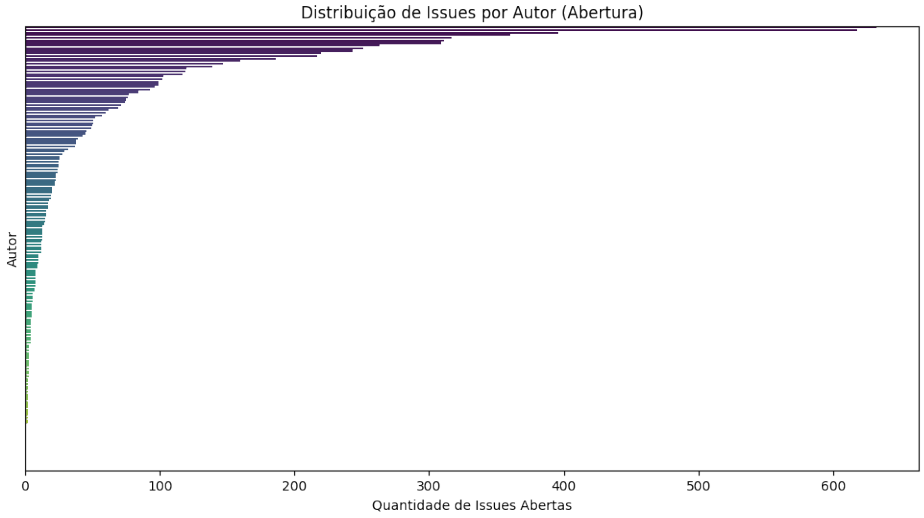

Figura 12: Distribuição de usuários que introduzem os *code smells*.

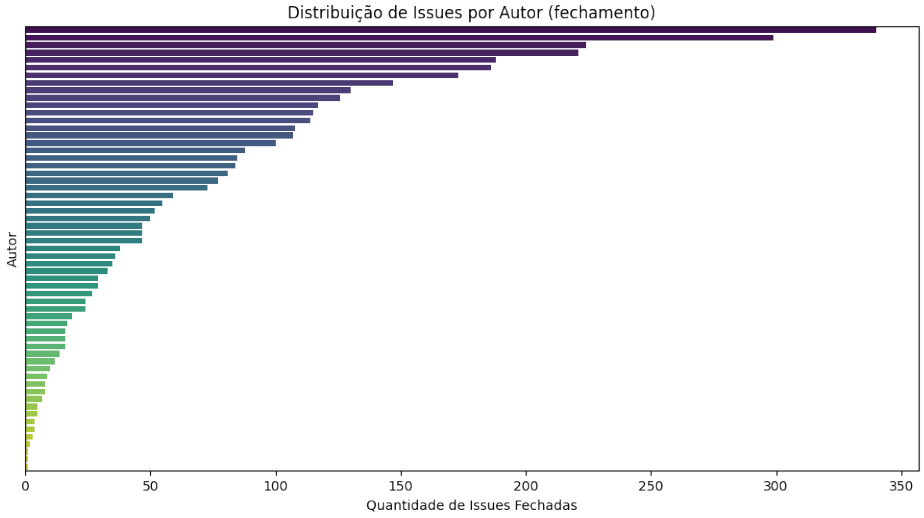

Figura 13: Distribuição de usuários que removem os *code smells*.

4.3 Análise dos dados gerados pelo *RefactoringMiner*

Nesta etapa do projeto, o *RefactoringMiner* foi executado em 100 repositórios *Java*, distribuídos entre os seguintes ecossistemas de *software*, conforme apresentado na Tabela 7. Desses projetos, foi analisado 13 112 *commits*, e foi encontrado 3 517 refatorações, em 85 projetos.

Ecossistema	Quantidade de Repositórios
spring-guides	53
aws-samples	22
Azure-Samples	20
spring-cloud-samples	5
googlesamples	1

Tabela 7: Distribuição dos repositórios

4.3.1 Métricas Coletadas

- **Média dos números de refatorações detectadas:**

  A Tabela 8 apresenta as médias das refatorações em diferentes dimensões relevantes aos repositórios analisados.

Média	Média
Refatorações por 1000 linhas de código	4,12
Refatorações a cada 100 dias de atividade	5,00
Refatorações por contribuidor	4,51
Refatorações por estrelas	8,22
Refatorações por forks	4,02
Refatorações por issues fechadas	1,20
Refatorações por *commit*	0,38

Tabela 8: Médias de refatorações observadas.

O Gráfico 14 demonstra a relação entre o tempo de vida dos projetos (em dias) e o número de refatorações realizadas. Observa-se que, apesar da dispersão significativa dos pontos, a linha de regressão linear apresenta uma leve inclinação negativa. Isso sugere que, conforme os projetos amadurecem, há uma tendência de redução no número de refatorações realizadas. No entanto, a alta variabilidade nos dados indica que essa tendência não é consistente para todos os repositórios, sendo influenciada por características específicas, como o tamanho do projeto, número de contribuidores e nível de atividade.

Como observado na Tabela 8 a métrica de 4,12 refatorações por 1 000 linhas de código destaca que, independentemente do tempo de projeto, o tamanho do código possui uma correlação importante com a frequência de refatorações. Da mesma forma, 5,00 refatorações a cada 100 dias de atividade reforçam que a dinâmica temporal do projeto desempenha um papel relevante, mas não determinante, como indicado pela tendência do gráfico.

Outro fator relevante está relacionado à colaboração e popularidade dos projetos. Refatorações por contribuidor apresentam uma média de 4,51, mostrando que equipes maiores tendem a distribuir mais uniformemente as tarefas de manutenção. Já a métrica de 8,22 refatorações por estrela sugere que projetos com maior visibilidade

e adoção frequentemente passam por melhorias mais intensas, possivelmente devido à pressão por qualidade e desempenho. A relação com forks (4,02 refatorações por fork) e issues fechadas (1,2 refatorações por issue) também ilustra como aspectos colaborativos e resoluções de problemas impactam o ritmo de refatorações.

No entanto, a métrica de refatorações por *commit* (0,38) indica que refatorações não acompanham diretamente o fluxo de desenvolvimento diário. Essa constatação ajuda a explicar a dispersão dos dados no Gráfico 14, onde alguns projetos apresentam alta frequência de refatorações mesmo com menor tempo de vida, enquanto outros possuem baixos níveis de refatoração, independentemente da idade.

Essas análises em conjunto reforçam que, embora o tempo de projeto influencie as refatorações realizadas, a complexidade, colaboração e contexto individual de cada repositório desempenham papéis cruciais para determinar os padrões observados.

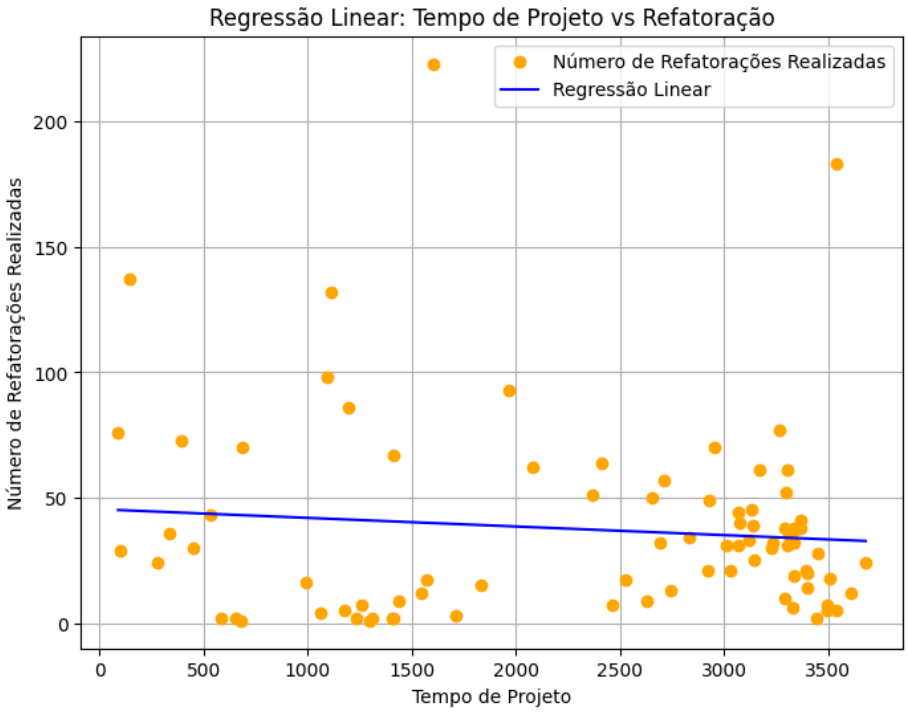

Figura 14: Regressão Linear: Tempo de Projeto vs Refatoração

- **Tipos de refatorações detectadas:**

  O Gráfico 15 apresenta a distribuição dos diferentes tipos de refatorações identificadas no estudo. O grupo **outros** inclui tipos de refatorações menos frequentes, que não foram analisados individualmente devido à sua menor relevância no conjunto de dados. Entre os tipos identificados, as refatorações mais comuns incluem *Remove Class Anotation (22,8%)*, *Move Class* (6,1%), *Add Class Anotation* (6,4%) e *Rename*

Method (2,7%), enquanto refatorações mais específicas, como *Move Package* (2,2%) e *Move And Rename Class* (2,3%), aparecem com menor frequência.

As refatorações podem ser classificadas de acordo com sua complexidade. As de **baixa complexidade**, como renomear métodos ou variáveis, geralmente impactam trechos localizados do código e são mais fáceis de implementar, uma vez que não exigem alterações significativas em outras partes do sistema. Por exemplo, *Rename Method* (2,7%) e *Rename Variable* (2,5%) são operações que não afetam a lógica de negócio, mas podem melhorar a clareza do código, sendo amplamente utilizadas em contextos de manutenção e melhorias incrementais.

Por outro lado, refatorações de **média complexidade**, como *Change Method Access Modifier* (4,0%) e *Move Class* (6,1%), envolvem mudanças que podem impactar diversas partes do sistema. Alterar o modificador de acesso de um método, por exemplo, pode exigir ajustes nos locais onde ele é chamado, especialmente se o método se tornar mais restrito. Da mesma forma, mover uma classe para outro pacote reorganiza a estrutura do projeto e exige atualizações nos *imports*, mas é uma prática comum para melhorar a modularidade do código.

Por fim, refatorações de **alta complexidade**, como *Move and Rename Class* (2,3%) ou *Change Return Type* (4,0%), possuem maior impacto no sistema e demandam maior cuidado. Alterar o tipo de retorno de um método pode exigir revisões em todas as chamadas associadas a ele, potencialmente introduzindo inconsistências se não forem realizados testes rigorosos. Já movimentar e renomear classes combina mudanças estruturais e semânticas, tornando esse tipo de refatoração mais raro, devido ao esforço necessário para garantir que não ocorram erros em partes dependentes do código.

Em termos gerais, observa-se que refatorações mais simples são realizadas com maior frequência, como esperado, pois exigem menor esforço e possuem menor impacto no sistema. Refatorações mais complexas, embora menos comuns, desempenham um papel importante na evolução do código, especialmente em projetos maiores e mais maduros.

- **Número de refatoração por tempo de projeto:**

  O Gráfico 16 apresenta a **distribuição das refatorações ao longo do tempo de vida dos projetos**, é importante ressaltar que o histograma não é acumulativo; os valores refletem o número de refatorações realizadas dentro de cada período específico.

  Nos primeiros 500 dias de vida dos projetos, observa-se uma frequência significativa de refatorações, com valores oscilando entre 200 e 300, o que pode indicar um esforço inicial em ajustes e melhorias nos códigos. No intervalo entre 500 e 1 000 dias, a frequência de refatorações diminui consideravelmente, mostrando um período de menor atividade nesse aspecto. Por outro lado, o intervalo próximo de 1 000 dias registra a maior quantidade de refatorações, com mais de 500 ocorrências. Isso sugere que projetos com aproximadamente três anos de vida enfrentam demandas significativas por melhorias ou reestruturações.

  Após o pico em torno de 1 000 dias, a frequência de refatorações diminui novamente,

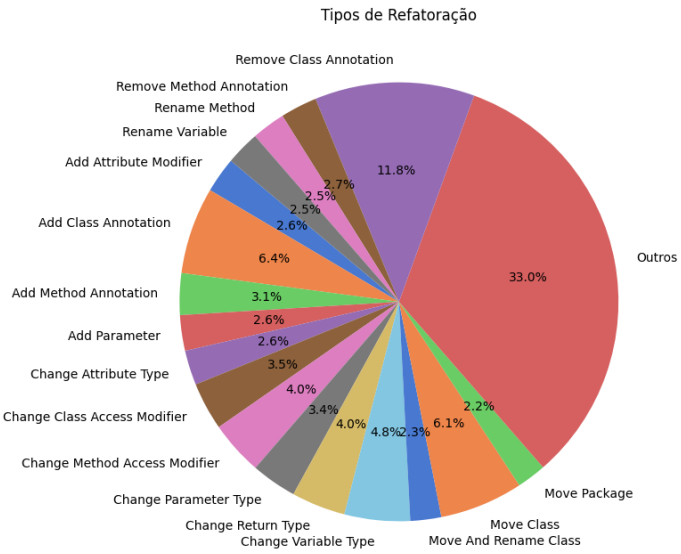

Figura 15: Tipos de Refatoração

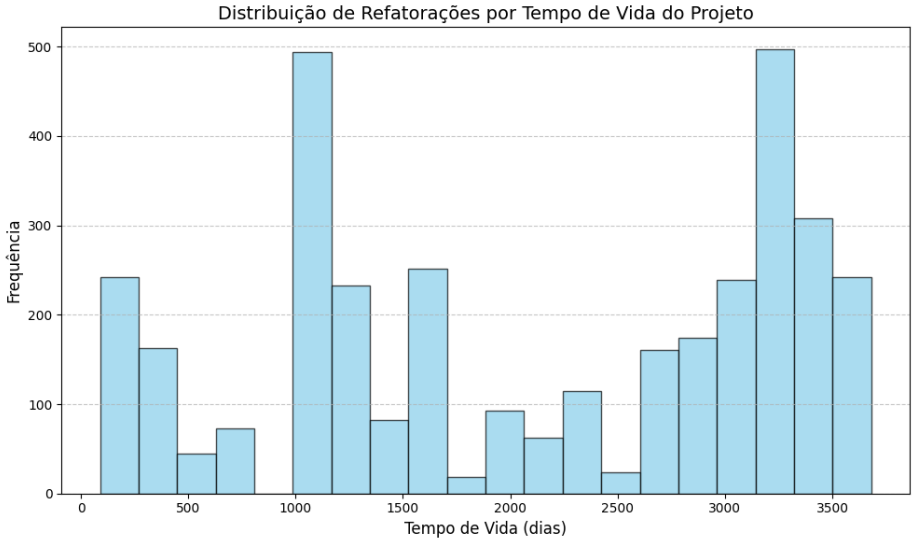

Figura 16: Distribuição de Refatorações por Tempo de Vida do Projeto

mas apresenta oscilações moderadas entre 1 000 e 2 500 dias, com períodos de maior estabilidade e menor necessidade de mudanças. Em projetos com mais de 3 000 dias de vida (aproximadamente oito anos), a frequência de refatorações aumenta novamente, culminando em outro pico de mais de 500 refatorações em torno de 3 500 dias. Esse comportamento pode refletir uma nova fase de reavaliações estruturais nos projetos mais maduros.

Em resumo, os dados indicam padrões cíclicos na realização de refatorações, com picos significativos em momentos específicos da trajetória dos projetos, possivelmente relacionados à maturidade, novas demandas ou reestruturações tecnológicas ao longo do tempo.

5 Discussão dos Resultados

As análises dos dados gerados pelo *SonarQube* e *RefactoringMiner*, aplicados aos repositórios de *code samples* selecionados, revelam implicações significativas para a compreensão da evolução e da qualidade do código nesses projetos. Contudo, é importante destacar que não foi possível estabelecer uma relação direta entre *code smells* e refatorações, pois os *code smells* analisados não contemplaram projetos *Java*, enquanto as refatorações observadas foram exclusivamente de projetos *Java*. Essa limitação reduz a abrangência das correlações entre os dados, demandando atenção em futuras análises.

5.1 Implicações da Análise de *code smells*

Os resultados apontam que a presença de *code smells* é uma constante em projetos de *code samples*, mesmo em ecossistemas renomados como *AWS* e *Azure*. A análise indica que muitos desses problemas permanecem por longos períodos, o que sugere que a resolução de *code smells* frequentemente não é priorizada, sobretudo em casos de severidade intermediária, como os classificados como *Minor* e *Major*. Por outro lado, os *code smells Blocker* recebem atenção imediata, demonstrando o foco em mitigar problemas críticos. Além disso, a predominância de *code smells Major* e *Critical* evidencia a importância de processos mais eficazes para sua detecção e resolução, visando minimizar impactos negativos na qualidade e na manutenibilidade do código. A dinâmica de introdução e resolução de *code smells* destaca pontos de melhoria nas práticas de desenvolvimento. Observa-se uma concentração de contribuições na introdução de *code smells*, o que aponta para a necessidade de aprimorar as revisões de código e a adoção de padrões de qualidade. Embora o esforço coletivo para resolver problemas seja evidente, a conscientização de todos os colaboradores sobre a importância da qualidade do código ainda precisa ser reforçada.

5.2 Implicações da Análise de Refatorações

A análise de refatorações revelou padrões cíclicos em sua realização, com picos em momentos específicos do ciclo de vida dos projetos. Esses picos podem estar relacionados a ciclos de desenvolvimento, introdução de novas funcionalidades ou demandas por reestruturações. No entanto, percebe-se uma redução gradual na frequência de refatorações em projetos mais maduros, possivelmente indicando uma estabilização do código. Essa estabilização, porém, exige atenção para evitar o acúmulo de débito técnico, que pode comprometer a manutenção futura.

Outro ponto relevante é a predominância de refatorações de baixa complexidade, que refletem uma preferência por melhorias incrementais e de baixo impacto. Apesar disso, refatorações mais complexas, embora menos frequentes, são essenciais em momentos específicos

do ciclo de desenvolvimento, pois tratam de mudanças estruturais necessárias. Além disso, a análise dos tipos de refatorações mais comuns fornece informações valiosas para otimizar ferramentas e processos, adaptando-os às necessidades de cada projeto.

6 Conclusão

Este estudo teve como objetivo principal investigar a evolução e a qualidade de *code samples* por meio da análise da presença, tratamento e impacto de *code smells* e refatorações, utilizando as ferramentas *SonarQube* e *RefactoringMiner* aplicadas a repositórios de *code samples* de ecossistemas como *AWS* e *Azure*. Os resultados alcançados demonstraram a presença constante de *code smells* nesses projetos, mesmo em repositórios de alta qualidade, com a persistência prolongada de problemas, especialmente aqueles de severidade intermediária. Foi observado que há uma priorização na resolução de *code smells* críticos, classificados como *Blocker*, enquanto problemas de severidade menor, como *Minor* e *Major*, recebem menos atenção. Adicionalmente, a concentração de contribuições relacionadas à introdução de *code smells* aponta para a necessidade de melhorar as práticas de desenvolvimento e revisão de código.

As análises também indicaram padrões cíclicos na realização de refatorações, com picos em momentos específicos do ciclo de vida dos projetos, possivelmente associados a introduções de novas funcionalidades ou necessidades de reestruturações. Refatorações de baixa complexidade foram predominantes, o que sugere um foco em melhorias incrementais, enquanto refatorações mais complexas, embora menos frequentes, lidaram com mudanças estruturais mais amplas.

Entretanto, este estudo apresenta algumas limitações. A análise automatizada, apesar de eficiente, pode não capturar todas as nuances e complexidades presentes no código. Outra limitação importante foi a impossibilidade de correlacionar diretamente os dados sobre *code smells* e refatorações, pois os primeiros não contemplaram projetos *Java*, enquanto as refatorações analisadas foram exclusivamente dessa linguagem.

Apesar dessas limitações, o trabalho fornece um panorama detalhado sobre a presença, persistência e tratamento de *code smells* em *code samples*, revelando padrões cíclicos na realização de refatorações e destacando a importância de abordagens estratégicas no gerenciamento do débito técnico. A identificação dos tipos mais frequentes de *code smells* e refatorações também fornece subsídios valiosos para o desenvolvimento de ferramentas e práticas mais eficazes, promovendo uma melhor gestão da qualidade do código.

No futuro, seria interessante ampliar a análise para outras linguagens de programação, incluindo *Java* na investigação de *code smells*, o que possibilitaria uma melhor compreensão da relação entre esses elementos e as refatorações. Também seria valioso explorar como fatores como o tamanho do projeto, a experiência da equipe e as metodologias de desenvolvimento influenciam a persistência de *code smells* e os padrões de refatorações. Além disso, o desenvolvimento de ferramentas mais sofisticadas para detecção e priorização de *code smells*, considerando aspectos contextuais e a complexidade do código, poderia oferecer avanços significativos na área. Outros estudos também poderiam investigar a relação entre *code smells* e indicadores de qualidade de *software*, como manutenibilidade, confiabi-

lidade e segurança.

Conclui-se, portanto, que a gestão de *code smells* e a aplicação de refatorações são essenciais para garantir a qualidade, a manutenibilidade e a longevidade de projetos de *software*, incluindo os *code samples*. Este estudo destacou a importância de compreender a dinâmica desses elementos para adoção de práticas e ferramentas mais eficazes, que promovam um código limpo e sustentável. Ao investir em pesquisa e desenvolvimento nessa área, a comunidade científica e profissional contribui para a evolução da qualidade do *software* e para o aprimoramento das habilidades dos desenvolvedores.

Referências

[1] K. Manikas, Revisiting software ecosystems research: A longitudinal literature study, Journal of Systems and Software 117 (2016) 84–103. doi:https://doi.org/10.1016/j.jss.2016.02.003.
URL `https://www.sciencedirect.com/science/article/pii/S0164121216000406`

[2] G. Menezes, B. Cafeo, A. Hora, How are framework code samples maintained and used by developers? the case of android and spring boot, Journal of Systems and Software 185 (2022) 111146. doi:https://doi.org/10.1016/j.jss.2021.111146.
URL `https://www.sciencedirect.com/science/article/pii/S0164121221002417`

[3] M. I. Murillo, M. Jenkins, Technical debt measurement during software development using sonarqube: Literature review and a case study, in: 2021 IEEE V Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI), 2021, pp. 1–6. doi:10.1109/JoCICI54528.2021.9794341.

[4] R. Verma, K. Kumar, H. K. Verma, A study of relevant parameters influencing code smell prioritization in object-oriented software systems, in: 2021 6th International Conference on Signal Processing, Computing and Control (ISPCC), 2021, pp. 150–154. doi:10.1109/ISPCC53510.2021.9609478.

[5] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, G. Pinto, Are static analysis violations really fixed? a closer look at realistic usage of sonarqube, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 209–219. doi:10.1109/ICPC.2019.00040.

[6] N. Tsantalis, A. Ketkar, D. Dig, Refactoringminer 2.0, IEEE Transactions on Software Engineering 48 (3) (2022) 930–950. doi:10.1109/TSE.2020.3007722.

[7] R. Solingen, E. Berghout, The goal/question/metric method: A practical guide for quality improvement of software development (01 1999).

[8] sonarsource, Sonar static code analysis, acesso em: 7 dez. 2024.
URL `https://rules.sonarsource.com/`