Impacto de Commits Grandes na Qualidade de Código e na Estabilidade de Features em Aplicações Java de código aberto

Augusto Noronha Leite[1], **David Leong Ho**[1], **Gabriel Lourenço Reis Resende**[1], **Larissa Pedrosa Silva**[1], **Paula de Freitas Camargos** [1], **Pedro Máximo Campos do Carmo** [1]

[1] Instituto de Ciências Exatas e Informática (ICEI)

Pontifícia Universidade Católica de Minas Gerais (PUC Minas)

Belo Horizonte - MG - Brasil

{anleite, david.ho, gabriel.lourenco, larissa.silva.1439578, paula.camargos.1235001, pedro.carmo.1273582}@sga.pucminas.br

Abstract. *This study analyzes the impact of large commits on code quality and feature stability in open-source Java projects. Based on 5,125 commits collected from 100 GitHub repositories, we examined the relationship between commit size, introduction of code smells, and structural metrics (CBO, WMC, RFC). Results show that large commits introduce more and higher-severity smells, while small commits generate fewer and less critical ones. The effect on structural metrics is limited, but large commits concentrate extreme values and localized degradation. The findings highlight the importance of granular, cohesive, and frequently reviewed commits to sustain software quality and maintainability.*

Resumo. *Este estudo analisa o impacto de commits grandes na qualidade de código e na estabilidade de features em projetos Java de código aberto. Com base em 5.125 commits de 100 repositórios do GitHub, investigou-se a relação entre o tamanho do commit, a introdução de code smells e as métricas estruturais (CBO, WMC e RFC). Os resultados indicam que commits grandes introduzem mais smells e de maior severidade, enquanto commits pequenos geram menos e menos críticos. O efeito sobre as métricas é limitado, mas grandes commits concentram degradações pontuais e maior variabilidade. Conclui-se que granularidade, coesão e revisão contínua favorecem a qualidade e a manutenibilidade do código em longo prazo.*

1. Introdução

A qualidade dos *commits* é um fator determinante para a manutenibilidade e evolução sustentável de sistemas de software, especialmente em ambientes colaborativos e projetos *open source*, onde múltiplos desenvolvedores interagem de forma contínua com o mesmo repositório. *Commits* bem estruturados refletem boas práticas de engenharia de software de modo a favorecer a rastreabilidade das mudanças, o que, consequentemente, facilita a compreensão do histórico de evolução e a detecção de defeitos introduzidos ao longo do tempo [10].

Contudo, a qualidade dos *commits* é afetada por múltiplos fatores, que vão desde aspectos técnicos e organizacionais até fatores humanos. Pressões de prazo, sobrecarga

de trabalho e contextos de desenvolvimento intensivos tendem a reduzir a granularidade das alterações e aumentar a introdução de *code smells* [10]. *Commits* muito extensos, ou seja, com um grande número de alterações, ou *commits* realizados em horários atípicos frequentemente refletem fadiga cognitiva e menor rigor na revisão de código, ampliando o risco de degradação estrutural do sistema e dificultando práticas de manutenção de código [12].

Além dos fatores processuais, a estrutura interna do *software* também influencia o padrão e a qualidade dos *commits*. Projetos com alto acoplamento entre módulos ou baixa coesão tendem a gerar *commits* que modificam simultaneamente múltiplos componentes, o que pode indicar dependências excessivas entre as classes e frágil modularização.

Ainda que diversos estudos explorem *code smells* e práticas de revisão de código [3, 2, 5], existem lacunas quanto à compreensão de como padrões de granularidade, acoplamento e esforço de *commit* se relacionam com a saúde evolutiva e a confiabilidade de um projeto. Compreender essas relações pode auxiliar equipes de desenvolvimento a adotar estratégias mais eficazes de versionamento e revisão.

A motivação deste estudo surge do pressuposto de que *commits* com um grande volume de modificações comprometem a estabilidade do sistema e promovem uma piora na qualidade estrutural de um *software*. A partir disso, busca-se compreender de que forma o volume e a granularidade das mudanças influenciam na probabilidade degradação estrutural do *software*.

O objetivo deste trabalho, então, é analisar a relação entre o tamanho dos *commits* e a introdução de *code smells*, buscando compreender em que medida alterações de grande porte, isto é, número alto de alterações em arquivos e/ou linhas do código, influenciam a qualidade interna do código e a manutenibilidade sob a perspectiva de desenvolvedores atuando em repositórios *Java* de código aberto.

Para apoiar o desenvolvimento deste estudo, foi utilizada a abordagem GQM (*Goal-Question-Metric*), proposta por Victor Basili, auxiliando na identificação de métricas significativas para o processo de medição. Essa abordagem parte da formulação de questões de pesquisa (RQs, do inglês, *Research Questions*), que permitem refinar o objetivo do estudo de forma quantitativa e orientar a coleta e análise dos dados. As RQs deste trabalho focam em: entender qual tamanho de *commit* mais se relaciona com a introdução de *code smells*, qual o impacto do tamanho dos *commits* na qualidade de código e qual a proporção de cada tipo de criticidade de *code smell* de acordo com o tamanho do *commit*.

A estrutura deste artigo está organizada da seguinte forma: a Seção 2 apresenta o referencial teórico, abordando os principais conceitos e fundamentos que sustentam o estudo; a Seção 3 descreve os trabalhos relacionados, situando esta pesquisa no contexto da literatura existente; a Seção 4 detalha a metodologia adotada, incluindo os critérios de coleta, extração e análise dos dados; a Seção 5 apresenta os principais resultados empíricos, respondendo às perguntas de pesquisa com base nas métricas definidas; a Seção 6 discute criticamente os achados, destacando suas implicações e relações com estudos anteriores; a Seção 7 expõe as ameaças à validade, abordando possíveis limitações metodológicas que possam ter influenciado os resultados; e a Seção 8 apresenta as conclusões do trabalho.

2. Referencial Teórico

O referencial teórico apresenta os principais conceitos, estudos e abordagens que fundamentam a pesquisa, fornecendo base científica para a análise e a discussão dos resultados. Nesta seção, são explorados os temas relacionados à qualidade de *software*, métricas, *code smells*, refatoração e qualidade dos *commits* para contextualização do problema.

2.1. Qualidade de *Software*

A qualidade de *software* pode ser definida como uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam [1]. Em outras palavras, um *software* de qualidade é aquele que atende às necessidades dos usuários e mantém-se confiável, eficiente e de fácil manutenção ao longo de sua evolução.

A busca pela qualidade não se limita ao produto final ou somente ao processo de codificação, mas envolve, também, processos utilizados durante o desenvolvimento, incluindo revisões sistemáticas de código, testes (exploratórios e automatizados) e práticas de versionamento de código. Desse modo, qualidade está relacionada tanto a fatores externos, como usabilidade e desempenho, quanto a fatores internos, como modularidade, acoplamento e coesão.

O foco deste estudo está na qualidade do código, onde o código-fonte e os artefatos relacionados devem estar em conformidade com os padrões locais de codificação e apresentar características que facilitem a manutenção [1], isto é, facilidade de compreensão por parte dos desenvolvedores, nível de acoplamento, coesão e complexidade do código.

2.2. *Code Smells*

Um *code smell* é um indício superficial que aponta para um problema mais profundo no sistema [2], em outras palavras, um sintoma associado a problemas de manutenibilidade de *software*. O termo *smell* (do inglês, “cheiro”) refere-se justamente a algo de fácil detecção, um sinal de que pode haver algo errado no código [2].

Porém um *smell* nem sempre representa um defeito real; ele atua, na verdade, como um indicador de possíveis fragilidades estruturais, sugerindo pontos onde a refatoração pode trazer benefícios de longo prazo. Segundo Emden e Moonen [3], os *code smells* funcionam como guias práticos para o processo de refatoração, auxiliando desenvolvedores a decidir quando e o que refatorar com base em evidências do código.

A identificação desses *smells* pode ser realizada tanto de forma manual, por meio de inspeção de código, quanto de maneira automatizada, utilizando ferramentas que aplicam métricas estruturais. Métricas como complexidade ciclomática, número de linhas de código (LOC), acoplamento entre classes (CBO) e coesão de métodos (LCOM) são amplamente empregadas para mensurar a presença e a intensidade dos *code smells* [4].

2.3. Métricas de *Software* e *CK*

As métricas de *software*, ou métricas de código, são medidas concebidas para gerenciar o processo de desenvolvimento de *software*, de modo que a capacidade de medir o processo é amplamente reconhecida como um componente importante para a melhoria deste [4].

O presente estudo se refere à linguagem *Java*, uma linguagem orientada a objetos, na qual métricas tradicionais, originalmente desenvolvidas para paradigmas estruturados, não capturam adequadamente os conceitos e princípios fundamentais da orientação a objetos, como encapsulamento, herança e polimorfismo. A partir dessa lacuna, Shyam R. Chidamber e Chris F. Kemerer propuseram, em 1994, um conjunto de métricas fundamentadas para o *design* orientado a objetos, conhecido como *CK Metrics Suite*.

Esse conjunto de métricas é composto por seis indicadores principais: WMC (*Weighted Methods per Class*), que mede a complexidade de uma classe com base na soma das complexidades de seus métodos; DIT (*Depth of Inheritance Tree*), que avalia a profundidade da hierarquia de herança; NOC (*Number of Children*), que indica o grau de reutilização e especialização da classe; CBO (*Coupling Between Objects*), que mede o nível de dependência entre classes; RFC (*Response For a Class*), que estima o potencial de resposta de uma classe frente a mensagens recebidas; e LCOM (*Lack of Cohesion in Methods*), que quantifica a falta de coesão entre os métodos de uma classe.

Métricas como CBO e LCOM, por exemplo, têm relação direta com a manutenibilidade e com a ocorrência de *code smells*, pois altos níveis de acoplamento e baixa coesão tendem a indicar fragilidades no *design* [5].

O objetivo dessas métricas é auxiliar os usuários a compreender a complexidade do *design*, detectar falhas de *design* e prever resultados como defeitos, esforço de teste e manutenção. As métricas de código também são usadas para identificar a presença de *code smells*, e as métricas relacionadas à complexidade, acoplamento, coesão e tamanho estão entre as mais frequentemente empregadas para esse propósito [6].

2.4. Versionamento de Código e *Commits*

O versionamento de código é a prática de registrar, organizar e controlar as modificações realizadas nos arquivos de um projeto de *software* ao longo do tempo, permitindo rastrear o histórico de alterações, identificar a autoria e o propósito de cada modificação, restaurar versões anteriores em caso de falhas e facilitar o trabalho colaborativo entre desenvolvedores por possibilitar o gerenciamento de múltiplas linhas de desenvolvimento.

Um *commit* é definido como a contribuição atômica de código-fonte para um repositório, composta por linhas de código, comentários e linhas vazias. Para estimar o tamanho de um *commit*, considera-se o total de linhas de código-fonte (SLoC, do inglês *Source Lines of Code*) adicionadas e removidas [7]. Segundo Arafat e Riehle [8], *commits* entre 1 e 100 SLoC são classificados como pequenos (*single commits*), enquanto *commits* com mais de 100 SLoC são considerados grandes (*aggregate commits*).

Essa classificação, derivada da análise de mais de 9.000 projetos *open source*, serve como base para a definição adotada neste estudo, pois permite distinguir mudanças rotineiras de alterações de alto impacto. Purushothaman e Perry [9] observam que a probabilidade de falhas aumenta significativamente em *commits* maiores, alcançando cerca de 35% em *commits* com mais de 50 SLoC e aproximadamente 50% em *commits* superiores a 500 SLoC.

Alguns *code smells* relacionados ao tamanho e à complexidade, como *Blob* e *Complex Class*, podem surgir gradualmente ao longo de várias modificações. Contudo, Tufano

et al. [10] apontam que, quando esses *smells* são introduzidos, tendem a estar associados a um crescimento abrupto nas métricas de *Lines of Code* (LOC) ou *Weighted Methods per Class* (WMC), indicando que a classe sofreu uma grande submissão de código de uma só vez, um comportamento típico de *commits* volumosos.

Em adição, Santana et al. [11] mostram que quando aglomerações de *code smells* ocorrem na mesma porção de código, se tornam mais prejudiciais à qualidade do que *smells* isolados. Classes com aglomerações de diferentes tipos de *smells* mudam com mais frequência e intensidade do que classes com um único *smell* ou classes limpas. Ou seja, grandes *commits* não apenas dificultam a rastreabilidade das mudanças, mas também podem contribuir para a degradação estrutural do código ao favorecer a concentração e a interação de *smells* em regiões específicas do sistema.

3. Trabalhos Relacionados

Esta seção apresenta pesquisas previamente realizadas que se relacionam, de forma direta ou indireta, aos objetivos deste estudo. As Tabelas 1 e 2 sintetizam os principais trabalhos analisados, destacando seus autores e a contribuição central de cada estudo para o contexto desta pesquisa.

Table 1: Referências Bibliográficas Selecionadas

Título	**Autores**	**Contribuição**
A Metrics Suite for Object Oriented Design	Shyam R. Chidamber, Chris F. Kemerer	Propõe uma suíte de métricas para medir complexidade, coesão e acoplamento em design orientado a objetos, auxiliando na avaliação da qualidade do software.
A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques	Mansi Agnihotri, Anuradha Chug	Apresenta uma visão sistemática sobre métricas de software, *code smells* e técnicas de refatoração, destacando abordagens que favorecem a manutenção e qualidade de projetos Java.
Automatic Classification of Large Changes into Maintenance Categories	Abram Hindle, Daniel M. German, Michael W. Godfrey, Richard C. Holt	Classifica grandes *commits* em categorias de manutenção usando apenas metadados.
Code Smells Detection and Visualization: A Systematic Literature Review	José P. dos Reis, Fernando B. e Abreu, Glauco de F. Carneiro, Craig Anslow	Levanta técnicas de detecção de *code smells* e evidencia a escassez de ferramentas de visualização voltadas à manutenção de software.
Commit Quality in Five High Performance Computing Projects	Kapil Agrawal, Sadika Amreen, Audris Mockus	Mostra que *commits* maiores em projetos de computação de alto desempenho (HPC) aumentam a complexidade e dificultam a manutenção.
Do Time of Day and Developer Experience Affect Commit Bugginess?	Jon Eyolfson, Lin Tan, Patrick Lam	Aponta que fatores humanos e contextuais, como horário de trabalho e experiência do desenvolvedor, influenciam a qualidade e a incidência de erros nos *commits*.

Continua na próxima página

Continuação da Tabela 1— Referências Bibliográficas

Título	Autores	Contribuição
Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects	Ramanath S., M. S. Krishnan	Evidencia que métricas de complexidade em design orientado a objetos impactam diretamente a qualidade e a ocorrência de defeitos no código.
Estimating Commit Sizes Efficiently	Philipp Hofmann, Dirk Riehle	Propõe um método eficiente para estimar o tamanho de *commits*.
Impact of Abstract Factory and Decorator Design Patterns on Software Maintainability: Empirical Evaluation using CK Metrics	Aisha Kurmangali, Muhammad E. Rana, Wan N. W. A. Rahman	Mostra que o uso de padrões de *design* melhora a qualidade estrutural e a manutenibilidade do código.
Java Quality Assurance by Detecting Code Smells	Eva Van Emden, Leon Moonen	Demonstra que a detecção automática de *code smells* pode apoiar a análise da qualidade dos commits e indicar pontos de refatoração em projetos *Java*.
On the Nature of Commits	Lile P. Hattori, Michele Lanza	Define padrões de tamanho de *commits* e os relaciona a tipos de atividade de desenvolvimento.
Predicting Bug Inducing Commits Using Commit-State Transition Analysis	Alireza T. Barzoki, Kostas Kontogiannis	Reforça a importância de monitorar métricas entre *commits* para prever falhas e compreender como variações de qualidade e dívida técnica antecedem *code smells* e degradação do código.
SmellyCode++: Multi-Label Dataset for Code Smell Detection	Nawaf Alomar, Amal Alazba, Hamoud Aljamaan, Mohammad Alshayeb	Oferece uma base multi-rótulo para detecção de *code smells*, permitindo análises mais precisas sobre a relação entre tipos de *smells* e a qualidade dos *commits* em projetos *Java*.
The Commit Size Distribution of Open Source Software	Oliver Arafat, Dirk Riehle	Define limites quantitativos para classificar *commits* e avaliar seu impacto na qualidade do código.
Toward Understanding the Rhetoric of Small Source Code Changes	Ranjith Purushothaman, Dewayne E. Perry	Demonstra empiricamente que mudanças pequenas tendem a introduzir menos erros, reforçando a importância da granularidade dos *commits* para manter a qualidade e estabilidade do código.
Unraveling the Impact of Code Smell Agglomerations on Code Stability	Amanda Santana, Eduardo Figueiredo, Juliana A. Pereira	Mostra que *code smells* aumentam com *commits* grandes, afetando a estabilidade do código.
Use of Relative Code Churn Measures to Predict System Defect Density	Nachiappan Nagappan, Thomas Ball	Mostra que o *code churn* é bom preditor de defeitos e da qualidade do código.

Continua na próxima página

Continuação da Tabela 1— Referências Bibliográficas

Título	Autores	Contribuição
When and Why Your Code Starts to Smell Bad	Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano D. Penta, Andrea De Lucia, Denys Poshyvanyk	Evidencia que *code smells* surgem majoritariamente em *commits* grandes e sob pressão de prazo, reforçando a relação entre granularidade das alterações e degradação da qualidade do código.

Um dos estudos mais relevantes é o de Hattori e Lanza, que analisou mais de 70 mil *commits* de nove projetos de código aberto para compreender a natureza e o tamanho das alterações realizadas. Os autores identificaram que a distribuição dos *commits* segue a lei de Pareto, sendo a maioria composta por pequenas modificações, enquanto os *commits* grandes, embora menos frequentes, concentram atividades de desenvolvimento e gestão de código. Essa observação é particularmente relevante para o estudo atual, pois reforça a importância de compreender o impacto das grandes submissões de código, que tendem a introduzir mudanças amplas e complexas, afetando diretamente a estabilidade e a qualidade das features em sistemas *Java* de código aberto.

De forma complementar, Agrawal et al. investigaram a qualidade dos *commits* em cinco grandes projetos de Computação de Alto Desempenho (HPC) e a compararam com projetos de código aberto tradicionais. Os resultados revelaram que, embora os *commits* desses projetos apresentassem mensagens mais detalhadas e bem documentadas, o tamanho médio das alterações era significativamente maior, o que dificulta o rastreamento de mudanças e aumenta o risco de introdução de defeitos. Esses resultados se conectam diretamente com o tema deste trabalho, ao mostrar que *commits* grandes, mesmo quando bem descritos, podem prejudicar a manutenção e favorecer o surgimento de *code smells* devido à baixa granularidade das alterações.

Em um esforço voltado à automatização da análise de grandes *commits*, Hindle et al. propuseram um modelo para classificar automaticamente grandes mudanças de código em categorias de manutenção, como corretiva, adaptativa e perfectiva. Utilizando técnicas de aprendizado de máquina aplicadas aos metadados e mensagens de *commit*, o estudo demonstrou que é possível identificar o tipo e o propósito das alterações com boa precisão. Essa abordagem contribui para o presente trabalho, pois ajuda a entender o contexto em que os *commits* grandes ocorrem, o que é essencial para relacionar seu tamanho com a introdução de *smells*.

Por outro lado, Eyolfson, Tan e Lam analisaram fatores humanos e sociais relacionados à probabilidade de um *commit* introduzir erros. O estudo, realizado sobre projetos como o *Linux kernel* e o *PostgreSQL*, mostrou que *commits* feitos de madrugada ou sob fadiga tendem a ser mais propensos a conter defeitos. Além disso, a experiência e a frequência de contribuição dos desenvolvedores influenciam diretamente na qualidade do código. Esses resultados reforçam a ideia de que pressões de tempo e carga de trabalho impactam a qualidade dos *commits*, especialmente quando envolvem grandes volumes de código.

Por fim, Hofmann e Riehle propuseram um método eficiente para estimar o tamanho real de *commits* em projetos de código aberto. O algoritmo desenvolvido, baseado em regressão linear, permite calcular com precisão o volume de alterações (*SLOC* adicionadas e removidas), sendo útil para análises em larga escala. Essa contribuição é importante para este estudo, pois oferece uma forma prática e confiável de medir o tamanho dos *commits* e investigar sua relação com a introdução de *code smells* e a qualidade do código.

4. Metodologia

A metodologia deste estudo foi organizada em um fluxo de quatro etapas principais, estruturadas de forma a permitir a coleta, processamento e análise sistemática dos dados. Cada etapa foi cuidadosamente planejada para garantir a reprodutibilidade dos resultados e a integridade das informações coletadas.

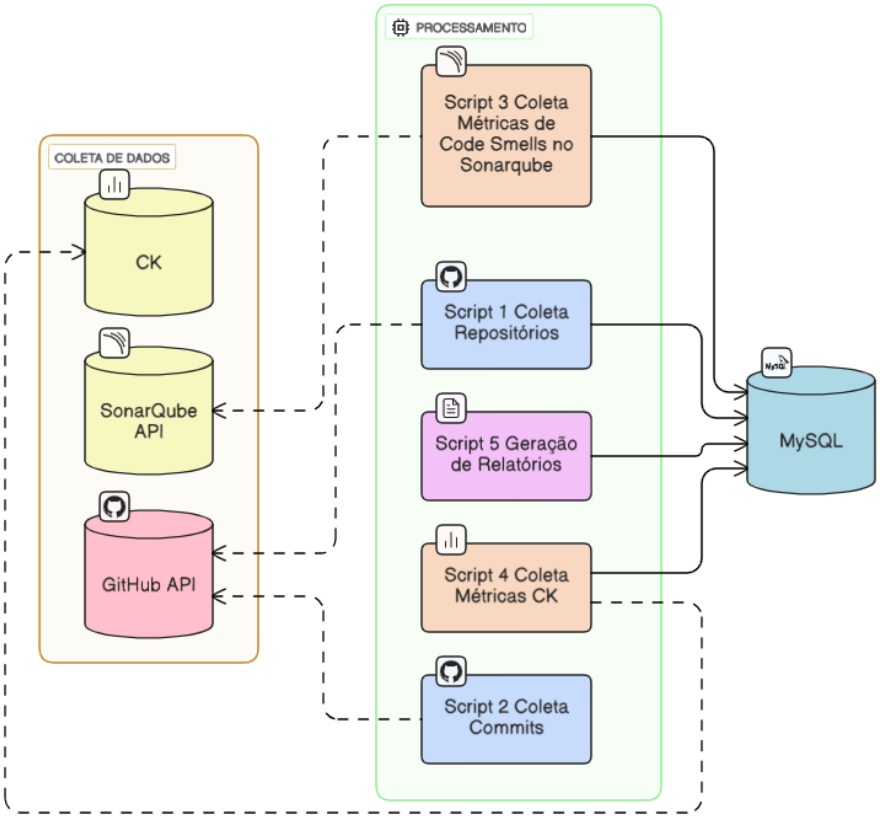

Figure 1. Modelo lógico relacional dos dados

4.1. Hipóteses nulas e alternativas

Para responder às RQs definidas no *Goal, Question, Metric* (GQM) , foram formuladas as seguintes hipóteses nulas (H0) e alternativas (H1).

RQ1: Qual tamanho de commits mais se relaciona com a introdução de *code smell*?

- H0: Não há correlação estatisticamente significativa entre o tamanho do *commit* e a quantidade de *code smells* introduzidos.
- H1: *Commits* pequenos, os quais têm menos de cem linhas alteradas, têm baixa incidência de *code smells*, indicando que granularidade fina ajuda a manter qualidade, portanto, *commits* maiores, que contêm mais de cem *SLOC* alteradas, introduzem proporcionalmente mais *code smells*.

RQ2: Qual o impacto do tamanho dos *commits* na qualidade do código?

- H0: O tamanho dos *commits* não impacta significativamente as métricas de qualidade do código.
- H1: *Commits* grandes, ou seja, com mais de cem *SLOC* alteradas, estão positivamente associados a aumentos imediatos e estatisticamente significativos na variabilidade de métricas de qualidade do código, como: variação positiva em WMC (*Weighted Methods per Class*), variação positiva em CBO (*Coupling Between Objects*) e variação negativa em um *proxy* de testabilidade calculado pela métrica RFC (*Response for a Class*).

RQ3: Qual a proporção de cada tipo de criticidade de *code smell* de acordo com o tamanho do *commit*?

- H0: A proporção dos tipos de criticidade de *code smells* é independente do tamanho dos *commits*.
- H1: O tamanho dos *commits* está associado ao nível de criticidade dos *code smells*, de forma que os *commits* pequenos tendem a introduzir *code smells* de baixa criticidade (*minor/info*) e os *commits* grandes tendem a introduzir uma quantidade maior de *code smells* de intermediária a alta criticidade (*critical/blocker*).

4.2. Etapa 1: Coleta e persistência dos repositórios

Nesta primeira etapa foi realizada a busca e coleta dos cem repositórios *open-source* mais populares do *GitHub* cuja linguagem predominante é *Java*. Para esta fase, foi utilizada a API pública do *GitHub*, com consultas automatizadas para filtrar os repositórios conforme os seguintes critérios:

- **Linguagem principal:** *Java*;
- **Estado:** público;
- **Ordenação:** número de estrelas (*stars*).

Os dados extraídos foram o nome, a *url* e a quantidade de estrelas do repositório. Dessa forma, essas informações foram persistidas em um banco de dados relacional *MySQL* na tabela destinada aos registros dos repositórios (*repositories*) para uso nas etapas seguintes. Essa estrutura permitiu o controle de repositórios já processados e a repetição das análises de forma incremental.

4.3. Etapa 2: Coleta e persistência dos *commits*

Após a persistência dos repositórios, foi realizada a coleta dos cem *commits* mais recentes de cada um deles, também por meio da API do *GitHub*. Além disso, para garantir a qualidade da extração de métricas nas etapas futuras, foi realizado um filtro para armazenar somente *commits* que contenham um arquivo de código *Java* na sua alteração. Cada *commit* foi armazenado com informações relevantes, incluindo:

- *Hash* (SHA) do commit;
- Quantidade de linhas adicionadas e removidas (*SLOC*);
- Número de arquivos alterados.

Esses dados foram armazenados em uma tabela chamada *commits*, vinculada ao repositório correspondente. Essa estruturação permitiu o rastreamento histórico das modificações realizadas no código e a correlação com as métricas obtidas posteriormente.

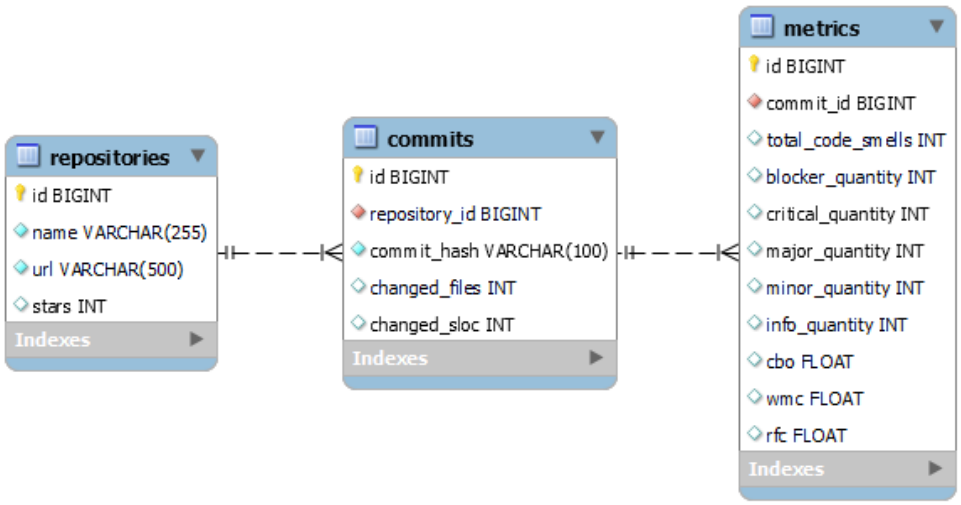

Figure 2. Modelo lógico relacional dos dados

4.4. Etapa 3: Análise automatizada das métricas de qualidade

Nesta etapa, cada *commit* coletado foi analisado automaticamente por meio de ferramentas de análise estática de qualidade de código, com o objetivo de extrair informações detalhadas sobre a manutenção e complexidade do projeto. Para isso, foram empregadas duas ferramentas complementares, abordadas separadamente a seguir.

4.4.1. Análise de *Code Smells* com *SonarQube*

O *SonarQube* foi utilizado para detectar o número total de *code smells* e a quantidade referente a cada tipo de criticidade, classificando-os em: *blocker, critical, major, minor e info*. A análise foi realizada de forma automatizada por meio de *scripts Python*, que efetuaram o checkout de cada *commit*, submeteram o código ao *SonarQube* e calcularam a diferença da análise estática do projeto no estado do *commit* analisado em relação ao estado do projeto anteriormente às alterações efetuadas. Com isso, os resultados foram armazenados no banco de dados *MySQL* na tabela *metrics*, a qual contém todos os dados de métricas de cada *commit* analisado .

4.4.2. Análise com *CK Metrics*

Paralelamente, a suíte de métricas *Chidamber & Kemerer (CK)* foi utilizada para extrair métricas orientadas a objetos, como WMC (*Weighted Methods per Class*), CBO (*Coupling Between Objects*) e RFC (*Response for a Class*). Assim como na análise com *SonarQube, scripts Python* automatizaram a execução das métricas para cada *commit*, calculando o *delta* dos valores obtidos com os resultados encontrados para o estado anterior do projeto e armazenando os resultados no banco de dados. Sendo assim, essa abordagem permitiu avaliar a complexidade, o acoplamento, o esforço para manutenção e a testabilidade das classes envolvidas no *commit*.

4.5. Etapa 4: Processamento e análise dos dados

Com os dados devidamente coletados e armazenados, foram executados *scripts* de processamento e análise estatística. Esses *scripts* tiveram como objetivos principais calcular as métricas referentes às três perguntas estabelecidas no *Goal-Question-Metric* (GQM) e plotar gráficos com os resultados encontrados.

Com isso, as métricas e gráficos produzidos nesta etapa serviram de base para a discussão dos resultados, permitindo identificar tendências e padrões que auxiliam na compreensão do impacto dos *code smells* e das métricas de qualidade no desenvolvimento dos sistemas estudados.

5. Resultados

Esta seção apresenta os resultados obtidos a partir do tratamento e interpretação dos dados. Os achados são organizados conforme cada questão de pesquisa, destacando padrões, variações e correlações relevantes entre as variáveis analisadas.

RQ1: Qual tamanho de commits mais se relaciona com a introdução de code smells?

A Figura 3 consiste em um gráfico de violino que apresenta a distribuição dos *code smells* em função do tamanho dos *commits*.

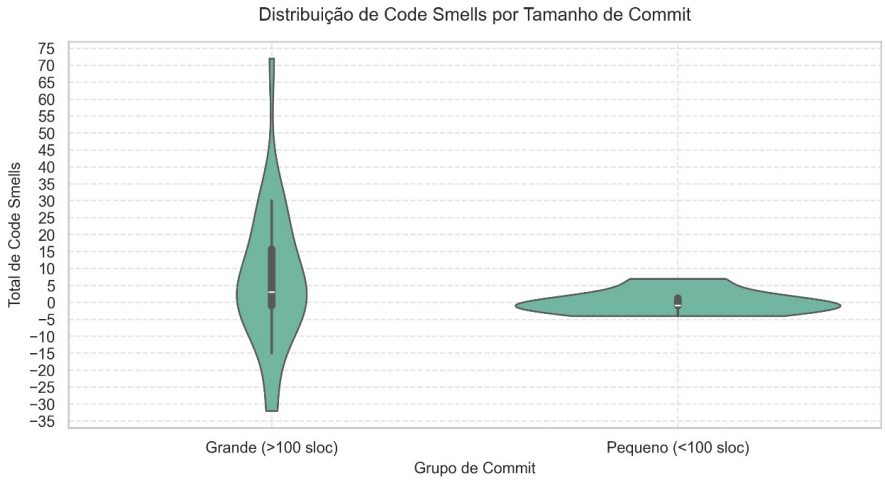

Figure 3. Distribuição dos code smells em função do tamanho dos commits.

Nota-se que *commits* grandes (> 100 *SLOC*) exibem uma maior dispersão e amplitude na quantidade de *smells* introduzidos, com valores que chegam a ultrapassar 70 ocorrências caracterizando-se como *outliers*. Essa maior amplitude de dispersão indica que há uma grande variabilidade no comportamento desses *commits*: enquanto alguns introduzem poucos *smells*, outros inserem uma quantidade extremamente elevada.

Em termos práticos, isso sugere que *commits* grandes tendem a ser menos homogêneos, podendo representar tanto alterações estruturadas e bem planejadas quanto modificações extensas e desorganizadas mais propensas a gerar *code smells* de diferentes tipos.

Já os *commits* pequenos (< 100 *SLOC*) concentram-se em torno de valores muito inferiores, com baixa variabilidade e mediana próxima de zero. Essa concentração evidencia maior consistência e previsibilidade na qualidade do código, indicando que mudanças menores costumam introduzir menos *code smells*. Em conjunto, o gráfico reforça a hipótese de que *commits*

com maior quantidade de arquivos modificados e/ou linhas de código alteradas estão mais sujeitos à introdução intensa e irregular de *code smells*, enquanto *commits* pequenos tendem a preservar maior estabilidade e qualidade no código.

A Tabela 3 resume os resultados obtidos para as métricas M1, M2 e a densidade de *code smells*, considerando os dois grupos de *commits*. Os resultados mostram que tanto o valor absoluto de *smells* por arquivo (M1) quanto a densidade de *smells* por *sLOC* (M2) são maiores em *commits* grandes.

Isso indica que *commits* grandes não apenas concentram mais *smells* em termos absolutos, mas também proporcionalmente em relação ao tamanho da alteração. O gráfico (Figura 3) evidencia ainda que *commits* grandes apresentam maior variabilidade e extremos mais altos de *code smells*, o que reforça que, quando mal planejados, esses *commits* podem introduzir *smells* em grande volume de uma só vez.

Table 2. Densidade de code smells em função do tamanho dos commits.

Métrica	*Commits Pequenos*	*Commits Grandes*	Relação (grande/pequeno)
M1 (*smells/arquivo*)	0,1176	0,4456	3,7873
M2 (*smells/sLOC*)	0,000025	0,000056	2,2538

Os achados sugerem que *commits* grandes estão mais associados à introdução massiva e pontual de *code smells*, enquanto *commits* pequenos, embora mais frequentes, apresentam impacto proporcionalmente menor por linha de código alterada. Confirmando-se parcialmente a hipótese de que o tamanho do *commit* influencia a degradação da qualidade do código, especialmente quando grandes volumes de alterações são submetidos de forma agregada e sem revisão granular.

RQ2: Qual o impacto do tamanho dos commits na qualidade do código?

A Figura 4 apresenta a distribuição das métricas de qualidade - *Coupling Between Objects* (CBO), *Weighted Methods per Class* (WMC) e *Response For a Class* (RFC) - de acordo com o tamanho dos *commits*.

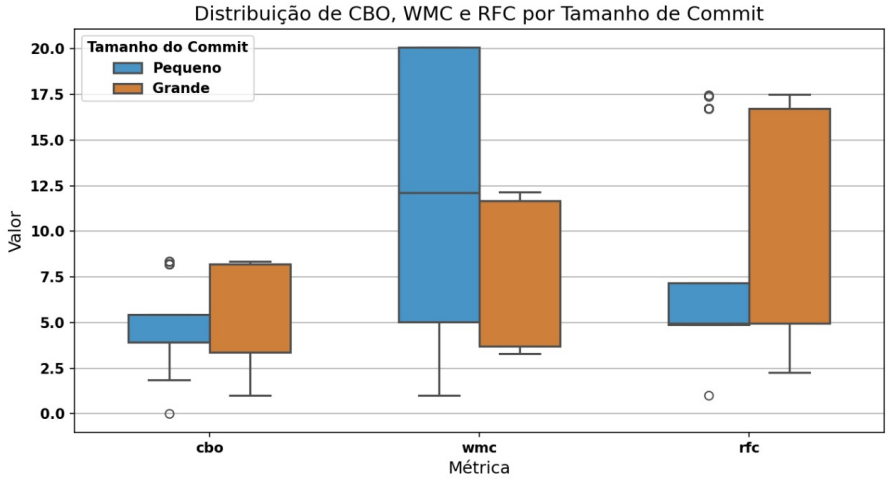

Figure 4. Distribuição de CBO, WMC e RFC por tamanho de commit.

Observa-se que as medianas das três métricas permanecem próximas entre *commits* grandes e pequenos, indicando que o tamanho da submissão exerce pouca influência sobre os valores centrais das métricas estruturais.

No entanto, há diferenças na dispersão e na presença de valores extremos, principalmente na métrica WMC, sugerindo que *commits* pequenos apresentam maior variação na complexidade ciclomática, enquanto *commits* grandes tendem a concentrar valores mais homogêneos, porém com casos isolados de maior acoplamento (CBO) e testabilidade (RFC).

A Tabela 2 detalha as médias e medianas observadas para cada métrica, considerando os dois grupos de *commits*. Nota-se que as diferenças são sutis, com variações inferiores a uma unidade na maioria dos casos.

Table 3. Densidade de code smells em função do tamanho dos commits.

Métrica	*Commits Pequenos*	*Commits Grandes*	Relação (grande/pequeno)
M1 (*smells/arquivo*)	0,1176	0,4456	3,7873
M2 (*smells/sLOC*)	0,000025	0,000056	2,2538

Esses resultados indicam que, em média, o tamanho do *commit* não altera de forma significativa os níveis de complexidade, acoplamento e responsabilidade das classes afetadas. A leve redução das medianas de WMC em *commits* grandes pode sugerir uma distribuição mais equilibrada da lógica, enquanto o pequeno aumento no CBO pode refletir a introdução de dependências entre módulos como consequência de grandes integrações.

Quanto à testabilidade, representada pela métrica RFC, os valores são praticamente equivalentes entre os grupos, sugerindo que o tamanho do *commit* não afeta substancialmente a quantidade de métodos invocados ou a responsabilidade das classes.

Os achados da RQ2 apontam que o tamanho do *commit* exerce pouco impacto sobre as métricas estruturais de qualidade de código, embora *commits* grandes apresentem maior variabilidade e casos extremos. Esses resultados complementam a RQ1, indicando que, apesar de *commits* grandes introduzirem mais *code smells*, seu efeito sobre a estrutura interna do código tende a ser menos sistemático, ocorrendo de maneira localizada e pontual.

RQ3: Qual a proporção de cada tipo de criticidade de code smell de acordo com o tamanho do commit?

A Figura 5 apresenta a proporção de criticidade dos *code smells* de acordo com o tamanho do *commit*. Pelo grafico, pode-se perceber que *commits* grandes (≥ 100 *SLOC*) concentram uma proporção mais elevada de *code smells* classificados como *major* e *minor*, representando conjuntamente cerca de 60% a 70% do total identificado.

Em contrapartida, *commits* pequenos (< 100 *SLOC*) exibem uma distribuição mais equilibrada, com predominância das categorias *info* ($\approx 45\%$) e *minor* ($\approx 35\%$), e baixa incidência de *smells* críticos (*critical* e *blocker*).

Podemos observar, também, valores negativos presentes no gráfico. Isso indica a correção efetiva de um *code smell* que estava presente no *commit* anterior.

Essa interpretação dos resultados entra em concordância com achados anteriores ([10], [11]), que afirmam que *commits* com maior quantidade de alterações frequentemente estão associados à implementação de novas funcionalidades ou grandes refatorações.

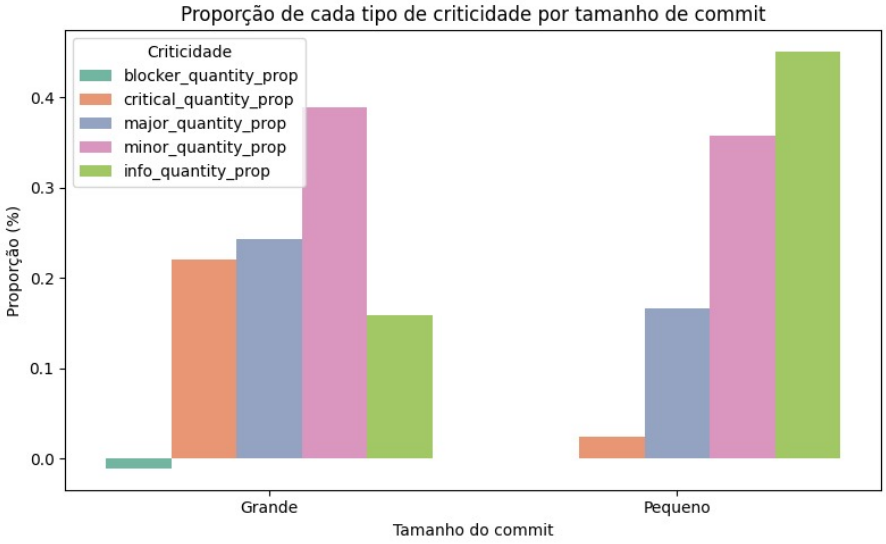

Figure 5. Percentual de criticidade por tamanho de commit.

A Tabela 3 resume as proporções observadas por tipo de criticidade, conforme os grupos de *commits*. Os valores percentuais foram estimados com base nas distribuições apresentadas no gráfico.

Table 4. Proporções observadas por tipo de criticidade.

Criticidade	*Commits Grandes* (≥ 100 *SLOC*)	*Commits Pequenos* (< 100 *SLOC*)
Blocker	$\approx$ 0–1%	$\approx 0\%$
Critical	$\approx 22\%$	$\approx 3\%$
Major	$\approx 24\%$	$\approx 17\%$
Minor	$\approx 38\%$	$\approx 35\%$
Info	$\approx 15\%$	$\approx 45\%$

Os resultados indicam que *commits* grandes tendem a acumular *code smells* de maior severidade, principalmente nas categorias *major* e *critical*, sugerindo que alterações maiores introduzem efeitos colaterais mais severos no *design* do código enquanto *commits* pequenos tendem a introduzir *smells* menos críticos, predominantemente de natureza informativa (*info*) ou de menor impacto estrutural (*minor*).

A análise da RQ3 evidencia que o tamanho do *commit* influencia não apenas a quantidade, mas também a criticidade dos *code smells* introduzidos. Alterações de maior porte estão mais propensas a incluir *smells* de criticidade elevada, enquanto modificações menores tendem a gerar *smells* de baixo impacto.

References

[1] Pressman, R. S. and Maxim, B. R. (2016). *Engenharia de Software: uma abordagem profissional*. 8ª ed. McGraw-Hill.

[2] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[3] Van Emden, E. and Moonen, L. (2002). Java Quality Assurance by Detecting Code Smells. In *Ninth Working Conference on Reverse Engineering (WCRE 2002)*, 97–107.

[4] Chidamber, S. R. and Kemerer, C. F. (1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6), 476–493.

[5] Reis, C. A., Santos, A. L., and Mendonça, M. G. (2020). Code Smells and Refactoring: A Systematic Mapping Study. *Journal of Systems and Software*.

[6] Agnihotri, M. and Chug, A. (2020). A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques. *International Journal of Information Technology and Computer Science*, 12(6), 1–16.

[7] Hofmann, F. and Riehle, D. (2009). The Commit Size Distribution of Open Source Software. In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS)*, 1–8.

[8] Arafat, O. and Riehle, D. (2009). The Commit Size Distribution of Open Source Software. In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS)*, 1–8.

[9] Purushothaman, R. and Perry, D. E. (2005). Toward Understanding the Rhetoric of Small Changes. *IEEE Transactions on Software Engineering*, 31(6), 511–526.

[10] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and Why Your Code Starts to Smell Bad. *IEEE Transactions on Software Engineering*, 43(11), 1063–1088.

[11] Santana, A., Figueiredo, E., and Pereira, J. A. (2021). Unraveling the Impact of Code Smell Agglomerations on Code Stability. *Empirical Software Engineering*, 26, 72–98.

[12] Eyolfson, J., Tan, L., and Lam, P. (2011). Do Time of Day and Developer Experience Affect Commit Bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR 2011)*, 153–162.

[13] Hofmann, P. and Riehle, D. (2017). Estimating Commit Sizes Efficiently. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2017)*, 295–306.

[14] Hattori, L. P. and Lanza, M. (2008). On the Nature of Commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 63–72.