Gigantes ou Moinhos de Vento? Um Estudo Empírico sobre a Evolução e Disseminação de Arquivos Grandes

Mateus Batichotti
Universidade Tecnológica Federal do Paraná
Campo Mourão, Brazil
matebatichotti@gmail.com

Tiago Defendi
Universidade Tecnológica Federal do Paraná
Campo Mourão, Brazil
tiagodefendidasilva@gmail.com

Igor Scaliante Wiese
Universidade Tecnológica Federal do Paraná
Campo Mourão, Brazil
igor@utfpr.edu.br

Ivanilton Polato
Universidade Tecnológica Federal do Paraná
Campo Mourão, Brazil
ipolato@professores.utfpr.edu.br

Reginaldo Ré
Universidade Tecnológica Federal do Paraná
Campo Mourão, Brazil
reginaldo@utfpr.edu.br

Resumo

O tamanho de um arquivo pode representar desafios à medida que um projeto de software evolui. Arquivos com um grande número de linhas podem dificultar a compreensão, complicar as revisões de código e tornar a manutenção mais difícil, aumentando, assim, o risco de bugs. Existe uma relação direta entre o tamanho de um arquivo e sua complexidade de manutenção, uma vez que arquivos maiores exigem maior esforço para serem compreendidos, modificados e testados. No entanto, não há uma caracterização padronizada desse fenômeno entre diferentes projetos ou linguagens de programação. Este estudo caracteriza a presença de arquivos grandes em projetos de software desenvolvidos em diferentes linguagens de programação, com foco na análise de seu comportamento evolutivo. Inicialmente, analisamos 450 repositórios de software em 15 linguagens de programação, detalhando aspectos como tempo de vida do arquivo, alterações de código, número de commits e autores contribuintes. Arquivos grandes estão presentes em 74,88% dos projetos analisados, sendo o JavaScript a linguagem com maior incidência. A maioria desses arquivos já é criada inicialmente com um tamanho considerável (superando 60%), tende a aumentar ao longo do tempo e está mais frequentemente envolvida em modificações relacionadas a defeitos. Além disso, costumam ser editados por um número maior de contribuidores, com uma média de 3 a 4 desenvolvedores responsáveis por mais de 70% das modificações. Nosso estudo mostra que arquivos grandes são comuns e tendem a crescer ainda mais, aumentando o risco de defeitos e exigindo a colaboração de mais desenvolvedores. Uma melhor compreensão desse fenômeno pode ajudar as equipes a adotarem práticas de refatoração aprimoradas. Portanto, monitorar e gerenciar arquivos grandes é um aspecto fundamental da governança de projetos de software.

Palavras-chave

Arquivos Grandes, Cheiro de Código, Mineração de Repositórios, Código Aberto, Métodos Longos, Objeto Deus

1 Introdução

Tarefas de manutenção e evolução são ferramentas ativas no arsenal de técnicas dos desenvolvedores de software [18] para combater o "envelhecimento de software". Compreender melhor a evolução de código complexo é fundamental para detectar áreas de risco no software e orientar futuras atividades de desenvolvimento [13].

Como forma de facilitar a melhoria da qualidade de software e, consequentemente, sua manutenção e evolução, diversos trabalhos foram realizados com o objetivo de identificar e remover maus cheiros (*bad smells*) no código-fonte de projetos de software [23]. De maneira geral, os maus cheiros são sinais de problemas no código, o que sugere a necessidade de refatoração para melhorar a qualidade. Em particular, os maus cheiros relacionados à complexidade e ao tamanho do código, como *God classes* e Métodos Longos, são geralmente percebidos como uma ameaça importante pelos desenvolvedores [16]. Essa importância está relacionada ao fato de que, quando concentrados, esses maus cheiros tendem a diminuir a qualidade do desenho e da arquitetura de software, atormentando os desenvolvedores [12].

Um dos motivos pelos quais esses maus cheiros relacionados ao tamanho do código existem é a ausência de uma definição clara de um limite de crescimento para classes. Embora existam diretrizes genéricas, perguntas como "Quantos métodos uma classe pode ter?", "Quão longo deve ser um método de teste?" ou "Qual é o tamanho máximo de uma classe de negócio?" raramente são respondidas adequadamente. Como exemplo de esforço para delimitar o crescimento de unidades de código, cita-se o trabalho de Pinto [20], que envolve práticas de CDD (Cognitive-Driven Development) ainda pouco adotadas no processo de desenvolvimento. Como resultado, as unidades tendem a crescer indefinidamente. Em buscas por projetos no GitHub não é incomum encontrar classes com mais de 10 mil, ou até 100 mil, linhas de código. Não obstante esse fato, muitos estudos sobre complexidade continuam restritos a uma única linguagem, principalmente Java [13, 30].

Embora existam trabalhos que tenham investigado o impacto de maus cheiros, como *God classes*, em diversos projetos de software [16, 23, 27, 29], nenhum deles investigou o fenômeno de grandes unidades de código em projetos de software livre.

O objetivo deste trabalho é avaliar empiricamente o que se entende por arquivo grande, comparando essa definição em diferentes linguagens de programação. Subsequentemente, aprofunda-se o estudo do impacto da existência destes arquivos em projetos de software livre de diferentes linguagens, investigando a evolução destes

arquivos nos projetos em que aparecem (Já nascem grandes?) e analisando também o impacto na manutenção entre os desenvolvedores (quem modifica, quando e por que mudam).

Para realizar este estudo, foram selecionados 450 projetos de software livre em 15 linguagens de programação distintas. Ainda, como parte importante do trabalho, propõe-se uma definição do que se entende por arquivos grandes em projetos de código aberto.

As principais contribuições deste trabalho são:

1. Um conjunto de dados composto por 450 projetos de software livre, com 15 linguagens de programação mais populares e suas respectivas listas de arquivos grandes, categorizados com base em uma definição empírica de arquivos grandes (1% com maior LoC).

2. O estudo analisou a evolução dos arquivos por meio da investigação de 1.435.683 commits, dos quais 849.854 em arquivos grandes e 831.388 em arquivos regulares (245.559 commits em ambos os grupos), desde a sua criação, com detalhamento sobre os autores das modificações e os impactos dessas manutenções nos arquivos.

3. Os resultados indicam que arquivos grandes nascem grandes (60.0%) e, ao longo do tempo, continuam aumentando de tamanho de maneira recorrente e consistente, mesmo desconsiderando aqueles de tamanho estacionário.

4. Arquivos grandes concentram um volume maior de alterações corretivas em comparação à adição de novas funcionalidades; nesses arquivos, o índice de commits para correção de defeitos é de 30,71% e para novas funcionalidades é de 11,2%, além de receberem uma baixa taxa de refatoração (0,4%).

5. Arquivos grandes exigem uma "colaboração forçada" de 3 a 4 autores em média e até de 15 em casos extremos.

2 Trabalhos Relacionados

O foco deste estudo é conceitualizar arquivos grandes em projetos de software livre e analisar suas implicações. Não foram encontrados artigos que tratem diretamente do fenômeno de arquivos grandes, mas sim como parte de outros fenômenos da engenharia de software, como os maus cheiros de código. Fowler [8] define maus cheiros como problemas estruturais expressos em antipadrões de programação. Bavota et al. [2] identificaram o número de linhas como principal fator na detecção de *Long Method*, *Lazy Class*, *Blob Class* e *Spaghetti Code*, verificando que classes longas têm 71% mais chance de serem refatoradas. Pereira dos Reis et al. [19] constataram que 83,1% dos estudos sobre maus cheiros ocorreram em projetos open source, com Java presente em 77,1% dos trabalhos, sendo *God Class* (51,8%), *Feature Envy* (33,7%) e *Long Method* (26,5%) os mais comuns, dois deles diretamente relacionados ao tamanho do arquivo.

Palomba et al. [17] destacam que *Long Methods* são detectados apenas pela contagem de linhas e argumentam que maus cheiros identificados textualmente são tão prejudiciais quanto os estruturais, mesmo sem ultrapassar métricas típicas recomendadas.

O tamanho de arquivos, classes ou métodos também é investigado como fator de aumento na probabilidade de defeitos. A métrica *Lines of Code (LoC)* [10] é um dos indicadores mais usados há mais de 20 anos em predição de defeitos. Saboury et al. [22] mostraram que arquivos JavaScript sem maus cheiros relacionados a tamanho apresentam taxas de risco 65% mais baixas do que aqueles com tais maus cheiros.

Por fim, os estudos sobre maus cheiros e modelos de predição utilizam métricas de tamanho de métodos, classes e arquivos, mas sem investigar o impacto específico de arquivos grandes nos projetos analisados.

3 Metodologia

Nesta seção é proposta uma definição própria de arquivo grande. Além dessa definição importante, são descritos os passos adotados para a coleta de dados, são pontuadas as questões de pesquisa e, por fim, é apresentada a análise detalhada.

3.1 Definição De Um Arquivo Grande Em Projetos De Software Livre E Coleta De Dados

É comum encontrar discussões entre desenvolvedores profissionais sobre malefícios e impactos de arquivos grandes em projetos de software. Por exemplo, foram apresentadas alegações de que um arquivo de 700+ linhas de código em C# é considerado grande. Neste artigo, Clare Sudbery [25] discute estratégias de refatoração para melhorar a manutenibilidade de código. Em fóruns de discussão, encontram-se discussões em que um arquivo de 1000+ linhas[1] é um arquivo grande, embora reconheçam que essa definição é arbitrária, o que é suficiente para desenvolvedores pensarem sobre o projeto de software e a modularização. Ainda mais, levanta uma clara discussão sobre os impactos de arquivos grandes existirem e permanecerem, sem refatoração, por muito tempo nos projetos.

Não foi encontrada na literatura uma definição adequada ou consensual para arquivos grandes. A definição mais próxima é a de "Classes Inchadas" (do inglês, "*Bloaters*"), que são conhecidas por serem um mau cheiro de código [8, 14]. Esse mau cheiro de código tem sido investigado empiricamente, como por exemplo no trabalho de Zhang et al. [31], que verificaram seu impacto na prática, e Fontana et al. [7], que investigaram tipos de mal cheiros de código relacionados ao tamanho de classes e métodos, tais como, "*God Class*", "*Large Class*", "*Long Method*" e "*Long Parameter List*". Geralmente, estes maus cheiros relacionados ao tamanho do arquivo são discutidos em estudos de programas escritos em linguagem Java [5]. Explora-se aqui um ponto não abordado por tais trabalhos: como o contexto (projeto, desenvolvedores e linguagem de programação) influencia a evolução e a manutenção dos grandes arquivos.

O primeiro passo para a realização do estudo foi definir o que seria um arquivo grande, conforme se observa na Figura 1. Assim, diante da dificuldade de encontrar uma definição de consenso, foi elaborada uma definição empírica, baseada na observação do tamanho dos arquivos minerados em repositórios de projetos de software livre, para propor uma definição própria de arquivo grande. Ainda mais, a hipótese está relacionada à influência do contexto em que ele é encontrado. Assim, para definir o que é um arquivo grande, foram selecionadas 15 linguagens de programação diferentes, identificadas a partir do ranking do GitHut 2.0 e do índice TIOBE, que mapeia as linguagens de programação mais populares do mundo, com base na quantidade de estrelas dos repositórios [2].

Após esse passo, foram identificados manualmente os 30 repositórios mais populares no GitHub para cada linguagem selecionada,

[1] https://medium.com/@olegalexander/how-to-write-large-programs-628c90a70615
[2] https://madnight.github.io/githut/#/stars/2024/1

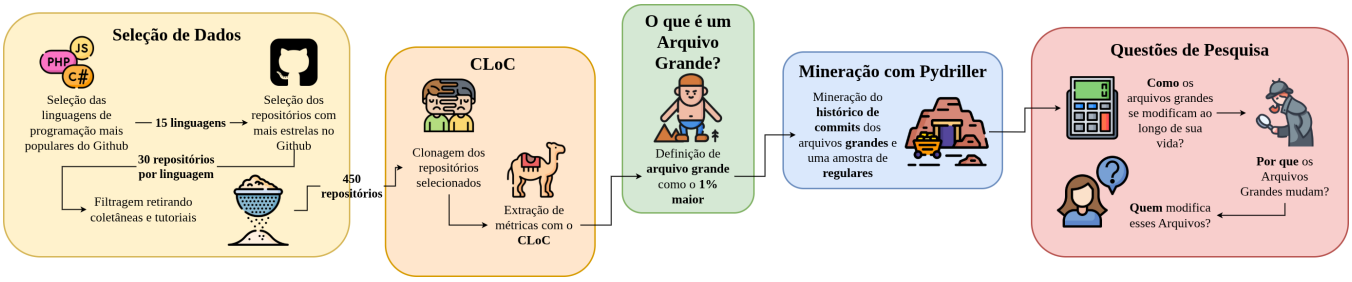

Figura 1: Fluxograma da Metodologia Aplicada

totalizando 450 repositórios. Esta abordagem é usada em estudos que envolvem a mineração de repositórios [3] e auxilia na identificação de projetos ativos, que atraem contribuidores e despertam grande interesse da comunidade de software livre. É importante mencionar que os repositórios selecionados eram projetos de software, excluindo aqueles que se tratavam de livros, tutoriais (como o "FreeCodeCamp"em JavaScript) ou repositórios de listas de tópicos interessantes, como, por exemplo, os "awesome projects".

O próximo passo foi clonar cada um dos 450 repositórios, totalizando assim um espaço amostral de aproximadamente 1,83 milhão de arquivos, 21,73 milhões de linhas totais de código e 273 extensões de arquivos diferentes. Como o objetivo inicial era definir um arquivo grande nas 15 linguagens-alvo selecionadas, foi excluído desta amostragem qualquer arquivo que não pertencia à lista de linguagens selecionadas, por exemplo, arquivos com extensão de linguagens de marcação (HTML, CSS, SCSS), shell scripts, arquivos de texto, scripts de banco de dados ou Markdown.

A Tabela 1 apresenta o total de arquivos com extensão de uma determinada linguagem de programação, e os percentis listam a quantidade mínima de linhas de código sem comentários e de linhas em branco (extraídas com a ferramenta Cloc[3]) para que um arquivo esteja no percentil correspondente. Observa-se que há uma grande variância entre as linguagens de programação. Enquanto na linguagem C, no seu percentil 99% os arquivos devem ter no mínimo 4.197 linhas, nas linguagens Kotlin, Swift e Ruby os arquivos nesse percentil têm menos de 700 linhas de código. Em comparação com a linguagem C, esses arquivos grandes são quase 6 vezes menores. Outros percentis também são apresentados: 95%, 75%, 50% e 25%.

Note que, para calcular os valores apresentados na tabela, um repositório foi classificado na lista de TOP-30 repositórios populares, considerando apenas a linguagem de maior incidência nele. Por exemplo, o projeto "PHP-SRC"[4] tem 68% de código em C, 29% em PHP, menos de 3% em Lua, C++, M4 e Shell. Nesse caso, o projeto "PHP-SCR" contou como um projeto da linguagem C para fins de seleção e amostragem, mas todos os arquivos de quaisquer extensões de interesse, como foi o caso dos arquivos das linguagens C, C++ e PHP, foram considerados para análise do percentil de cada linguagem correspondente.

Após o uso do CLoC, o limiar de corte que foi definido para estabelecer os arquivos grandes é o menor arquivo de cada linguagem de programação no percentil 99, considerando a última aparição do arquivo até a data da mineração dos commits (31 de Agosto de 2025). Dessa forma, **propõe-se, para este estudo, uma definição de arquivo grande como aqueles arquivos 1% maiores de cada linguagem**. Por exemplo, em JavaScript, um arquivo é considerado grande se tiver pelo menos 959 linhas de código. Os arquivos não grandes, ou seja, aqueles que não estão no conjunto de 1% maiores arquivos, são chamados de arquivos regulares.

Tabela 1: Percentil de linhas de código por linguagem.

		Percentil de LOC				
Linguagem	**# Arq.**	**25%**	**50%**	**75%**	**95%**	**99%**
C	56.781	85	219	538	1.818	4.197
C#	94.157	20	47	120	520	1.760
C++	43.316	48	118	299	1.230	3.504
Dart	39.389	14	42	116	587	1.830
Go	60.070	25	70	188	736	2.014
Java	102.655	21	50	117	421	1.050
JavaScript	157.710	6	12	37	249	959
Kotlin	74.911	8	17	44	186	491
Objective-C	14.542	11	11	26	316	1.070
PHP	57.004	17	32	71	299	977
Python	55.753	22	65	164	664	1.657
Ruby	85.296	16	33	73	249	662
Rust	48.601	9	23	98	526	1.403
Swift	27.544	5	16	54	256	672
TypeScript	89.805	8	23	75	334	913

Apesar da escolha dos 1% maiores arquivos ser uma decisão arbitrária, optou-se por esse valor como um *baseline* para trabalhos futuros que visem estudar a variação do número de linhas na determinação de arquivos grandes e seus efeitos na manutenibilidade. O uso do percentil 99 garante que o estudo se concentre exclusivamente nos casos mais extremos de cada linguagem. Isso isola arquivos atípicos que têm maior probabilidade de violar princípios de design como o de Responsabilidade Única (SRP, *Single Responsability Principle*).

3.2 Mineração De Commits De Arquivos Grandes E Regulares Para Realização De Análises

Com o conceito de arquivo grande definido, cada arquivo dos 450 repositórios foi categorizado como arquivo grande ou regular. Isso foi feito por meio da clonagem dos repositórios no tempo presente. Em seguida, foi realizada a mineração dos commits que continham arquivos categorizados como grandes, utilizando a biblioteca PyDriller [24], implementada em *threads*, para nos beneficiarmos do paralelismo na extração. A mineração foi feita de 31 de Agosto de

[3]https://github.com/AlDanial/cloc
[4]https://github.com/php/php-src

2025 até a data de criação dos repositórios, de maneira que se obteve, então, todo o histórico de variação de tamanhos dos arquivos grandes anteriormente detectados com a ferramenta CLoC.

Paralelamente, com a intenção de conduzir estudos comparativos, também foi minerada uma amostra de arquivos regulares que, assim como para os arquivos grandes, foi feita por meio da clonagem no tempo presente e da subsequente mineração até a criação do repositório. Para isso, foi calculado o tamanho ideal da amostragem para cada linguagem de programação com uma calculadora amostral [5]. Foi utilizada uma distribuição amostral mais homogênea (80/20), margem de erro de 1% e nível de confiança de 99% para aumentar a representação dos arquivos regulares. Após definir o tamanho da amostra, houve a identificação da porcentagem de arquivos de cada linguagem presente em cada projeto. Quando um projeto não possuía arquivos suficientes para compor a amostragem desejada, foram utilizados todos os arquivos disponíveis da respectiva linguagem, sem comprometer a representatividade da amostra.

Por fim, foram sorteados aleatoriamente arquivos dentro de cada projeto para compor a amostra, e foi realizada a mineração dos commits desses arquivos regulares. Com os commits minerados, uma série de análises foi conduzida com o objetivo de comparar as características dos arquivos grandes com as dos arquivos regulares para responder as questões de pesquisa abaixo descritas.

3.3 Questões Da Pesquisa

Com o propósito de caracterizar a existência e a evolução de arquivos grandes em projetos de software livre, quatro questões de pesquisa são investigadas. Observa-se que a primeira questão de pesquisa é exploratória e leva em consideração todos os 450 projetos coletados, já as questões de pesquisa 2, 3 e 4 levam em consideração somente 448 projetos (Linux[6] e Pytorch[7] são os faltantes). Essa decisão foi tomada em decorrência de uma limitação em relação à mineração e ao processamento local dos dados, no entanto, tal implicação será tratada na Seção 7.

QP1. Qual a distribuição de arquivos grandes nos projetos de software livre analisados?

Entender a frequência de arquivos grandes nos projetos é importante para caracterizar como o fenômeno ocorre na prática. Esta questão visa explorar a ocorrência do fenômeno, quantificando e investigando se há uma concentração de arquivos grandes em poucos projetos ou se há uma relação de espalhamento da ocorrência entre projetos. Também, para investigar se projetos majoritariamente escritos em uma linguagem de programação concentram arquivos grandes dessa linguagem ou se também podem conter arquivos grandes de outras linguagens.

Embora a existência de arquivos grandes não seja fundamentalmente prejudicial, há discussão entre desenvolvedores de que isso pode indicar que o arquivo/classe/método está assumindo muitas responsabilidades, violando boas práticas como o princípio da responsabilidade única (*SRP*)[8].

[5]https://comentto.com/calculadora-amostral/
[6]https://github.com/torvalds/linux
[7]https://github.com/pytorch/pytorch
[8]https://dev.to/cecilelebleu/in-programming-is-it-better-to-have-many-small-files-or-one-large-file-1oom

Para esta questão, adotou-se estatística descritiva. Além de descrever um fenômeno ainda não quantificado na literatura, essa questão contribuirá para a construção de um conjunto de dados útil a pesquisas futuras.

QP2. Como é a evolução dos arquivos grandes quando comparados com arquivos regulares?

Nessa questão de pesquisa, identifica-se em que momento os arquivos grandes são introduzidos em um projeto de software livre, com o objetivo de descobrir se são criados com muitas linhas de código ou se novas linhas são adicionadas ao longo do tempo, tornando-os ainda maiores. Para isso, foram investigadas a evolução em termos de linhas adicionadas e removidas, a quantidade de commits associados a esses arquivos e a distância temporal entre esses commits. Para responder esta questão de pesquisa, foram coletadas informações sobre: o hash do commit, nome do projeto, caminho do arquivo, números de arquivos por commit, linhas adicionadas e removidas em cada arquivo no commit, mensagem do commit, tipo da mudança, se era um merge commit, horário em que o commit foi realizado, além de dados do autor e do commiter, como por exemplo, o nome e email.

Para as questões de pesquisa 2, 3 e 4, foram usados os 9.944 arquivos grandes identificados nos 448 projetos considerados (excluindo Linux e Pytorch). Para fins de comparação, utilizou-se ainda uma amostra de 128.089 arquivos regulares dos mesmos projetos, selecionados aleatoriamente segundo a estratégia mencionada anteriormente.

QP3. Quais as razões das modificações em arquivos grandes?

Nesta questão de pesquisa, o objetivo principal foi entender como os arquivos grandes podem afetar o processo de evolução e manutenção do projeto de maneira geral, avaliando se as mudanças em arquivos grandes são diferentes das mudanças em arquivos regulares. Também são exploradas quais mudanças são mais e menos comuns, identificando quais têm maior probabilidade de acontecer em arquivos grandes de maneira a alertar os desenvolvedores sobre os impactos de um arquivo grande no projeto.

A primeira análise focou em identificar as razões implícitas em que um arquivo grande é modificado a partir do método utilizado por Kruger et al. [11], que categorizou as motivações, evidências e técnicas relacionadas às intenções de mudança de software (SCIs). Para isso, foi realizada uma análise automática de 1.435.683 mensagens de commit, procurando por palavras-chave que descrevessem o objetivo da mudança. A Tabela 2 apresenta a intenção de mudança descrita no commit, a descrição e a heurística replicada do artigo para identificar as intenções de mudança de software (SCIs).

É importante notar que um commit pode ser classificado disjuntamente em duas ou mais categorias, pois ele pode conter mais de uma palavra-chave de tipos de alteração distintos. Isso pode acontecer quando alterações de código não relacionadas são realizadas juntas, como, por exemplo, uma correção de defeitos e uma refatoração, criando os chamados commits emaranhados (*tangled commits*) [6]. Por isso, foi decidido seguir a ordem de classificação da tabela: a primeira classificação encontrada será a do commit, a menos que seja um commit feito por um bot, o que leva à classificação "Automatizado".

Tabela 2: SCI: Categorias e Heurísticas

SCI e Descrição	Heurísticas
Operações de GIT	Contém: *"branch", "merge", "integrate", "revert"*
Compilação do software	Contém: *"build"*
Correção de Bugs	1. Contém: *"bug", "fix"* **E** 2. Não contém: *"test case", "unit test"*
Alteração em Recursos	1. Contém: *"conf", "license", "legal"* **OU** 2. Arquivos não incluem código/teste
Alt. em Funcionalidade	1. Contém: *"update", "add", "new", "create", "implemented feature"* **OU** 2. Categoria *"Outro"* com: *"enable", "add", "update", "implement", "improve"*
Teste	1. Contém: *"test"* **OU** 2. Arquivos são exclusivos de teste
Refatoração	Contém: *"refactor"*
Rem. Descontinuidades	Contém: *"deprecat", "delete", "clean up"*
Automatizado	Submetido por bots/contas automatizadas

Após realizar a classificação automática dos commits, tanto para arquivos grandes quanto para regulares, foi avaliado se essas duas variáveis (arquivo grande? [sim|não] e tipo de mudança apresentado na Tabela 2) são relacionadas ou independentes por meio de comparações quantitativas da incidência das classificações.

QP4. Quem são os principais responsáveis por estas alterações?

Foi avaliada a autoria da criação e manutenção de arquivos grandes. Com base na ideia e no cálculo do *Truck Factor* [1, 9], foi obtido o número de autores responsáveis por pelo menos 70% dos commits de um arquivo. A intenção foi avaliar se um arquivo grande era mantido por um subconjunto pequeno de desenvolvedores ou por um grupo grande, em comparação com os regulares.

4 Resultados

Esta seção apresenta os resultados e análises das quatro questões de pesquisa investigadas neste trabalho.

4.1 QP1. Qual A Distribuição De Arquivos Grandes Nos Projetos De Software Livre?

Para responder à primeira questão de pesquisa, realizou-se a inspeção da proporção de arquivos grandes nos 450 projetos analisados, denominada hipótese do espalhamento. A conjectura é de que há arquivos grandes espalhados entre os projetos. Foram encontrados 337 projetos com arquivos grandes, distribuídos aleatoriamente, nas 15 linguagens de programação analisadas, o que corresponde a 74.88% do total de projetos coletados. O espalhamento desses arquivos entre os 337 projetos pode ser observado na Figura 2. Existe uma concentração de projetos que possuem entre 1 e 100 arquivos grandes e, após esse limite, existe pelo menos um projeto até o limite de 590 arquivos grandes.

A Tabela 3 apresenta a lista de linguagens de programação investigadas, o nome do projeto que apresenta o maior número de arquivos grandes dessa linguagem e a linguagem mais predominante no repositório/projeto. Alguns projetos são implementados em mais de uma linguagem de programação, e denomina-se linguagem predominante a que possui a maior parte das linhas de

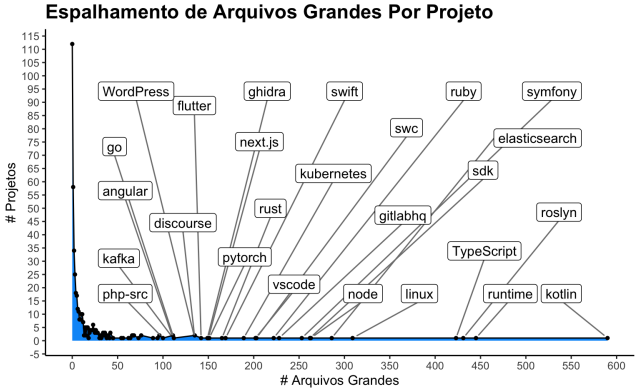

Figura 2: Espalhamento de arquivos grandes por projeto.

código do projeto. Na maioria dos casos, os arquivos grandes estavam implementados na linguagem predominante. No entanto, um ponto interessante a ser observado é que, em algumas linguagens, os arquivos grandes foram implementados em linguagens não predominantes no projeto.

Observou-se que o projeto com a maior quantidade de arquivos grandes é o "Kotlin"[9], com 590 arquivos: 415 grandes em Kotlin, 172 em Java e 3 em Swift. Este projeto apresentou um percentual de arquivos grandes de 5,67% em relação ao total de arquivos grandes (590/10.404). Entretanto, a proporção de arquivos grandes no total de arquivos do projeto é de 0,61% (590/89.993). Dessa forma, o projeto que relativamente contém a maior quantidade de arquivos grandes é o "lodash"[10], com 13,51% dos arquivos sendo grandes (10/74). Além disso, uma análise do percentil 99 de cada linguagem de programação, por projeto, resultou na Tabela 3.

Tabela 3: Repositórios com mais arquivos grandes.

Linguagem	Repositório	# Arq.	Total	(%)
C	linux	309	874	35,35
C#	roslyn	431	943	45,71
C++	tensorflow	44	434	10,14
Dart	sdk	231	394	58,63
Go	kubernetes	189	601	31,45
Java	elasticsearch	286	1027	27,85
JavaScript	node	139	1579	8,80
Kotlin	kotlin	415	755	54,97
Objective-C	gnachman	49	146	33,56
PHP	symfony	253	572	44,23
Python	pytorch	132	562	23,49
Ruby	ruby	200	855	23,39
Rust	rust	162	487	33,26
Swift	realm-swift	31	276	11,23
TypeScript	vscode	188	899	20,91

Entretanto, quando considerada apenas a linguagem majoritária do projeto para a comparação, o projeto "roslyn"[11] é o com maior quantidade absoluta de arquivos grandes, totalizando 431 dos 934 arquivos grandes em C#, o que corresponde a 45.71% dos arquivos

[9] https://github.com/JetBrains/kotlin
[10] https://github.com/lodash/lodash
[11] https://github.com/dotnet/roslyn

da linguagem, e o "sdk"[12] passa a ser o projeto com mais arquivos grandes relativamente à sua linguagem.

Resposta para QP1: A análise da hipótese do espalhamento mostrou que 74,88% dos 450 projetos contêm ao menos um arquivo grande, com a maior parte concentrando entre 1 e 100 desses arquivos, enquanto poucos chegam a centenas (até 590 no projeto Kotlin). Considerando apenas os repositórios que contêm ao menos um arquivo grande (~75% dos repositórios), os arquivos grandes representam, em média, ~14% dos arquivos dentro desses repositórios e na maioria dos casos, estão implementados na linguagem predominante do projeto.

4.2 QP2. Como É A Evolução Dos Arquivos Quando Comparados Com Arquivos Regulares?

Para responder a esta questão de pesquisa, foi analisado o número de linhas adicionadas e removidas dos arquivos grandes. Primeiramente, constatou-se que os 9.944 arquivos grandes foram modificados em 849.854 commits.

Tabela 4: Alterações e intervalos em arquivos por linguagem

	Média de Alterações		Intervalo Médio (dias)	
Linguagem	*Grandes*	*Regulares*	*Grandes*	*Regulares*
C	21,5	11,4	255	275
C#	8,4	4,5	224	211
C++	16,4	8,5	173	174
Dart	14,0	7,8	230	265
Go	11,4	7,2	262	268
Java	6,2	5,7	330	259
JavaScript	8,8	5,6	295	227
Kotlin	6,8	4,4	389	252
Objective-C	14,6	8,3	379	293
PHP	10,4	9,2	256	230
Python	19,4	7,1	224	224
Ruby	10,1	6,4	281	380
Rust	11,8	7,9	133	73
Swift	10,5	6,6	180	151
TypeScript	8,2	5,6	315	243

A Tabela 4 mostra que arquivos grandes são alterados, em média, cerca de 1,67 vezes (mediana 1,57) mais do que arquivos regulares, considerando todas as linguagens. O destaque fica para a linguagem Python, cujos arquivos grandes são alterados cerca de 2,75 vezes mais do que arquivos regulares. Linguagens como C, C# e C++ têm seus arquivos grandes alterados quase duas vezes mais do que os regulares (1,88, 1,88, 1,93, em média, respectivamente). Entretanto, cabe ressaltar que o número de alterações não afeta o tempo médio entre elas. Das 15 linguagens estudadas, 7 ficam com uma diferença de menos de 30 dias nas médias dos intervalos de modificações em arquivos grandes e regulares, 2 com até 60 dias de diferença, e o restante com um intervalo superior a 60 dias. A maior diferença é observada na linguagem Kotlin, onde, em média, arquivos grandes são alterados a cada 389 dias e arquivos regulares a cada 252 dias. As menores diferenças estão em C++ e Python, que mantêm a mesma média em ambos os tipos de arquivos.

[12]https://github.com/dart-lang/sdk

A Tabela 5 apresenta o cenário de aumento de tamanho, em linhas de código, dos arquivos nos projetos, agrupados por linguagem. Considerando as métricas adotadas e a definição de arquivos grandes proposta, a coluna *LOC/Arq.* apresenta a mediana de linhas de código nos arquivos grandes em uma dada linguagem. A coluna *+ LOC* apresenta a mediana de linhas aumentadas em arquivos grandes em uma determinada linguagem, enquanto a coluna *- LOC* apresenta a mediana de linhas reduzidas nesses mesmos arquivos. As colunas *%* apresentam os percentuais relativos.

Tabela 5: Variação de tamanho em arquivos grandes.

Linguagem	LOC/Arq.	+ LOC	(%)	- LOC	(%)
C	5737	509	8,9	−26	−0,5
C#	2939	359	12,2	−84	−2,8
C++	5269	84	1,6	−4	−0,1
Dart	3052	40	1,3	−23	−0,8
Go	3200	92	2,9	−25	−0,8
Java	1536	38	2,5	−7	−0,5
JavaScript	1725	340	19,7	−61	−3,5
Kotlin	697	251	36,0	−57	−8,2
Objective-C	1881	418	22,2	−36	−1,9
PHP	1659	474	28,6	−22	−1,3
Python	2290	1442	63,0	−181	−7,9
Ruby	981	409	41,6	−8	−0,8
Rust	1973	362	18,3	−46	−2,3
Swift	986	423	42,9	−49	−5,0
TypeScript	1346	356	26,4	−1	−0,1

Nota-se que um arquivo grande pode crescer ao longo da vida de um projeto. Em especial, algumas linguagens se destacam pelo elevado percentual de aumento, sendo as cinco maiores Python (63%), Swift (42,9%), Ruby (41,6%), Kotlin (36%) e PHP (28,6%). No outro extremo, há linguagens com menor número de LOCs inseridas ao longo da vida de um arquivo. Com menos de 5% de aumento, os repositórios analisados apresentam Dart (1,3%), C++ (1,6%), Java (2,5%) e Go (2,9%). Ao considerar todas as linguagens, os arquivos grandes apresentam um aumento de 19% no número de LOC ao longo do histórico de mudanças analisado. Ainda há uma redução de linhas em arquivos grandes, que representa um percentual bem menor em comparação ao aumento. Destacam-se as três linguagens com maior remoção de linhas, que também figuram no Top-5 de aumento: Kotlin (-8,2%), Python (-7,9%) e Swift (-5%).

A Tabela 6 apresenta os mesmos dados dos arquivos considerados regulares neste estudo. Observa-se que, para essa tabela, a mediana de LOC dos arquivos da amostra foi analisada, e representa uma amostra do total de arquivos regulares dos projetos, conforme descrito na seção de metodologia.

Nota-se que, proporcionalmente, as reduções de LOC dos arquivos regulares são maiores do que em arquivos grandes. Assim, há um crescimento lento de arquivos regulares, e em alguns casos quase uma estabilidade. Considerando os valores medianos de aumento de LOC apresentados nas Tabelas 5 e 6, os arquivos regulares apresentam um aumento de aproximadamente 5% no número de LOC em comparação aos arquivos grandes.

Observa-se que 26,5% dos arquivos grandes tendem a aumentar de tamanho, 10,6% tendem a diminuir, 57,2% tendem a se manter com a mesma quantidade de linhas desde o nascimento. Comparativamente, os arquivos regulares apresentaram 27,6% de tendência

Tabela 6: Variação de tamanho em arquivos regulares.

Linguagem	LOC/Arq.	+ LOC	(%)	- LOC	(%)
C	220	28	12,7	−8	−3,6
C#	54	14	25,9	−5	−9,3
C++	158	23	14,6	−7	−4,4
Dart	44	26	59,1	−10	−22,7
Go	67	29	43,3	−11	−16,4
Java	53	13	24,5	−5	−9,4
JavaScript	24	21	87,5	−7	−29,2
Kotlin	23	9	39,1	−4	−17,4
Objective-C	51	16	31,4	−7	−12,7
PHP	35	12	34,3	−4	−11,4
Python	86	35	40,7	−8	−9,3
Ruby	35	17	48,6	−5	−14,3
Rust	62	32	51,6	−10	−16,1
Swift	22	17	77,3	−5	−22,7
TypeScript	41	9	22,0	−2	−4,9

de aumento, 11% de tendência de diminuição, 53,1% de manutenção do número de linhas.

Ainda na caracterização das mudanças, os arquivos grandes eram alterados em conjunto com outros arquivos ou modificados isoladamente. Observa-se que, em 600.834 commits, arquivos grandes eram alterados junto com outros arquivos, o que equivale a 70,7% do total, com uma média de aproximadamente 12 arquivos alterados por vez e uma mediana de 2. Em relação ao total, 25,4% dos commits envolvem alterações em ao menos 2 arquivos grandes por vez. Já para os arquivos regulares, um total de 181.960 commits alteravam mais de um arquivo de uma só vez, equivalente a 21,9% do total, carregando consigo média de 31 arquivos alterados em um único commit, dos quais 40,9% do total representam alterações em commits que também envolviam um arquivo grande.

Essas modificações podem alterar a classificação dos arquivos de acordo com as métricas deste trabalho. Ao questionar se um arquivo grande já nasce (é criado) grande ou se torna grande, primeiro houve a separação dos commits dos arquivos grandes previamente detectados pelo cloc, conforme a definição anteriormente descrita. A Tabela 7 apresenta os resultados dessa análise.

Tabela 7: Classificação temporal de arquivos grandes.

Linguagem	Criação		Transformação	
	#	(%)	#	(%)
C	447	79,1	118	20,9
C#	658	69,8	285	30,2
C++	278	64,1	156	35,9
Dart	213	54,1	181	45,9
Go	390	64,9	211	35,1
Java	569	55,4	458	44,6
JavaScript	950	60,2	629	39,8
Kotlin	531	70,3	224	29,7
Objective-C	88	60,3	58	39,7
PHP	399	69,8	173	30,2
Python	122	29,7	289	70,3
Ruby	330	38,6	525	61,4
Rust	291	59,7	196	40,3
Swift	104	37,7	172	62,3
TypeScript	509	56,6	390	43,4

Observa-se que a maioria dos arquivos grandes já nasce grande, conforme mostra a coluna *Criação*. Em média, considerando todas as linguagens analisadas, cerca de 60% dos arquivos já são criados grandes, índice que chega a quase 80% em C. Em contraste, Ruby (38,6%), Swift (37,7%) e Python (29,7%) apresentam os menores percentuais, sendo justamente as linguagens com maior índice de *Transformação*, isto é, de arquivos que se tornam grandes ao longo do tempo por meio da adição de linhas.

Propôs-se ainda uma terceira categoria, a de arquivos *Flexíveis*, atualmente grandes, mas que, ao longo do histórico de commits, oscilaram de categoria, por remoção e/ou posterior adição de linhas. Esses arquivos correspondem a 98,92% dos casos (9837 arquivos), o que pode indicar refatoração (do projeto ou do próprio código) ou alteração de responsabilidades e funções.

A classificação "flexíveis", portanto, representa arquivos grandes que oscilaram de tamanho o suficiente para sair e retornar ao percentil de 1% com mais linhas de código.

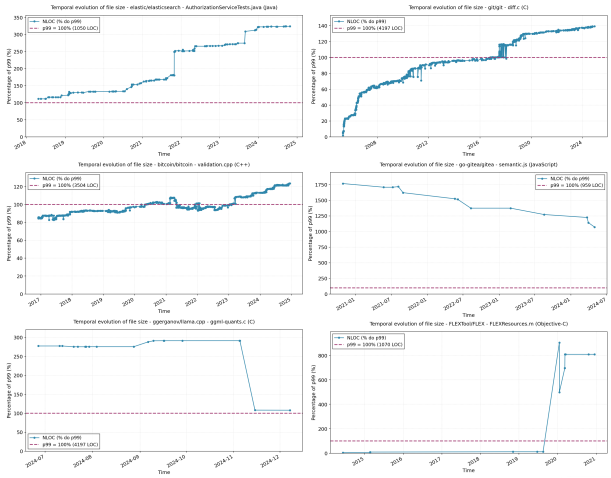

Figura 3: Padrões de crescimento em arquivos grandes.

A Figura 3 exibe casos de arquivos originalmente regulares que se tornam grandes gradativamente ao longo da vida. Esses casos relacionam-se aos commits de correção de bugs e alterações de recursos identificados na questão anterior: arquivos de teste/validação que apenas acumulam responsabilidades e não param de crescer, e arquivos com funcionalidades centrais que ficam mais complexos com o tempo sem serem devidamente refatorados.

A análise identificou ainda os arquivos flexíveis, que alternam sua classificação de tamanho em diversos momentos da existência, como ilustrado na Figura 3. Essa volatilidade, em que um arquivo grande diminui e, após um período, volta a crescer, associa-se à alta instabilidade decorrente da natureza ativa do projeto e de refatorações constantes, sendo comum em arquivos de teste e validação frequentemente refatorados.

Certos arquivos grandes diminuem por meio de refatorações sistemáticas que combatem "maus cheiros" como "Métodos Longos" e "Objeto Deus", que acumulam responsabilidades e funcionalidades obsoletas. A aplicação do SRP redistribui tarefas e elimina código morto, gerando componentes mais enxutos. Esse padrão ocorreu no arquivo semantic.js, criado para centralizar configurações, mas cujo tamanho foi reduzido com a remoção de módulos obsoletos.

Identificaram-se também quedas repentinas de linhas, com redução exponencial de tamanho (Figura 3), padrão restrito a parcela

reduzida da amostra e geralmente associado a refatoração. No arquivo ggml-quants.c, por exemplo, uma refatoração transformou o backend do projeto em estrutura de bibliotecas, esvaziando o arquivo, diluindo suas responsabilidades e facilitando sua manutenção, "matando o Gigante".

O cenário oposto também foi detectado: arquivos com ganho abrupto de linhas, caracterizando crescimento exponencial (Figura 3). Esse comportamento, muitas vezes superando o P99 da amostra, pode decorrer de novas funcionalidades ou de alterações arquiteturais. No caso analisado, o crescimento consolidou-se com a população de uma grande estrutura de dados, algo que naturalmente implica muitas linhas de código, mas que, na maioria das vezes, não representa risco por si só.

Resposta para QP2: Cerca de 60% dos 9.944 arquivos grandes analisados já nascem grandes. Comparativamente, observa-se uma tendência de que arquivos grandes são duas vezes mais alterados do que os regulares ao longo do histórico de mudanças, ainda que em intervalos de tempo mais longos. Esse cenário sugere que tais arquivos atuam como pontos de acúmulo de dívida técnica, em que a alta frequência de alterações em código complexo perpetua a instabilidade do sistema.

4.3 QP3. Quais As Razões Das Modificações Em Arquivos Grandes?

Para responder à terceira questão de pesquisa, foram analisados os tipos de mudança, com o objetivo de entender se a maioria das adições/remoções de linhas nos commits era relativa a atividades de correção de defeito (manutenção corretiva), refatoração, remoção de código descontinuado ou adição de alguma nova funcionalidade, recurso ou teste.

A Tabela 8 apresenta o percentual de mudanças entre os arquivos grandes e regulares. De maneira geral, o tipo mais comum de mudança é a correção de defeitos. Arquivos grandes têm, proporcionalmente, menos mudanças relativas à refatoração e a novos recursos.

Observou-se que arquivos grandes sofrem mais commits ao longo do tempo (como visto na QP2), e o maior percentual deles foi relativo a mudanças que corrigiram defeitos, em comparação aos arquivos regulares. Outra distinção relevante reside na categoria de Alteração em Funcionalidade: arquivos regulares apresentam maior frequência desse tipo de mudança (14,23%) em comparação aos grandes (11,20%). Em contrapartida, as demais categorias analisadas (como alterações em recursos e configurações) apresentam variações pouco significativas entre os dois grupos.

A comparação realizada permitiu identificar as principais disparidades entre as categorias, evidenciando que os motivos para modificações divergem conforme o tamanho do arquivo. Contudo, é fundamental avaliar a evolução dos arquivos grandes em uma investigação mais aprofundada das causas dessas transformações.

Ao analisar os arquivos grandes, constatou-se que 2904 (30%) contêm "test" ou "validation" em seus nomes. Este dado contribui para a incidência de testes e correções de bugs na Tabela 8. Arquivos de teste crescem ao acumular mais responsabilidades e responsabilidades mais complexas, porém, se pouco refatorados, criam testes complexos o suficiente para precisarem ser testados.

Tabela 8: Proporção dos commits em arquivos grandes.

	Proporção (%)	
Classificação	**Arq. Grandes**	**Arq. Regulares**
Operação de Commit	4,13	4,75
Configuração de Build	2,76	3,10
Correção de Bug	30,71	26,43
Alteração em Recurso	28,90	28,35
Alteração em Funcionalidade	11,20	14,23
Refatoração	0,40	0,81
Remoção de descontinuidade	0,59	0,64
Teste	8,69	9,60
Automatizados	2,32	2,27
Outras	10,30	9,82

Conforme apresentado na Questão de Pesquisa 2, constatou-se que arquivos grandes geralmente nascem grandes, mantendo uma tendência de crescimento contínuo. Este fenômeno é ilustrado na Figura 3 e atribui-se ao fato de serem arquivos constantemente alterados; nesse caso, é um teste que é sempre incrementado conforme o projeto cresce, porém não sofre refatoração e, consequentemente, segue aumentando de tamanho. Embora seja esperado que os arquivos de teste aumentem à medida que novas funcionalidades sejam adicionadas, a ausência de refatoração pode levar ao acúmulo de responsabilidades e de validações heterogêneas em um único artefato. Esse crescimento desestruturado pode estar associado à ocorrência de *Test Smells* [26, 28], reconhecidos na literatura como indícios de degradação estrutural em código de teste e potencialmente correlacionados ao aumento excessivo de seu tamanho.

Suspeita-se que arquivos grandes atuam como depósitos de práticas inadequadas e débitos técnicos, nos quais o acúmulo de responsabilidades e de código inútil impulsiona o crescimento desses arquivos. Esses "Gigantes" concentram maus cheiros, elevando a incidência de defeitos e dificultando a manutenção, o que os torna cada vez maiores. Entretanto, com a prática de refatorações é possível "tratar" esses arquivos e devolvê-los à categoria regular ao redistribuir responsabilidades e eliminar código morto.

Resposta para QP3: Arquivos grandes concentram mais correções de bugs (30,71%) e menos alterações de funcionalidade (11,2%) e metade das refatorações (0,4%) do que arquivos regulares (26,43%, 14,23% e 0,81%, respectivamente).

4.4 QP4. Quem São Os Principais Responsáveis Por Estas Alterações?

Levando em conta a quantidade de desenvolvedores que modificaram o arquivo, procura-se saber se essas atividades eram concentradas em poucos desenvolvedores ou distribuídas entre os contribuidores da comunidade que auxilia na manutenção do software livre. Por fim, foi avaliada a quantidade de autores associados à manutenção e evolução de arquivos grandes, calculando quantos autores são responsáveis por 70% dos commits. A média dos autores responsáveis pela maioria das alterações nos arquivos regulares segue o mesmo padrão observado nos resultados do estudo de Bus Factor da JetBrains[13], em que a média dos grandes responsáveis varia de 1 a 2, e a mediana conta com apenas 1 pessoa. Entretanto, quando o foco se volta para os arquivos grandes, a proporção de

[13] https://github.com/JetBrains-Research/bus-factor-explorer

autores responsáveis pela maioria das alterações varia. Tendo em vista que a média para os arquivos regulares estava entre 1 e 2 autores, para os arquivos grandes essa média varia de 3 a 4 autores, já a mediana tende a 2 autores.

Tabela 9: Percentis dos autores de 70% dos commits.

	Arq. Grandes		Arq. Regulares	
Percentil	**Média**	**Mediana**	**Média**	**Mediana**
Geral	3,64	2	1,44	1
50%	5,92	3	1,85	1
25%	8,71	5	2,24	1
10%	14,32	10	2,79	2

Ao analisar os 50% dos arquivos com o maior número de autores responsáveis pelo projeto, é notável uma diferença bastante acentuada e crescente entre os grandes e os regulares. Enquanto os arquivos regulares mantêm uma média de 1 a 2 autores, os grandes já apresentam 5 a 6 autores em média. Essa diferença se amplia significativamente à medida que se restringe o foco a percentis mais seletivos: nos 25% maiores, os arquivos grandes atingem, em média, de 8 a 9 autores, contra 2 a 3 dos arquivos regulares. No top 10%, os grandes mantêm entre 14 e 15 autores, enquanto os regulares permanecem estáveis em 2 a 3 autores. Quanto mais se afunila a métrica (de 50% para 10%), maior se torna o abismo entre os dois grupos, evidenciando que arquivos grandes demandam colaboração exponencialmente maior, enquanto os regulares mantêm padrões consistentes de autoria.

Resposta para QP4: Arquivos grandes apresentam uma autoria mais fragmentada do que os arquivos regulares. Enquanto os regulares mantêm entre 1 e 2 autores, os grandes concentram médias de 3 a 4, podendo chegar a 15. Isso indica uma colaboração forçada: embora o conhecimento sobre o arquivo esteja mais disperso (menor *Truck Factor*), o excesso de intervenções simultâneas eleva os custos de coordenação e comunicação, diluindo a responsabilidade e aumentando a propensão a defeitos.

5 Discussão

Este trabalho preenche uma lacuna na literatura ao definir arquivos grandes com base no percentil superior (1%) da linguagem predominante dos projetos. Constata-se que tais arquivos não são anomalias, mas característica intrínseca a 74,88% dos projetos analisados. A QP2 indica um problema estrutural e precoce: 60% desses arquivos já "nascem grandes", sugerindo falhas no design inicial e negligência quanto à modularização, evidenciando o envelhecimento de software [18], em que priorizar desempenho ou entrega rápida em detrimento da arquitetura insere a dívida técnica no núcleo da lógica de negócio.

Uma vez estabelecidos, esses arquivos geram um ciclo vicioso de instabilidade e alto custo de manutenção. Segundo a QP3, concentram maior taxa de correções de bugs (30,71%) que de novas funcionalidades, ao mesmo tempo em que recebem metade das refatorações destinadas a arquivos regulares. Essa resistência à limpeza, alinhada aos achados de Cedrim et al. [4] sobre os riscos de alterar código problemático, confirma que arquivos grandes atuam como "Objetos Deus": em vez de evoluir, o software apenas envelhece, pois o receio das equipes em refatorá-los torna a dívida técnica permanente.

A complexidade técnica é ainda agravada por fatores humanos (QP4): arquivos grandes sofrem com autoria fragmentada, exigindo "colaboração forçada" de até 15 desenvolvedores, contra 1 ou 2 em arquivos regulares. Essa diluição de ownership eleva custos de coordenação e propensão a erros. Ainda assim, o cenário é reversível: refatorações baseadas no Princípio da Responsabilidade Única (SRP) [15] mostraram-se eficazes para desmontar esses "Gigantes", transformando-os em "Moinhos de Vento" e restaurando a saúde arquitetural do projeto.

Suspeita-se também que arquivos grandes atuem como depósitos de más práticas e maus cheiros de código, convergindo problemas arquiteturais e organizacionais que refletem decisões de design alheias à modularidade e à separação de responsabilidades, remetendo à concentração de responsabilidades em entidades centrais e ao aumento da complexidade estrutural. No longo prazo, esse acúmulo pode contribuir para o envelhecimento do software descrito por Parnas [18], dificultando a compreensão, modificação e evolução do sistema.

Trata-se, porém, de hipótese empírica, não de conclusão definitiva. O contexto do projeto pode influenciar a formação desses arquivos: em sistemas de larga escala, fatores como desempenho, limitações arquiteturais ou geração automática de código podem gerar arquivos extensos sem indicar necessariamente problemas estruturais graves. Assim, apesar da associação encontrada entre arquivos grandes e indicadores de débito técnico, são necessárias investigações adicionais sobre como o fenômeno se manifesta em diferentes linguagens, domínios e arquiteturas.

6 Implicações

Este estudo inova metodologicamente ao definir empiricamente "arquivos grandes" como o percentil superior (1%) por linguagem, demonstrando que métricas de tamanho absoluto são insuficientes sem considerar o contexto tecnológico. Pesquisas futuras podem adotar essa métrica relativa para investigar a dívida técnica em ecossistemas industriais ou em códigos gerados por Inteligência Artificial, além de viabilizar estudos longitudinais sobre ferramentas de detecção de mau cheiro voltadas a prever a degradação arquitetural.

Para líderes e praticantes, os resultados alertam que arquivos grandes preveem instabilidade e altos custos de manutenção. Como esses artefatos representam 30% das correções de defeitos, as equipes devem configurar ferramentas estáticas e pipelines de CI/CD (integração e implantação contínuas) para bloquear o crescimento desordenado acima do percentil 99, adotando a refatoração contínua para decompor componentes gigantes em módulos menores e coesos.

Em projetos de Software Livre, a presença de arquivos grandes em 75% dos repositórios ameaça a sustentabilidade diante da alta rotatividade de contribuidores. A "colaboração forçada" nesses artefatos dificulta o onboarding e dilui o code ownership. Mantenedores devem ser rigorosos nas revisões, rejeitando contribuições que ampliem arquivos monolíticos (60% já nascem grandes) e incentivando a modularidade desde o primeiro commit.

No ensino, a pesquisa reforça a urgência dos princípios SOLID como práticas de sobrevivência do software. A correlação entre tamanho do arquivo e incidência de defeitos pode ilustrar aos alunos que "código que funciona" difere de "código de qualidade", com estudos de caso de refatoração servindo de exercícios práticos para formar engenheiros capazes de transformar sistemas legados em arquiteturas sustentáveis.

7 Limitações

Em estudos empíricos, há sempre limitações que podem afetar os resultados e as conclusões. A seguir, são listadas as limitações deste estudo e as decisões adotadas para mitigá-las.

Definição de arquivos grandes: Sem consenso na literatura, adotou-se o percentil 99 global, com base no conceito de mau cheiro de código. Essa definição pode variar conforme o projeto, a linguagem ou o período analisado, podendo omitir arquivos grandes em contextos específicos, cabendo a estudos futuros investigar o fenômeno sob outras perspectivas.

Borda rígida: Um limiar fixo é sensível a variações pontuais, já que uma única linha pode reclassificar um arquivo. Métricas como a Curva ROC [21] poderiam reduzir esse impacto, mas não foram aplicadas, ficando como sugestão para trabalhos futuros.

Adoção de outros limiares: A escolha do P99 foi empírica, baseada na distribuição de LOC das 15 linguagens (Tab. 1). Percentis menores gerariam volumes excessivos de arquivos (100961 para P90, 50463 para P95, 30287 para P97), inviabilizando a análise (até 4 dias de processamento por projeto), enquanto o P99.5 resultaria em amostra ainda mais restritiva (5048 arquivos). Reconhece-se que essa decisão metodológica, embora limitante, não reduz a relevância dos resultados.

Exclusão dos projetos Linux e pytorch da QP2, QP3 e QP4: Feita por restrições computacionais na mineração de dados (limite de RAM, conexão e tempo hábil), reduzindo a generalização dos resultados. Este trabalho é um primeiro passo para compreender um fenômeno multifacetado, a ser explorado por estudos futuros.

Filtragem de arquivos gerados automaticamente: Não foram filtrados arquivos gerados automaticamente; a amostragem excluiu apenas categorias específicas (HTML, CSS, shell etc.), conforme a Seção 3.1. Como esses arquivos também são versionados e recebem manutenção, sua inclusão é válida, embora a origem do código seja uma variável relevante para estudos futuros.

Normalização para KLOC: Não foi realizada, pois a Tabela 4 mostra que arquivos grandes sofrem mais manutenções (1,6x). As Tabelas 5 e 6 indicam que, em 14 das 15 linguagens, a densidade de alteração por linha é maior em arquivos regulares (ex.: JavaScript cresce 116,7% em regulares vs. 23,2% em grandes; exceção: Python, 70,9% vs. 50,0%). Frequência de manutenção e densidade de mudança são fenômenos distintos e complementares.

Testes estatísticos: Aplicou-se o teste de Wilcoxon-Mann-Whitney (α = 0,05) com Cohen's d para métricas contínuas, observando-se maior variação de tamanho (d=0,51) e menor intervalo entre modificações (d=-0,43) em arquivos grandes. Para a distribuição de tipos de commit, aplicou-se o teste qui-quadrado (α = 0,05, 9 graus de liberdade) com Cramer's V, com efeitos moderados em Swift (V=0,231), JavaScript (V=0,213) e pequeno em C# (V=0,198). Arquivos grandes concentram mais commits de Bug-Fix (30,7% vs. 26,4%), enquanto regulares têm mais Features (14,2% vs. 11,2%), reforçando o maior esforço de manutenção associado a arquivos grandes.

Validação da classificação automática para as SCIs: Baseada na heurística de Kruger [11]. Validação manual estratificada (768 commits, 95% de confiança, margem de erro de 5%) resultou em concordância geral de 90,5% (695/768), com maior divergência na categoria *Other*.

Influência dos arquivos de teste nas razões de modificações: 70% dos arquivos grandes não possuem "test" ou "validation" no nome, e a taxa de correção de bugs (30,71%) incide sobre todo o conjunto. A categoria Teste é até menor em arquivos grandes (8,69%) que em regulares (9,60%) (Tabela 8), descartando a hipótese de que a concentração de testes explique isoladamente essa diferença.

8 Conclusão

Este estudo caracterizou a existência e a evolução de arquivos grandes em 450 repositórios de software livre em 15 linguagens populares. Usando o percentil 99 de cada linguagem como definição, constatou-se que o fenômeno é frequente, atingindo 74,88% dos repositórios. A maioria desses arquivos (60%) já nasce grande e tende a crescer, sugerindo falhas no design inicial e na modularização, além de acúmulo orgânico. Esses "Gigantes" são duas vezes mais complexos que arquivos regulares, alterados com o dobro da frequência, embora em intervalos maiores, e cerca de 30% deles são arquivos de teste.

A manutenção é predominantemente reativa, respondendo por 30,71% das correções de bugs, mas recebendo apenas metade das refatorações destinadas a arquivos regulares, possível indício de receio das equipes em lidar com códigos extensos. Há também fragmentação de autoria: enquanto arquivos regulares têm de 1 a 2 responsáveis principais, os grandes chegam a envolver até 15 desenvolvedores, atuando como centros de colaboração forçada que diluem o conhecimento técnico.

Conclui-se que arquivos grandes são polos de instabilidade que ameaçam a saúde dos projetos a longo prazo, mas esse cenário é reversível por meio de refatorações baseadas em SRP [15], capazes de redistribuir funcionalidades e eliminar código morto. Ao isolar responsabilidades, é possível transformar "Gigantes" nocivos em "Moinhos de Vento": grandes, porém modulados, gerenciáveis e favoráveis à evolução sustentável do software.

Disponibilidade De Artefatos

O pacote de replicação deste estudo está disponível publicamente no Zenodo, em zenodo.org/uploads/21366286 sob o DOI 10.5281/zenodo.21366286, e no Github, em github.com/batichotti/Giants-or-Windmills. O pacote de replicação é disponibilizado sob a *Apache License 2.0*.

Agradecimentos

Os autores agradecem à Universidade Tecnológica Federal do Paraná de Campo Mourão, ao CNPq (409359/2024-6, 444802/2024-0) e à Fundação Araucária/Governo do Paraná (PRD2023361000043).

Referências

[1] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2016. A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, Association for Computing Machinery, New York, NY, USA, 1–10.

[2] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1–14.

[3] H. Borges, A. Hora, and M. T. Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Association for Computing Machinery, New York, NY, USA, 334–344. https://doi.org/10.1109/ICSME.2016.31

[4] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 465–475. https://doi.org/10.1145/3106237.3106259

[5] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering* 47, 1 (2018), 17–66.

[6] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Cham, 341–350. https://doi.org/10.1109/SANER.2015.7081844

[7] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.* 11, 2 (2012), 5–1.

[8] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, USA.

[9] E. Jabrayilzade, M. Evtikhiev, E. Tuzun, and V. Kovalenko. 2022. Bus Factor in Practice. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, Los Alamitos, CA, USA, 97–106. https://doi.org/10.1109/ICSE-SEIP55303.2022.9793985

[10] Yasutaka Kamei and Emad Shihab. 2016. Defect Prediction: Accomplishments and Future Challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. IEEE, USA, 33–45. https://doi.org/10.1109/SANER.2016.56

[11] Jacob Krüger, Yi Li, Kirill Lossev, Chenguang Zhu, Marsha Chechik, Thorsten Berger, and Julia Rubin. 2024. A meta-study of software-change intentions. *Comput. Surveys* 56, 12 (2024), 1–41.

[12] Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2018. An empirical study of architectural decay in open-source software. In *2018 IEEE International conference on software architecture (ICSA)*. IEEE, IEEE, USA, 176–17609.

[13] Mateus Lopes and Andre Hora. 2022. How and why we end up with complex methods: a multi-language study. *Empirical Softw. Engg.* 27 (sep 2022). https://doi.org/10.1007/s10664-022-10144-3

[14] M. Mantyla, J. Vanhanen, and C. Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* ICSM, USA, 381–384. https://doi.org/10.1109/ICSM.2003.1235447

[15] Robert C. Martin. 1996. The Single Responsibility Principle. *C++ Report* 8, 5 (May 1996), 48–51.

[16] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, USA, 101–110. https://doi.org/10.1109/ICSME.2014.32

[17] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2018. The scent of a smell: an extensive comparison between textual and structural smells. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18')*. Association for Computing Machinery, New York, NY, USA, Article 740, 1 pages. https://doi.org/10.1145/3180155.3182530

[18] David Lorge Parnas. 1994. Software aging. In *Proceedings of 16th International Conference on Software Engineering*. IEEE, IEEE, USA, 279–287.

[19] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering* 29, 1 (2022), 47–94.

[20] Alberto Pinto, Gustavo de Souza. 2023. Cognitive Driven Development helps software teams to keep code units under the limit! *Journal of Systems and Software* 206 (2023), 111830.

[21] Anthony Robinson, Marissa Huber, Eathan Breaux, Erika Pugh, and Matthew Calamia. 2022. Failing the b Test: The Influence of Cutoff Scores and Criterion Group Approaches in a Sample of Adults Referred for Psychoeducational Evaluation. *Journal of Clinical and Experimental Neuropsychology* 44, 9 (Oct. 2022), 619–626. https://doi.org/10.1080/13803395.2022.2153805

[22] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in JavaScript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, USA, 294–305. https://doi.org/10.1109/SANER.2017.7884630

[23] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2021. A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering* 47, 1 (2021), 17–66. https://doi.org/10.1109/TSE.2018.2880977

[24] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[25] Clare Sudbery. 2020. Refactoring: This class is too large. (14 April 2020). https://martinfowler.com/articles/class-too-large.html

[26] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. ACM, New York, NY, USA, 4–15.

[27] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088. https://doi.org/10.1109/TSE.2017.2653105

[28] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, University Park, PA, USA, 92–95.

[29] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. WCRE, USA, 242–251. https://doi.org/10.1109/WCRE.2013.6671299

[30] Morteza Zakeri-Nasrabadi, Saeed Parsa, Ehsan Esmaili, and Fabio Palomba. 2023. A systematic literature review on the code smells datasets and validation mechanisms. *Comput. Surveys* 55, 13s (2023), 1–48.

[31] Min Zhang, Tracy Hall, Nathan Baddoo, and Paul Wernick. 2008. Do bad smells indicate"trouble"in code?. In *Proceedings of the 2008 workshop on Defects in large software systems*. zhang, China, 43–44.