Uma Análise da Co-Evolução de Teste em Projetos de Software no GitHub

Charles Miranda, Guilherme Avelino, Pedro Santos Neto e Victor da Silva
{charlesmiranda,gaa,pasn,victor.silva}@ufpi.edu.br
Universidade Federal do Piauí
Teresina, Piauí, Brasil

RESUMO

Os sistemas de software evoluem e essa evolução requer modificações em seu código-fonte para a realização de alterações, como correções de *bugs*, melhorias de desempenho ou adição de novas funcionalidades. Tendo em vista a importância da realização de testes para garantir a qualidade, modificações no código-fonte devem ser acompanhadas de alterações e incrementos do código de teste. Entretanto, testes e a co-evolução desse muitas vezes são negligenciados no desenvolvimento de projetos de software, podendo resultar em maior esforço e custo para manter o projeto. Neste trabalho, através da análise de um grande *dataset*, composto pelo histórico de desenvolvimento de 3.000 projetos hospedados no Github, investigamos como artefatos de código-fonte e teste evoluem. Através da aplicação de técnicas de clusterização identificamos cinco padrões comuns de crescimento de teste. Adicionalmente, ao contrastar dados dos repositórios identificados com co-evolução e sem co-evolução foi observado que os primeiros apresentam maiores níveis de contribuição (commits, colaboradores e forks).

PALAVRAS-CHAVE

Co-evolução, teste, mineração de repositório de software, GitHub.

1 INTRODUÇÃO

Os sistemas de software estão cada vez mais presentes na vida das pessoas, deixando de ser apenas ferramentas de apoio e se tornando algo intrínseco às nossas vidas. Eles estão presentes em diversos dispositivos, sendo utilizados para lazer, estudo, trabalho e outras atividades de nosso cotidiano. Dada sua importância, espera-se que o software faça o que ele foi desenvolvido para fazer, ou seja, o software deve possuir o correto funcionamento [10, 11].

Durante a evolução de um projeto de software, o código-fonte do sistema muda continuamente para lidar com novos requisitos ou possíveis problemas que possam surgir. Essa evolução requer a alteração e adição de diversos artefatos ao longo do ciclo de vida do software [9]. Entre esses artefatos, têm-se o código de teste, o qual deve co-evoluir com o código-fonte do sistema [6]. Essa co-evolução deve acontecer por pelo menos dois motivos: 1) novas funcionalidades devem ser testadas e 2) ao realizar mudanças a preservação do comportamento deve ser verificada [14].

Nesse contexto, alguns estudos foram desenvolvidos buscando compreender a relação da evolução do código de teste e do código-fonte em repositórios de software. Tais pesquisas buscam identificar comportamentos e padrões relacionados a co-evolução de teste [5, 6, 12, 14]. Entretanto, tais estudos se limitaram a uma única linguagem de programação e analisaram poucos projetos.

Assim, este trabalho pretende investigar a co-evolução do código de teste e código-fonte em múltiplas linguagens de programação, buscando identificar que características dos repositórios podem ser influenciadas pela co-evolução. Para tanto foi utilizada uma grande base de dados composto pelo histórico de desenvolvimento de 3.000 repositórios hospedados no Github, desenvolvidos em cinco linguagens de programação diferentes (JavaScript, Java, Python, PHP, Ruby). O estudo é conduzido tendo como objetivo prover respostas para três questões de pesquisa: QP_1 - *Como testes evoluem em projetos de software?*, QP_2 - *Com que frequência o código-fonte e testes co-evoluem em projetos de software?* e QP_3 - *Que características distinguem projetos onde há co-evolução de código de teste de projetos onde essa prática não é comum?*

Este trabalho se estrutura da seguinte forma. Na Seção 2 é descrito a metodologia utilizada. Na Seção 3e 4 são descritas as análises e os resultados obtidos. Na Seção 5 é fornecido um breve resumo sobre trabalhos relacionados. Por fim, na Seção 6 é apresentada as considerações obtidas e direções para trabalhos futuros.

2 METODOLOGIA

Nessa seção descrevemos a metodologia adotada para a análise da co-evolução de teste em repositórios de software. O estudo é desenvolvido em seis etapas: 1) seleção dos repositórios e extração dos dados; 2) classificação de arquivos de teste; 3) extração do histórico da proporção de teste; 4) identificação de padrões de crescimento; 5) identificação de repositórios com co-evolução; e 6) análise da influência da co-evolução de teste nos repositórios.

2.1 Seleção de Repositórios e Extração de Dados

A extração de dados foi feita a partir de projetos de software populares presente na plataforma de desenvolvimento GitHub. O *dataset* desta pesquisa foi construído através do download de 3.000 projetos hospedados no GitHub em dezembro de 2019. O GitHub disponibiliza uma API para fazer a seleção e o download dos projetos. A seleção foi baseada no número de estrelas e na linguagem de programação principal do projeto. Foram selecionados os 600 projetos mais populares nas linguagens Javascript, Java, Python, PHP e Ruby, totalizando 3.000 projetos.

A Tabela 1 apresenta o total de *commits*, colaboradores (*devs*), *forks*, *issues*, tamanho (*size*) e estrelas (*stars*) dos repositórios de nosso *dataset*. Observando os valores totais, temos mais de 9 milhões de commits feitos por mais de 260 mil colaboradores. Observando por linguagem, temos PHP e Ruby como as linguagens que possuem mais *commits*. Ruby e Python mais desenvolvedores. Javascript e Java possuem mais *forks*. Javascript e Python possuem mais *issues*. Java e Javascript ocupam mais espaço (*size*). Considerando ao número de estrelas, Javascript e Python são as linguagens com repositórios mais populares, possuindo 10.79 e 5.51 milhões de estrelas respectivamente. Esses dados mostram que utilizando a

abordagem de seleção de estrelas consegue-se construir um *dataset* com relevante volume de dados. A distribuição dos repositórios pode ser verificada na Figura 1.

Tabela 1: Dados do dataset (K = milhares, M = milhões)

Linguagem	Commits	Devs	Forks	Issues	Size	Stars
Javascript	1.74M	52.62K	1.67M	141.69K	28.35GB	10.79M
Java	2.00M	37.95K	1.35M	101.22K	51.89GB	4.77M
Python	1.96M	61.67K	1.20M	135.67K	27.78GB	5.51M
PHP	2.11M	47.37K	461.74K	63.76K	17.81GB	2.46M
Ruby	2.11M	64.34K	392.38K	42.56K	14.22GB	2.12M
Todos	**9.95M**	**263.97K**	**5.08M**	**484.92K**	**140.08GB**	**25.68M**

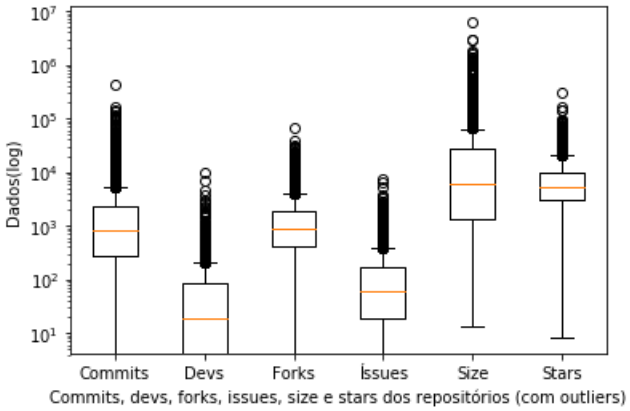

Figura 1: Dados dos repositórios

2.2 Classificação de arquivos de teste

Nesta etapa são identificados os arquivos de teste presente nos projetos do *dataset*. Após a realização de um levantamento da literatura sobre técnicas e práticas de classificação de arquivos de teste, decidiu-se adotar a abordagem proposta por Gonzales et al. [3]. Para identificar arquivos de teste, os autores construíram um Catálogo de *Frameworks* de automação de teste e definiram duas heurísticas para detecção de casos de teste. Os principais aspectos da abordagem para classificação de teste são descritos a seguir:

- Catálogo de *Frameworks* de Automação de Teste. É um catálogo constituído por uma lista de *frameworks* de teste e expressões regulares que representam as declarações de import em cada *framework*. O catálogo possui 251 *Frameworks* de teste para 48 linguagens de programação.
- Detecção de Casos de Teste. Um arquivo só é classificado como de teste se atender a dois requisitos: 1) possuir pelo menos um *import* contemplado no Catálogo de Frameworks de Automação de Teste; 2) possuir uma chamada de função pertencente a um *framework* de teste. Por exemplo, usar a anotação @Test no *framework* JUnit.

Segue um exemplo para ilustrar o funcionamento da abordagem de classificação de teste. Ao analisar um repositório Z temos dois arquivos da linguagem java denominados A1 e A2. O arquivo A1 possui no seu conteúdo o *import* do *framework* JUnit e a chamada da anotação *@Test* marcando uma de suas funções/métodos. O arquivo A2 possui em seu conteúdo o *import* do *framework* JUnit, porém não foram identificadas funções/métodos que representam casos de teste (no caso, anotadas com *@Test*). Nesse exemplo, seguindo a abordagem, somente o arquivo A1 é considerado de teste, pois, possui *import* para um *framework* de teste e uma função/método que representam um caso de teste .

2.3 Extração do histórico de proporção de teste

Para investigar a evolução dos artefatos, e consequentemente a co-evolução ou não desses, o histórico de desenvolvimento de cada repositório é dividido em dez intervalos. Dada a lista de *commits* de um repositório, ordenados pela data de *commit*, são selecionados os *commits* que dividem o histórico em 10 porções iguais. Sendo assim, a seleção de cada recorte é proporcional e baseada no total de *commits* do repositório. Para cada *commit* selecionado é realizado *checkout* no repositório e executado o programa *cloc* [1] em todos os arquivos do repositório para realizar a coleta do total de Linhas de Código (*LOC*). Dessa forma, é realizada a coleta de dados dos *commits* a cada 10% da história de cada projeto. São realizadas, separadamente, as coletas de *LOC* de arquivos de teste e de código-fonte. Com essa informação é possível calcular a proporção de LOCs de teste para cada recorte, conforme descrito na Equação 1.

$$\text{proporcaoTeste} = \frac{totalLOCTeste}{totalLOCTeste + totalLOCRepositorio} \quad (1)$$

2.4 Identificação de padrões de crescimento de teste

Neste trabalho, utilizamos o KSC [13] como algoritmo de clusterização, pois ele apresenta agrupamentos mais significativos considerando agrupamentos de séries temporais. O algoritmo foi utilizado em outros estudos para clusterizar séries temporais de popularidade de repositórios de software [1], de vídeos do youtube [2] e twiter [4]. Na sua execução é informado o número máximo de grupos (k) que serão gerados e a lista de séries temporais na entrada do algoritmo. Então, iniciando em k igual 2, o k é incrementado em uma unidade até chegar ao número máximo de k. Para cada k da iteração, o KSC define os *centroids* baseado nas formas das séries temporais informadas na entrada do algoritmo e só então cada série temporal é associada a um *cluster*.

No final da execução, para cada k, têm-se um arquivo com os resultados de similaridade, *centroids* e agrupamentos que permitirão identificar os padrões de crescimento. Uma parte importante do processo é identificar o valor de k que melhor agrupa a amostra. Para selecionar o melhor k, utilizamos a heurística β_{CV} [8]. Baseado nessa heurística, o menor valor de k depois que a proporção do β_{CV} estabiliza deve ser escolhido [1, 8].

2.5 Identificação de repositórios com co-evolução

O objetivo desta etapa é identificar os repositórios com co-evolução de teste. A partir das séries temporais que representam a proporção de LOCs de teste (total LOCs de teste dividido por total LOCs do

[1] http://cloc.sourceforge.net/

repositório) é calculado o desvio padrão da proporção de teste em cada um dos repositórios.

Da distribuição do desvio padrão do percentual de testes são computados os quartis dos repositórios baseados na variância da evolução do código de teste em relação ao código-fonte no histórico de desenvolvimento. Essa informação é então utilizada para identificar os repositórios que co-evoluem em nosso *dataset*, ao selecionar aqueles onde a variação da proporção ao longo do tempo de vida do projeto é mínima. Para tanto, definimos como critério de corte o valor de desvio padrão correspondente ao primeiro quartil da distribuição.

2.6 Influência da co-evolução de teste nos repositórios

Nesta etapa da pesquisa, os dados coletados de repositórios são comparados entre os repositórios classificados com co-evolução e os não sem co-evolução. Os dados coletados da distribuição de número de *commits*, número de colaboradores, número de *forks* e número de *issues* são analisadas. O método *Wilcoxon/Mann-Whitney* [7] é utilizado para avaliar se existem diferenças significativas (*p-value* <= 0,05) relacionadas às características dos repositórios entre os grupos, os que apresentaram co-evolução e os que não possuem co-evolução, e caso sejam significativas utilizamos o *Cohen's D* para quantificar o tamanho da diferença.

2.7 Artefatos da pesquisa

Os dados utilizados neste trabalho estão publicamente disponíveis em https://zenodo.org/record/5222965#.YR55lUBv-Uk.

3 RESULTADOS

Esta seção apresenta os resultados obtidos ao investigar as questões de pesquisa alvo deste estudo.

***QP1**) Como testes evoluem em projetos de software?*

Como primeira parte da investigação da QP1 foram descartados repositórios nos quais não foi identificado arquivos de testes. Para a identificação dos arquivos de teste foi aplicada a abordagem descrita na Seção 2.2. Do total de 3.000 repositórios inicialmente selecionados, foram identificados arquivos de testes em 1.914 (63,8% dos repositórios).

A Figura 2 apresenta a distribuição do percentual de LOCs de testes nos repositórios. Considerando todos os repositórios, é possível observar que na maioria dos repositórios (75%) o percentual de arquivos de testes é inferior a 41% e metade deles tem no máximo 20% de testes. Por linguagem, observa-se que o segundo *quartil* (mediana) das linguagens JavaScript, Java, Python, PHP e Ruby são, respectivamente, 31%, 18%, 19%, 23%, 16%.

Para analisar a evolução do código de teste em relação ao código-fonte nos repositórios foram construídos gráficos representando a proporção de testes considerando a divisão do tempo de vida do repositório em 10 intervalos iguais. Buscando identificar padrões evolucionários nesses dados, as linhas de tempo foram agrupadas (clusterizadas) adotando o algoritmo KSC (vide Seção 2.4).

Para identificar o melhor número de grupos para o nosso *dataset*, utilizamos a heurística β_{CV} [8], conforme dito na metodologia. No

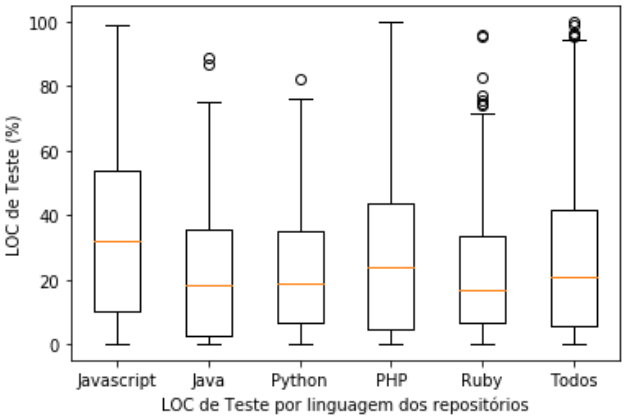

Figura 2: Percentual de LOC de Teste

nosso exemplo, os valores do coeficiente de variação para a proporção de teste dos repositórios foram 0.531, 0.608, 0.652, 0.655, 0.641, 0.630, 0.609, 0.592 e 0.585 para os k 2, 3, 4, 5, 6, 7, 8, 9 e 10 respectivamente. Neste caso, o β_{CV} se estabiliza com k igual a 5. Os resultados dessa classificação podem ser observados na Figura 3.

Na Figura 3 nos mostra os *clusters* encontrados, em cada cluster, temos o nome do *cluster*, total de repositórios, *centroid* (destacado na cor preta) e os repositórios que o compõe (cada repositório representado por uma cor tracejada). Podemos visualizar os cinco padrões de co-evolução e seus comportamentos da proporção de teste durante o histórico dos projetos. Os *clusters I e II* são os que apresentam o maior número de repositórios.

> **Resumo da QP1:** Foram identificados 5 padrões de evolução de teste. Os dois padrões com mais repositórios (*clusters I* e *V*), modelam um comportamento em que a proporção de testes cresce de forma gradual no início do projeto, se estabilizando posteriormente.

***QP2**) Com que frequência o código-fonte e testes co-evoluem em projetos de software?*

Para identificar os repositórios com co-evolução utilizamos o desvio padrão da proporção de teste, conforme detalhado na Seção 3.5. Após a análise da distribuição, foi adotado como valor de corte na identificação de repositórios com co-evolução o valor de XX, correspondente o primeiro quartil. Dessa forma, repositórios onde a proporção de testes tem variação inferior a XX, foram classificados como repositórios com co-evolução de testes. Adotando essa estratégia, foram identificados 422 repositórios com indícios de co-evolução de teste. Observando por linguagem, encontramos diferenças significativas no percentual de repositórios. As linguagens com maior co-evolução são Java (36%), seguida por Python e Ruby (23% e 22%). Javascript é a que possui menor percentual (13%).

Em relação a clusterização, os repositórios com co-evolução encontram-se com mais frequência no *Cluster I*, com 205 repositórios, e no *Cluster V*, que possui 126. A quantidade de repositórios com co-evolução nos demais *clusters* é de 32, 31 e 28 repositórios para os *Clusters III, II* e *IV* respectivamente.

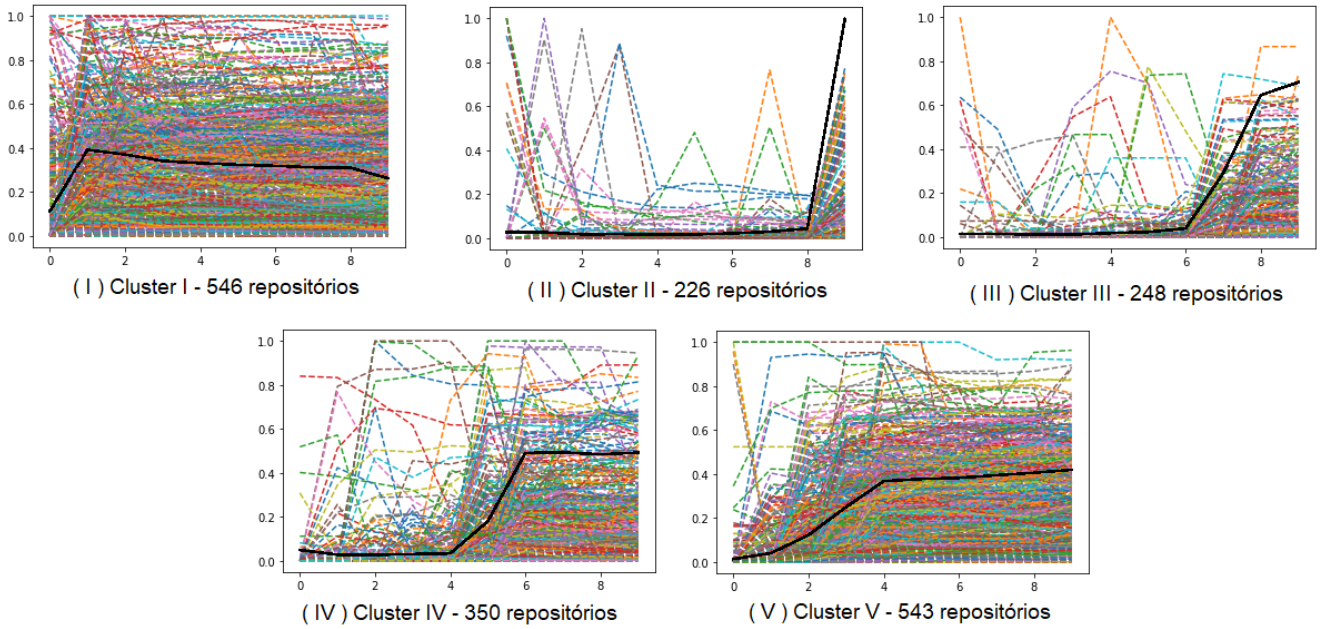

Figura 3: *Clusters* gerados do histórico de proporção de teste. Linha preta representa o centróide de cada cluster.

Resumo da QP2: Utilizando o desvio padrão como métrica para identificar a co-evolução nos repositórios, encontramos 422 repositórios com indícios de co-evolução de teste, representando 22,04% dos repositórios com teste.

QP3*) Que características distinguem projetos onde há co-evolução de código de teste de projetos onde essa prática não é comum?*

A Figura 4 apresenta a distribuição dos repositórios sem e com co-evolução em relação ao número de *commits*, número de colaboradores, número de *forks* e número de *issues*. Conforme o método *Wilcoxon/Mann-Whitney* as distribuições de *commits*, colaboradores, número de *forks* e número de *issues* são estatisticamente diferentes (p-*value* <= 0,05) entre os grupos com e sem evolução. Adicionalmente, ao aplicar o *Cohen's D*, as diferenças entre as distribuições podem ser consideradas pequenas (*small*) para *commits* (0.25), colaboradores (0.27) e *forks* (0.20) e insignificante (*negligible*) para *issues* (0.17). Assim, os resultados mostram que repositórios com co-evolução apresentam de forma significativa maior número de *commits*, de colaboradores, de *forks* em comparação aos repositórios que não possuem.

Em relação às características comparadas entre os repositórios com co-evolução e aqueles que não possuem, segue os resultados por linguagem de programação. Distribuição de *commits*: os repositórios com co-evolução apresentam mais *commits* em todas as linguagens. Distribuição de Colaboradores: os repositórios com co-evolução possuem mais colaboradores nas linguagens Java, PHP e Ruby, e são menores para Javascript e Python. Distribuição de *forks*: os repositórios com co-evolução possuem mais *forks* nas linguagens JavaScript, Java e Ruby, e são semelhantes para Python e PHP. Distribuição de *issues*: As *issues* são mais utilizadas em repositórios com co-evolução para JavaScript, Java, Python e Ruby. O número de *issues* não possui diferenças significativas para PHP.

Resumo da QP3: A comparação dos dois grupos de repositórios demonstrou a existência de diferenças significativas entre repositórios com e sem co-evolução. Repositórios identificados com co-evolução possuem mais *commits*, colaboradores, *forks*.

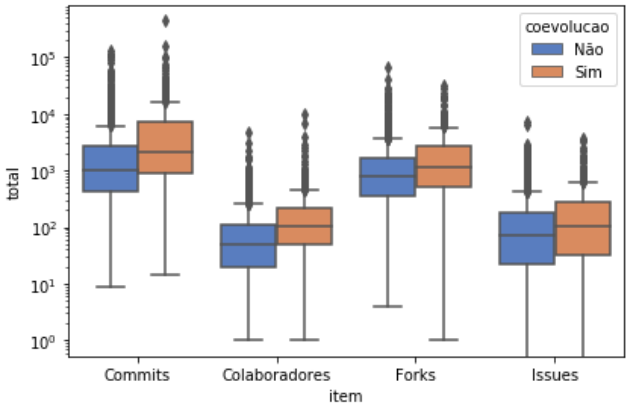

Figura 4: Distribuições entre projetos sem e com co-evolução em relação ao número de *commits*, colaboradores, *forks* e de *issues*

4 DISCUSSÃO

Nossos resultados mostram que 63,8% dos repositórios possuem arquivos de teste. Conforme visto na Figura 2, Javascript é a linguagem com mais repositórios com teste, 31% na mediana, e, em contrapartida temos Ruby com apenas 16%. A mediana de todas as linguagens correspondem a 20%. Esses resultados demonstram que algumas linguagem de programação podem ter efeito positivo na adoção de testes no processo de desenvolvimento de software.

Adicionalmente, realizamos a clusterização do histórico da proporção de testes buscando identificar como os testes evoluem. Na figura 3, é observado que os repositórios com teste são agrupados em cinco padrões de crescimento. Os *clusters I* e *V* apresentam o maior número de repositórios, e apresentam como característica a presença de teste desde as fases iniciais do desenvolvimento do projeto e uma estabilização desse percentual com o tempo. Essa estabilização é um indício de co-evolução nos repositórios desses *clusters*, sendo confirmado ao observarmos que esses dois *clusters* possuem a maior parte dos repositórios identificados com co-evolução.

Na Figura 4 observa-se que a mediana de *commits* dos repositórios com co-evolução é maior que a dos que não possuem. O mesmo acontece quando observa-se as medianas de colaboradores, *forks* e *issues*, contudo a diferença relacionada as *issues* se mostrou insignificante ao aplicar o teste *Cohen's D*. Esse fato indica que a co-evolução de teste é mais comum em repositórios que são alterados com maior frequência (*commits*) e possuem maior contribuição por parte da comunidade, medido através do número de colaboradores e *forks*. Os resultados são esperados, visto que projetos com mais alterações precisam de mais testes para garantir a qualidade. Da mesma forma, a existência de teste é fundamental para permitir a contribuição da comunidade, devendo haver uma evolução nos testes para que tais contribuições possam acontecer de forma segura e gerenciável.

5 TRABALHOS RELACIONADOS

Marsavina et al. [6] apresentam uma abordagem para investigação de padrões de co-evolução de teste e código-fonte. A abordagem utiliza a extração de mudanças do histórico e o uso de um algoritmo de regras de associação para gerar os padrões de co-evolução. Foram utilizados cinco projetos *open source* no *dataset*. Zaidman et al. [14] realizam um estudo sobre co-evolução de teste e código-fonte. Os resultados obtidos mostram que o desenvolvimento pode ser faseado ou síncrono. Levin et al. [5] exploram a co-evolução código de teste e código-fonte através de mudanças semânticas, ou seja, que adicionar ou remover funções/métodos/classes são ações significativas na manutenção de teste. Os resultados obtidos mostram que na maior parte das vezes os desenvolvedores realizam correções de código sem realizar manutenção de teste no mesmo *commit*. Vidács et al. [12] realizaram uma extensão do trabalho de Marsavina et al. [6]. Nele é realizado um estudo para identificar padrões de co-evolução de teste através de regras de associação. Os resultados obtidos mostram que o código-fonte e teste crescem em sincronia no projeto analisado. Foi verificado que em muitos casos código-fonte e teste são alterados separadamente.

Este trabalho se diferencia dos demais ao estender a investigação sobre co-evolução de testes a um conjunto maior de dados, composto por projetos desenvolvidos em diferentes linguagens de programação. Adicionalmente, são realizadas investigações não contempladas nos demais trabalhos, tais como a identificação de padrões de crescimento de testes e a influência da co-evolução de teste nas características dos projetos.

6 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi realizado um estudo sobre a co-evolução de teste em repositórios de software, tendo sido investigado 3.000 projetos reais hospedados no Github. No estudo utilizamos técnicas de classificação de arquivos de teste, mineração de repositório de software e clusterização para extração, análise e síntese dos resultados. Os resultados deste estudo podem ser sumarizados como segue: i) dos 3.000 repositórios inicialmente analisados, verificou-se que 1.914 (63,8%) possuem teste e que destes, apenas 422 (22,04%) possuem co-evolução de teste; ii) foram identificados cinco padrões de crescimento de teste, sendo a maior parte dos repositórios agrupados em dois *clusters* que apresentam estabilidade na proporção de teste na maior parte do histórico do projeto. iii) foram identificadas evidências de que repositórios com co-evolução de teste são alterados com maior frequência (*commits*) e possuem maior interesse da comunidade em contribuir (número de colaboradores e *forks*).

Como proposta futura, com o objetivo de validar os resultados e ampliar o conhecimento sobre o tema, pretende-se realizar um *survey* com os desenvolvedores dos sistemas analisados.

REFERÊNCIAS

[1] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 334–344.

[2] Flavio Figueiredo. 2013. On the prediction of popularity of trends and hits for user generated videos. In *6th International Conference on Web Search and Data Mining*. 741–746.

[3] Danielle Gonzalez, Joanna CS Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension. In *14th International Conference on Mining Software Repositories (MSR)*. IEEE, 391–401.

[4] Janette Lehmann, Bruno Gonçalves, José J Ramasco, and Ciro Cattuto. 2012. Dynamical classes of collective attention in twitter. In *21st International Conference on World Wide Web*. 251–260.

[5] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *International Conference on Software Maintenance and Evolution (ICSME)*. 35–46.

[6] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 195–204.

[7] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[8] Daniel A Menasce and Virgilio Almeida. 2001. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR.

[9] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. 2005. Challenges in software evolution. In *8th International Workshop on Principles of Software Evolution (IWPSE)*. IEEE, 13–22.

[10] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[11] Ian Sommerville. 2011. *Software engineering 9th Edition*. Pearson.

[12] László Vidács and Martin Pinzger. 2018. Co-evolution analysis of production and test code by learning association rules of changes. In *Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 31–36.

[13] Jaewon Yang and Jure Leskovec. 2011. Patterns of temporal variation in online media. In *4th International Conference on Web Search and Data Mining*. 177–186.

[14] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (2011), 325–364.