Universidade Católica do Salvador
Bacharelado em Engenharia de Software

Rodrigo Soares da Silva

SED99 - Software Evolution Dataset - 99 repositories

Salvador
2020

Rodrigo Soares da Silva

SED99 - Software Evolution Dataset - 99 repositories

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como parte dos requisitos necessários para a obtenção do Título de Engenheiro de Software.
Orientador: Prof. Me. André Brasil Vieira Wyzykowski

Universidade Católica do Salvador

Salvador
2020

Rodrigo Soares da Silva

SED99 - Software Evolution Dataset - 99 repositories

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como requisito parcial para a obtenção do título de Engenheiro de Software.

Comissão Examinadora

Prof. Me. André Brasil Vieira Wyzykowski
Universidade Católica do Salvador
Orientador

Prof. Esp. Antônio Cláudio Neiva
Universidade Católica do Salvador

Prof. Me. Osvaldo Requião Melo
Universidade Católica do Salvador

Salvador, 29 de janeiro de 2021

Dedico à Maria de Lourdes Soares Lira, minha avó materna.

Agradecimentos

Desde a escolha do Curso Superior, do resultado do vestibular e do ingresso à Universidade Católica do Salvador, um universo de possibilidades e desafios abriu-se para mim. Parece que foi ontem, mas já se passaram quase quatro anos.

A Deus agradeço por minha existência e por sempre me mostrar através dos seus pequenos milagres diários o meu verdadeiro propósito de vida. Sua presença forte em minha vida me fez igualmente forte nos momentos de celebrar vitórias, de vencer dificuldades e administrar as perdas ao longo desse período.

Família é alicerce, afeto e proteção. Na minha Família eu tive amor, incentivo e apoio necessários para seguir firme no meu objetivo profissional. Minha mãe pedagoga sempre foi incansável na minha formação e me oportunizou todos os meios ao seu alcance para o meu sucesso acadêmico. Ela é minha inspiração de caráter e profissionalismo na minha construção pessoal de homem íntegro, de cidadão consciente e de profissional de excelência. Infelizmente, meus avós são falecidos. Aqui enalteço o meu avô materno, que me deixou há treze anos, mas que foi o precursor da minha profissão, pois presenteou-me aos seis anos com meu primeiro computador, uma máquina usada que ele mesmo montou a partir de muitas outras da sua velha oficina. A minha avó materna esteve presente no dia do meu vestibular e tenho certeza que ela gostaria de estar aqui nesse momento de conclusão de curso, mas não foi a vontade de Deus. Sei que ela continua cuidando de mim no "Céu das Avós". Ao meu único tio agradeço pela admiração e incentivo em todas as minhas demandas.

Meus professores foram os melhores provocadores para a minha formação como engenheiro de software. A palavra desafio define a nossa relação ao longo desses quase quatro anos. Seus ensinamentos, compreensão e amizade foram de fundamental importância para a minha construção profissional.

Resumo

O SED99 é um dataset contemplado por projetos de cunho *Open Source* / *Software* livre para auxílio no estudo da Evolução de *Software* e na análise do comportamentos de projetos diversificados ao longo de 5 anos em pontos de controle quadrimestrais. O trabalho tem como objetivo apresentar as motivações de sua criação, seu processo de compilação, formas de extrair informações relevantes sobre seus dados e incentivar à outras formas de exploração de seu conteúdo, além de representar uma contribuição à ciência da Evolução de *Software*, que comparada a outras áreas carece de mais estudos a seu respeito.

Palavras-Chave: 1. Evolução de Software. 2. Leis de Lehman. 3. Open Source. 4. Dataset

Abstract

The SED99 is a dataset contemplated by Open Source / Free Software projects to aid in the study of Software Evolution and in the analysis of the behavior of diversified projects over 5 years at four-month control points. The work aims to present the motivations for its creation, its compilation process, ways to extract relevant information about its data and encourage other ways of exploring its content, in addition to representing a contribution to the science of Software Evolution, which compared other areas need more studies about it.

Keywords: 1. Software Evolution. 2. Lehman's Laws. 3. Open Source. 4. Dataset

Lista de figuras

Lista de Siglas e Abreviaturas

SED99	*Software Evolution Dataset - 99 repositories*
CCN	*Cyclomatic Complexity Number*
NLOC	*Lines of Code without comments*
TCC	*Total Cyclomatic Complexity*
LOC	*Lines of Code*
KLOC	*Kilo Lines of Code*
LCOM	*Lack of Cohesion Methods*
DAM	*Data Access Metrics*
CAM	*Cohesion Among Methods in Class*
CVS	*Concurrent Version System*
SVN	*Apache Subversion*
HTTP	*Hypertext Transfer Protocol*

Sumário

1 Introdução

A tecnologia está atuando cada vez mais nas diversas áreas da sociedade, potencializando a produtividade dos indivíduos em diferentes segmentos. *Softwares* são criados, levando-se em consideração um conjunto de fatores que os tornam úteis em um determinado contexto, estando estes sempre em constantes transformações e exigindo, por consequência, que se reestruturem para continuarem atendendo as demandas de maneira satisfatória.

De acordo com Okwu e Onyeje (2014), "O processo pelo qual os programas são modificados e adaptados em um ambiente em mudança é conhecido como evolução de *software*". O ambiente é o contexto no qual o *software* está inserido, ou seja, causando e receptando influência. Muitas características como integração dos usuários, nível de satisfação, velocidade de resposta, funcionalidades, usabilidade e muitas outras, que serão abordadas posteriormente, compõem o ambiente/contexto de um determinado *software*.

No estudo da Evolução de *Software*, Meir Lehman ganhou destaque através dos seus estudos de natureza empírica que cobrem um período significativo do século XX, resultando nas Leis de Lehman, que têm por objetivo caracterizar propriedades invariantes para serem observadas nos projetos de *software* (HERRAIZ et al., 2013). Uma série de publicações ao longo das décadas de 70 até 90 culminaram em uma referência que permite analisar a fundo o ciclo de vida de um *software*.

O conceito de ambiente em constante mudança permite a constatação da necessidade de se pensar sobre os estudos que norteiam a Evolução de *Software*. A forma de se fazer *software* e as interações resultantes também evoluem e abrem a possibilidade de observação do comportamento das Leis de Lehman para aplicações que fogem à forma proprietária de desenvolvimento que, neste trabalho, tem como um dos objetos de estudo, o mundo *open source*.

O estudo das Leis de Lehman no ciclo de vida dos projetos de desenvolvimento de *software*, contribui na evolução da própria Evolução de *Software*, que necessita de mais estudos empíricos para dar suporte à aplicabilidade dessas leis (KAUR; RATTI; KAUR, 2014), destacando-se neste trabalho a adoção de projetos *open source*, como um dos processos disponíveis para se criar e manter *software*.

Para que ocorra o estudo, *datasets* com coletâneas de informações de suporte são necessários, e, como foi abordado previamente, a área se encontra carente de estudos e materiais para confecção de novas abordagens, tornando a compilação de informações uma contribuição significativa para a área da Evolução de *Software*.

1.1 Aplicabilidade e Motivação

A tecnologia evolui rapidamente, se comparada com outras áreas, e cria um universo cronologicamente heterogêneo. É interessante a aplicação das Leis de Lehman que foram concebidas, considerando-se uma realidade do século XX ao longo de três décadas, em um contexto mais recente que é o *open source*, sendo esta terminologia adotada em 1998 e que gerou um movimento em massa no que diz respeito ao acesso do código fonte de forma livre.

Essa união de dois mundos da computação gera uma contribuição simbiótica que incrementa a área de Evolução de *Software*, através de uma análise empírica, revelando formas de interpretação das Leis, métricas que podem ser utilizadas, estruturas organizacionais singulares, diferentes pontos de vistas, entre várias outras características. Para patrocinar essa relação, a criação de um *dataset* com registros históricos de softwares *Open Source* para a promoção de novos estudos voltados a este objetivo, avaliado por um conjunto de métricas, respondendo perguntas acerca de sua composição na visão do(s) criador(es) e do(s) consumidor(es) Gebru et al. (2018), compõe uma contribuição que incentiva a geração de novos estudos na área da Evolução de *Software*.

1.2 Objetivos

O objetivo deste trabalho é documentar o processo de criação de um *dataset* personalizado, voltado ao estudo da aplicabilidade das Leis de Lehman em projetos *Open Source*, gerando uma compilação de dados oriundos da análise dos códigos fonte e informações ofertadas pela estrutura de seus repositórios como saída final do documento, tornando-se disponível para pesquisadores continuarem importantes avanços na área.

1.3 Objetivos Específicos

- Criar um *dataset* com registros históricos de *softwares Open Source* para a promoção de novos estudos.
- Disponibilizar publicamente o *dataset* criado.

2 Fundamentação Teórica

Nesta etapa serão esclarecidos temas pertinentes sobre a criação e inspiração do *data-set*, baseada na literatura que compreende as Leis de Lehman (Seção 2.1), suas definições, uma introdução ao *open source* (Seção 2.2) e seu respectivo impacto, e a própria estrutura de repositório (Seção 2.3) adotada neste trabalho.

2.1 Leis de Lehman

As Leis de Lehman foram concebidas historicamente através do incremento do estudo de projetos de *software*, analisando propriedades invariantes em seus arquivos de códigos fonte. Sistemas *E-type*, como são mencionados nas leis abaixo, são sistemas reais, que são inseridos no contexto para lidar com negócios, interação com pessoas, onde não existe uma precisão em relação a sua especificação, o que garante sua constante mudança(E-TYPE...,).

2.1.1 Mudança Contínua

Um sistema desenvolvido para um contexto, seus usuários, objetivos, limitações, com variáveis que afetam direta ou indiretamente seu ciclo de vida, deve sofrer interferências adaptativas e evolutivas constantes para corresponder às mudanças que o próprio ambiente sofre. Essas mudanças podem ser oriundas de diversas variáveis, tais como: concorrência, tendências, novas tecnologias, entre outras. Se esse processo não for realizado, essa aplicação torna-se progressivamente menos útil a esse ambiente específico. (HERRAIZ et al., 2013; LEHMAN, 1996)

Segundo Lehman (1996), a lei se define por "Um programa do tipo E que é usado deve ser continuamente adaptado, caso contrário, torna-se progressivamente menos satisfatório.". Uma das formas disponíveis de se visualizar esta lei é através do gráfico de contribuição no projeto que a ferramenta *Github* oferece, que realiza um somatório dos *commits* realizados, com a exclusão dos *merge commits*, em uma linha do tempo desde o início do projeto, como mostra a figura a seguir.

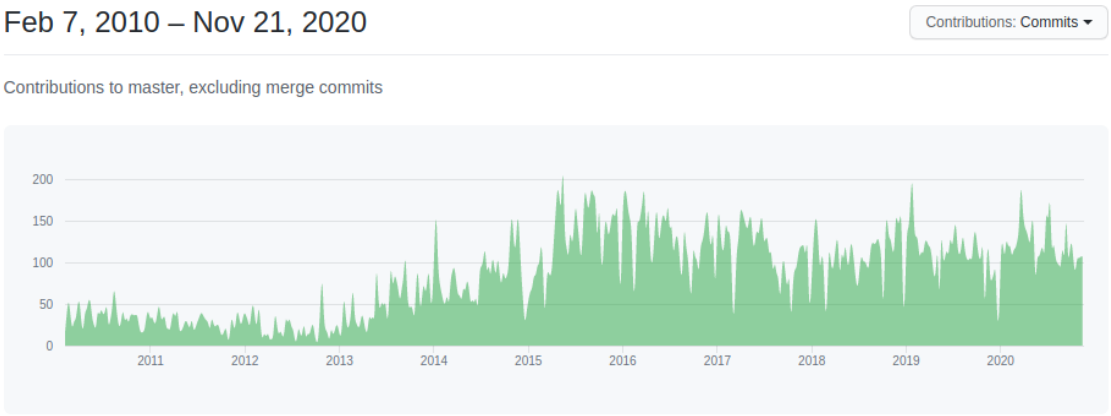

Figura 1 – Gráfico de contribuição do projeto *Elasticsearch* extraído do *Github*.

A mudança contínua faz-se presente através do somatório de *commits* em um intervalo determinado, que indica as alterações realizadas no projeto, sejam adições ou deleções, provando que *softwares* precisam se modificar para se adequarem ao seu propósito, e que isso é inerente ao seu ciclo de vida, nunca chegando a um nível de estabilização que intervenções no código fonte não sejam necessárias.

2.1.2 Complexidade Crescente

Com as mudanças ocasionadas em um *software*, como discutido na primeira lei acima, acaba-se degradando a estrutura do sistema, natural à intervenção de código novo estabelecido para algum fim, seja de evolução, correção e/ou adaptação. Degradar significa fragmentar, naturalmente aumentando a complexidade da aplicação em si, com dependências, variáveis, funções, entre outros. Esse processo torna-se cumulativo chegando ao ápice da não viabilidade de intervenção do sistema, exigindo esforços na reestruturação dos componentes da aplicação para diminuir a complexidade e permitir o prosseguimento do seu ciclo de vida.

É interessante destacar a necessidade de refatoração, análise de qualidade, padrão de codificação, decisões, tecnologias utilizadas e várias outras métricas/processos necessários nos projetos, pois o contexto pode ser crucial e determinar o sucesso/fracasso de uma aplicação.

Esta lei, de acordo com Lehman (1996), define-se por: “Conforme um programa evolui, sua complexidade aumenta, a menos que seja feito um trabalho para mantê-la ou reduzi-la.”.

Um fator histórico pertinente a essa lei foi que no artigo de 1974, no qual as três primeiras leis foram publicadas, essa lei em específico utilizava-se do termo "entropia"ao invés de "complexidade". A mudança ocorreu devido ao fato da introdução da desordem, tornando difícil a compreensão da estrutura dos programas após intervenção(HERRAIZ et al., 2013).

Uma forma de representar numericamente a complexidade, é através do cálculo da Taxa de Complexidade Ciclomática, que será abordada mais adiante neste trabalho. Para representar esta lei, um gráfico da variação desta taxa ao longo de um período determinado auxilia na compreensão, como na figura 2:

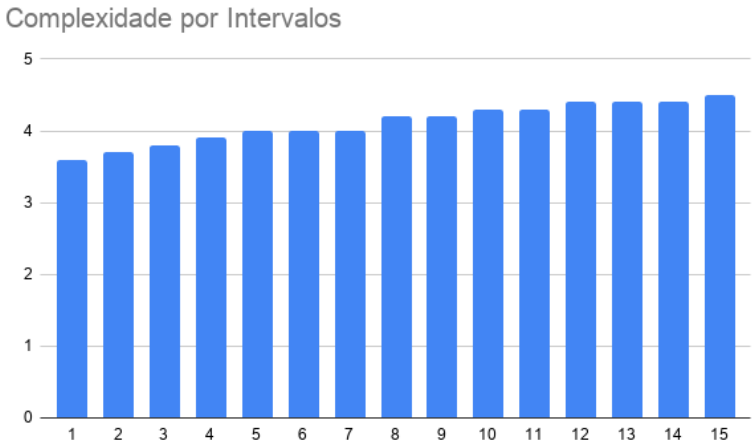

Figura 2 – Gráfico da Complexidade Ciclomática de um *software* fictício em intervalos mensais.

A figura 2 permite a suposição de hipóteses em relação ao ciclo de vida deste *software*. Percebe-se um incremento gradual na complexidade à medida que os intervalos passam, alertando para a necessidade de refatoração, por exemplo.

2.1.3 Auto Regulação

Autorregulação está intrinsecamente ligado à estabilização. Considerando-se uma corporação que possui interesses bem determinados, pontos de controle são estabelecidos para o cumprimento de objetivos, munidos de parâmetros estabelecidos. Naturalmente, as pessoas envolvidas no projeto empreendem esforço no *software* com intervenções incrementais que se mantém constantes entre as versões do sistema.

A definição desta lei segundo Lehman (1996) se dá por: "O processo de evolução do programa é autorregulado com distribuição próxima da normal de medidas de atributos de produto e processo."(LEHMAN, 1996).

Uma forma de se manter o controle das modificações realizadas entre as diferentes versões do sistema é através do arquivo de modificações, mais conhecido por "*change log*".

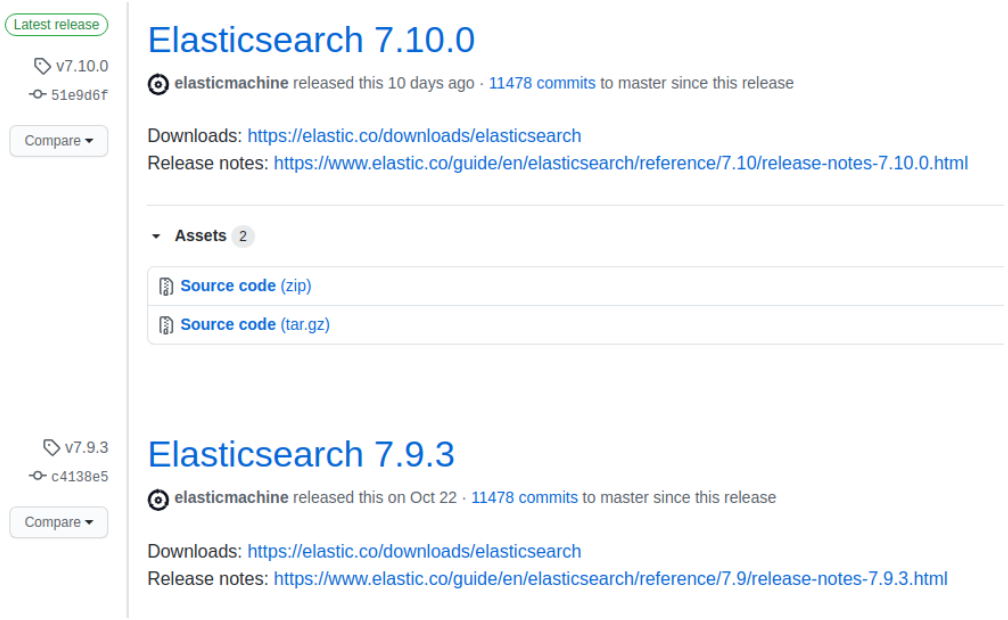

Figura 3 – Versões do *Elasticsearch*, com a distribuição das *Release Notes*

A figura 3 remete a uma marcação das modificações entre as versões do sistema, as quais estão disponíveis para consulta através dos links fornecidos, e que existe uma tendência natural da volumetria das modificações seguirem bem próximas, justamente para se manter um maior controle, e consequentemente, qualidade, alcançando os objetivos propostos. Uma comparação pode ser realizada entre as versões:

- 7.9.2 : <https://www.elastic.co/guide/en/elasticsearch/reference/7.9/release-notes-7.9.2.html>
- 7.9.1: <https://www.elastic.co/guide/en/elasticsearch/reference/7.9/release-notes-7.9.1.html>

É notável a proximidade em relação a quantidade de alterações realizadas, mesmo em um projeto de código aberto, que possui várias versões, inclusive as que possui suporte para terceiros, com funções extras.

2.1.4 Estabilidade Organizacional

"A taxa média de atividade global efetiva em um sistema em evolução é invariável em relação ao produto e ao tempo de vida"(LEHMAN, 1996).

Essa lei refere-se à equipe ativa no desenvolvimento do *software*. Quando fala-se em equipe, remete-se aos indivíduos envolvidos no processo. Por mais que mude, a taxa da atividade no *software* se mantém constante durante seu ciclo de vida, além de equipes muito grandes causarem *overhead* na comunicação, sendo prejudicial ao desenvolvimento.

No caso de projetos *open source*, as pessoas são livres para se juntarem a um projeto, respeitando suas diretrizes, e de fato contribuírem. O controle atua mais no conteúdo gerado do que os participantes envolvidos, oferecendo liberdade, mas com foco na qualidade. A figura a seguir mostra os maiores contribuidores do projeto.

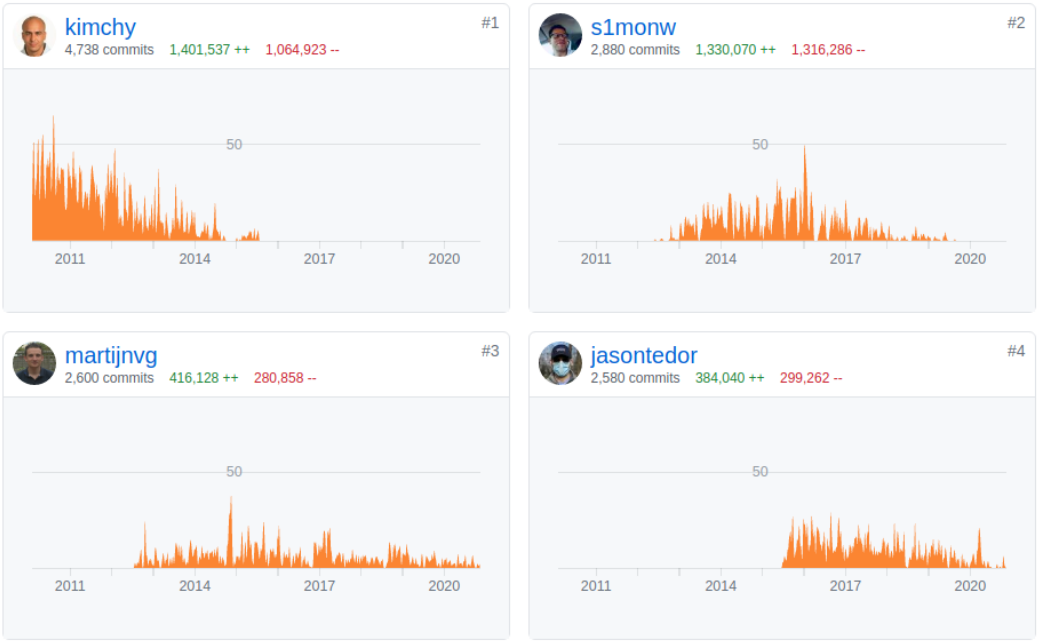

Figura 4 – Contribuidores do *Elasticsearch* em ordem decrescente.

Mesmo não possuindo uma equipe totalmente definida, geralmente os projetos *open source* possuem mantenedores, pessoas que ficam encarregadas em analisar os *pull requests* realizados pelos contribuidores, como uma espécie de controle de qualidade.

2.1.5 Conservação de Familiaridade

Faz-se necessário a percepção de como as leis se conectam no contexto. Com o *background* oferecido pelas leis anteriores, logicamente, quando um *software* sofre intervenção, mudanças ocorrem, e toda mudança causa efeitos, positivos e/ou negativos.

A lei define-se por: "Durante a vida ativa de um programa em evolução, o conteúdo de lançamentos sucessivos é estatísticamente invariante"(LEHMAN, 1996).

Quando novas funcionalidades são adicionadas, consequentemente, o número de defeitos tende a subir, gerando gastos para refatoração e com grandes chances de impactar negativamente o ciclo de lançamentos de novas versões. Para evitar isso, as modificações no sistema são limitadas entre os lançamentos.

Essa estratégia também visa garantir a qualidade e manter o time com conhecimento satisfatório sobre o *software*, pois mudanças constantes e descontroladas causam maior dificuldade a uma equipe em estar ciente dos objetivos, já que o *software* corresponde a um determinado negócio.

Pode-se perceber isso ao analisar o arquivo de mudanças de um *software* de acordo com a sua versão, como demonstra as diferenças entre as versões do reprodutor de mídia VLC.

```
Changes between 3.0.9.2 and 3.0.10:
-----------------------------------

Misc:
 * Update Twitch & VLSub scripts


Changes between 3.0.9.1 and 3.0.9.2:
-----------------------------------

Misc:
 * Properly bump the version in configure.ac


Changes between 3.0.9 and 3.0.9.1:
-----------------------------------

Misc:
 * Fix VLSub returning 401 for earch request
```

Figura 5 – Modificações incluídas entre diferentes versões do VLC.

É importante destacar o porquê de um *software*. Qual dor ele procura sanar? Seu objetivo? De forma geral, *sofwares* necessitam ser pontuais em suas funções, para permitirem um ciclo de vida saudável das suas funções e a equipe que o intervém. Como a figura 5 demonstra, mudanças pequenas e pontuais entre as versões garantem que as partes envolvidas saibam sobre o produto no qual operam e consigam dar prosseguimento às modificações, tornando cada vez menor o número de erros gerados.

2.1.6 Crescimento Contínuo

Uma aplicação está atuando em um determinado ambiente, sendo esse não estático. As influências são as mais diversas possíveis. Externamente, como concorrentes, tendências, necessidades, entre outros. Internamente, como capacidade, disponibilidade, segurança etc.

A lei define-se por: “O conteúdo funcional de um programa deve ser aumentado continuamente para manter a satisfação do usuário ao longo de sua vida”(LEHMAN, 1996).

Muitas das características acima, mesmo que não diretamente relacionadas, influenciam no nível de satisfação do usuário com o sistema, exigindo que seu conteúdo funcional seja aumentado para que sempre agregue valor à respectiva experiência. Também é característica importante, a manutenção do que já está implementado, seja melhorando sua performance, adicionando novos conteúdos e/ou reformulando a função. Essa é uma es-

tratégia muito adotada pelas grandes indústrias para não perderem o engajamento dos usuários com suas respectivas plataformas. As mudanças nem sempre são planejadas, acarretando em lançamentos com códigos desestruturados, *bugs*, aumentando não só o conteúdo, mas a complexidade, custos, entre os mais diversos fatores.

Percebe-se na adição de funcionalidades à aplicações, como por exemplo o recurso "*stories*", que permite a postagem de fotos que ficam disponíveis durante 24 horas para uma rede de amigos em plataformas como o *Facebook*, *WhatsApp* e *Instagram*.

2.1.7 Qualidade Decrescente

A percepção do usuário é algo essencial em sua interação com uma aplicação. Com o conhecimento prévio das leis anteriores, percebe-se que uma aplicação possui um ritmo de modificações. Por mais que um *software* esteja muito bem implementado, se não seguir o mercado, pode acabar em desuso, pois não foi a sua qualidade enquanto construção que sofreu decréscimo, mas o ambiente em que ele está inserido mudou, exigindo novas soluções para atender às necessidades e expectativas, resultando na diminuição da satisfação.

Se um *software* hoje funciona muito bem e atende a sua demanda, é necessário esforços para detectar e atuar em incertezas que podem comprometer o futuro da sua aplicação, necessitando assim, até mesmo sair do escopo da tecnologia da informação, por exemplo. Resguardadas as devidas proporções na área da tecnologia da informação, a análise de mercado, marketing, estratégia de negócios, questões legais, entre outros, são tópicos não inerentes em escopo, mas que andam em sintonia quando o protagonista de um negócio é um *software*.Tudo faz parte dentro de um determinado contexto. Evidencia-se dessa forma, a influência que essa convergência cria, auxiliando na evolução que as áreas de estudos sofrem.

A definição formal da lei segundo Lehman é: "Os programas do tipo E serão percebidos como de qualidade declinante, a menos que sejam rigorosamente mantidos e adaptados a um ambiente operacional em mudança"(LEHMAN, 1996).

Aproveitando-se do exemplo anterior, percebe-se que os *softwares* concorrentes estão numa disputa para dominar cada vez mais parte do mercado. *Instagram* contra *Snapchat*, *WhatsApp* contra *Telegram*, esses são alguns dos mais variados casos em que *softwares* funcionais podem ter sua taxa de utilização variada pela percepção de seus usuários, que modificam os contextos nos quais estão inseridos.

2.1.8 Sistema de *Feedback*

Uma aplicação é criada com um propósito de atender uma determinada finalidade, seguindo características especificadas pelas leis anteriores. Uma forma de se determinar futuras ações no *software* e em seu processo, é coletar o *feedback* gerado mediante seu uso. *Feedback* relaciona-se a um conjunto extenso de variáveis, como utilização dos usuários,

informações de ambiente, estatísticas de uso, taxa de erros, tempo de resposta, ou seja, métricas que possam direcionar a evolução, não só do produto, como de seu ciclo de desenvolvimento.

Para Lehman, essa lei define por: “Os Processos de Programação do tipo E constituem sistemas de *feedback* Multi-loop e Multi-nível e devem ser tratados como tal para serem modificados ou melhorados com sucesso”(LEHMAN, 1996).

O próprio *Github* CI/CD, como demonstrado na figura 6, oferece um ambiente que fornece *feedback*, com informações sobre as ações realizadas no repositório, compilação de métricas relacionadas aos *commits*, contribuidores, adições, deleções, entre outras, além de permitir o próprio engajamento da comunidade.

/

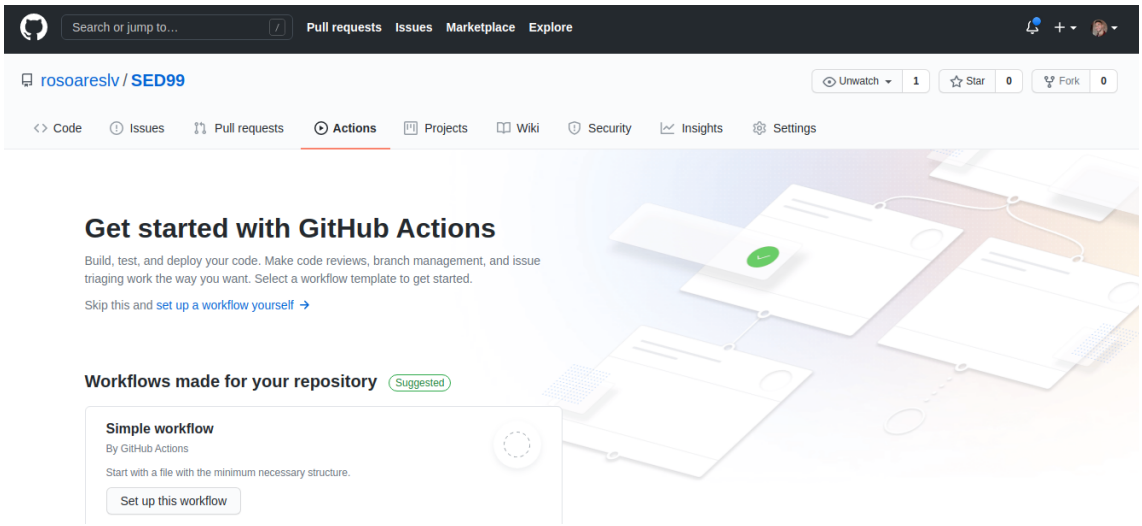

Figura 6 – Ferramenta de CI/CD oferecida pelo *Github*

2.2 *Open Source*

Software de código aberto é um conceito que remete desde o advento da computação, com o início da utilização dos computadores pelas universidades com fins de pesquisas e a sua introdução no contexto de negócios empresariais, o que resultou na capitalização do *software*, componente intangível que começou a representar ganhos monetários. Para contrariar essa tendência, Richard Stallman desde 1984 fomentou a ideia de código aberto, mesmo ano em que formou a "Free Software Foundation

e o *"GNU Project"*, com a premissa de oferecer liberdade, e as pessoas de fato aproveitá-la, considerando o contexto de *softwares*(PERENS et al., 1999).

Softwares de código aberto representam uma nova forma de se criar aplicações, confundindo os limites da contribuição pública e privada (KOGUT; METIU, 2001). Em uma comparação direta, empresas possuem ferramentas que atuam em seus negócios e oferecem vantagens competitivas que não podem ser disponibilizadas publicamente. Os

avanços resultantes de pesquisas científicas possuem forte tendência e apoio para a dispersão livre de resultados. No mundo *Open Source*, percebe-se uma convergência entre estruturas hierárquicas, controles e governança com a produção de código fonte aberto. Muitos projetos *Open Source*, como o *Linux*, foram patrocinados por empresas privadas que captaram capital intelectual envolvido com a perspectiva científica.

Existem inúmeros projetos que possuem seu código fonte publicado de forma aberta na internet com diferentes sistemas de armazenamento e versionamento. O movimento *Open Source* ganha cada vez mais força, pois movimenta uma comunidade ativa que auxilia no ciclo de vida das aplicações. Percebe-se uma tendência sociológica quando se discute sobre *Open Source*, pois se trata de conhecimento aplicável construído com a contribuição de vários indivíduos. Além da polarização dos modelos de negócio para a inserção do contexto de acesso público em suas realidades, cita-se a estratégia clássica de disponibilizar uma aplicação de forma gratuita e apoiada pela comunidade, e outra restrita voltada a um público alvo específico que necessita de um suporte especializado.

2.3 Repositórios

Os repositórios escolhidos para o *dataset* utilizam o sistema de controle e versionamento *Git*, lançado em 2005 por Linus Torvalds, um dos mais recentes e amplamente utilizados pelos desenvolvedores nos projetos de *software*, Open Sources ou não.

A tecnologia *Git* mostra-se muito à frente dos seus concorrentes como o *CVS* e o *SVN* para possibilitar a colaboração múltipla em um mesmo projeto. Com sua excelente performance, possibilidade de trabalho remoto com uma cópia local, compressão de objetos, entre outros destaques, colocam o *Git* na frente dos seus concorrentes como opção ao versionamento(OTTE, 2009).

Para a confecção do *dataset*, o *Git* foi escolhido por, além desses motivos acima citados, oferecer funções que permitem a manipulação de uma grande massa de dados através de ordem temporal, cujas suas diferentes versões encontram-se distribuídas de acordo com os *commits* realizados. *Commit* pode-se entender como um conjunto de alterações (inclusões e deleções) realizado nos arquivos de um repositório, que será monitorado e registrado em uma linha do tempo que permitirá o controle e, consequentemente, o versionamento daquele projeto de *software*. Em uma comparação educativa, funciona como uma foto retirada de um momento que está em fluxo contínuo de modificações. Essa foto sendo arquivada em um álbum de memória, poderá ser sempre acessada quando requisitada for.

Aproveita-se para demonstrar a relação estreita entre *Git* e a oitava lei de Lehman a respeito dos agentes de *feedback*, pois o *Github* é uma ferramenta que disponibiliza indicadores e oferece suporte à criação de novas perspectivas, como o exemplo claro das ferramentas de hospedagem de códigos fonte que se beneficiam da tecnologia *Git*, como

o *Github*, plataforma escolhida para a extração dos códigos fonte desse *dataset*, *Gitlab*, *Bitbucket*, entre outras.

O *Github* representa um avanço da tecnologia *Git* para a experiência de contribuição em projetos, expandindo mais as fronteiras que esse sistema de versionamento pode alcançar. Com o passar do tempo, o *Github* evoluiu e representa uma rede social voltada aos desenvolvedores e simpatizantes, oferecendo um ambiente de colaboração e compartilhamento que agrega conhecimento e provoca a curiosidade.

Como resultado, as pessoas envolvidas nessa plataforma aprendem e compartilham conhecimentos, promovem sua imagem, recebem e fornecem *feedbacks* na comunidade, ficam em contato mais próximo com os usuários das plataformas nas quais seus códigos fonte estão hospedados no *Github*, além do *networking* que a própria rede impulsiona(DABBISH et al., 2012).

2.4 Métricas

O *software* utilizado para a extração das métricas foi o *Lizard*, um analisador de código que trabalha com diversas linguagens, que oferece informações relevantes, sendo as duas métricas a seguir extraídas e incluídas no *dataset*.

2.4.1 NLOC

O *NLOC* oferecido pelo *Lizard* é responsável pela contagem das linhas de código, com exclusão de comentários, do diretório indicado para leitura, com o somatório final como resultado.

A métrica pode muito bem abraçar todo o conceito e importância que o *LOC* oferece, indicando uma forma de volumetria do *software* que já permite observações iniciais, mesmo que não tão precisas.

Adições, alterações, deleções são capturas pela métrica *LOC*, que pode sofrer particionamento de acordo com o tempo considerado e que combinada com outras métricas oferece uma visão precisa de um momento do *software* e indica características em relação à codificação, modelo de negócio, tamanho, entre outras características.

O *LOC*, por mais simples que pareça, já é um bom indicativo para a área da Evolução de *Software*, ainda mais se combinada com outras métricas, como o trabalho de Israel e Gregorio (2006), que analisa o comportamento das linhas de código na evolução de projetos de *software*, buscando a captação de comportamentos padronizados com combinação desta métrica, mais a quantidade de arquivos presentes, com a utilização de equações quadráticas e posterior categorização dos resultados obtidos.

2.4.2 Complexidade Ciclomática

A Complexidade ciclomática é umas das informações disponíveis no CSV gerado junto ao *dataset*. Representa um auxílio inicial na navegação dos diretórios criados, pois seus números revelam diferentes níveis de atividade, que permitem a correlação direta com as leis de Lehman previamente visitadas.

O conceito desta métrica é inerente ao conceito de algoritmo. Um algoritmo, educativamente, é similar a uma sequência de passos que no caminho possui decisões influenciadoras no resultado final. Esta métrica, criada por Thomas J. McCabe em 1976, trabalha com a transformação das estruturas de decisão em grafos, logo depois ocorre uma contagem dos possíveis caminhos de execução independentes (MADI; ZEIN; KADRY, 2013). A fórmula 2.1 da complexidade ciclomática é formada por:

$$CC = e - n + 2 \tag{2.1}$$

Onde:

- "CC" é o resultado, a Complexidade Ciclomática propriamente dita,
- "e" é o total de percursos entre os nós
- "n" representa as condicionais/expressões existentes no código

De acordo com Madi, Zein e Kadry (2013), a complexidade ciclomática leva em consideração as seguintes estruturas:

- Contagem das estruturas *if/else*
- Contagem das estruturas *try/catch*
- Contagens dos *loops* no programa
- Contagens de estruturas de seleção/*switch* e os possíveis desvios de fluxo baseado nos cases, sem a seleção do desvio *default*

Uma representação visual da complexidade ciclomática está na tradução de código fonte em um grafo, previamente citado, da seguinte forma:

Node	Statement
(1)	`while(x<100){`
(2)	`  if (a[x] % 2 == 0) {`
(3)	`      parity =  0;`
	`    }`
	`    else {`
(4)	`      parity =  1;`
(5)	`    }`
(6)	`    switch(parity){`
	`      case 0:`
(7)	`        println( "a[" + i + "] is even");`
	`      case 1:`
(8)	`        println( "a[" + i + "] is odd");`
	`      default:`
(9)	`        println( "Unexpected error");`
	`     }`
(10)	`    x++;`
	`  }`
(11)	`p = true;`

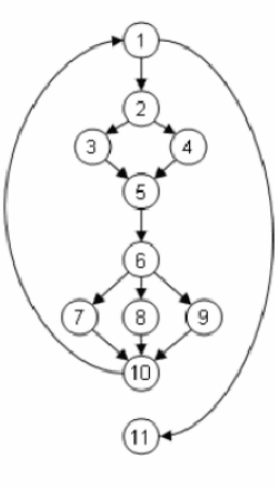

Figura 7 – Exemplo prático da teoria da Complexidade Ciclomática

3 Trabalhos Relacionados

3.1 *Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution?*

O trabalho de Skoulis, Vassiliadis e Zarras (2014) tem uma relevância grande para este trabalho, pois coloca em questão as Leis de Lehman em projetos de software *Open Source*, utilizando seus bancos de dados publicados por outros pesquisadores.

Percebe-se uma análise, sempre referindo-se às leis, com marcações visuais, como demonstra a figura 8, dos comportamentos obtidos e observações pontuais em comparação com as definições das leis, provando sua aplicabilidade ou questionando-a. É perceptível uma aplicação similar deste estudo ao dataset SED99, que oferece condições ideais para a manipulação de gráficos com um tempo de vida dos projetos considerável e múltiplas interpretações a favor ou contra as leis.

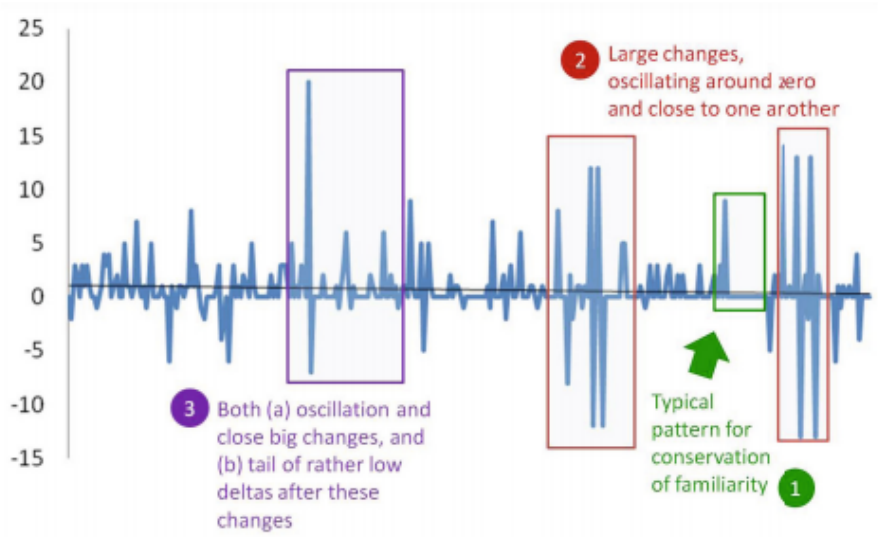

Figura 8 – Análise Gráfica das Leis de Lehman (SKOULIS; VASSILIADIS; ZARRAS, 2014).

Este gráfico por exemplo, tange à lei da Conservação de Familiaridade, explicada na Seção 2.1.5, que analisa os padrões de grandes modificações, seguidos por menores e tenta capturar momentos de estabilização para a comprovação da lei, que não consegue ser provada de forma concreta, não sendo confirmada, mas possível.

São feitas conclusões a respeito do *dataset* utilizado, das leis que puderam ser avaliadas, trabalhos futuros relacionados à expansão da pesquisa a bancos não relacionais e proprietários, problemas encontrados com o uso dos *schemas* e as métricas utilizadas. É

interessante reforçar a importância deste trabalho no que tange as possibilidades de uso do SED99.

3.2 Acompanhamento da Evolução de Software via Métricas

O trabalho de (RIBEIRO et al., 2012) traz um foco importante nas métricas utilizadas na avaliação de um projeto, considerando a linguagem *Java*, gerenciado pelo *Maven* e com o sistema de controle e versionamento *Subversion*. O sistema escolhido foi o Publico Core, um módulo do sistema iDUFF, responsável pela gestão acadêmica, auxiliando no acesso a dados oriundos de outras fontes, como informações de alunos e funcionários.

Algumas métricas exploradas, como o *TCC (Total Cyclomatic Complexity)* e o *LOC (Lines of Code)* foram exploradas na construção deste *datasets*, para fornecer suporte inicial aos futuros pesquisadores na navegação entre os diretórios presentes e auxiliar na própria pesquisa. Outras foram citadas, como o *LCOM (Lack of Cohesion of Methods)*, que indica o quanto os métodos de uma classe se relacionam, o *DAM (Data Access Metric)*, responsável por indicar a razão entre atributos privados e o total de atributos, o *CAM (Cohesion Mong Methods in Class)*, que indica a coesão entre os métodos de uma classe, e várias outras métricas nas quais softwares como o *Locc*, *JavaNCSS*, *JDepend*, *Dependecy Finder*, entre outros, foram responsáveis por extrair.

O trabalho foca na compilação de gráficos e exploração de métricas do projeto de software citado anteriormente, com a adição de comentários a respeito da análise dos materiais gerados e correlação com as métricas analisadas. Um exemplo seria a redução das linhas totais de código (LOC) induzir a uma diminuição na taxa de complexidade (TCC).

3.2.1 *Android Apps and User Feedback: A Dataset for Software Evolution and Quality Improvement*

O artigo (GRANO et al., 2017) representa uma alternativa ao estudo da Evolução de *Software* através da compilação de um *dataset* com foco nos *feedbacks* de usuários em 395 aplicativos da *Playstore* que também estavam disponíveis no *F-Droid*, um catálogo de aplicativos *Android Open Source*. Foram coletadas 288.065 revisões, relacionadas de acordo com a versão do aplicativo considerado e servindo de base para a aplicação de 22 métricas de qualidade de código.

Ocorreu um processo de peneiração dos aplicativos na plataforma do *F-Droid* para escolha, considerando as aplicações ativas e captura dos comentários dos usuários de acordo com o software da *Playstore*. Foram utilizados softwares como o ARDOC para

classificar as revisões extraídas e a categorização dos aplicativos com sua área de atuação, quantidade de versões lançadas e a quantidade de revisões.

Houve o levantamento numérico de informações extraídas dos aplicativos e a percepção da relação entre a interação externa dos usuários com a estrutura interna dos *softwares*, evidenciando uma relação que impacta no grau de satisfação, na qualidade do projeto, auxilia no estudo da tendência de evolução, entre outras observações que o estudo procurou clarificar.

3.2.2 *COMETS: A Dataset for Empirical Research on Software Evolution using Source Code Metrics and Time Series Analysis*

O artigo (COUTO et al., 2013) contempla a descrição do *dataset COMETS (Code Metrics Time Series)*, que tem por objetivo estudar a evolução de 17 *softwares* baseados em *Java*, em intervalos de 14 dias com a inclusão dos códigos fontes dos programas de acordo com a versão considerada, baseado no intervalo temporal determinado. Foi utilizada a plataforma *Moose* para a extração das métricas.

O objetivo maior foi prover um *dataset* para continuação de estudos na área, objetivo muito semelhante ao deste trabalho, com perspectiva de evolução do *dataset*, que como trabalho futuro cita a inclusão do número de defeitos nas classes.

Foram gerados alguns gráficos com a comparação entre métricas de um projeto escolhido, compartilhadas informações empíricas sobre as versões encontradas, janela de tempo estudado entre cada software individualmente, além da inclusão de arquivos csv para auxiliar a utilização do *dataset*, estratégia semelhante adotada neste trabalho.

4 Ferramentas e considerações

4.0.1 Introdução

Nesta seção será inciada a descrição do processo de criação do *dataset* que auxiliará novos estudos na área da Evolução de *Software*, possibilitando novas abordagens. As três linguagens escolhidas foram o *C++*, *Java* e *Python*. Essas três linguagens de programação encontram-se amadurecidas e popularizadas entre a comunidade de desenvolvedores e, consequentemente, protagonizam a construção de *softwares* livres / Open Source importantes, o que viabilizou a criação do SED99 *(Software Evolution Dataset - 99 repositories)*, que é um *dataset* que contempla 99 repositórios divididos igualmente entre essas linguagens de características, comportamentos e grandezas variadas, oferecendo um leque de possibilidades a futuros trabalhos.

Os projetos escolhidos traduzem-se em soluções utilizadas, por pequenas e grandes corporações, evidenciando a força que projetos de *software* livre / Open Source possuem e sua capacidade de adequação à realidades corporativas, seja na sua codificação, ou na forma de distribuição, suporte, entre outros.

4.0.2 Critérios de seleção

Os critérios de seleção de projetos procuraram compreender a realidade do momento, abrangendo *softwares* com utilizações, citações e atividades variadas para criar uma coletânea diversificada, que permita inferir perspectivas, hipóteses e análises diversificadas no estudo de suas evoluções, mas de forma estruturada para permitir bases empíricas de resultados.

Para a escolha dos projetos de código aberto e/ou livres que constituem o SED99, foram considerados repositórios que cumpriram com os seguintes requisitos:

- Sistema de controle e versionamento *Git*, para viabilizar a manipulação de grandes massas de arquivos com funções que o *Git* CLI oferece.
- Mínimo de 5 anos de existência, confirmando a consolidação de um projeto na comunidade
- Mínimo de 1000 *commits* totais realizados, garantindo uma sequência de alterações nos arquivos do repositório, para auxiliar na captura de diversas versões dos arquivos que realmente possuíssem modificações consideráveis.
- Mínimo de 1 *commit* em Outubro/2020 para verificar existência de atividade recente e não abandono do projeto.

- Cada repositório possui no mínimo 50% do seu código fonte de acordo com a linguagem de programação na qual foi classificada, para que a extração de arquivos fonte conseguisse capturar quantidades significativas de arquivos para compor o *dataset* e permitir o estudo histórico da evolução de software.

4.0.3 Repositórios selecionados

Consequentemente, os repositórios escolhidos foram:

C++	**Java**	**Python**
caffe	dagger	ansible
Catch2	dbeaver	aws-cli
Chaste	dubbo	bottle
cJSON	elasticsearch	celery
ClickHouse	ExoPlayer	compose
cm-compiler	fastjson	cookiecutter
cppcheck	flink	core
cryptopp	glide	django
dash	guava	erpnext
electron	hadoop	falcon
fmt	hazelcast	flask
folly	hibernate-orm	gensim
gdal	j2objc	httpie
godot	jadx	kivy
Halide	jdk	librosa
imgui	jenkins	luigi
json	lucene-solr	Mailpile
JUCE	mockito	matplotlib
kakoune	MPAndroidChart	mopidy
libtorrent	mybatis-3	nilearn
Marlin	neo4j	numpy
mongo	netty	pandas
mysql-server	openapi-generator	requests
opencv	OpenRefine	scikit-learn
pcl	picasso	scrapy
rdkit	realm-java	sentry
rocksdb	retrofit	spaCy
seastar	RxJava	statsmodels
solidity	scribejava	sympy
tesseract	sonarqube	tornado
TileDB	spring-boot	you-get
TrinityCore	spring-framework	youtube-dl
xbmc	zxing	zulip

Figura 9 – Tabela com os repositórios escolhidos por linguagem

5 SED99

5.1 Processo de Criação

Nesta parte do trabalho será descrito o processo de criação, com detalhes da lógica e ferramentas utilizadas para a execução. Eventualmente, irão surgir trechos de código fonte para clarificar o entendimento, principalmente para os indivíduos da área da Engenharia de *Software* e afins. Esses trechos são parciais, devido à natureza da montagem do SED99 ser híbrida, e os repositórios não seguirem o mesmo padrão, como *branches*, estruturas e a própria atividade dos repositórios serem diferentes, o que acarreta manual intervenção, além de partes do processo não serem possíveis/inviáveis à automatização.

Para criar o SED99, um processo simples e robusto foi desenhado para performar a manipulação dos arquivos no sistema Operacional, dividido em 6 etapas no que concebe ao início da confecção, com os repositórios previamente selecionados, até sua publicação no *Github*, refletindo o decorrer das seções 5.1, 5.3 e 5.4.

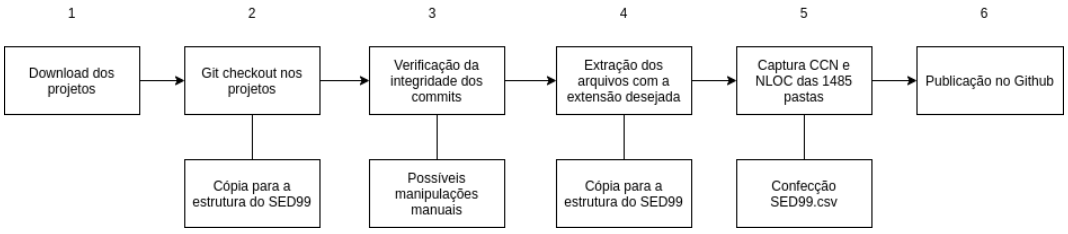

Figura 10 – Processo de criação do SED99 dividido em etapas.

A primeira parte do processo foi o *download* dos repositórios escolhidos, que foi automatizado utilizando a linguagem *Shell* para percorrer uma lista previamente carregada com os links de acesso dos repositórios previamente demonstrados, de acordo com o trecho do código fonte apresentado abaixo.

```
1  for i in 1 2 3; do
2      if [ $i -eq 1 ]
3      then
4          cd /<caminho no computador>/repositorios/c++
5          file=/<caminho no computador>/listaCpp.txt
6      fi
7      if [ $i -eq 2 ]
8      then
9          cd /<caminho no computador>/repositorios/java
```

```
10          file=/<caminho no computador>/listaJava.txt
11      fi
12      if [ $i -eq 3 ]
13      then
14          cd /<caminho no computador>/repositorios/python
15          file=/<caminho no computador>/listaPython.txt
16      fi
17      while read -r line; do
18          git clone $line
19      done < $file
20  done
```

Após ocorrer o download, foi montado outro *script* em *Shell* responsável pela extração dos códigos fonte, considerando 1º de Janeiro de 2015 à 31 de Dezembro de 2019, sempre em intervalos de 4 meses, o que oferece um *snapshot* de 3 momentos distintos dos arquivos fonte presentes no repositório por ano, 15 no total se considerar apenas um *software* da lista utilizada e os 5 anos de estudo.

Depois de possuir os repositórios localmente, foi utilizado o comando *git pull*, passando como parâmetro as datas para se capturar o *hash* do último *commit* dentro do intervalo especificado. A regra geral utilizada, independente dos anos, foi de pegar sempre o último dia do mês correspondente ao período de 4 meses, então as datas utilizadas foram montadas e utilizadas no padrão:

- 04/30/<ano>
- 08/31/<ano>
- 12/31/<ano>

Para retornar versões antigas dos arquivos em ordem cronológica, foi utilizado o comando *git checkout*, que permite retroceder as versões dos arquivos baseado no *commit* escolhido, sendo o *commit* o resultado da operação anterior. A montagem desta lógica foi executada no próximo trecho de código.

Esse é um dos pontos mais importantes, pois essa foi a forma utilizada para o controle em ordem cronológica dos *commits*, com uma lógica que se aproveita do comando *git log*, aplicado a um tratamento em *Shell* para captura do *Hash* do *commit*, servindo de *input* para o comando *git checkout*, responsável por retroceder o repositório a um ponto temporal específico.

```
1       year=<ano>
2       cd <repositório>
```

```
3       commit=`git log --max-count=1 --until=04/30/2019
4       | head -1 | cut -d" " -f2`
5       #Este comando gera como output apenas o número do hash do commit
6       git checkout $commit
7       #Modifica-se os arquivos para o commit escolhido
8       cd ..
9       cp -r ./<repositório> /<caminho do computador>/repositorios_intervalados
10      /<linguagem de programação>/<repositorio>/<ano>/<período>
11      #realiza-se a cópia do repositório para uma estrutura adequada
12      cd <repositório>
13      git checkout <branch padrão>
14      # volta para a branch default do projeto
15      cd ..
```

Esse trecho de código foi executado em *loop*, por repositório, sendo as variáveis modificadas para o contexto necessário.

A estrutura adequada mencionada nos comentários se assemelha muito à estrutura final do SED99. Para facilitar a compreensão, tomando como base um caminho único percorrido, mas que é uniforme em todos os níveis apresentados, ela pode ser representada pela figura 11:

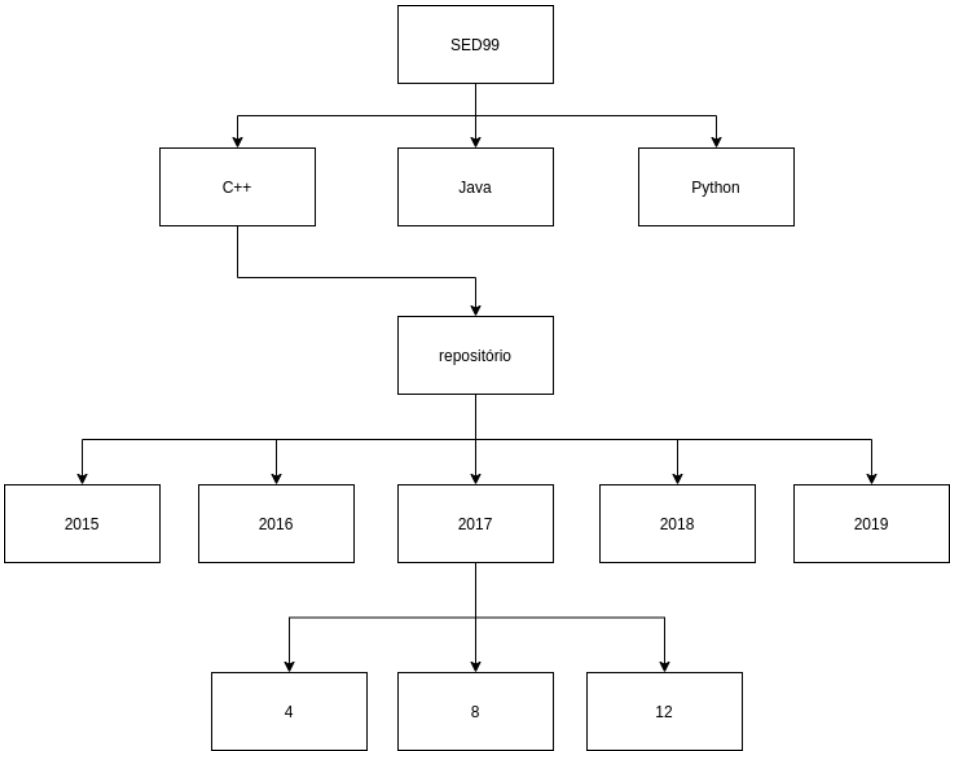

Figura 11 – Estrutura de diretórios

O *dataset* contém inicialmente 3 pastas, identificando a linguagem, e em cada pasta contém 33 repositórios escolhidos, que possuem uma divisão seguida pelos anos e logo depois o período, sendo o identificador “4” os primeiros 4 meses, “8” o segundo período de 4 meses e “12” o último período quadrimestral.

Essa estrutura também é utilizada na versão final do *dataset*, que conterá apenas os códigos fonte dos arquivos de acordo com a linguagem escolhida. Neste momento, os repositórios como um todo estão presentes, incluindo arquivos indesejados/irrelevantes, como o ".git", para o intuito do *dataset*.

Logo após fazer a separação, foi aplicado uma peneiração dos arquivos, com uma busca automatizada dentro das pastas de cada repositório, procurando arquivos com as extensões escolhidas para contemplar o *dataset*, sendo elas:

C++	Java	Python
“*.cpp”, “*.h”, “*.cc”	“*.java”	“*.py”’

A peneiração foi realizada com o percorrimento dos repositórios localmente, e utilizando comandos em *Shell* que serão demonstrados a seguir, com comentários, sendo o resultado a extração dos arquivos de acordo com as extensões demonstradas, em uma estrutura de diretórios que é uma cópia da apresentada na figura acima, mas agora só com os arquivos relevantes que irão auxiliar na propagação de novos estudos.

```
1       cd <caminho no computador>/repositorios\ intervalados/<linguagem>/
2       # Acessar a pasta com os repositórios
3       year=<ano>
4       cd <repositorio>/<ano>/<periodo/<repositorio>
5       #Acessar o destino final dos arquivos
6       find . -name '*.h' -exec cp {}
7       /<caminho no computador>/SED99/<linguagem>/<repositorio/<ano>/<periodo> \;
8       #Realizar a captura dos arquivos com a
9       #extensão utilizada (nesse caso o .h) e
10      #redirecionar para a pasta desejada.
```

Este foi o fluxo realizado para a compilação do SED99. O resultado final, como é de conhecimento, é um *dataset*, que contempla três linguagens, com um total de 99 repositórios distribuídos igualmente entre essas linguagens, sendo seus tamanhos e comportamentos variados de acordo com seu histórico de modificações.

5.2 Informações Úteis

A seguir, serão reunidas informações úteis sobre o *dataset* e o ambiente de criação.

1. *Dataset*: SED99 *(Software Evolution Dataset 99 repositories)*
2. Distribuição: *Github*
3. Licença: *MIT*
4. Linguagens de programação contempladas: *C++, Java* e *Python*
5. Tamanho total : 28 giga (incluso também arquivos de configuração *Git*, arquivo "readme.md" e o "SED99.csv")
6. Sistema Operacional de criação: *Linux Mint 20 Ulyana*
7. Processador: *Intel Core* i5 4200U 4ª geração
8. Memória *RAM*: 8 giga
9. Tamanho SED99:

	C++	Java	Python
Giga	9.6	15.0	2.7
LOC	96244823	86033112	29611227

Figura 12 – Linguagens, seu espaço ocupado e *LOC* total

5.3 Validações e Verificações

As validações e verificações ocorreram de forma manual, com o objetivo de garantir a integridade das etapas mencionadas anteriormente e correto resultado final.

A busca pelos repositórios foi realizada em uma página específica do *Github*, a *Github Trendings*, responsável por catalogar repositórios que de acordo com o algoritmo proprietário, que destaca repositórios que possuíram um maior nível de procura, baseado no número de "estrelas"ganhas em um período determinado e regulável pela interface.

Para a escolha dos repositórios foi realizada uma consulta manual à sua respectiva página no *Github* e foi verificado na seção de "*Insights*", e logo adiante na opção "*Contributors*", o tempo de existência do repositório, para que fosse possível capturar 5 anos de atividade, para condução da criação do *dataset*.

Em seguida, foi verificado na página principal do repositório o último *commit* realizado, seguindo a regra de o mais recente ter sido realizado no mês de Outubro de 2020 para verificar indícios de atividade, e não abandono do projeto. Na mesma página, foi analisado a distribuição dos códigos fonte no projeto, verificando se pelo menos 50% de sua composição era de acordo com a linguagem de programação na qual foi classificado.

Logo após a distribuição dos códigos fonte na estrutura do repositório do SED99, sem a peneiração dos arquivos baseada na extensão, foi realizada uma verificação manual em arquivos de *log*, que continham o código identificador do *commit* (*hash*) e data correspondente, para verificação da integridade dos intervalos explicados anteriormente, o que requisitou intervenções manuais e especiais para alguns repositórios selecionados.

Cada repositório possui um conjunto de 15 pastas verificadas, pois cada ano de 2015 à 2019 possui 3 pastas (4,8 e 12) nas quais foram realizadas verificações, e como o *dataset* possui 99 repositórios, gerou-se 1485 pastas que foram verificadas e logo a mesma quantidade em arquivos de *log*. A seguir, o trecho do código fonte responsável pela criação dos *logs* citados, no qual foi executado em *loop* por estas pastas.

```
1       year=<ano>
2       echo <repositório>
3       cd <repositorio>/<ano>/<período>/<repositorio>
4       git log --max-count=1 >>  <caminho no computador>/log/<linguagem>
5       /<repositorio>year-4.txt
6       #Busca último commit e coloca o output em um arquivo com identificação
7       cd /<caminho no computador>/dataset/<linguagem>
```

Uma verificação realizada, e que auxiliou a composição do arquivo .csv, foi a captura da Média da Complexidade Ciclomática *(CCN)* e o número de linhas de código sem comentários *(NLOC)*, que foi feita através da aplicação *Lizard*, explicada anteriormente. A inserção dos resultados foi manual, mas a captura foi automatizada, seguindo o mesmo padrão de supervisionamento já mencionado. A seguir o trecho executado em *loop*, com lógica semelhante à captura dos *commits*.

```
1       echo 'Executando Lizard no repositório ' <repositorio>
2       cd <repositorio>
3       touch /<caminho no computador>/lizard\ relatorios/<linguagem>/
4       <repositorio>.txt
5       #criado arquivo vazio
6       lizard >> /<caminho no computador>/lizard\ relatorios/<linguagem>/
7       <repositorio>.txt;
8       #Executando a ferramenta e capturando o resultado no arquivo vazio gerado
9       cd ..
```

5.4 Disponibilização de Resultados

O SED99 está disponível no *Github*, através do seguinte *link*: <https://rosoareslv.github.io/SED99/>, que contém as pastas com as respectivas linguagens, um arquivo ".csv"com as informações das métricas previamente selecionadas nos momentos capturados de cada projeto ao longo do tempo e outros arquivos pertinentes à organização do projeto.

Para executar o *download* do projeto existem algumas formas. Para os utilizadores do sistema de versionamento *Git* e da página de hospedagem *Github*, não encontrarão maiores dificuldades. De forma simples, o *clone* do projeto através da *url* fornecida através do terminal no computador será suficiente para ganhar acesso aos arquivos.

5.5 Análise Gráfica

Nesta seção serão demonstrados gráficos pertinentes ao SED99, construídos através do arquivo CSV compilado e suas respectivas considerações.

5.5.1 Média da Complexidade Ciclomática

A média da métrica *CCN* extraída, serve para informar aos possíveis utilizadores, qual a Complexidade predominante em um repositório específico, para otimização da sua utilização, e comparar os repositórios, a fim de se enxergar se existe necessidade do uso total ou parcial do SED99.

Uma forma de se interpretar o valor obtido, aproveitando-se do conceito da métrica e segundo Ajala et al. (2016), é permitir a reflexão sobre formas de codificação, para otimizar a criação de código fonte, evitando erros, caminhos desnecessários, a partir dos grafos visuais que podem ser obtidos, e, consequentemente, uma melhora de habilidade dos indivíduos envolvidos.

Estes gráficos oferecem uma visualização mais rápida que os gráficos de distribuição apresentados na seção 5.6, que trazem no detalhe em cada repositório sua variação ao longo dos períodos. Servem como uma identificação visual de repositórios que apresentam complexidades condizentes com o objetivo do pesquisador, como metodologias de encontro de *bugs*, estilos de codificação, aspectos das linguagens estudadas que possam interferir nesses valores, entre outros.

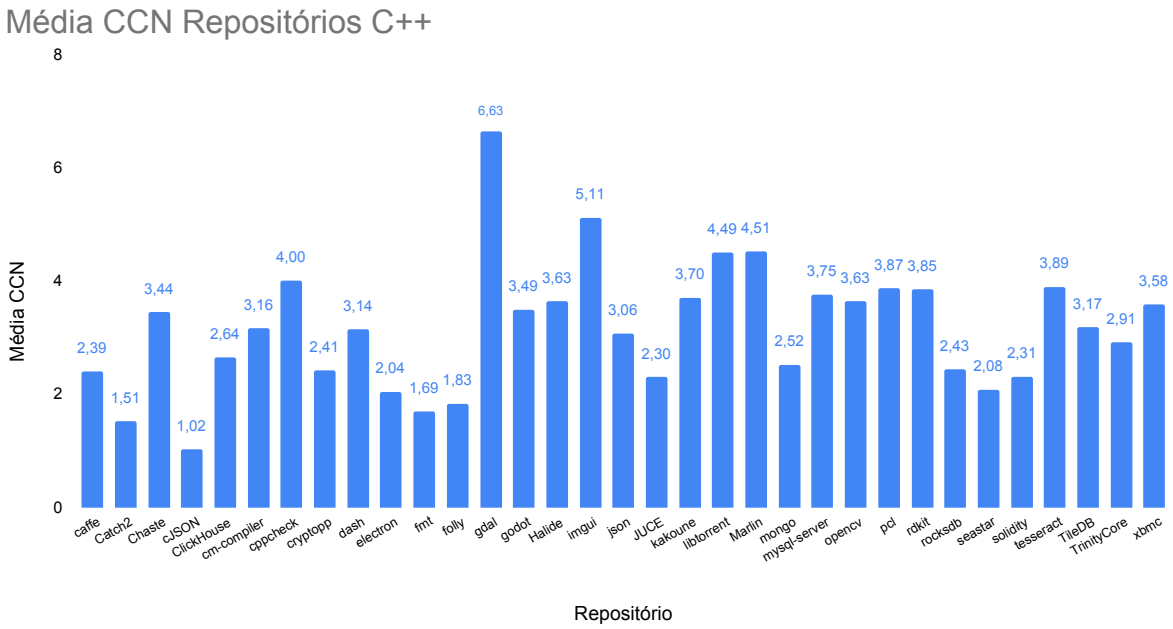

Figura 13 – Média da Complexidade dos repositórios escolhidos para compor a parte C++ do SED99

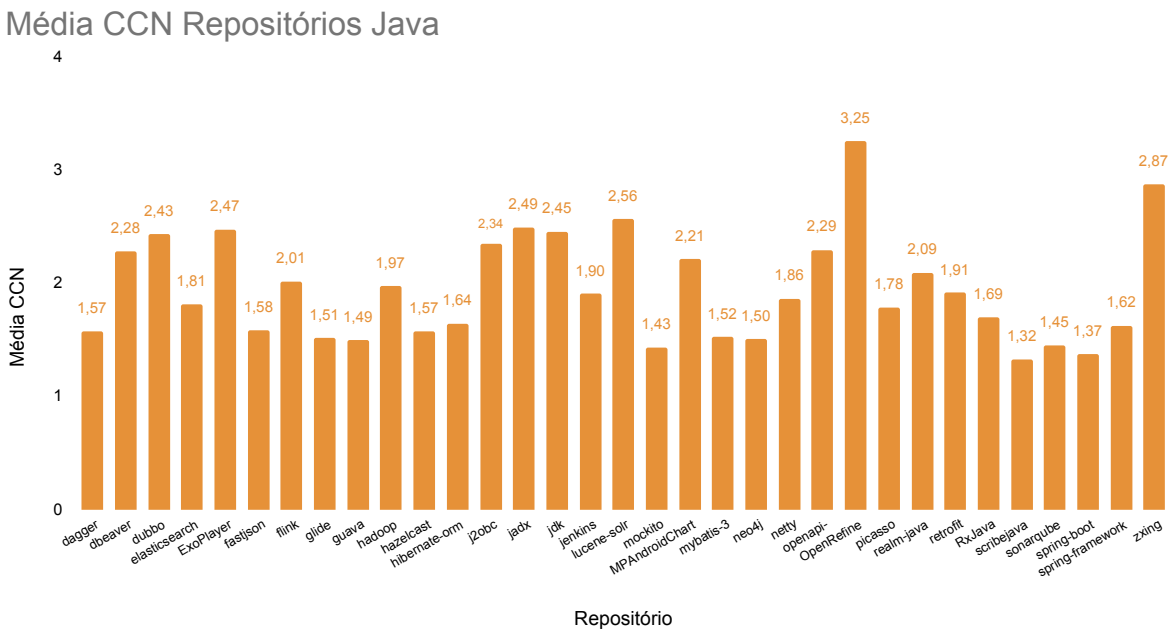

Figura 14 – Média da Complexidade dos repositórios escolhidos para compor a parte C++ do SED99

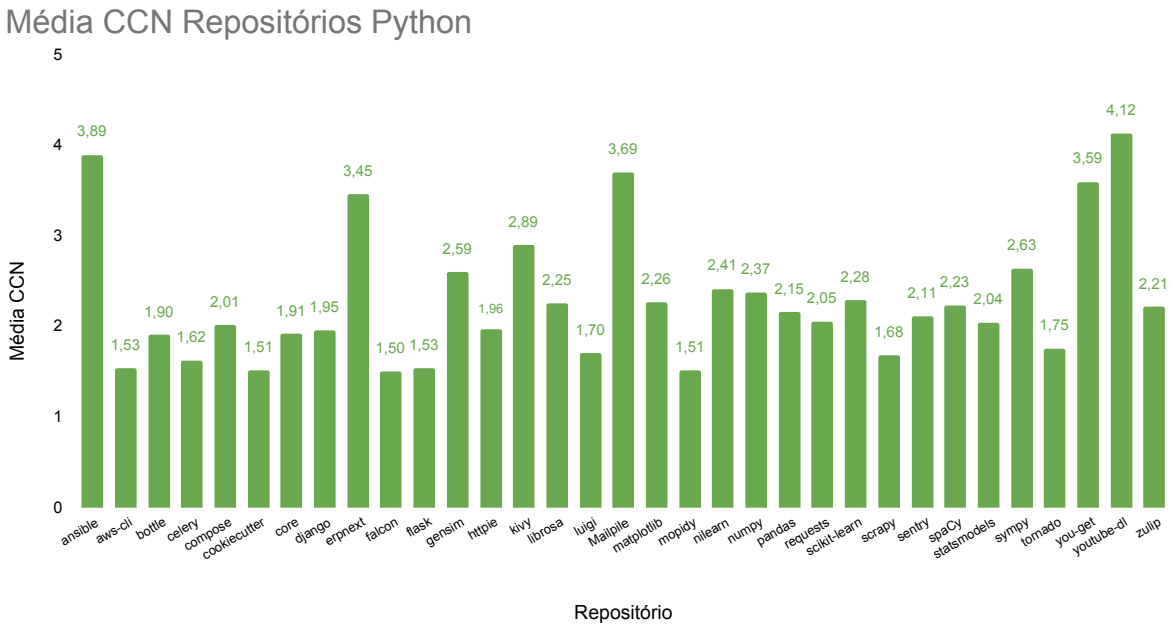

Figura 15 – Média da Complexidade dos repositórios escolhidos para compor a parte C++ do SED99

5.5.2 Média do *LOC*

O *LOC* é uma métrica bastante utilizada que oferece uma volumetria inicial interessante sobre os projetos. Existe uma preocupação a respeito dessa métrica, como enfatiza o trabalho de Gebru et al. (2018), que coloca em questão o conteúdo do código em relação ao conhecimento aplicado, a validação dos comentários como parte da contagem, pois para existir o comentário, um certo esforço foi empreendido, então por que não considerar? Independente dos pontos levantados, o *LOC* fornece uma visão inicial do comportamento de um software ao longo do período.

Para os futuros utilizadores, também oferece uma visão de magnitude, pois dependendo do objetivo, repositórios muito grandes ou pequenos não atendam as necessidades, para a aplicação de outras métricas no código fonte.

Existem projetos grandes, como o *"mysql-server"* e o *"mongo"* e projetos menores, como o *"caffe"* e *"Catch2"*, que podem ser aplicados em diferentes contextos. Para a criação de uma metodologia de encontro de *bugs*, qual repositório seria mais indicado, se o uso do SED99 fosse parcial? Para o entendimento de um padrão de código em um projeto, qual repositório viabilizaria o estudo? A diversidade do SED99 permite a aplicabilidade em diferentes níveis e contextos, em projetos reais de *software* livre / *Open Source* com foco em linguagens específicas.

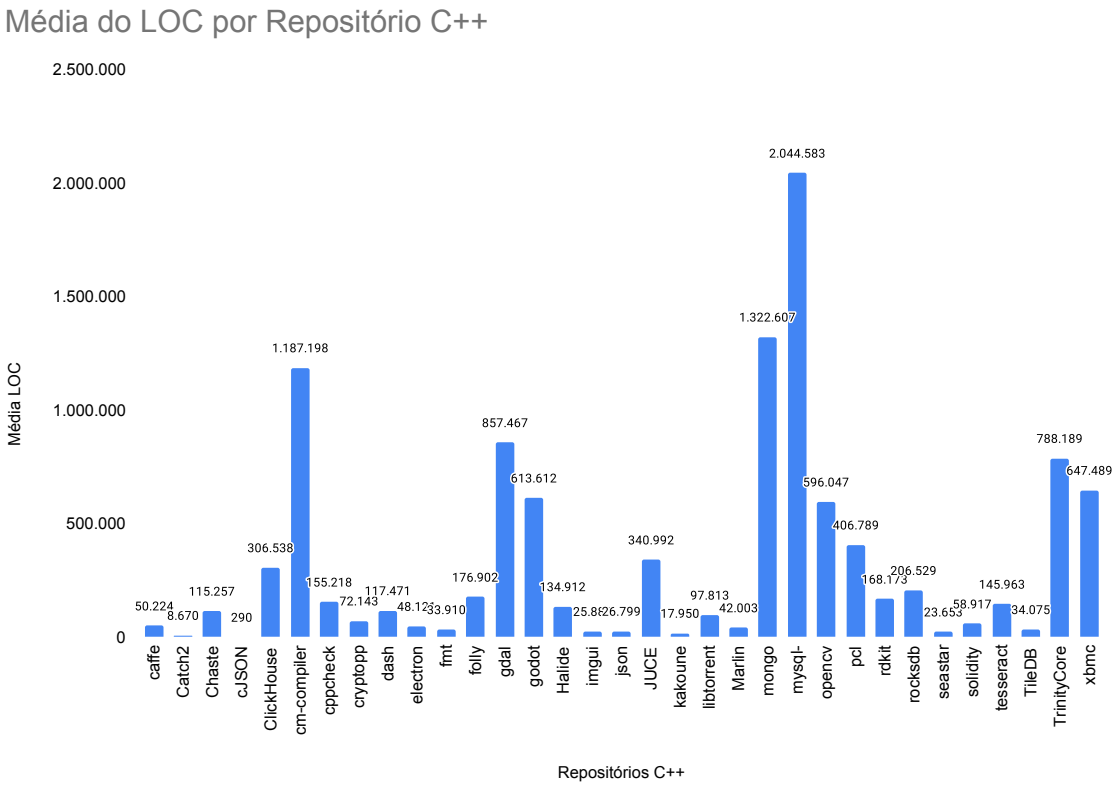

Figura 16 – Média do LOC dos respositórios escolhidos para compor a parte C++ do SED99

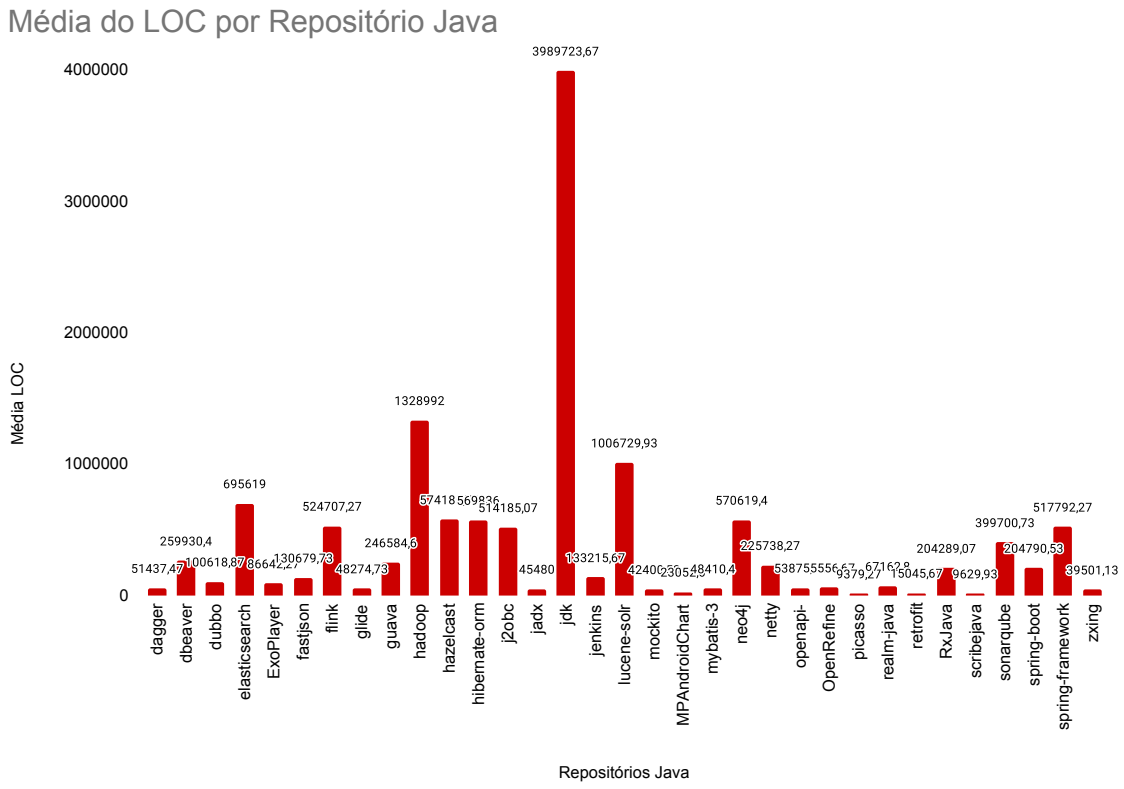

Figura 17 – Média do LOC dos respositórios escolhidos para compor a parte Java do SED99

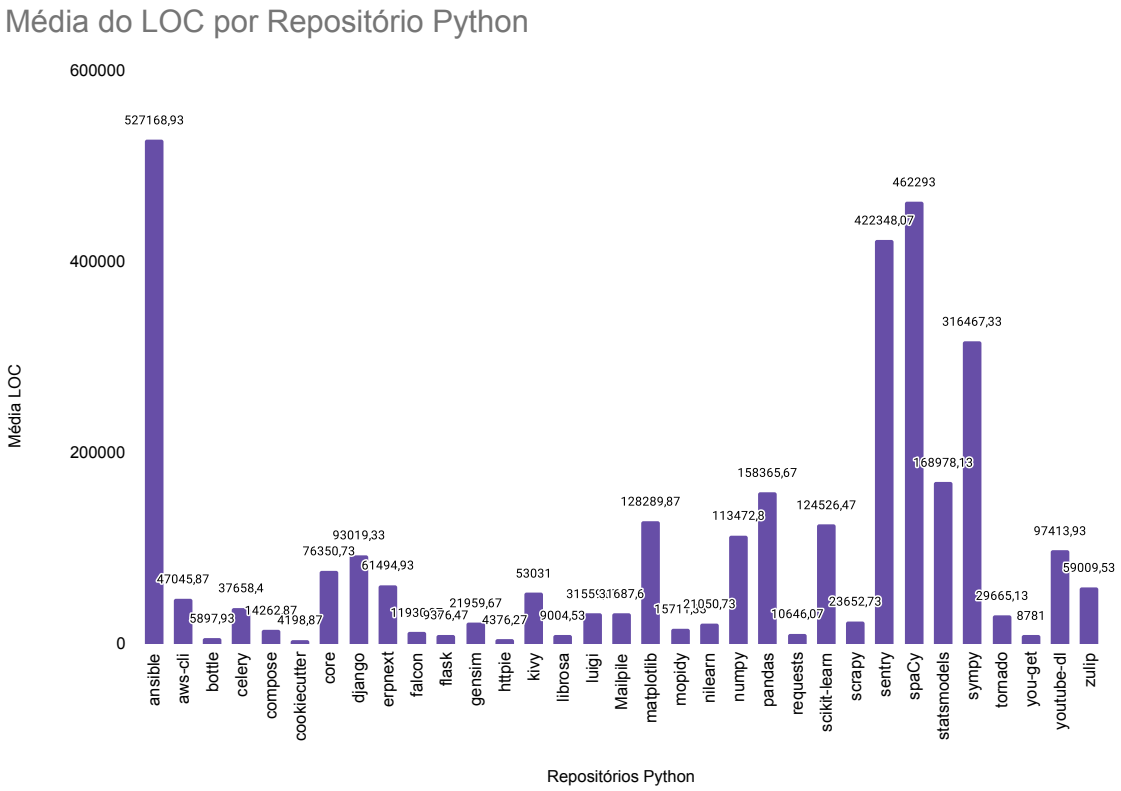

Figura 18 – Média do LOC dos respositórios escolhidos para compor a parte Python do SED99

5.6 Distribuição da Complexidade Ciclomática

A sequência de gráficos 22, 20, 21 traz com maiores detalhes as complexidades capturadas dos repositórios em cada momento criado no SED99, sendo o eixo X a complexidade e o eixo y o período, marcado pela formação "ano - pasta", sendo a pasta a representação do período quadrimestral.

Os gráficos oferecem um *overview* sobre a distribuição das complexidades dos repositórios no período, indicando estabilizações, aumentos e diminuições, que aprofundada com outras métricas e embasamento teórico, podem expressar muito sobre o *software* em si, como foi construído, decisões realizadas, entre outros.

Por exemplo, um *software* com uma linha muito uniforme, sem alterações em seu grau, pode indicar uma complexidade inerente ao próprio *software*, devido ao negócio que está inserido, pode inferir sobre a capacitação das pessoas envolvidas para lidar com o *software*, ou até mesmo indicar o nível de atividade do repositório, que pode ser baixo ou com incrementos que não procuram a refatoração. Essas são hipóteses que podem ser levantadas, mas exigem comprovação combinadas com outras métricas e perspectivas.

Por outro lado, uma linha que possui muitas variações indica movimentos da equipe na

refatoração de código, com complexidades que acabam decrescendo à medida do tempo, ou descuido, se a tendência for o aumento. Existem questões como mudança de foco, estruturas dos *branches*, engajamento da equipe que afetam diretamente os projetos, ainda mais de *software* livre / *Open Source*, que possuem estruturas mais flexíveis e estão sujeitos a interferência de várias fontes em sua estrutura.

O estudo realizado por Chen (2019) propõe a relação entre *bugs* e a Complexidade encontrada nos código utilizando como base os *datasets Defects4j* e *BugSwarm*, mostrando uma forma de utilização do SED99. O estudo concluiu que a complexidade pode ser uma das formas de se prever *bugs* para a geração de testes.

Foram discutidas algumas formas de aproveitamento das métricas extraídas, mas é necessário o reforço da utilização delas para otimização do uso do SED99 para a promoção de novos estudos.

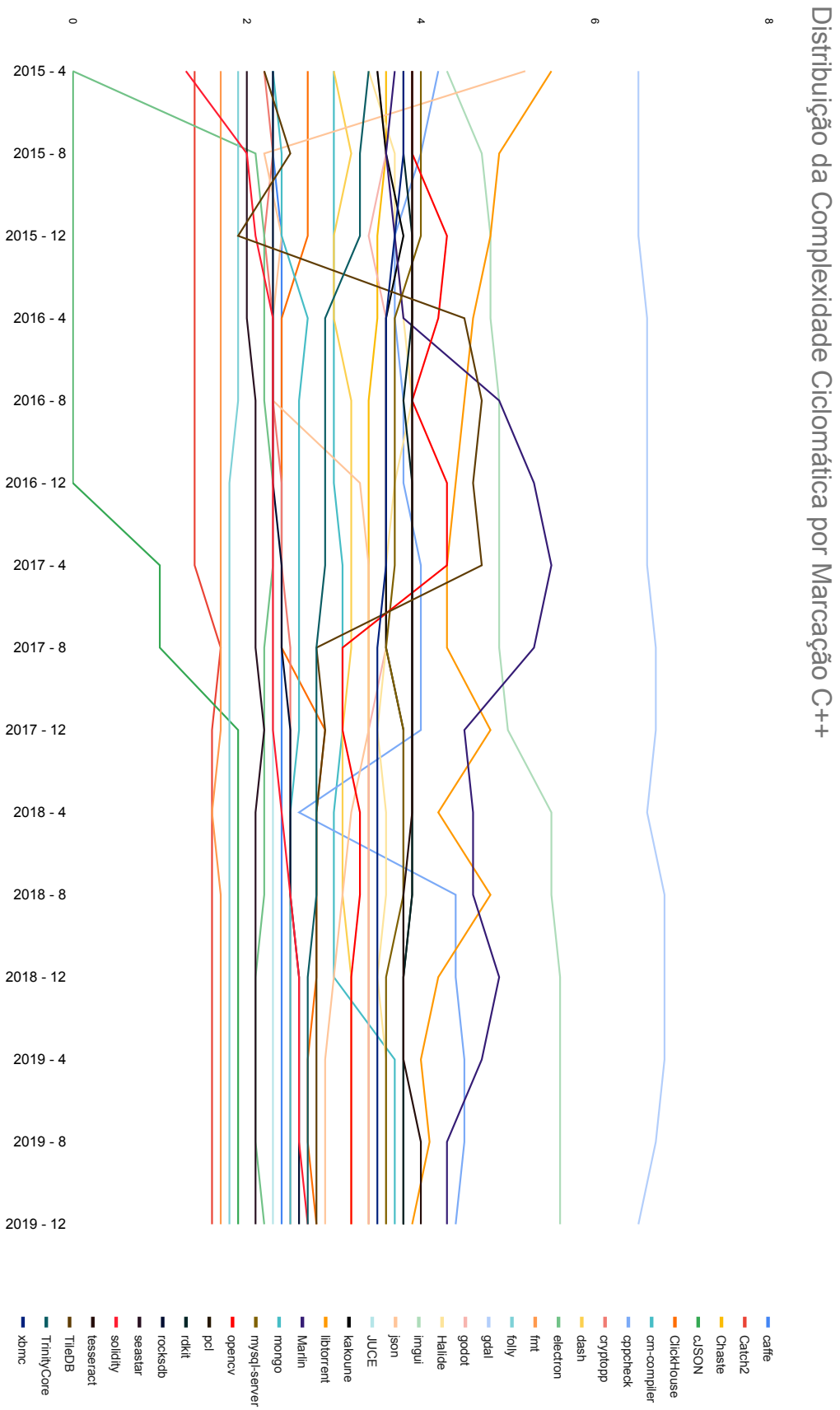

Figura 19 – Distribuição da Complexidade Ciclomática ao longo do período C++

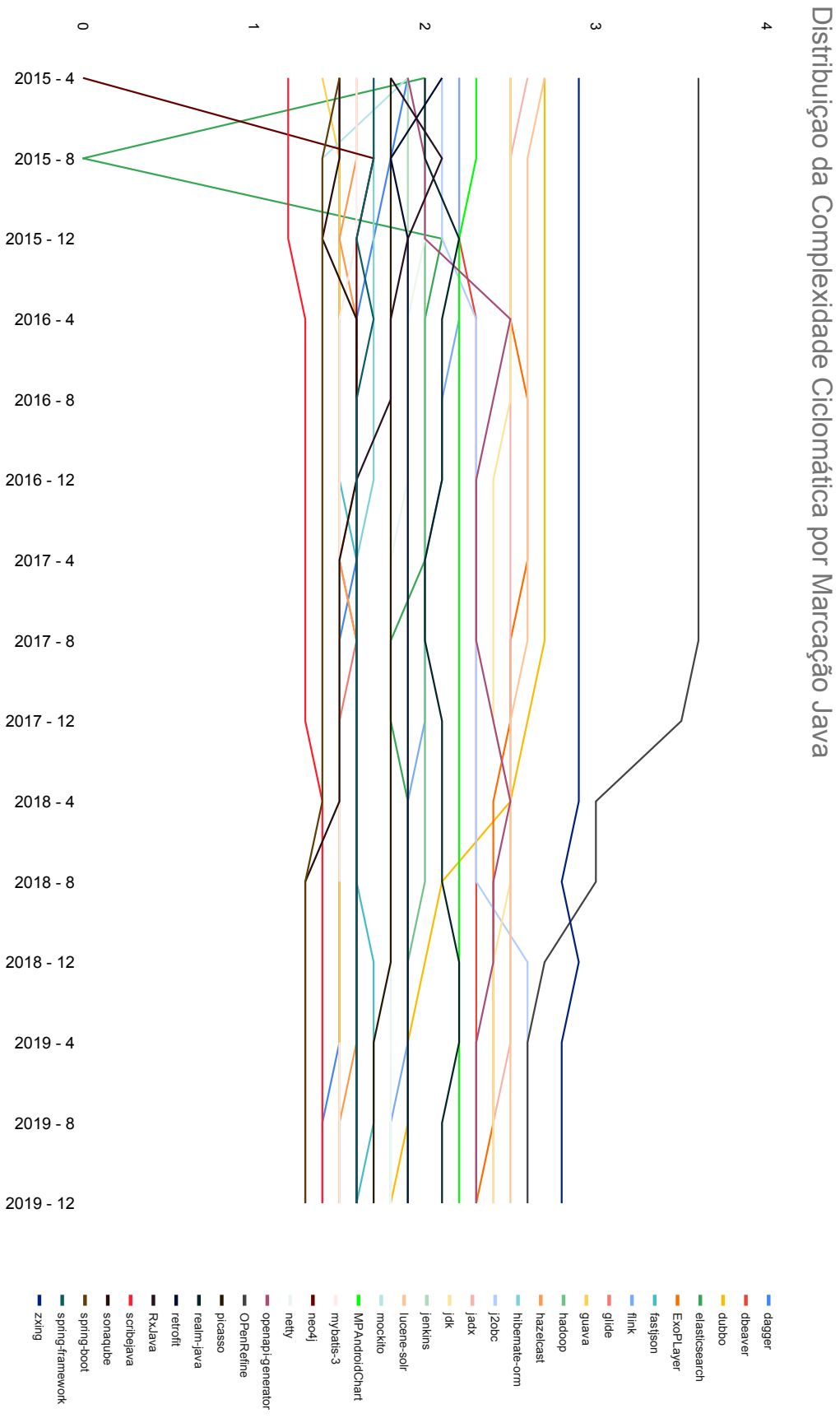

Figura 20 – Distribuição da Complexidade Ciclomática ao longo do período Java

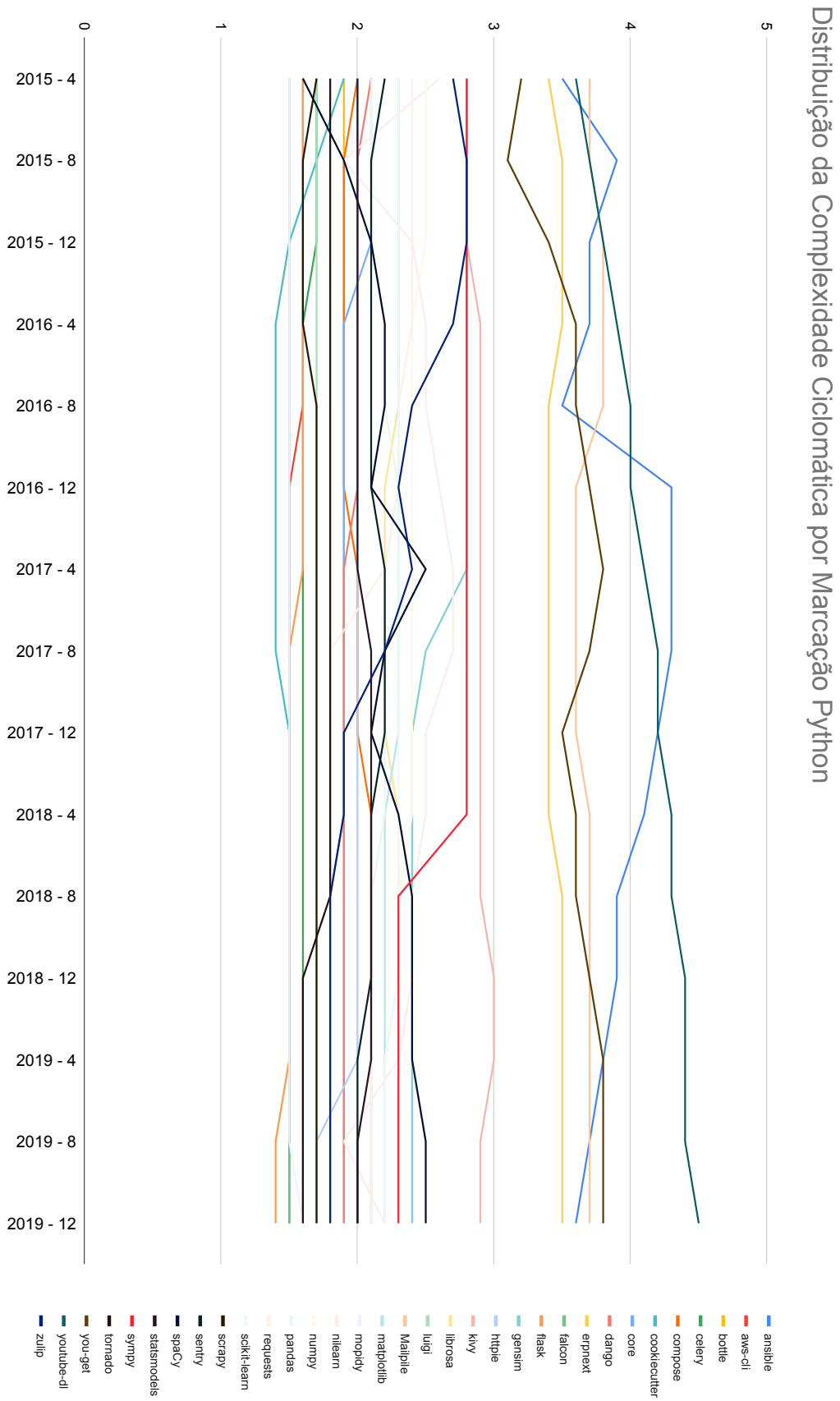

Figura 21 – Distribuição da Complexidade Ciclomática ao longo do período Python

5.7 Comparações com outros datasets

5.7.1 *Bugs.jar*

Para se ter uma ideia de tamanho e importância do SED99, uma comparação com alguns *datasets* disponíveis no meio acadêmico, que sejam relevantes às possibilidades e propósitos do SED99, é válida, reforçando sua importância.

O primeiro *dataset* relacionado é *Bugs.jar*, encontrado no trabalho de Saha et al. (2018), voltado para a linguagem *Java* e com foco em *bugs* e *patches* lançados em 8 grandes repositórios *open source*. Sua construção foi baseada nos aspectos da relevância, diversidade, reprodução e automação.

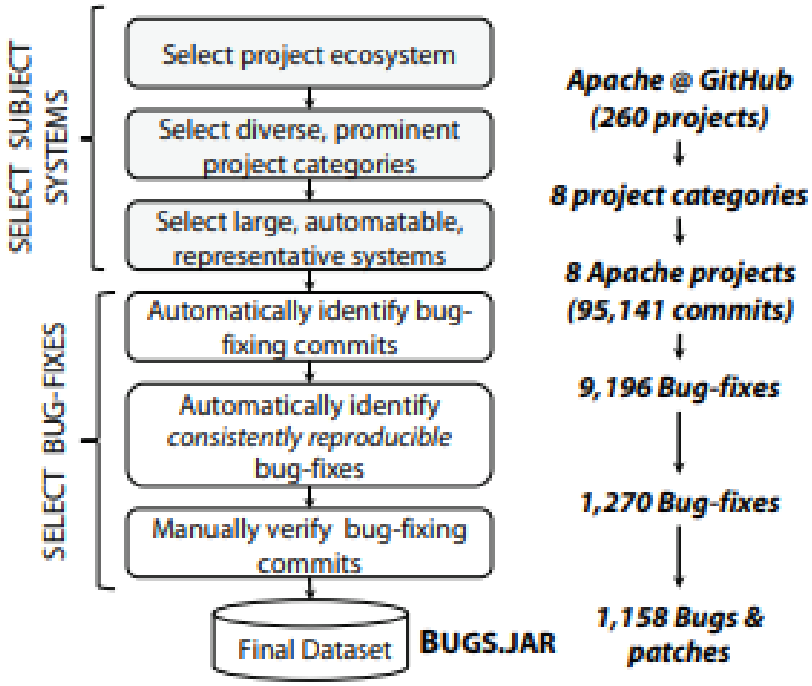

Figura 22 – Processo de Construção do *Bugs.jar* (SAHA et al., 2018)

O *dataset* foi construído em etapas, como as identificadas acima, que geraram cerca da captação de 1158 *bugs* e *patches* através de uma peneiração realizada pelos criadores. O SED99 também foi construído em etapas de processos descritos na seção 5.1, que resultaram na coletânea de repositórios, com arquivos que também podem ser utilizados na perspectiva do trabalho com *bugs* e *patches*. É interessante destacar a utilização de ferramentas, como o *Jira* no caso do *Bugs.jar*, mas também processos manuais para garantir a qualidade. No caso do *Bugs.jar*, foi para verificar a correta categorização de *bugs* na plataforma *Jira*, e no SED99, para verificar se a ordem temporal dos *commits* estava correta.

Existe uma seção de comparação do *Bugs.jar* com o *Defect4j*, afirmando possuir 2.93 vezes mais *bugs* e 3.46 vezes maior em relação as linhas de código, medidas em *KLOC*. O SED99 naturalmente por possuir 91 repositórios a mais e uma abrangência mais genérica, acaba sendo muito maior em relação ao seu volume de dados e também pode se beneficiar em estudos do mesmo estilo do *Bugs.jar* sem prejuízo.

Percebe-se o viés científico no que este *dataset* pode ser empregado, através da aplicação de técnicas de identificação de *bugs*, categorização e utilização de ferramentas automatizadas. Essa característica, com um processo de peneiração em níveis similar ao SED99 e documentação da pesquisa desenvolvida com os mesmos focos, além do fato do *Apache Flink* estar presente nos dois repositórios, oferece suporte ao SED99 por atuar no mesmo universo, mas com outras possibilidades, maiores ainda e que enaltece o *Bugs.jar*, que deve ser utilizado como referência quando se busca uma perspectiva mais voltada ao tratamento de erros em códigos fonte.

5.7.2 *Public Git Archive*

O trabalho de Markovtsev e Long (2018) possui cerca de 3 tera de tamanho, muito superior ao SED99, contemplando cerca de 182014 repositórios com destaque no *Github*, disponíveis através de requisições HTTP com os códigos fonte, metadados e histórico de desenvolvimento.

Esse *dataset* acaba seguindo a mesma perspectiva do SED99 em ser algo mais genérico, por se aproveitar das estruturas que o *Git* oferece, gerar visualizações com o *dataset* e arquivos de consulta, como o csv do SED99. O SED99 se inspira em menor magnitude neste trabalho, também destacável a seleção por linguagens de programação, como demonstra a figura 23:

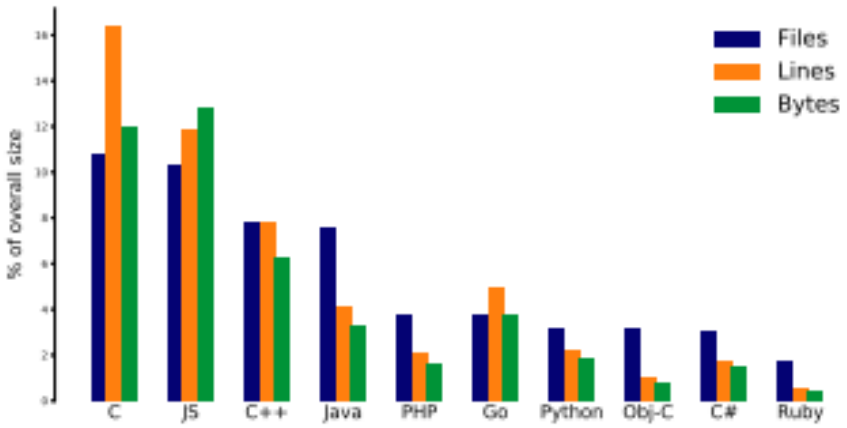

Figure 2: Statistics of 10 most popular programming languages in PGA.

Figura 23 – Distribuição das linguagens e suas linhas, arquivos e bytes do *Public Git Archive* (SAHA et al., 2018)

No caso do SED99, foram escolhidos as linguagens *C++*, *Java* e *Python* devido a sua maturidade no mercado, o que tendenciosamente levaria ao encontro de projetos em diferentes tamanhos para compor o *dataset*, frente as outras tecnologias.

Um fato interessante é em relação à aplicabilidade do *Public Git Archive*, é a possibilidade da aplicação *machine learning* e processamento de linguagem natural, devido ao seu tamanho. O SED99 foca mais em aspectos pertinentes à Evolução de *Software*, mas também, pode ser utilizado para esse fim, seja em uma pesquisa final ou um período de testes de ferramentas, entre outros.

5.7.3 *A Historical Dataset of Software Engineering Conferences*

Não menos importante, o fator histórico por trás do SED99 auxilia no entendimento de comportamentos, levantamento de hipóteses e análise de um período de 5 anos dos projetos reunidos. A pesquisa de Vasilescu, Serebrenik e Mens (2013) trabalha no mesmo viés, como todos os outros apresentados, mas com um maior destaque, pois seu objeto de estudo é conhecimento. Este *dataset* concentra os artigos publicados por 11 conferências bem conceituadas, com interesse maior na saúde da comunidade de pesquisa da Engenharia de *Software*.

Foi utilizado o *MySQL* para o armazenamento dos dados extraídos, e um fator interessante é que sua aplicabilidade pode ser voltada à análise de *networking*, em relação aos conteúdos e às subcomunidades que são formadas em tópicos de interesse.

O *dataset* contempla uma janela em média de 15 anos entre cada conferência escolhida, mas fica a atenção especial na formulação histórica, e no paralelo que publicações tem com

código fonte, pois mesmo não possuindo uma sintaxe tão acessível, suas linhas contam a história de projetos e suas trajetórias até o momento mais recente capturado.

No geral, os 3 *datasets* escolhidos para comparação compartilham técnicas e características com o SED99, pois não existe uma competição de qual *dataset* é melhor, apenas os mais adequados a um determinado contexto, além de aspectos relacionados aos seus tamanhos. É notável a utilização da mesma estratégia por Vasilescu, Serebrenik e Mens (2013) para a visualização de uma das métricas selecionadas, que é o gráfico de distribuição temporal, também apresentado neste trabalho na seção 5.6.

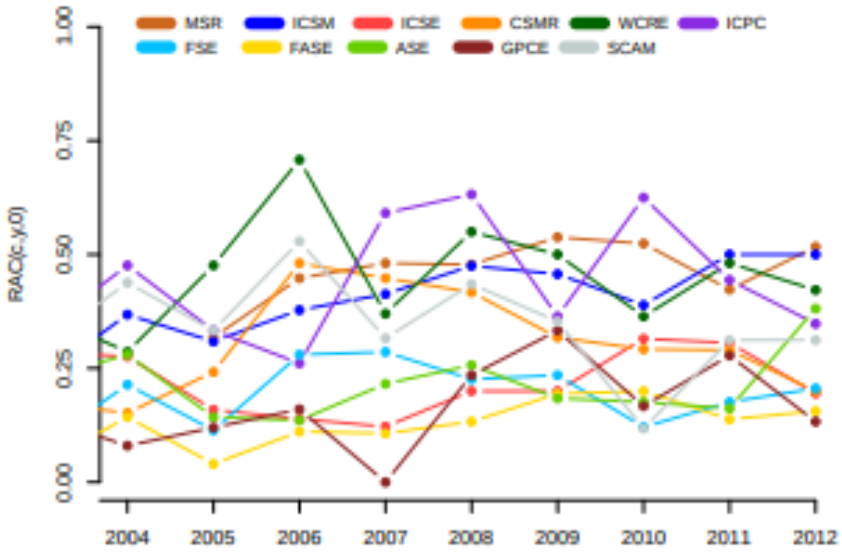

Figura 24 – Distribuição da métrica *"Inbreeding Ratio"*pelas conferências escolhidas (VASILESCU; SEREBRENIK; MENS, 2013)

6 Conclusões

A Evolução de *Software* é uma área que, em comparação a outras da tecnologia da informação, necessita de maiores estudos e contribuições. O SED99 apresenta-se com uma proposta de oferecer suporte às novas pesquisas, baseado em estudos já existentes, como as Leis de Lehman e toda a cultura que *softwares* disponíveis publicamente agregaram à comunidade.

Procurou-se criar um processo de extração que utiliza-se de ferramentas simples, mas robustas como o *Git*, *Shell* e o *Lizard* para viabilizar a documentação de 5 anos de modificações em 15 intervalos totais de projetos de *software* livre / Open Source, com a compilação de um arquivo .csv para ajuda na navegação entre os diretórios.

Percebe-se muito presente as Leis de Lehamn nestes projetos, considerando seus aspectos do estilo de desenvolvimento, como a livre contribuição, acesso democratizado, envolvimento da comunidade que possibilita a releitura das leis para um novo contexto, mas, também, permite a aplicação de observações já difundidas e compartilhadas através da análise de softwares proprietários, como foi iniciado no desenvolvimento das leis.

Este trabalho também descreveu o processo de criação na tentativa de auxiliar novas pesquisas no mesmo estilo, oferecendo suporte para a criação de novas metodologias de compilação de *datasets*, com inspiração nos métodos adotados neste trabalho.

Para a publicação do SED99 foi utilizado o próprio *Github*, plataforma amplamente utilizada, na tentativa de convidar mais pessoas à conhecerem o projeto e extrair os benefícios dele, reforçando cada vez mais a importância que o SED99 vai causar para os estudos na área de Evolução de *Software*.

6.1 Trabalhos futuros

Para trabalhos futuros em relação ao SED99 foram escolhidos os seguintes pontos de implementação/melhoria:

- Criação de um site de promoção do *dataset*, para aumento na divulgação e disponibilização de possíveis atualizações.
- Incorporação de novas métricas ao csv, para agregar cada vez o mais o conteúdo fornecido pelo SED99.
- Geração de mais gráficos, com visões voltadas à identificação de *bugs*, classes, funções com o auxílio de outras ferramentas, muito semelhante ao trabalho realizado por Saha et al. (2018) com o *Bugs.jar*

- Exploração do dataset com uma visão voltada mais a Inteligência Artificial, mineração de dados, entre outros.
- Viabilização para a inclusão de mais uma linguagem de programação e repositórios relacionados.

Referências

AJALA, V. et al. Análise da complexidade ciclomática como apoio ao processo de desenvolvimento do pensamento algorítmico. In: SBC. *Anais do V Workshop de Desafios da Computacao aplicada a Educacao*. [S.l.], 2016. p. 31–40. 40

CHEN, C. An empirical investigation of correlation between code complexity and bugs. *arXiv preprint arXiv:1912.01142*, 2019. 46

COUTO, C. et al. Comets: a dataset for empirical research on software evolution using source code metrics and time series analysis. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 38, n. 1, p. 1–3, 2013. 31

DABBISH, L. et al. Social coding in github: transparency and collaboration in an open software repository. In: *Proceedings of the ACM 2012 conference on computer supported cooperative work*. [S.l.: s.n.], 2012. p. 1277–1286. 26

E-TYPE, P-Type, S-Type systems. Disponível em: <https://denrox.com/post/e-type-p-type-s-type-systems>. 17

GEBRU, T. et al. Datasheets for datasets. *arXiv preprint arXiv:1803.09010*, 2018. 16, 42

GRANO, G. et al. Android apps and user feedback: a dataset for software evolution and quality improvement. In: *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics*. [S.l.: s.n.], 2017. p. 8–11. 30

HERRAIZ, I. et al. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 46, n. 2, p. 1–28, 2013. 15, 17, 18

ISRAEL, H.; GREGORIO, R. Comparison between slocs and number of files as size metrics for software evolution analysis. In: *Washington, USA: Proceedings of the Conference on Software Maintenance and R-eengineering. IEEE Computer Society*. [S.l.: s.n.], 2006. p. 206–213. 26

KAUR, T.; RATTI, N.; KAUR, P. Applicability of lehman laws on open source evolution: a case study. *International Journal of Computer Applications*, Citeseer, v. 93, n. 18, p. 0975–8887, 2014. 15

KOGUT, B.; METIU, A. Open-source software development and distributed innovation. *Oxford review of economic policy*, Oxford University Press, v. 17, n. 2, p. 248–264, 2001. 24

LEHMAN, M. M. Laws of software evolution revisited. In: SPRINGER. *European Workshop on Software Process Technology*. [S.l.], 1996. p. 108–124. 17, 18, 19, 20, 21, 22, 23, 24

MADI, A.; ZEIN, O. K.; KADRY, S. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and Its Applications*, v. 7, n. 2, p. 67–82, 2013. 27

MARKOVTSEV, V.; LONG, W. Public git archive: A big code dataset for all. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2018. p. 34–37. 51

OKWU, P.; ONYEJE, I. Software evolution: past, present and future. *American Journal of Engineering Research (AJER)*, v. 3, n. 05, p. 21–28, 2014. 15

OTTE, S. Version control systems. *Computer Systems and Telematics*, p. 11–13, 2009. 25

PERENS, B. et al. The open source definition. *Open sources: voices from the open source revolution*, Sebastopol, CA: O'Reilly, v. 1, p. 171–188, 1999. 24

RIBEIRO, W. et al. Acompanhamento da evolução de software via métricas. In: *Workshop de Manutenção de Software Moderna (WMSWM)*. [S.l.: s.n.], 2012. 30

SAHA, R. K. et al. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2018. p. 10–13. 11, 50, 52, 54

SKOULIS, I.; VASSILIADIS, P.; ZARRAS, A. Open-source databases: Within, outside, or beyond lehman's laws of software evolution? In: SPRINGER. *International Conference on Advanced Information Systems Engineering*. [S.l.], 2014. p. 379–393. 11, 29

VASILESCU, B.; SEREBRENIK, A.; MENS, T. A historical dataset of software engineering conferences. In: IEEE. *2013 10th Working Conference on Mining Software Repositories (MSR)*. [S.l.], 2013. p. 373–376. 11, 52, 53