Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

Uma Análise da Co-Evolução de Teste em Projetos de Software

Charles José Lima de Miranda

Teresina-PI, 30 de Novembro de 2021

Charles José Lima de Miranda

Uma Análise da Co-Evolução de Teste em Projetos de Software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (linha de pesquisa: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Prof. Dr. Guilherme Amaral Avelino

Coorientador: Prof. Dr. Pedro de Alcantara dos Santos Neto

Teresina-PI

30 de Novembro de 2021

Charles José Lima de Miranda
Uma Análise da Co-Evolução de Teste em Projetos de Software/ Charles José Lima de Miranda. – Teresina-PI, 30 de Novembro de 2021-
53 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Guilherme Amaral Avelino

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação, 30 de Novembro de 2021.

1. Coevolução. 2. Código de Teste. 3. Repositórios de Software. 4. Mineração de Dados.

CDU 02:141:005.7

Charles José Lima de Miranda

Uma Análise da Co-Evolução de Teste em Projetos de Software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (linha de pesquisa: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Trabalho aprovado. Teresina-PI, 30 de Novembro de 2021:

Prof. Dr. Guilherme Amaral Avelino
Orientador

Prof. Dr. Pedro de Alcantara dos Santos Neto
Co-Orientador

Prof. Dr. Raimundo Santos Moura
Docente Interno ao Programa

Prof. Dr. Davi Viana dos Santos
Docente Externo ao Programa

Teresina-PI
30 de Novembro de 2021

Dedico aos meus pais José Lima de Miranda (in memoriam) e Maria da Cruz Lima Miranda pelo amor, carinho e apoio aos meus estudos.

Agradecimentos

Gostaria de agradecer primeiramente aos meus orientadores, Prof. Guilherme Amaral Avelino e Prof. Pedro de Alcântara dos Santos Neto por terem me acolhido não apenas como aluno, mas como amigo. De certo, para mim esses anos no Mestrado foram recheados de momentos de crescimento pessoal e muito aprendizado. Quero também agradecer aos meus pais, a minha esposa, aos meus colegas da UFPI e colegas do trabalho que muito me ajudaram. Obrigado!

Resumo

Durante a evolução de um software, é natural a necessidade de modificações em seu código-fonte para a realização de alterações, como correções de *bugs*, melhorias de desempenho ou adição de novas funcionalidades. Essas modificações precisam garantir as funcionalidades e qualidade do software, assim, essas modificações no código-fonte devem ser acompanhadas de alterações e incrementos do código de teste. Isso faz da identificação de como ocorre a co-evolução de teste uma informação importante para o desenvolvimento de software. Entretanto, pesquisas relacionadas a co-evolução de teste estão focadas na linguagem Java. Desse contexto surge a necessidade de ampliar esse estudo para outras populares linguagens em projetos de software. 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 forte e fraca co-evolução foi observado que os primeiros apresentam maiores níveis de contribuição em relação ao número de *commits*, colaboradores, *forks* e *issues*.

Palavras-chaves: Co-evolução de Teste, Código de Teste, Repositórios de Software, Mineração de Repositórios.

Abstract

During the software evolution, it is natural the need for modifications in its source code to carry out changes, such as bug fixes, performance improvements, or the addition of new features. These modifications need to guarantee the functionality and quality of the software, so these modifications to the source code must be accompanied by changes and increments to the test code. This makes identifying how test co-evolution occurs an important piece of information for software development. However, research related to test co-evolution is focused on the Java language. From this context arises the need to extend this study to other popular languages in software projects. In this work, through the analysis of a large dataset, comprising the development history of 3,000 projects hosted on Github, we investigate how source code and test artifacts evolve. Through the application of clustering techniques, we identified five common test growth patterns. Additionally, when comparing data from repositories identified with strong co-evolution and weak co-evolution, it was observed that the first ones present higher levels of contribution about the number of commits, collaborators, forks, and issues.

Keywords: Test Co-Evolution, Test Code, Software Repositories, Repository Mining.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
ES	*Engenharia de Software*
LOC	*Linha de Código*
SGBD	*Sistema de Gerenciamento de Banco de Dados*
QP	*Questão de Pesquisa*
UFPI	*Universidade Federal do Piauí*

Sumário

1 Introdução

1.1 Contexto e Motivação

Os sistemas de software estão 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. Uma importante atividade no desenvolvimento desses sistemas de software é o teste. O teste de software é uma atividade complexa (DELAMARO; JINO; MALDONADO, 2013) e é realizado para reduzir custos a longo prazo, garantir funcionalidades, garantir confiabilidade e medir a qualidade do software. Apesar de contribuir em diversas atividades no desenvolvimento de software, o teste de software é especialmente associado à identificação e correção de *bugs* (PRESSMAN, 2007). Assim, o teste é realizado no desenvolvimento de software visando minimizar o número de erros no software e, consequentemente, aumentando a sua qualidade.

A tabela 1 mostra que um software de qualidade precisa atender a quatro atributos essenciais para garantir a satisfação do cliente. Desse modo, todos os softwares deveriam ser projetados e desenvolvidos de modo a assegurar a manutenibilidade, confiança e proteção, eficiência e aceitabilidade. Contudo, ainda é comum encontrar sistemas de *Software* com uma grande quantidade de erros.

Tabela 1 – Atributos essenciais de um bom software (SOMMERVILLE, 2011)

Características	**Descrição**
Manutenibilidade	A mudança é um requisito inevitável de um negócio em mudança.
Confiança e proteção	Não deve causar prejuízos físicos ou econômicos devido falhas.
Eficiência	Inclui disponibilidade, velocidade e custo.
Aceitabilidade	Deve ser compreensível, usável e compatível.

A Atividade de Teste de Software é uma das atividades mais custosas em projetos de desenvolvimento de software. Assim, testes automatizados vêm sendo utilizados para agilizar o desenvolvimento de projetos de software. Pesquisas mostram claramente os efeitos positivos da automação de testes no custo, qualidade e tempo de lançamento do software no mercado (KUMAR; MISHRA, 2016). Os modelos de desenvolvimento de software tradicionais realizam a atividade de teste observando o processo de desenvolvimento como um todo, ou seja, desenvolve todo o projeto e, somente então, testa todo o projeto. No entanto, ao longo dos anos a atividade de teste foi adaptada a novos modelos como, por exemplo, os modelos iterativos, diminuindo erros de planejamentos e o risco de defeitos se propagarem por todo o software.

Em suma, teste é algo recomendado (MYERS et al., 2004; WONG et al., 2011), mas nem sempre é realizado (WANGENHEIM; SILVA, 2009; PETTICHORD; BACH; KANER, 2013; CLEGG; ROJAS; FRASER, 2017). Isso pode ser ocasionado pelos custos associados a atividade, por falta de profissionais habilitados, falta de infraestrutura, ou mesmo a ausência de tempo para essa atividade (WONG et al., 2011). Com o objetivo de reduzir os custos referentes a atividade de teste, diversas técnicas, têm sido alvos de pesquisas (ZAIDMAN et al., 2011; LEVIN; YEHUDAI, 2017; VIDÁCS; PINZGER, 2018; VIEIRA et al., 2018).

Outro fator a ser considerado é que durante a evolução de um projeto de software o código-fonte do sistema muda continuamente para lidar com novos requisitos ou correção de problemas que possam surgir. Essa evolução requer a alteração e adição de diversos artefatos ao longo do ciclo de vida do software (MENS et al., 2005). Entre esses artefatos, têm-se o código de teste, o qual deve co-evoluir com o código-fonte do sistema (MARSAVINA; ROMANO; ZAIDMAN, 2014). 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 (ZAIDMAN et al., 2011).

Neste contexto, apesar da importância dos testes sobre a qualidade do software, existem desafios relacionados à realização de teste em projetos de softwares. Esses desafios têm motivado diversas pesquisas e, neste trabalho em especial, pesquisar como os testes evoluem à medida que o software também evolui e o impacto da ausência dessa co-evolução nos projetos de software.

1.2 Definição do Problema

A engenharia de software prescreve a criação de testes , como estratégia para garantir qualidade do software produzido (MYERS et al., 2004; WONG et al., 2011). Entretanto, a simples implementação de teste não é suficiente para garantir a melhoria na atividade de manutenção, já que o código de teste, assim como código de produção exige domínio de conhecimento e manutenção por parte da equipe.

Com isso, necessita-se compreender não somente as vantagens e benefícios da realização de testes, mas da sua evolução junto com o projeto. Compreender a co-evolução de teste é relevante para gerentes de projetos dentro da equipe de desenvolvimento, pois, precisam de evidências convincentes para persuadir seus colegas em tomadas de decisão relacionadas às atividades de teste (ZAIDMAN et al., 2011).

A literatura sobre co-evolução de teste de software, de forma geral, apresenta evidências que apoiam a compreensão da co-evolução de teste somente para a linguagem Java. Diversas pesquisas dessa área abordam o problema com foco somente em Java, e em *datasets* muito pequenos (ZAIDMAN et al., 2011; MARSAVINA; ROMANO; ZAIDMAN,

2014; LEVIN; YEHUDAI, 2017; VIDÁCS; PINZGER, 2018). A generalização dos resultados destes trabalhos requer um maior aprofundamento do estudo, em especial, uma avaliação em *datasets* maiores, composto por projetos de software desenvolvidos em diferentes linguagens de programação. Portanto, apesar da existência de trabalhos relacionando a co-evolução de teste de software, algumas questões ainda dificultam a compreensão da co-evolução de teste. Em suma, seguem as principais dificuldades que afetam a compreensão da co-evolução de teste:

- Maioria das pesquisas abordam somente a linguagem Java. É necessário, portanto, evidenciar quais fatores afetam a co-evolução de teste em múltiplas linguagens;
- Maioria das pesquisas apresentam dataset pequeno, o que pode representar um obstáculo para a generalização dos resultados;

Dessa forma, há a necessidade de uma investigação aprofundada sobre práticas de co-evolução de testes em projetos reais e diversos, objetivando investigar seu impacto no desenvolvimento de software de qualidade.

1.3 Questões de Pesquisa

Nesta seção são apresentadas as questões de pesquisa que levaram às investigações deste trabalho. O estudo é conduzido tendo como objetivo prover respostas para três questões de pesquisa:

- QP_1 - *Como testes evoluem em projetos de software?*
  Motivação: Nosso objetivo nesta primeira questão é investigar a representatividade do teste junto ao código de produção e como ele se comporta mediante as modificações realizadas durante o desenvolvimento do projeto. Em seguida, buscamos identificar a existência de padrões de teste junto à evolução do projeto, a fim de fornecer percepções sobre como os desenvolvedores vem conduzindo o teste no desenvolvimento. Esta investigação visa fornecer informações gerais sobre os padrões de teste, como comportamentos encontrados, distribuição dos repositórios e características associadas a cada padrão.
- QP_2 - *Com que frequência o código-fonte e testes co-evoluem em projetos de software?*
  Motivação: Nosso objetivo nesta segunda questão é investigar a frequência de repositórios com forte co-evolução de teste, concentrando a análise na correlação da evolução do LOC de teste junto ao LOC de produção. Esta investigação visa identificar repositórios com fortes evidências de equilíbrio no desenvolvimento orientado a testes.

- QP_3 - *Que características distinguem projetos onde há co-evolução de código de teste de projetos onde essa prática não é comum?*
  Motivação: Nesta última questão, nós investigamos se *commits*, colaboradores, *forks* e *issues* possuem diferenças significativas entre os repositórios com alta co-evolução e os com fraca co-evolução. Nesta investigação estamos interessados em observar se a co-evolução de teste pode refletir em outras características do repositório e se essas características podem indicar um resultado positivo ou negativo para o desenvolvimento de software.

1.4 Visão Geral da Proposta

Nossa proposta investiga como artefatos de código-fonte e teste evoluem e comparar características comuns a repositórios de *software*, buscando identificar se, e como, a co-evolução desses artefatos impactam no desenvolvimento dos projetos. Para identificar se ocorre co-evolução de teste no desenvolvimento do projeto, é necessário primeiramente identificar se existem arquivos de teste, assim como as modificações nos arquivos de teste evoluem junto com os arquivos de produção. Com isso, a abordagem proposta neste trabalho é dividida em seis etapas: 1) Seleção de Repositórios e Extração de Dados; 2) Classificação de arquivos de teste; 3) Extração do histórico de proporção de teste; 4) Identificação de padrões de crescimento de teste; 5) Identificação de repositórios com co-evolução e 6) Investigação do impacto da co-evolução de teste nos repositórios.

A abordagem proposta utiliza conceitos relacionados a mineração de repositórios de software, busca de arquivos de teste através de expressão regular e clusterização de dados para descoberta de padrões de crescimento. Para detectar o nível de co-evolução de teste dos repositórios, a abordagem utiliza o Coeficiente de *Pearson* (SEDGWICK, 2012) para medir a correlação existente entre o histórico de LOC de teste e produção dos repositórios, isso permite identificar os repositórios com forte co-evolução de teste e os em que a co-evolução não se faz tão presente.

Neste trabalho temos uma abordagem mais ampla em relação às encontradas na literatura, possibilitando o estudo da co-evolução de teste em projetos desenvolvidos em cinco linguagens de programação. Os resultados provêm bons indicadores, viabilizando o uso da abordagem para trabalhos futuros e avaliação da co-evolução de teste em projetos reais.

1.5 Objetivos

Os sistemas de software, em regra geral, possuem muitas funcionalidades e tornando um desafio o teste de todas elas. Para facilitar a atividade de teste, tem-se como alternativa

realizar a automação dos processos repetitivos, ou seja, o uso de testes automatizados.

Contudo ainda existem muitos desafios relacionados à realização de testes no processo de desenvolvimento de software. A co-evolução de teste representa um equilíbrio entre alterações no código-fonte e identificação de falhas, representando assim, uma preocupação não apenas com a realização de teste, mas como fazê-lo.

O objetivo principal deste trabalho é investigar como artefatos de código-fonte e teste evoluem e como a evolução destes pode impactar no processo de desenvolvimento de software. Para que seja possível concretizá-lo, é essencial atingir os seguintes objetivos específicos:

- **Construir uma grande base de dados para investigação da co-evolução de testes em projetos de desenvolvimento de software.**

  - Selecionar projetos desenvolvidos nas linguagens Javascript, Java, Python, PHP e Ruby, hospedados no Github;

  - Obter dados do histórico de versionamento dos projetos;

  - Aplicar técnicas de classificação de arquivos de testes;

- **Identificar os padrões de crescimento de teste**

  - Aplicar técnicas de clusterização para identificação de padrões de co-evolução nos repositórios investigados;

- **Identificar repositórios com forte co-evolução de teste**

  - Analisar a correlação entre LOC de teste e LOC de produção dos repositórios investigados;

- **Identificar o impacto da co-evolução de teste**

  - Analisar a co-evolução de teste com outros indicadores ligados a projetos no GitHub;

1.6 Estrutura do Trabalho

Este trabalho está estruturado em seis capítulos.

O Capítulo 1 mostra a introdução, onde são apresentados o contexto e a motivação para a escolha do tema do projeto, definição do problema, as questões e visão geral da proposta de investigação que são importantes para organizar as práticas de pesquisa. São apresentados também o objetivo geral e específicos.

O Capítulo 2 apresenta o referencial teórico e conceitos importantes para o entendimento de testes de software e co-evolução de teste.

No Capitulo 3 é fornecido detalhes sobre as publicações relevantes relacionadas ao tema desta pesquisa.

No Capítulo 4 é descrito as etapas realizadas, como também detalhada as técnicas utilizadas.

No Capítulo 5 são descritas as análises e os resultados obtidos a partir da investigação relacionadas às questões de pesquisa e objetivos deste trabalho.

Por fim, no capítulo 6 é apresentada a conclusão, assim como principais contribuições, limitações, trabalhos futuros, e publicações obtidas.

2 Referencial Teórico

Neste capítulo são apresentados os principais conceitos relacionados ao trabalho. Primeiramente, apresenta-se conceitos sobre teste de software. Em seguida são apresentados conceitos fundamentais sobre testes automatizados. Por fim, são apresentados conceitos sobre a co-evolução entre código de produção e código de teste.

2.1 Teste de Software

Existem diversas abordagens sobre a qualidade no desenvolvimento de software. Geralmente, existem duas atividades que são realizadas quando se trabalha no desenvolvimento concorrente, elas são: testes estáticos e testes dinâmicos (RIOS; MOREIRA, 2006). Os testes estáticos referem-se às revisões de código-fonte e documentação. Já os testes dinâmicos referem-se a verificações durante a execução do software.

O desenvolvimento de software possui um conjunto de atividades relacionadas à sua existência, desde o planejamento até o seu desuso. Existem etapas comuns entre os processos de desenvolvimento, como: especificação de requisitos, desenvolvimento do sistema, validação e verificação e evolução do software (SOMMERVILLE, 2011). A forma como essas atividades são executadas depende do modelo adotado.

O teste de software encontra-se na atividade de validação e verificação de software, a atividade de validação envolve outros processos como verificações, inspeções e revisões. Contudo, ocorre a predominância de testes. O teste de software é definido como o processo de executar o software de maneira controlada para verificar se ele se comporta conforme o especificado, sendo fundamental para a sua avaliação durante seu desenvolvimento (CRESPO et al., 2004).

Dessa forma, no ciclo de desenvolvimento de software, a atividade de teste é uma etapa relevante (VIDÁCS; PINZGER, 2018; WHITE; KRINKE; TAN, 2020). O teste é uma análise dinâmica, ou seja, existe a necessidade de colocar o software em execução para avaliar o comportamento. O principal objetivo da atividade de teste é identificação e correção de *bugs*, porém existem outras razões para realizar testes, tais como: reduzir riscos, garantir funcionalidades e garantir confiabilidade.

O *Bug* é a manifestação do defeito, originado de um erro, que ocorre durante a execução do programa (FIRESMITH, 2014). Detectar todos os defeitos de um software em seu período de desenvolvimento é uma tarefa quase impossível, mas isso não deve ser visto como fator desmotivador, é recomendado investir em teste. Como publicado no jornal *National Institute of Standards and Technology (NIST)*, pesquisas relacionadas mostram

que detectar defeitos no período de desenvolvimento reduz de forma significativa o custo do software (PLANNING, 2002).

Adicionalmente, testes podem ser utilizados para medir a qualidade do Software (POL; TEUNISSEN; VEENENDAAL, 2002; AGARWAL; TAYAL; GUPTA, 2010), pois, a qualidade do software pode ser associada ao número de erros encontrados. Um software com poucos erros possui muita qualidade. É importante reforçar a necessidade de identificar e corrigir possíveis erros ainda no ambiente de desenvolvimento, evitando prejuízos, transtornos e frustrações para os usuários finais.

Dependendo do processo adotado no desenvolvimento de software, a atividade de teste pode ser organizada de diferentes maneiras. A Figura 1 apresenta o modelo cascata que possui como característica a execução das atividades de modo sequencial. Os estágios desse modelo são: 1) definição de requisitos, 2) projeto de sistema e software, 3) implementação e teste unitário, 4) integração e teste de sistema e 5) operação e manutenção.

O modelo cascata é um modelo dirigido a planos, primeiro deve planejar todas as atividades que devem ser realizadas antes de começar a trabalhar nelas. Por causa dos custos de produção e aprovação de documentos, as iterações podem ser dispendiosas e envolver significativo retrabalho (SOMMERVILLE, 2011). Além disso, esse modelo possui como principal limitação a realização do teste após todo o Software ser desenvolvido. A identificação de erros no modelo cascata torna-se bastante custosa, uma vez, que os defeitos podem se propagar por todo o produto e podem causar perda de praticamente todo o trabalho. Dentre outras limitações, destacam-se as relatadas por Pressman e outros (PRESSMAN, 2005): 1) Projetos reais raramente seguem o fluxo sequencial; 2) Em geral, é difícil para clientes identificarem todos os requisitos explicitamente; e 3) O cliente precisa ter paciência, uma vez que a primeira versão executável do produto somente estará disponível no fim do projeto.

Para a maioria dos sistemas, o modelo cascata não oferece custo-benefício significativo sobre outras abordagens para o desenvolvimento de sistemas (SOMMERVILLE, 2011). Para lidar com as limitações do modelo cascata, foi proposto um novo modelo de processo denominado modelo incremental (Figura 2).

O modelo incremental facilita o desenvolvimento de sistemas, pois, é baseado na ideia de entregar ao cliente versões parciais do produto, para que sejam validadas e colocadas em produção. Ao entregar uma versão, é iniciado uma nova iteração e são realizadas várias iterações até que o sistema esteja adequadamente desenvolvido. O modelo incremental representa uma evolução em relação a realização de testes, a cada iteração é realizado descrição do esboço, especificação, desenvolvimento e validação. Ao desenvolver um software de forma incremental, é mais barato e mais fácil fazer mudanças no software durante seu desenvolvimento (SOMMERVILLE, 2011). O modelo incremental permite a identificação de erros de forma antecipada, diminuindo riscos e custos e aumentando a

confiabilidade do Software.

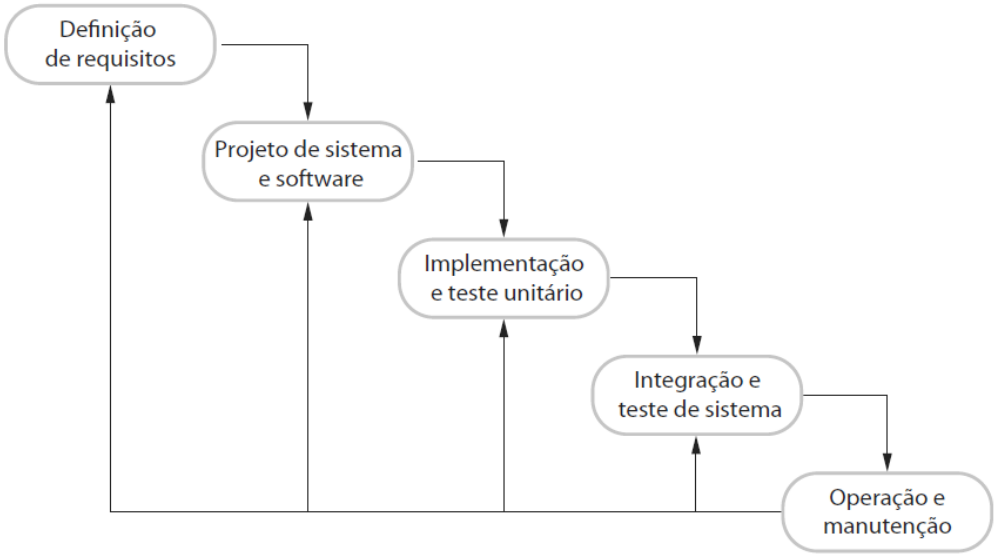


Figura 1 – A atividade de teste no modelo cascata (SOMMERVILLE, 2011)

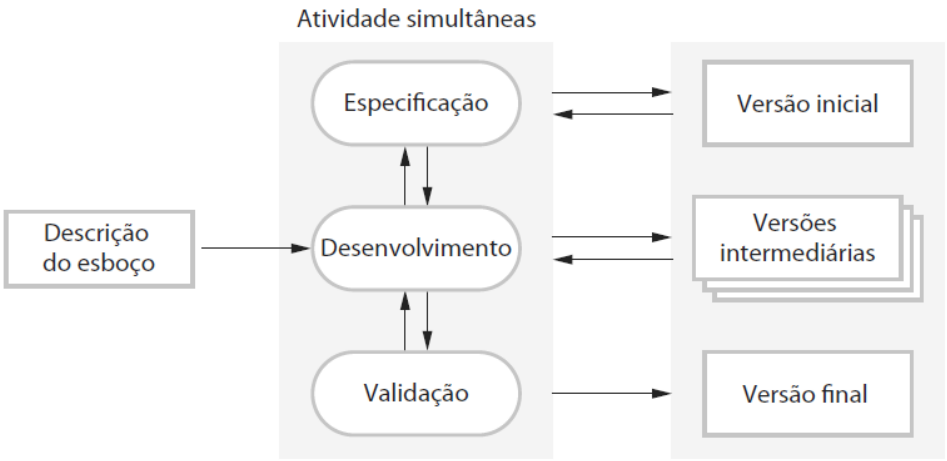


Figura 2 – A atividade de teste no modelo incremental (SOMMERVILLE, 2011)

É frequente durante o desenvolvimento de software os requisitos de negócios mudarem devido às necessidades de usuários mudarem, ou a equipe de desenvolvimento sofrer alterações e novas ameaças surgirem (PRESSMAN, 2007). Os métodos ágeis têm predominado no mercado, pois, utilizam modelo de desenvolvimento de software baseado no modelo iterativo, ou seja, com entregas de versões e com diversas iterações até o software está completo.

Neste contexto, todo software pode ser considerado como um projeto em desenvolvimento, um sistema em contínua mudança e evolução. Assim, o software precisa ser desenvolvido de forma que facilite a sua manutenção e evolução (LEHMAN, 1996). Com isso, surge a necessidade de o código de teste acompanhar a evolução do código-fonte, garantindo que as novas funcionalidades e correções também sejam testadas. A co-evolução

de teste pode ser analisada através da relação do histórico de mudanças entre o código-fonte e o código de teste em um projeto de software (MARSAVINA; ROMANO; ZAIDMAN, 2014).

Como visto, a atividade de teste difere conforme o modelo adotado no desenvolvimento de software. No modelo em cascata as atividades são organizadas em sequência, enquanto que no desenvolvimento incremental são intercaladas. Além disso, o teste também possui diversos estágios (Figura 3).

A Figura 3 apresenta como os testes podem ser executados, desde o estágio mais baixo (componente) ao mais alto (aceitação). O primeiro estágio contempla o teste de componente, que são os testes realizados pelos desenvolvedores. Os componentes podem ser testados de forma isolada (teste de unidade) ou agrupamentos coerentes de funções ou classes de objetos. No segundo estágio, o teste de sistema avalia o comportamento dos componentes quando interagem. Por fim, o teste de aceitação, com base em dados fornecidos pelo cliente, possui como objetivo avaliar se o sistema está pronto para uso operacional.

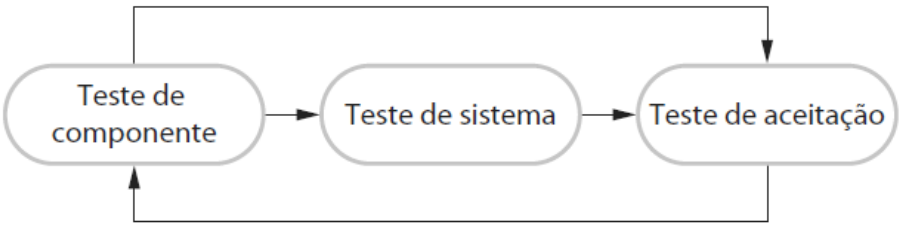


Figura 3 – Estágios de testes (SOMMERVILLE, 2011)

2.2 Testes Automatizados

Na Tabela 2 é apresentada a comparação entre os tipos de teste. É verificado as características dos testes manuais e automatizados em relação ao 1) processo de teste, 2) vulnerabilidades e ataque ao SGBD e 3) relatórios dos testes. Percebemos que os testes automatizados visam facilitar tarefas repetitivas, especialmente, quando envolvem testes de regressão.

Em testes automatizados são criados *scripts* que são executados toda vez que o sistema é testado. Nos testes automatizados são utilizadas ferramentas e *scripts*, esses visam repetir ações predefinidas comparando os dados de teste.

Dessa forma, para auxiliar nessa atividade é comum o uso técnicas e ferramentas que buscam maior automação no processo de teste, permitindo diminuir o esforço sem comprometer a qualidade dos testes (BERNARDO; KON, 2008). Dentre essas técnicas e ferramentas, se destacam as ferramentas de testes de unidade (também conhecidos como

Tabela 2 – Comparação entre testes manuais e testes automatizados (STEFINKO; PISKOZUB; BANAKH, 2016) (Livre tradução do autor)

	Manual	Automatizado
Processo de teste	Manual, não possui padrão no processo; Requer muito esforço e dinheiro; Exige alto custo para customização;	Rápido, possui padrão no processo; Facilmente aplicado para testes repetitivos;
Vulnerabilidades e ataque ao SGBD	Manual; Precisa acesso ao banco de dados; Precisa reescrever o código de ataque;	Ataque ao banco de dados é mantido e atualizado; Reuso dos códigos de ataque;
Relatórios	Manual;	Relatórios são automatizados e customizados;

testes unitários). Desenvolvida inicialmente para a linguagem Smalltalk por Kent Back, ela inspirou o desenvolvimento de *frameworks* de testes para diferentes tecnologias e linguagens de programação, conhecido como família *xUnit*[1].

Existem diversos *frameworks* que facilitam a automatização de testes de unidade, tornando o teste de unidade uma forma mais favorável para desenvolvimento de teste. Esses *frameworks* definem um conjunto de componentes e regras para escrever casos de teste. Assim, os desenvolvedores se utilizam desses componentes, para auxiliar na execução de testes para as unidades de código que produzem. Entre as principais vantagens da automatização de testes encontram-se: confiabilidade, velocidade e repetibilidade. Os testes automatizados permitem ser executados e repetidos sempre que necessário.

Seguindo essa proposta, os testes automatizados a nível de teste unitário são os mais fáceis de serem realizados, tornando mais confortável para a implementação dos desenvolvedores. Contudo essa não é a única forma, a automatização de teste pode ocorrer nos seguintes níveis, unitário, de serviços, e de interface do usuário (COHN, 2010).

Contudo, a criação de testes automatizados ainda possui alguns desafios. A criação de testes automatizados esbarra em fatores como tempo de entrega do recurso ou produto, falta de conhecimento da equipe ou mesmo interesse dos gestores em investir nessa prática. Mas é importante ressaltar que negligenciar ou colocar em segundo plano o tempo para as atividades de teste afetam diretamente as atividades de manutenção quando o software estiver em produção. Além disso, incentivar testes automatizados ajuda na interação entre os desenvolvedores da equipe, bem como minimizar conflitos no desenvolvimento e a existência ou adição de novas falhas (*bugs*) no software.

Por fim, observa-se que a automação de teste assegura qualidade ao software que é produzido de forma incremental e iterativa, pois assegura a entrega em tempo hábil e a presença de testes a cada ciclo de desenvolvimento (OLUWOLE, 2013). Os testes automatizados são úteis para substituição de tarefas repetitivas e garantem que novas modificações não adicionem falhas em estruturas já existentes, mas não significa que testes

[1] <https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks>

manuais também não devam ser realizados (BERNER; WEBER; KELLER, 2005).

2.3 Co-evolução de Código de Teste e Produção

O software evolui (LEHMAN, 1979) e essa evolução envolve alterações no código de produção para atividades de correções e evolução do software. Essas alterações no código de produção, visando a qualidade de software, devem ser acompanhadas da co-evolução de outros artefatos, como requisitos, documentação e teste (MENS et al., 2005). Assim, para um processo de desenvolvimento visando qualidade o código de teste também deve evoluir junto com o código de produção (ZAIDMAN et al., 2011).

Várias situações podem ocasionar erros no processo de desenvolvimento concorrente de software, como: conflitos em processos de *merge*, sobrescrever mudanças com novos *commits*, reaparecimento de erros, assim, vê-se a importância do SCV em gerenciar e controlar essas situações, facilitando o processo de desenvolvimento de software em equipe.

O SCV pode ser utilizado para armazenar diversos artefatos, mas quando se trata de repositório de software a parte principal é o próprio código fonte do projeto (SPINELLIS, 2005). Entre os dados armazenados pode-se obter versões de arquivos; informação de arquivos e linhas adicionadas, removidas ou alteradas; autores e datas de *commits*; diretório do arquivo e outras informações. O *commit* é a ação de persistir as alterações no SCV, alterações na base de armazenamento somente são possíveis com a submissão de um novo *commit*. Segundo (CHACON; STRAUB, 2014), dentre os principais benefícios do uso de um SCV estão: 1) Permitir recuperar uma versão específica do software; 2) Auxiliar o desenvolvimento concorrente; 3) Compartilhar o código fonte; 4) Visualizar histórico de mudanças (adição, remoção ou modificação).

Neste contexto, no desenvolvimento de projetos de software tem-se que considerar a sua natureza dinâmica. Os Softwares precisam de correções e adições de novas funcionalidades, potencializando a inserção de novos *bugs*. Assim, não há garantias que as funcionalidades já existentes no sistema continuarão a funcionar mediante essas mudanças no código-fonte.Com isso, surge a necessidade de o código de teste acompanhar a evolução do código de produção, garantindo que as novas funcionalidades e correções também sejam testadas. A co-evolução de teste pode ser analisada através da relação do histórico de mudanças entre o código de produção e o código de teste em um projeto de software (MARSAVINA; ROMANO; ZAIDMAN, 2014).

A co-evolução de teste ajuda a compreender padrões no desenvolvimento do projeto, podendo ser útil para melhorias no contexto da manutenção do software. A co-evolução auxilia a compreender as atividades de manutenção, qualidade e evolução do software (LEVIN; YEHUDAI, 2017).

2.4 Considerações Finais

Este capítulo apresenta de forma resumida os principais conceitos utilizados neste trabalho. Inicialmente, foram apresentados conceitos relacionados a Teste de Software e, em seguida, testes automatizados foram abordados.

Dessa forma, percebe-se oportunidades para pesquisas sobre o uso de testes no desenvolvimento de software. Procurando conhecer com mais profundidade o campo de estudo, o próximo capítulo apresenta a revisão da literatura de diversos trabalhos sobre a co-evolução de teste.

3 Trabalhos Relacionados

Este capítulo apresenta uma revisão da literatura com o objetivo de identificar pesquisas realizadas sobre teste de software, especificamente sobre a co-evolução de teste, para entender quais técnicas, métodos e ferramentas são utilizadas.

3.1 Seleção dos Trabalhos

A busca de trabalhos relacionados foi realizada no repositório de computação *Scopus* [1], pois, é uma reconhecida e significante base de pesquisa que agrega diversas bibliotecas digitais. A formação da *String de busca* é baseada em palavras-chave, sinônimos ou palavras equivalentes relacionadas ao estudo. Neste trabalho foi definido como palavras-chaves relacionadas *Mining software repositor** e *software test**. Pesquisou-se trabalhos que contenham no título, resumo ou palavras-chave a *String de busca*, resultando na seguinte busca avançada *TITLE-ABS-KEY (("Mining software repositor*") AND ("software test*"))*. Obteve-se 60 trabalhos encontrados na busca da base de dados. Na seleção dos trabalhos, aplicou-se os critérios de inclusão e exclusão na leitura do título e resumo dos trabalhos, restando 18 trabalhos para a leitura completa.

Após a análise dos artigos foram identificados cinco trabalhos relacionados. A tabela 3 lista as principais características dos trabalhos relacionados que nos inspiraram.

Tabela 3 – Trabalhos relacionados

Autor	**Foco da pesquisa**	**Aborda Co-evolução de teste**
(MARSAVINA; ROMANO; ZAIDMAN, 2014)	Identificação de padrões de co-evolução de teste	Sim
(VIDÁCS; PINZGER, 2018)	Identificação de padrões de co-evolução de teste	Sim
(ZAIDMAN et al., 2011)	Visualização da co-evolução de teste	Sim
(LEVIN; YEHUDAI, 2017)	Co-evolução de teste e mudanças semânticas	Sim
(GONZALEZ et al., 2017)	Manutenção de software e teste de unidade	Não

3.2 Descrição dos Trabalhos Relacionados

Um resumo dos estudos abordando contexto, *dataset*, principais resultados é apresentado a seguir.

Marsavina e outros (MARSAVINA; ROMANO; ZAIDMAN, 2014) apresentam uma abordagem para investigação de padrões de co-evolução de teste e código de produção. 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 5 projetos *open source* para construção do *dataset*, todos desenvolvidos na linguagem Java. Os resultados

[1] <https://www.scopus.com/home.uri>

obtidos mostram os padrões de co-evolução encontrados. Dois padrões identificam se as classes/métodos de produção são adicionadas junto às classes/métodos de teste, enquanto um dos padrões verifica se as classes de teste são removidas junto com as de classes de produção. Adicionalmente, outros padrões verificam se os casos de teste são alterados de acordo com as modificações de atributos/métodos da classe de produção e se o código de teste é modificado de acordo com mudanças em instruções condicionais no código de produção Cada padrão possui uma subdivisão de positivo ou negativo, o positivo significa que foi encontrado co-evolução e o negativo sua ausência. Foi verificado que os padrões positivos são mais propensos a ser encontrados em sistemas de software totalmente testados.

Vidács e outros (VIDÁCS; PINZGER, 2018) realizam uma expansão do trabalho de (MARSAVINA; ROMANO; ZAIDMAN, 2014), buscando identificar padrões de co-evolução de teste através de regras de associação. A abordagem é uma replicação do estudo anterior, com análises estendidas das propriedades do projeto, como experimentos na ponderação e análise de composição de *commits*. No conjunto de dados foi utilizado apenas 1 projeto na linguagem Java. Os resultados obtidos mostram que o código de produção e teste crescem em sincronia no projeto analisado. Foi verificado que em muitos casos código de produção e teste são alterados separadamente.

Zaidman e outros (ZAIDMAN et al., 2011) realizam um estudo sobre co-evolução de teste e código de produção. A abordagem apresenta três visões para o estudo da co-evolução: i) Histórico de Mudanças, ii) Histórico de crescimento, iii) Evolução da Cobertura de Teste. Foram utilizados 3 projetos, 2 *open source* e 1 projeto privado de uma empresa de desenvolvimento na construção do *dataset*. Todos os projetos foram desenvolvidos na linguagem Java. O Histórico de Mudanças tem como objetivo verificar se o código de produção possui um código de teste associado (teste de unidade) e se o arquivo de produção e de teste são adicionados e alterados juntos. O Histórico de Crescimento tem como objetivo identificar se o projeto possui co-evolução de teste síncrona ou faseada, se antes de uma *release* maior ocorre maior implementação de teste, e se existem evidências de desenvolvimento orientado a testes. A Evolução da Cobertura de Teste tem como objetivo indicar a qualidade do teste. A qualidade do teste é identificada através da relação do percentual de cobertura pelo percentual da proporção de teste. Os resultados obtidos mostram que embora o código de teste e produção possam ser commitados em momentos diferentes (faseado) ou juntos (síncrono), o padrão de desenvolvimento síncrono foi encontrado com maior frequência.

Levin e outros (LEVIN; YEHUDAI, 2017) exploram a co-evolução código de teste e produção através de mudanças semânticas, ou seja, investiga se adicionar ou remover funções/métodos/classes são ações significativas na manutenção de teste. A abordagem utiliza mineração de repositórios e modelos preditivos para fazer a relação entre manutenção

de teste, manutenção de código de produção e mudanças semânticas. Foram utilizados no *dataset* 61 projetos *open source* desenvolvidos na linguagem Java. Os resultados obtidos mostram que na maior parte das vezes os desenvolvedores realizam correções de código sem realizar manutenção de teste no mesmo *commit*. Foi verificado que o número de métodos e classes de teste podem ser previstos utilizando os modelos da abordagem e que o tipo de manutenção aumenta ou diminui a chance de a manutenção do teste acontecer.

Gonzalez e outros (GONZALEZ et al., 2017) realizaram um estudo em larga escala explorando padrões de atributos de manutenibilidade através da manutenção de teste de unidade. A abordagem utiliza mineração de repositório de software, identificação de arquivos de teste e detecção de padrões de teste. Foram utilizados 82.447 projetos *open source* no *dataset*. Foram analisadas diversas linguagens de programação, como Javascript, Java, Ruby, Pyhton, PHP, C++, C, C#, Shell, Objetive-C, e outras. Os resultados obtidos mostram que os projetos *open source* não adotam padrões de teste que podem ajudar com atributos de manutenibilidade. Foi verificado que projetos menores aplicam padrões com mais frequência e que a adoção de padrões é uma decisão individual do desenvolvedor. É importante ressaltar que esse trabalho não trata especificamente de co-evolução de teste.

Dentre os trabalhos relacionados, os trabalhos de Gonzalez (GONZALEZ et al., 2017) e Zaidman (ZAIDMAN et al., 2011) contribuíram significativamente com o estudo desenvolvido nesse estudo. A abordagem para identificação de casos de teste proposto por Gonzalez e outros (GONZALEZ et al., 2017) utiliza expressões regulares baseadas no Catálogo de *Frameworks* de automação de testes. A abordagem proposta nesse trabalho foi escolhida por considerar as linguagens Javascript, Java, Python, PHP e Ruby e ter apresentado bons resultados em um grande número de repositórios. A abordagem proposta por Zaidman e outros (ZAIDMAN et al., 2011) influenciou este trabalho, pois é o primeiro trabalho a nos apresentou a forma de estudar co-evolução de teste em projetos de software.

3.3 Considerações finais

Os trabalhos apresentados mostram que a co-evolução de teste tem sido estudada predominantemente em projetos de software da linguagem Java. Eles investigam, como por exemplo, as formas de visualização da co-evolução, como também quais os padrões de co-evolução.

Este trabalho assemelha-se e se inspira em conceitos e abordagens definidas nesses trabalhos anteriores. Contudo, neste trabalho é realizada a análise de um número maior de repositórios e de linguagens de programação (JavaScript, Java, Python, PHP e Ruby). O trabalho também utiliza a distribuição de proporção de teste para classificar os repositórios com forte co-evolução de teste e realiza a comparação da co-evolução com algumas características de repositórios de software.

4 Metodologia

Neste capítulo descrevemos a metodologia adotada para a análise da co-evolução de teste em repositórios de software. A Figura 4 apresenta as etapas da abordagem proposta. 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 do impacto da co-evolução de teste nos repositórios.

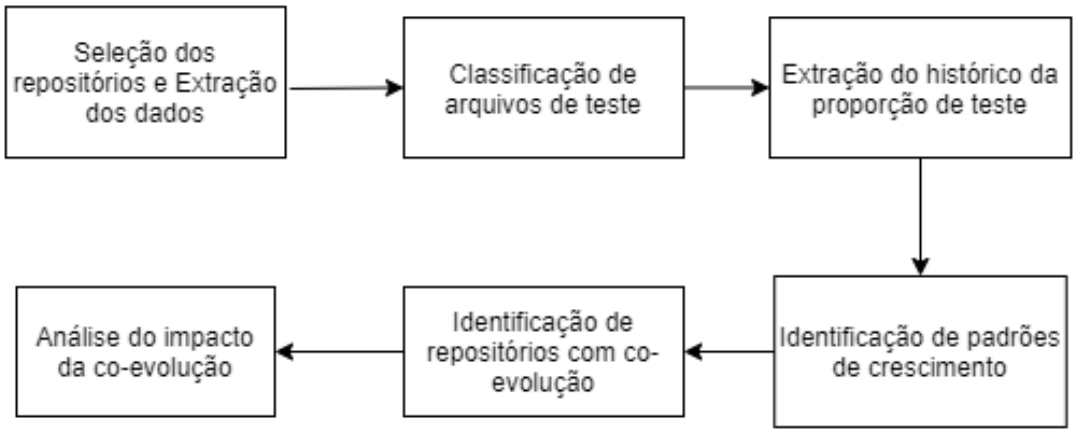


Figura 4 – Fluxo Detalhado da Abordagem

Na primeira etapa foi realizado o download de 3.000 projetos hospedados no GitHub, selecionados dentre os mais populares em cinco linguagens de programação. A partir de então, foram extraídos dados do histórico de desenvolvimento, a partir de seus repositórios Git, para a coleta de informações utilizadas na mineração. Na segunda etapa é realizada a identificação dos arquivos de teste, seguindo um conjunto de regras propostas por (GONZALEZ et al., 2017). Como resultado, obtém-se dois conjuntos, os arquivos de código de produção e os arquivos de código de teste. Então, na terceira etapa, ocorre a extração do histórico da proporção de teste ao realizar a coleta do número de linhas de código (LOC) de produção e de teste dos repositórios. Posteriormente, na quarta etapa, ocorre a identificação de padrões de crescimento utilizando as séries temporais com o histórico da proporção de teste dos repositórios. Utiliza-se o algoritmo KSC(YANG; LESKOVEC, 2011) para realizar a clusterização das séries temporais baseado na proporção de teste. Para cada k da iteração, o KSC define os *centroids* e associa cada repositório a um *cluster* baseado na forma das série temporal da proporção de teste do repositório. Na quinta etapa, ocorre a identificação da co-evolução, a partir dos dados extraídos, via criação de séries temporais. Por fim, na sexta etapa são comparados dados gerais, tais como número de *commits*, colaboradores, *forks* e *issues* entre os repositórios classificados com forte co-evolução e os com fraca co-evolução.

4.1 Seleção de Repositórios e Extração de Dados

Esta primeira etapa consiste em realizar a extração de dados de populares projetos hospedados no GitHub. O GitHub é uma plataforma que disponibiliza, além do código-fonte do projeto, diversas informações sobre alterações relacionadas ao desenvolvimento do mesmo. Além disso, não é obrigatório fazer download dos repositórios para ter acesso às informações do desenvolvimento, o GitHub possui uma API que provê esses dados. Contudo, no nosso caso, para análise do conteúdo dos arquivos de código durante as diversas versões dos mesmos, fizemos o clone dos repositórios. Para cada arquivo de código-fonte dos repositórios são salvos os seguintes metadados nesta etapa: *hash* do *commit*, nome e e-mail do desenvolvedor, data do *commit*, *status* da alteração, e por fim, caminho completo do arquivo.

4.1.1 Seleção de repositórios

O GitHub é uma popular plataforma para hospedagem de repositórios. Além de ferramentas para gerenciamento de versões de código, são disponibilizados recursos comuns de redes sociais para conectar e impulsionar a interação entre os desenvolvedores. Atualmente, o GitHub possui mais de 73 milhões de usuários e mais de 200 milhões de repositórios.

O *dataset* desta pesquisa foi construído em dezembro de 2019 através do download de 3.000 projetos hospedados no GitHub. Para a seleção dos projetos foi utilizado a API[1] disponibilizada pelo próprio GitHub. 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[2].

As estrelas são um dos recursos de interação social disponibilizados pelo GitHub, são semelhantes ao *like* de outras redes sociais (BORGES, 2018). Dar uma estrela a um repositório é a maneira do usuário do GitHub mostrar interesse no trabalho realizado pelo desenvolvedor do repositório, como também facilita a busca posterior desse repositório. O número de estrelas é comumente utilizado em diversos tipos de listas pela plataforma.

As Tabelas 4 e 5 mostram características dos repositórios relacionadas a última atualização e idade dos repositórios. Podemos constatar que são projetos predominantemente ativos, recentemente atualizados e a maioria possui entre 2 e 10 anos.

A Tabela 6 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.

[1] https://docs.github.com/pt/rest
[2] https://github.com/, acessado em 19/01/2022

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 Figura 5 apresenta as distribuições dos valores das características contidas na Tabela 6. Podemos concluir que o nosso *dataset* possui projetos maduros e com um grande número de *commits*, colaboradores, *forks*, *issues*, estrelas e que estes sistemas possuem um grande volume de dados para serem analisados (tamanho). Os detalhes são apresentados e discutidos no Capítulo 5.

Tabela 4 – Última Atualização dos repositórios

Ano	**Qtde**	**%**
2019	3	0.1
2020	2997	99.9

Tabela 5 – Idade dos repositórios

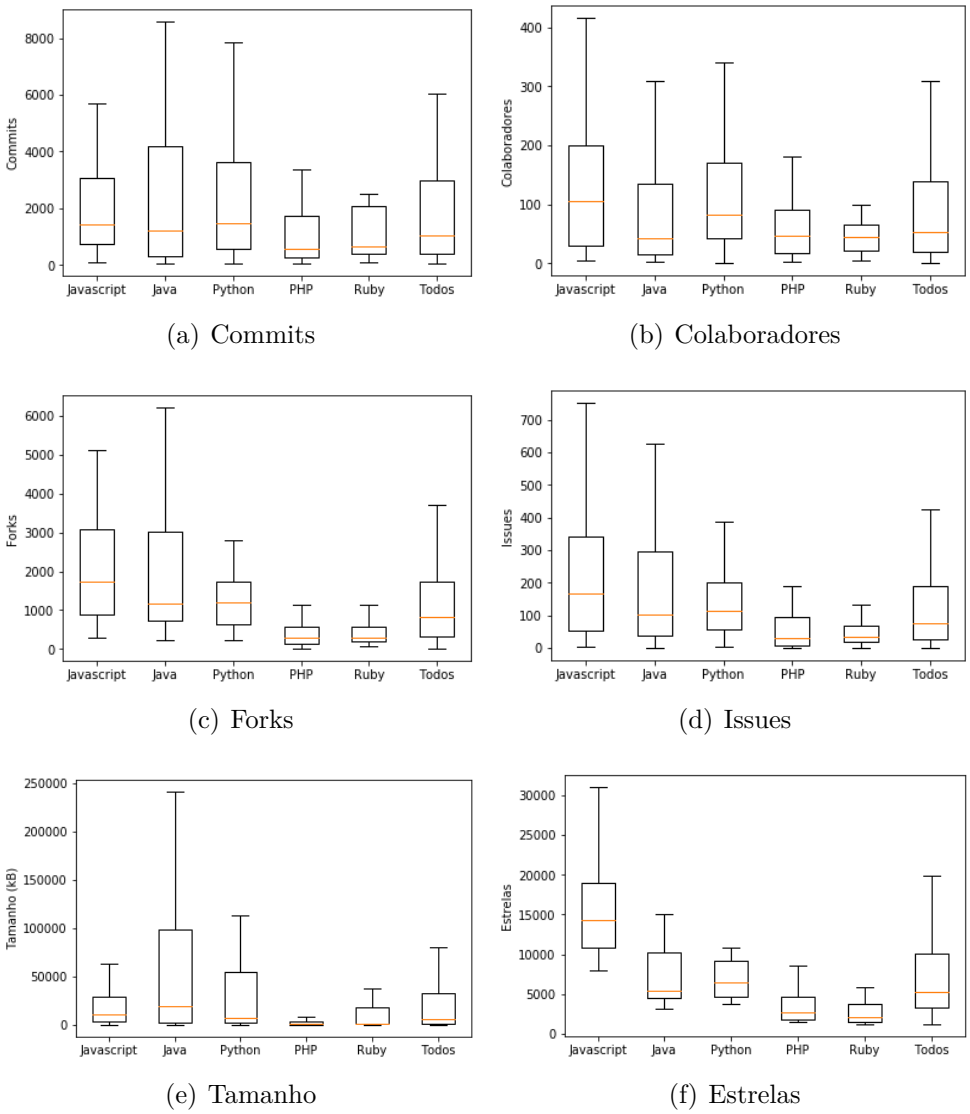
Característica	**Qtde**	**%**
Possuem idade <= 2 anos	200	6.7
Possuem 2 < idade <= 5 anos	1001	33.4
Possuem 5 < idade <= 10 anos	1572	52.4
Possuem idade > 10 anos	226	7.5

Tabela 6 – Dados dos repositórios por linguagem (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**

4.1.2 Extração dos dados

Para obter os dados sobre o histórico do versionamento dos projetos utilizou-se o comando "*git log –name-status –follow -p – varPathFile > varPathLogCommit*" para obter o *hash* do *commit*, nome do autor, e-mail do autor, data do *commit*, status do arquivo, caminhos do arquivo (Tabela 7). No comando, *varPathFile* representa uma variável com



(a) Commits (b) Colaboradores

(c) Forks (d) Issues

(e) Tamanho (f) Estrelas

Figura 5 – Dados dos repositórios por linguagem

o caminho do arquivo analisado e *varPathLogCommit* representa uma variável com o caminho em que é salvo o log.

Pode-se verificar na Tabela 7 que o arquivo Avatar.js foi adicionado no projeto em 23 de fevereiro de 2017 às 23h pelo desenvolvedor Mon***9 através do commit 68c3d2***c12fe5 e modificado em 14 de outubro de 2019 às 0h e 45min pelo desenvolvedor Ky***ch. Esses dois registros de *commits* foram selecionados do histórico de versionamento do arquivo Avatar.js para mostrar os dados que foram extraídos nessa etapa.

4.2 Classificação de Arquivos de Teste

A etapa de classificação de arquivos de teste ocorre após a extração de dados dos repositórios. É importante ressaltar que a abordagem e *scripts* utilizados para a classificação de arquivo de teste são de um trabalho relacionado (GONZALEZ et al., 2017), já os *scripts*

Tabela 7 – Amostra de dados extraídos do SCV. Hash, autor e e-mail foram ocultados.

Hash	68c3d2***c12fe5	4a19e4c***99bdb
Autor	Mon***9	Ky***ch
E-mail	Mon***9@users.noreply.github.com	kr***rk@gmail.com
Data	Thu Feb 23 23:00:50 2017 -0800	Mon Oct 14 00:45:03 2019 -0400
Situacao	ADDED	MODIFIED
Caminho	src/avatar/Avatar.js	src/avatar/Avatar.js

utilizados para salvar os metadados da classificação e realizar análises posteriores são dos autores deste trabalho. Os metadados são: nome completo do repositório, caminho completo do arquivo, se é de teste, se possui *import* de teste, e por fim, se possui chamada de função de teste.

4.2.1 Implementação da etapa

Como dito, nesta etapa são identificados os arquivos de teste presente nos projetos do *dataset* e para isso foi feito uma revisão da literatura sobre técnicas e práticas de classificação de arquivos de teste, então decidiu-se adotar a abordagem proposta por Gonzales e outros (GONZALEZ et al., 2017). 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 (representado na tabela 8).
- 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.

É relevante destacar que foi necessário fazer uma adição de expressões regulares na Tabela 8, pois nossos primeiros experimentos apresentaram resultados muito baixos para a linguagem Ruby e, após verificação manual de alguns projetos, identificou-se que para a linguagem Ruby o algoritmo de busca não estava reconhecendo quando o *import* do *framework* de teste era realizado através de um *helper*. Ou seja, não identificava quando o arquivo fazia o *import* de um *helper* que fazia o *import* do *framework* de teste. Dessa forma, foram adicionadas as expressões regulares (“|’)[\w]*test[\w]*(“|’) e (“|’)[\w]*spec[\w]*(“|’) para a linguagem Ruby no Catálogo de *frameworks* de automação de teste. Os resultados

obtidos após a modificação no catálogo, como apresentado no Capítulo 5, mostram que repositórios Ruby tem a distribuição do percentual de testes similar a Java e Python.

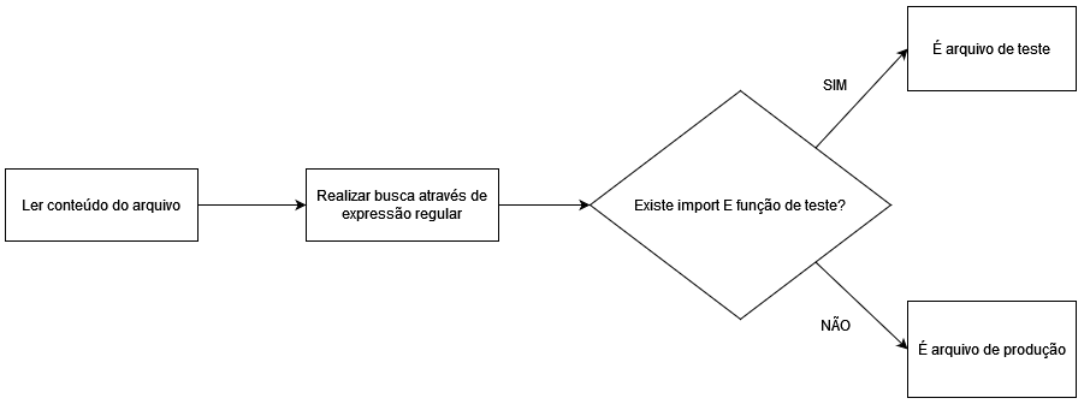


Figura 6 – Fluxo da classificação de arquivo de teste (GONZALEZ et al., 2017)

```
def test_include(testLookup, fileExtension, fileContents):
    """
        testIncludeVote determines if a file references a testing framework within
        it and returns a 1 if that is the case and a 0 otherwise

        Parameters: includeLookup - dictionary that stores the test framework
                    reference by file extension
                    file - Path object representing a path to a file
        Return: 0 or 1 if the file contains a reference to a test framework
    """
    includeList = testLookup[fileExtension]
    for includeToken in includeList:
        #print(includeToken.tokenString)
        includePattern = re.compile(includeToken.tokenString)
        match = includePattern.search(fileContents, re.IGNORECASE|re.MULTILINE)

        if(match):
            #print("match: " + str(includeToken.framework))
            return 1

    return 0
```

Figura 7 – Código em Python da busca de *import* de teste (GONZALEZ et al., 2017)

```
def test_keyword(keywordLookup, fileExtension, fileContents):
    """
        testIncludeVote determines if a file references a testing framework within
        it and returns a 1 if that is the case and a 0 otherwise

        Parameters: keywordLookup - dictionary that stores the keywords
                    by file extension
                    file - Path object representing a path to a file
        Return: 0 or 1 if the file contains a keyword
    """
    if (fileExtension in keywordLookup):
        keywordList = keywordLookup[fileExtension]

        for keyword in keywordList:
            keywordPattern = re.compile(keyword)
            match = keywordPattern.search(fileContents, re.IGNORECASE|re.MULTILINE)
            if(match):
                return 1

    return 0
```

Figura 8 – Código em Python da busca de chamada de função de teste (GONZALEZ et al., 2017)

A Figura 6 ilustra o fluxo realizado para a classificação de arquivo de teste. Inicialmente é realizada a leitura do conteúdo do arquivo. São seguidos três passos: 1) ler o conteúdo do arquivo; 2) Realizar busca através de expressão regular; 3) Comparar se existe *import* e função de teste. O passo 1) ler o código do arquivo que está sendo analisado no momento. O passo 2) baseado no dicionário de expressões regulares realiza duas buscas no conteúdo do arquivo, a primeira verifica se o arquivo possui *import* e a segunda verifica se o arquivo possui função de teste. Baseado no dicionário de expressões regulares. Por fim, o passo 3 realiza a comparação com o conector E, se possuir *import* e função de teste do dicionário, e retorna com verdadeira a classificação de teste do arquivo.

As Figuras 7, 8 mostram o código em Python que realiza a busca de *imports* e chamadas de função através de expressão regular. Em resumo, o código recebe como entrada o dicionário de expressões regulares relacionadas a *imports* ou chamadas de função, a extensão do arquivo e o conteúdo do arquivo. Inicialmente são recuperadas do dicionário as expressões regulares associadas a extensão do arquivo, logo então, para cada expressão regular do dicionário é realizada a busca para verificar se existe padrão correspondente no conteúdo do arquivo. Caso seja encontrado retorna 1, senão retorna 0 indicando que não foi encontrado.

As Figuras 9, 10 ilustram dois arquivos classificados, respectivamente, como de teste e de produção de acordo com a abordagem adotada. Ao analisar o repositório aosp-mirror temos dois arquivos da linguagem Java denominados AccountManagerPerfTest.java e SomeService.java. O arquivo AccountManagerPerfTest possui no seu conteúdo o *import* do *framework* JUnit e a chamada da anotação *@Test* marcando uma de suas funções/métodos. O arquivo SomeService não possui em seu conteúdo o *import* do *framework* de teste e também não foram identificadas funções/métodos que representam casos de teste (por exemplo *@Test*). Nesse exemplo, seguindo a abordagem, somente o arquivo AccountManagerPerfTest é considerado de teste, pois, possui *import* para um *framework* de teste e uma função/método que representa um caso de teste.

4.3 Extração do Histórico de Proporção de Teste

O objetivo principal desta etapa é realizar a extração do histórico de proporção de teste. Inicialmente o histórico de desenvolvimento é divido em 10 partes, representando 10 recortes (O conceito de recorte será descrito detalhadamente nos próximos parágrafos). Para cada recorte é realizada classificação de teste, e após a geração dos metadados de teste, cada arquivo de código-fonte é analisado e calculado o LOC. Caso o arquivo seja de teste o LOC é somado ao LOC de teste do repositório, se não for arquivo de teste o LOC é somado ao LOC de produção do repositório. No fim do processamento é salvo dois metadados, uma lista de dados de produção e uma lista de dados de teste, cada

```java
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
@LargeTest
public class AccountManagerPerfTest {

    @Rule
    public PerfStatusReporter mPerfStatusReporter = new PerfStatusReporter();

    @Test
    public void testGetAccounts() {
        BenchmarkState state = mPerfStatusReporter.getBenchmarkState();
        final Context context = InstrumentationRegistry.getTargetContext();
        if (context.checkSelfPermission(Manifest.permission.GET_ACCOUNTS)
                != PackageManager.PERMISSION_GRANTED) {
            fail("Missing required GET_ACCOUNTS permission");
        }
        AccountManager accountManager = AccountManager.get(context);
        while (state.keepRunning()) {
            accountManager.getAccounts();
        }
    }
}
```

Figura 9 – Exemplo de arquivo de teste

```java
/** Service in separate process available for calling over binder. */
public class SomeService extends Service {

    private File mTempFile;

    @Override
    public void onCreate() {
        super.onCreate();
        try {
            mTempFile = File.createTempFile("foo", "bar");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private final ISomeService.Stub mBinder =
            new ISomeService.Stub() {
                public void readDisk(int times) {
                    for (int i = 0; i < times; i++) {
                        mTempFile.exists();
                    }
                }
            };
```

Figura 10 – Exemplo de arquivo de produção

Tabela 8 – Amostra do Catálogo de *Frameworks* de automação de teste (GONZALEZ et al., 2017). Expressões em negrito foram adicionadas ao Catálogo original

Tipo	**Expressão regular**
Import Javascript	JS.Test [\.[\w]*]*, jsUnity [\.[\w]*]*, ...
Função Javascript	test\(, describe\(
Import Java	import +[\w\.]*JUnit, import +[\w\.]*jmock, ...
Função Java	@Test, @Before, @After, @RunWidth, assert.*, ...
Import Python	((from)|(import)) +unittest, ...
Função Python	def test_, testmod\(
Import PHP	extends PHPUnit_Framework_TestCase, ...
Função PHP	test, assert, @test, CLEAN, EXPECT ...
Import Ruby	**(“|’)[\w]*test[\w]*(“|’), (“|’)[\w]*spec[\w]*(“|’)** ...
Função Ruby	assert\(, assert_, describe

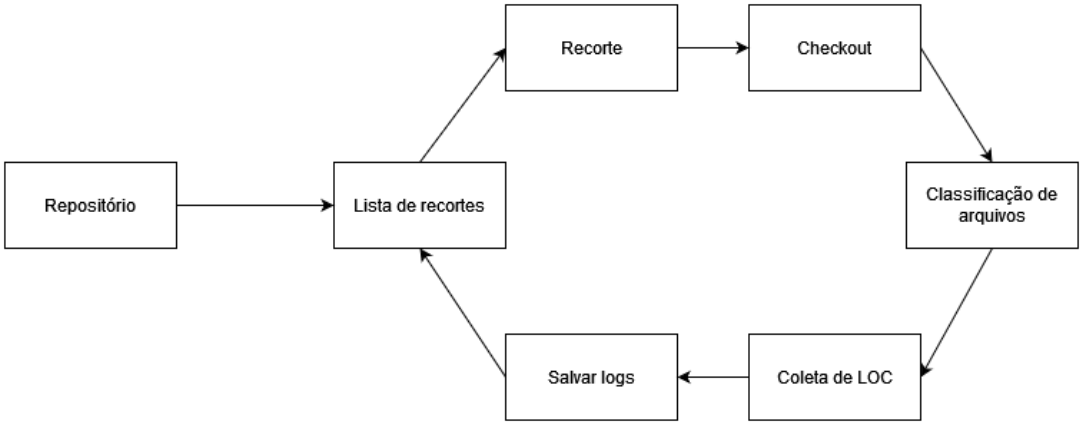


Figura 11 – Detalhamento do fluxo das etapa 3 da abordagem

lista contém os seguintes dados dos repositórios: id do repositório, nome completo do repositório, lista do total de LOC para cada recorte. Por exemplo, [{"id": "3", "full_name": "hashie/hashie", "timeseries": [13, 576, 1327, 1568, 2010, 2520, 2989, 3445, 4021, 4705]}, {"id": "11", "full_name": "ircmaxell/password_compat", "timeseries": [0, 111, 166, 170, 164, 161, 163, 159, 159, 209]}, ...].

4.3.1 Implementação da etapa

Para investigar a evolução dos artefatos, e consequentemente a co-evolução desses, a primeira etapa para a extração do Histórico de Proporção de Teste consiste em normalizar o histórico de desenvolvimento dos projetos. Essa primeira etapa tem como característica dividir o histórico de desenvolvimento em dez intervalos que serão representados por *commits* de cada repositório. A Figura 11 detalha as atividade realizadas para extração do histórico da proporção de teste.

No início da extração do Histórico de Proporção de Teste é realizada a captura dos

commits realizados no histórico de desenvolvimento. Esse processo consiste em extrair e selecionar *commits* para representar recortes do processo de desenvolvimento do projeto. Assim, são selecionados os *commits* que dividem o histórico em 10 partes iguais. Por exemplo, se um repositório tiver 100 *commits* serão selecionados os *commits* que dividem o histórico em 10 partes, de forma que, cada parte representa um recorte, ou seja, um salto no histórico de desenvolvimento de 10 *commits*.

Após identificação dos pontos de divisão do histórico de desenvolvimento, para cada recorte selecionado, são realizadas as seguintes atividades: *checkout* no *commit* que divide o recorte, classificação de arquivos de teste e coleta do total de Linhas de Código (LOC) de cada arquivo do repositório através da execução do programa *cloc* [3]. 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 teste para cada recorte, conforme descrito na Equação 4.1.

$$\text{proporcaoTeste} = \frac{totalLOCTeste}{totalLOCTeste + totalLOCRepositorio} \tag{4.1}$$

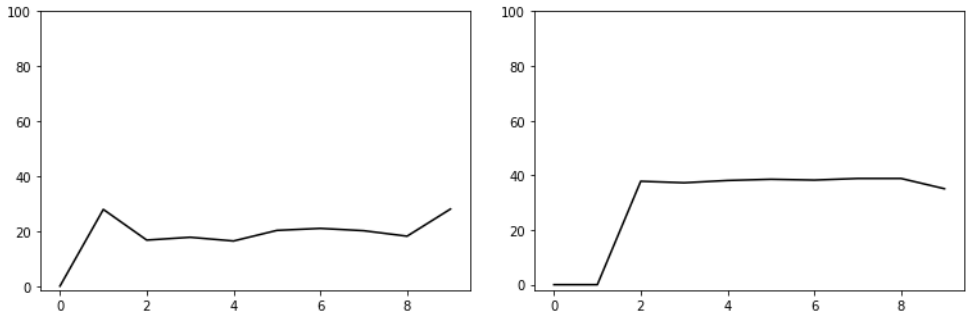
Como resultado da coleta de LOC de arquivos de teste e de código-fonte tem-se uma série temporal da proporção de teste. A série temporal representa uma coleção sequencial ao longo do tempo, permitindo observações sobre a co-evolução durante o histórico de desenvolvimento. Um exemplo prático é permitir verificar se o percentual de teste está evoluindo junto com o projeto.

Na Figura 12 é apresentado as proporções de teste dos repositórios hashie/hashie e ircmaxell/password_compat respectivamente. É possível observar que o repositório ircmaxell/password_compat possui mais teste do que hashie/hashie e que a co-evolução deste teste se mantém estável durante a maior parte do desenvolvimento. Observando o comportamento da co-evolução é possível classificar essa co-evolução em grupos, como forte co-evolução, moderada co-evolução e fraca co-evolução.

4.4 Identificação de Padrões de Crescimento de Teste

A etapa de Identificação de Padrões de Crescimento de Teste deve identificar comportamentos comuns no crescimento da proporção de teste dos repositórios. Para alcançar tal objetivo, deve-se agrupar os repositórios baseado na evolução do teste durante o histórico de desenvolvimento. Diante do grande volume de dados que possuímos, é necessário realizar a clusterização para a formação de grupos com comportamentos comuns. A clusterização também irá nos permitir realizar análises dos grupos de forma mais precisa.

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



(a) hashie/hashie (b) ircmaxell/password_compat

Figura 12 – Exemplo de séries temporais geradas da proporção de teste dos recortes dos repositórios

4.4.1 Implementação da etapa

Neste trabalho, utilizamos o KSC (YANG; LESKOVEC, 2011) 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 (BORGES; HORA; VALENTE, 2016), de vídeos do youtube (FIGUEIREDO, 2013) e twiter (LEHMANN et al., 2012). 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} (MENASCE; ALMEIDA, 2001). Baseado nessa heurística, o menor valor de k depois que a proporção do β_{CV} estabiliza deve ser escolhido (MENASCE; ALMEIDA, 2001; BORGES; HORA; VALENTE, 2016). A Figura 13 ilustra a seleção do melhor k, na figura podemos observar que, no exemplo, o melhor k é quatro.

4.5 Identificação de Repositórios com Co-evolução

A etapa de Identificação de Repositórios com Co-evolução tem como objetivo identificar o nível de co-evolução dos repositórios. Inicialmente, seu funcionamento consiste em calcular o nível de correlação entre a evolução do LOC de teste e LOC de produção em cada um dos recortes. Isso irá nos permitir observar a distribuição do Coeficiente de *Pearson*, e então observando os quartis da distribuição do coeficiente, é possível classificar

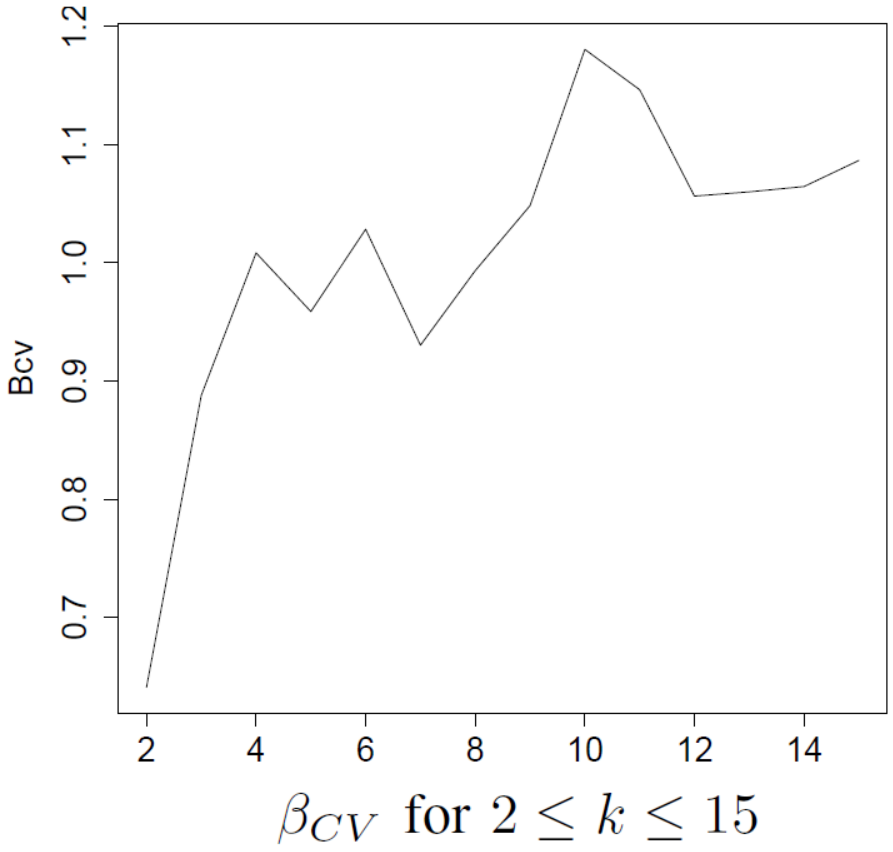


Figura 13 – Ilustração para identificação do melhor k (BORGES; HORA; VALENTE, 2016)

o nível de co-evolução dos repositórios. Em suma, a partir da distribuição do Coeficiente de *Pearson* obtido através da evolução do LOC dos repositórios, é possível identificar os repositórios que possuem forte co-evolução (4º quartil), moderada co-evolução (3º e 2º quartis) e com fraca co-evolução (1º quartil).

4.6 Impacto da Co-evolução de Teste nos Repositórios

Nesta etapa deve-se identificar inicialmente se existem diferenças entre as características dos repositórios com diferentes níveis de co-evolução e posteriormente o tamanho dessas diferenças. Para isso, iremos comparar número de *commits*, número de colaboradores, número de *forks* e número de *issues* entre os repositórios com forte co-evolução e os com fraca co-evolução. É importante ressaltar que os repositórios com forte co-evolução possuem o teste mais presente no seu desenvolvimento, logo, esperamos que esses apresentem características mais positivas para o desenvolvimento de software.

4.6.1 Implementação da etapa

Nesta etapa da pesquisa, os dados coletados de repositórios são comparados entre os repositórios classificados *com forte co-evolução* e *com fraca co-evolução*.

Os repositórios de software são ricos em informações e a api do Github nos disponibiliza diversas informações, algumas relacionadas ao histórico de desenvolvimento, outras relacionadas a popularidade do projeto. Tais dados permitem diversas análises diferentes, dependendo dos objetivos a serem alcançados. Neste estudo investigamos a correlação entre co-evolução e os seguintes atributos do repositório: número de *commits*, número de colaboradores, número de *forks* e número de *issues*.

Após a coleta, os dados da distribuição de número de *commits*, número de colaboradores, número de *forks* e número de *issues* são analisadas. Para tanto, o método *Wilcoxon/Mann-Whitney* (MCKNIGHT; NAJAB, 2010) 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* (COHEN, 2013) para quantificar o tamanho da diferença.

5 Resultados e Discussão

Este capítulo apresenta os resultados obtidos ao investigar as questões de pesquisa alvo deste estudo.

5.1 Resultados

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

A Figura 14 apresenta a distribuição do percentual de LOC de testes nos repositórios. Considerando todos os repositórios, é possível observar que em 75% dos repositórios (terceiro quartil) o percentual de LOC de testes é inferior a 41% e em metade deles (mediana/segundo quartil) tem-se no máximo 20% de testes. Comportamento similar é observado ao analisarmos a distribuição por linguagem. Com exceção dos repositórios Javascript, que possui um percentual de LOC de testes superior aos demais (3º quartil = 53.99% e 2º quartil=31.99%), os repositórios das demais linguagens tem uma distribuição bem similar. Nesses, os percentuais do 3º quartil varia entre 33.51% (Ruby) e 43.75% (PHP), enquanto a mediana varia entre 16.73% (Ruby) e 23.96% (PHP).

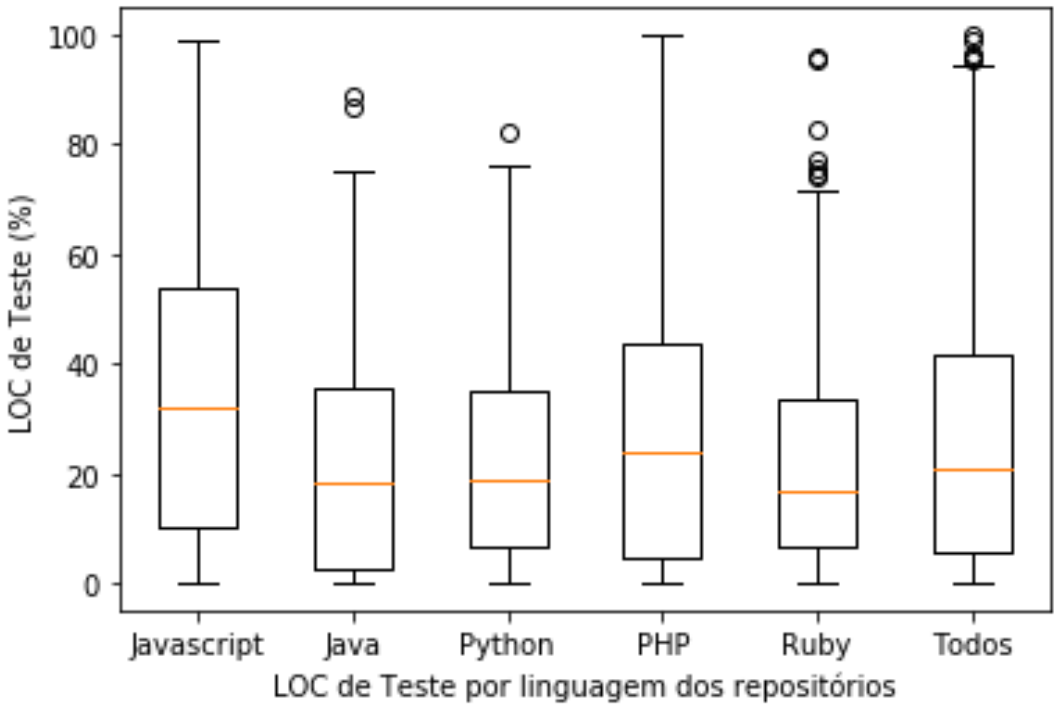


Figura 14 – Distribuição do Percentual de LOC de Teste

Como etapa preliminar para investigar a evolução de testes foram descartados repositórios nos quais não foram identificados arquivos de testes. Do total de 3.000

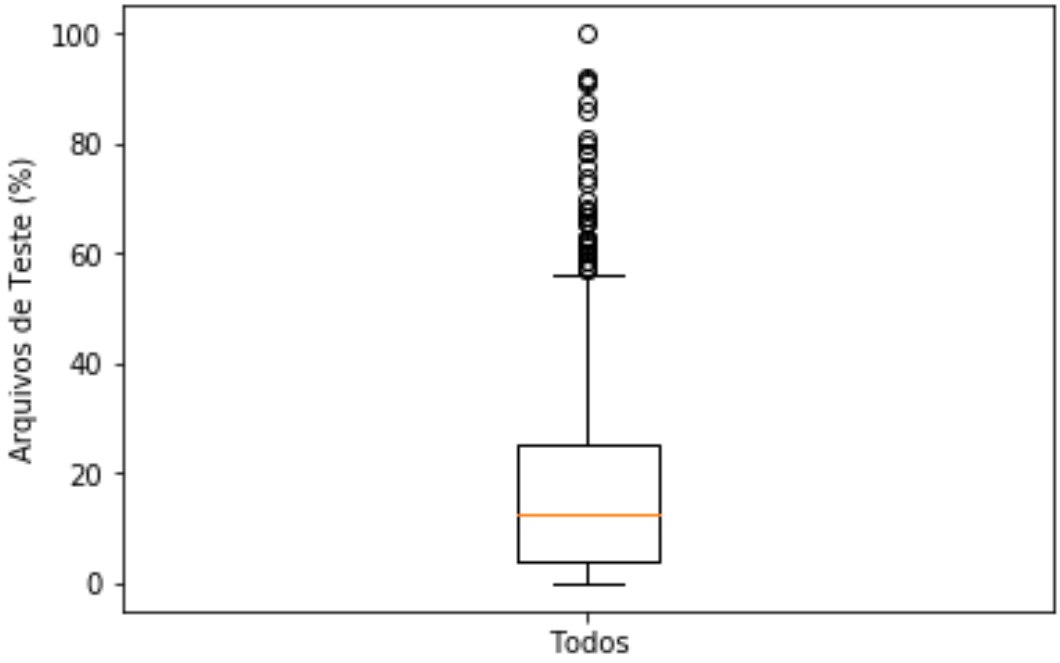
repositórios inicialmente selecionados, foram identificados arquivos de testes em 1.914 (63,8% dos repositórios).

Adicionalmente, após a construção dos gráficos representando a proporção de testes considerando a divisão do tempo de vida do repositório em 10 intervalos, como descrito na Seção 4.3, foram identificados problemas ao analisar repositórios com poucos arquivos/LOCs de teste. A Figura 16 ilustra o problema identificado. O gráfico de evolução de testes do repositório Marak/faker.js, por ter poucos arquivos/LOCs de teste, alterna grandes variações para cima e para baixo da proporção de testes. Esse comportamento ocorre porque mesmo pequenas modificações no código de teste produzem grandes variações na proporção de testes, tendo em vista a pequena quantidade de código de teste do repositório. Dessa forma, buscando minimizar esse problema, foram removidos repositórios no primeiro quartil do percentual de arquivos de teste (menor igual a 4.05% de arquivos de teste) e primeiro quartil do percentual de LOC de teste (menor igual a 6.02% de LOC de teste), resultando em 1.203 repositórios no nosso *dataset* final. A distribuição do número de arquivos e LOCs de testes dos repositórios pode ser observado na Figura 15.

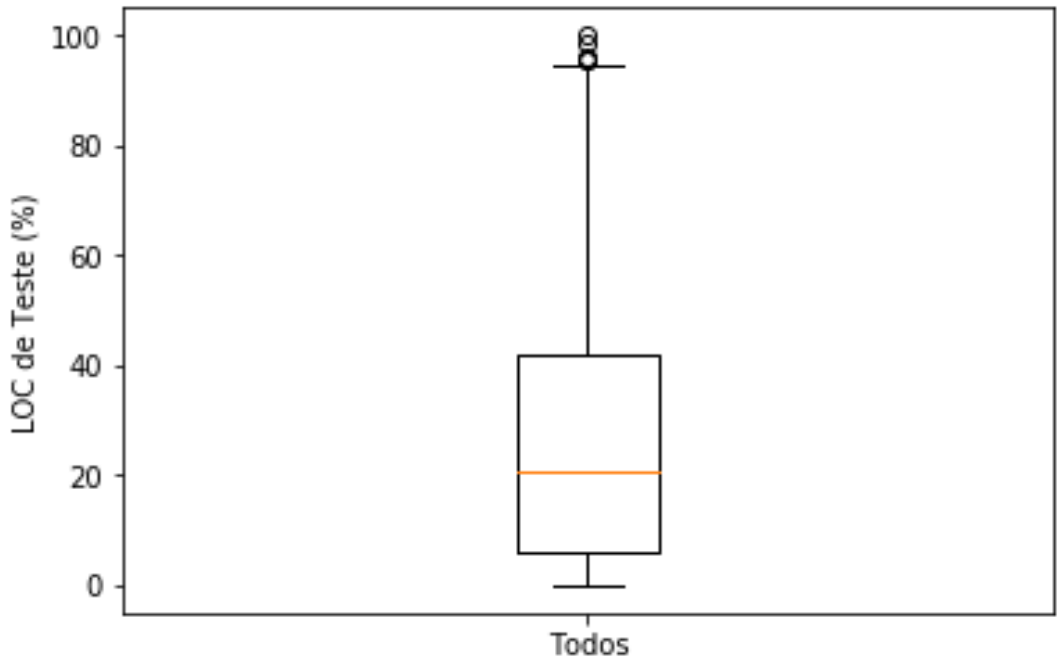
Após a filtragem e, posterior, construção do histórico da evolução da proporção de testes, buscando identificar padrões evolucionários nesses dados, as linhas de tempo foram agrupadas (clusterizadas) adotando o algoritmo KSC (vide Seção 4.4). Para identificar o melhor número de grupos para o nosso *dataset*, utilizamos a heurística β_{CV} (MENASCE; ALMEIDA, 2001). Conforme pode ser observado na Figura 17, em nosso estudo, o β_{CV} se estabiliza com k igual a 5 (β_{CV}=0,85). Dessa forma, o algoritmo KSC foi aplicado com k=*5*, resultando na classificação dos repositórios analisados em cinco grupos, como observado na Figura 18.

A Figura 18 apresenta os cinco *clusters* encontrados. Em cada *cluster* temos o nome do *cluster* (Cluster + numeração em algarismo romano), *centroid* (linha destacada na cor preta) e os repositórios que o compõe (cada repositório representado por uma cor tracejada). Com essas informações, podemos visualizar os cinco padrões de co-evolução e seus comportamentos com relação à evolução da proporção de teste durante o histórico dos projetos.

Os *clusters* foram organizados (e nomeados) de forma a destacar o padrão de comportamento da evolução dos testes dos repositórios que os compõem. É possível observar, que os repositórios do *Cluster I* tem um rápido crescimento do percentual de testes no início do desenvolvimento, o qual posteriormente se estabiliza no restante do tempo de vida do projeto. Comportamento similar é observado no *Cluster II*, porém a estabilização da proporção requer um tempo maior do que o verificado nos repositórios do *Cluster I*. Nos demais *clusters* não é possível observar essa estabilização da proporção de testes, isso pode ter relação com o processo de desenvolvimento aplicado ao projeto. Adicionalmente, vale destacar o momento e a velocidade de crescimento da proporção



(a) Arquivos



(b) LOC

Figura 15 – Distribuição do Percentual de Teste por arquivos e LOC

de testes nesses três *clusters*. Os repositórios do *Cluster III* utilizam testes desde o início do desenvolvimento e a proporção de testes cresce gradativamente ao longo da evolução deste, enquanto que os repositórios do *Clusters IV* e *V* demoram mais a iniciar o desenvolvimento de testes e, uma vez introduzido os testes, apresentam um crescimento acentuado da proporção de testes.

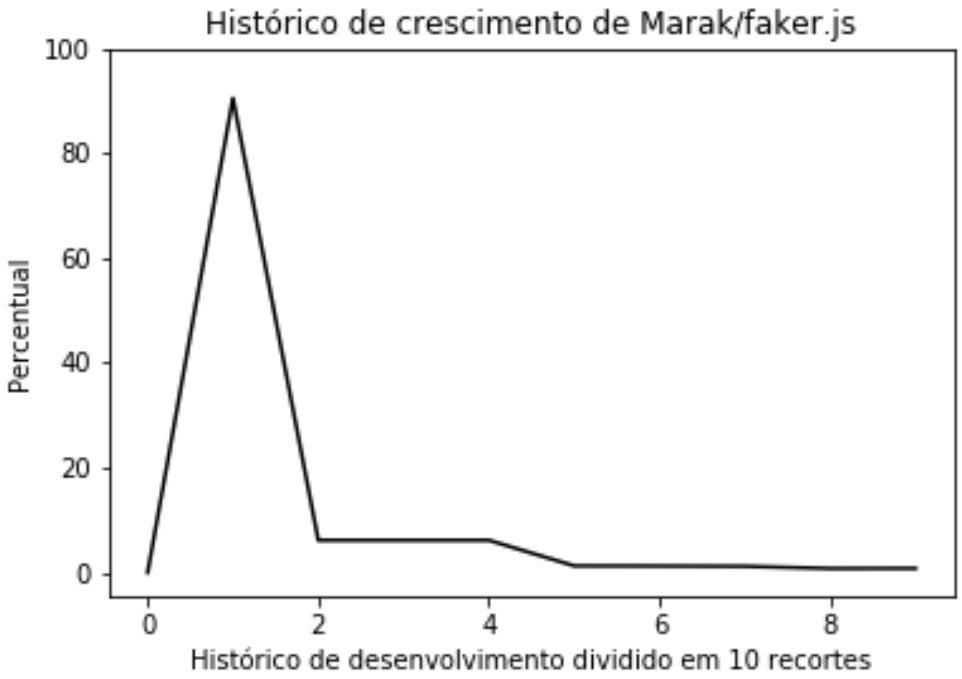


Figura 16 – Proporção de Teste do Repositório Marak/faker.js

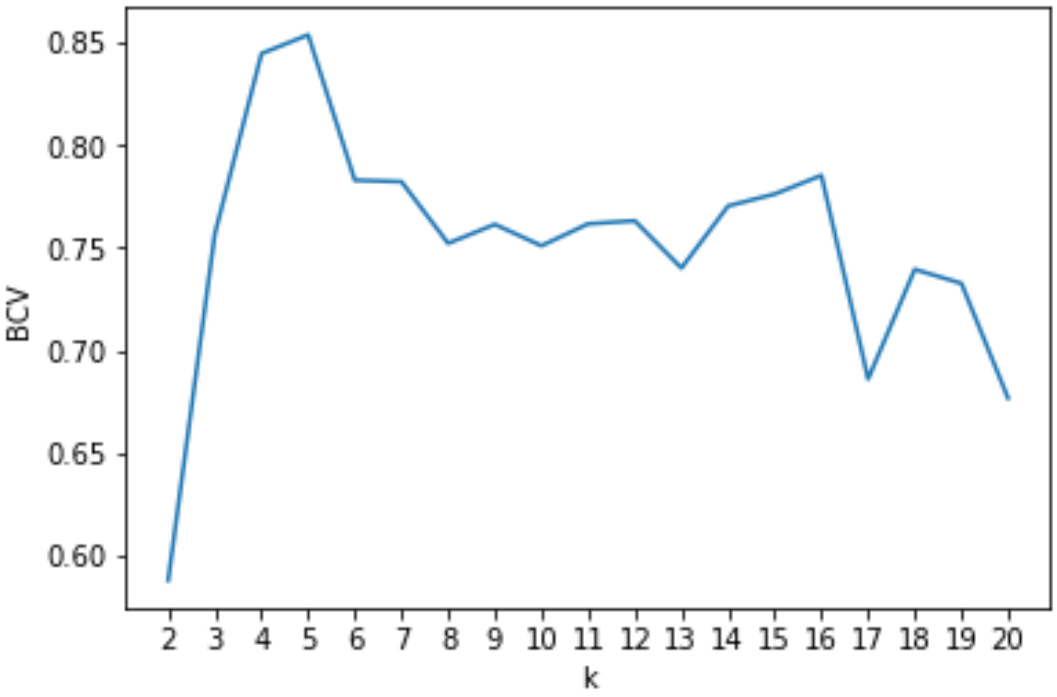
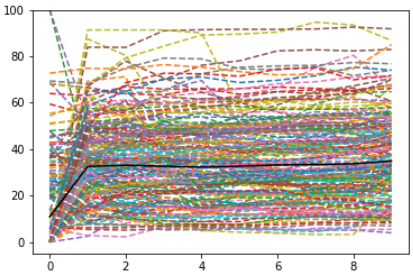
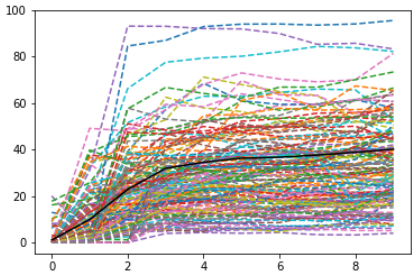


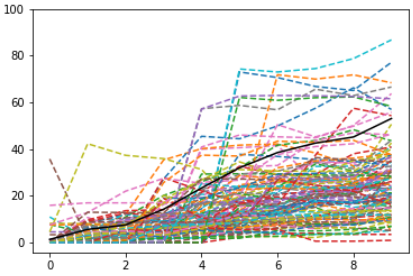
Figura 17 – β_{CV} por k >=2 e k <= 20



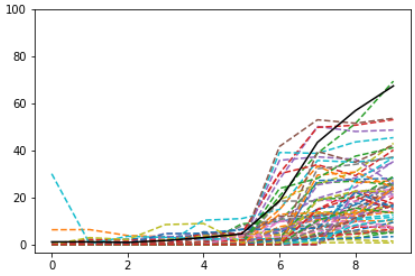
(a) Cluster I



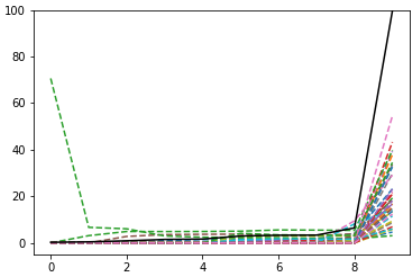
(b) Cluster II



(c) Cluster III



(d) Cluster IV



(e) Cluster V

Figura 18 – *Clusters* gerados do histórico de proporção de teste. O centróide está representado com a cor preta.

Como podemos observar na Tabela 9 os *clusters I e II* são os que apresentam o maior número de repositórios, respectivamente, 35% e 26%. Esse comportamento também é observado na distribuição dos repositórios considerando o agrupamento por linguagem de programação. O *Cluster I* é o que possui mais repositórios na maioria das linguagens, com exceção de Python que possui mais repositórios (31%) no *Cluster II*.

Tabela 9 – Distribuição dos repositórios da linguagem por *cluster*

Cluster	**Característica**	**Total(%)**	**Javascript(%)**	**Java(%)**	**Python(%)**	**PHP(%)**	**Ruby(%)**
I	Testes estáveis desde o início	35	**38**	**52**	24	**30**	**29**
II	Testes estáveis	26	23	30	**31**	25	20
III	Crescimento contínuo dos testes	20	23	11	26	18	25
IV	Testes tardios e em crescimento	12	10	5	15	12	18
V	Testes tardios e em crescimento intensivo	7	6	3	4	15	8

Outra interpretação dos dados pode ser observado na Tabela 10. Nessa tabela é

possível observar a representatividade das linguagens em cada um dos *clusters*. Repositórios Java são maioria nos *clusters I* (37%) e *II* (28%, empatado com Python), enquanto repositórios Python dominam os *clusters II* (28%, empatado com Java), *III* (31%)e *IV* (30%). Por fim, PHP é a linguagem que domina o *Cluster V* com a maior parte dos repositórios desse *cluster* (40%).

Tabela 10 – Representatividade da linguagem em cada *cluster*

Cluster	**Característica**	**Total(%)**	**Javascript(%)**	**Java(%)**	**Python(%)**	**PHP(%)**	**Ruby(%)**
I	Teste estáveis desde o início	35	18	**37**	16	17	12
II	Teste estáveis	26	15	**28**	**28**	16	13
III	Crescimento contínuo dos testes	20	20	13	**31**	16	20
IV	Testes tardios e em crescimento	12	14	11	**30**	19	26
V	Testes tardios e em crescimento intensivo	7	16	11	13	**40**	20

Resumo da QP1: Foram identificados 5 padrões de evolução de teste. Os dois padrões mais comuns (*clusters I* e *II*), modelam um comportamento em que a proporção de testes cresce rapidamente no início do projeto, se estabilizando em seguida. Os demais (*clusters III*, *IV* e *V*) agrupam repositórios em que a proporção de testes ainda não se estabilizou e, em especial nos últimos dois, representam uma adoção tardia de testes.

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

Para identificar os repositórios com co-evolução utilizamos a distribuição do Coeficiente de *Pearson* entre as linhas de crescimento do código de produção e testes, conforme detalhado na Seção 4.5. Na Figura 19 é possível observar a distribuição do Coeficiente de *Pearson* . A classificação do nível de co-evolução foi calculado baseado no coeficiente de *Pearson*, foi adotado o 1º quartil da distribuição de *Pearson* para caracterizar repositórios com fraca co-evolução e o 4º quartil para forte co-evolução.

Os repositórios com p maior que 0.96 foram considerados com forte co-evolução, os repositórios com p maior que 0.82 e menor igual a 0.96 foram considerados com moderada co-evolução, e os menor igual a 0.82 foram considerados como possuindo uma fraca co-evolução. Adotando essa estratégia, foram identificados 289 repositórios com fortes indícios de co-evolução, 602 com moderada e 312 com fraca.

Analisando a Figura 20 e a Tabela 11, encontramos diferenças significativas no percentual de repositórios em cada uma das categorias de co-evolução. A linguagem Java se destaca por possuir maior percentual de repositórios classificados com forte co-evolução (41%), contrastando com PHP que possui maior percentual de repositórios classificados como tendo fraca-coevolução (40%).

Em relação a clusterização, os repositórios com co-evolução encontram-se com mais frequência no *Cluster I*, com 67% repositórios, e no *Cluster II*, que possui 24%.

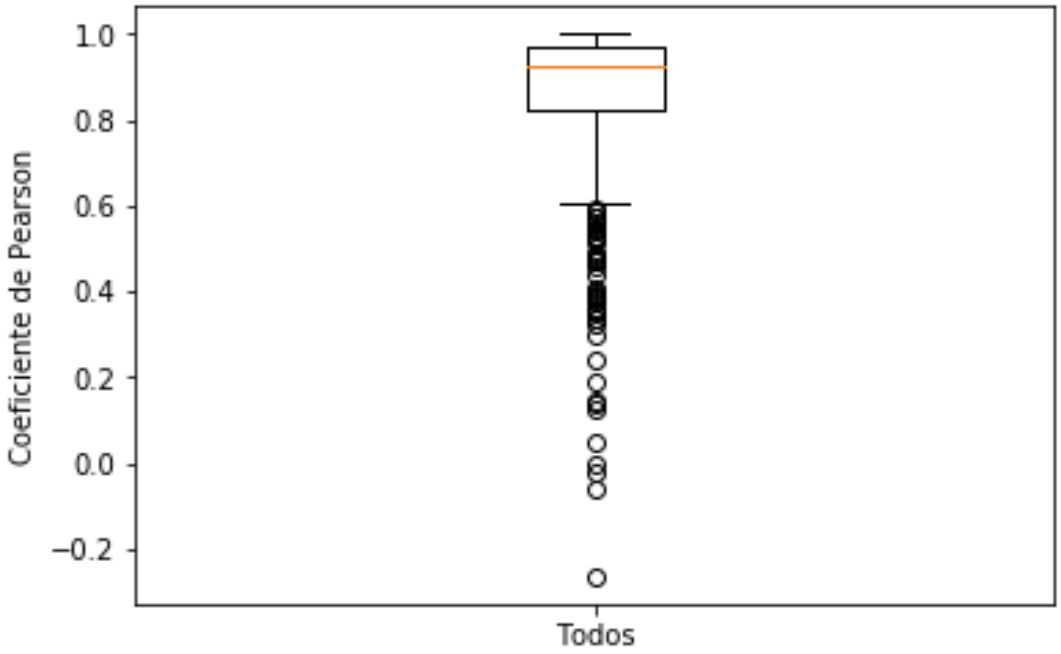


Figura 19 – Distribuição do Coeficiente de *Pearson*

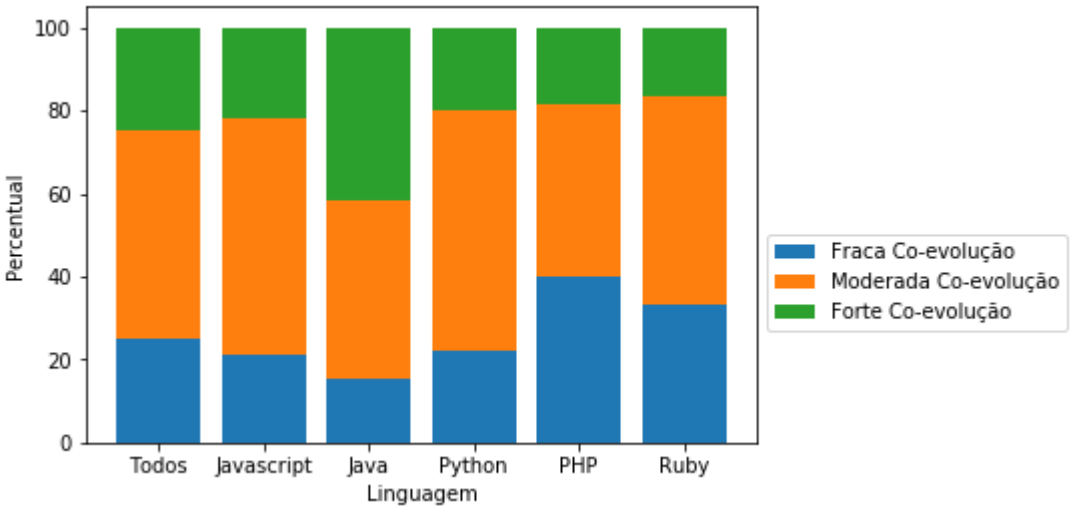


Figura 20 – Distribuição dos repositórios por nível de Co-evolução (%)

A quantidade de repositórios com co-evolução nos demais *clusters* é de 8%, 1% e 0% repositórios para os *Clusters III, IV* e *V* respectivamente.

Tabela 11 – Distribuição dos repositórios por nível de Co-evolução (%)

	Todos	**Javascript**	**Java**	**Python**	**PHP**	**Ruby**
Forte Co-evolução	24	22	41	20	18	17
Moderada Co-evolução	50	57	43	58	42	50
Fraca Co-evolução	26	21	16	22	40	33

Resumo da QP2: Utilizando o Coeficiente de *Pearson* como métrica para identificar a co-evolução nos repositórios, encontramos 289 repositórios com fortes indícios de co-evolução de teste, representando 24% dos repositórios. A linguagem Java se destaca como tendo maior percentual de repositórios com alta co-evolução de código de teste.

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

Para investigar se as características distinguem entre projetos onde há co-evolução de código de teste de projetos e onde essa prática não é comum utilizamos os repositórios agrupados em forte e fraca co-evolução de teste, e os comparamos através de cinco variáveis relacionadas a repositórios de software. As variáveis comparadas são: 1) número de *commits*, 2) número de colaboradores, 3) número de *forks* e 4) número de *issues*. A Figura 21 apresenta a distribuição e as Tabelas 12, 13, 14, 15 apresentam os resultados da comparação entre os repositórios. É aplicado o teste *Wilcoxon/Mann-Whitney* para verificar se a diferença das medianas entre os repositórios com forte e fraca co-evolução são estatisticamente diferentes. O *p-value* menor ou igual a 0,05 representa que a diferença é estatisticamente diferente. Se sim, aplicamos o *Cohen's D* para verificar o tamanho da diferença. Assim, o tamanho da diferença pode ser: insignificante (|d|<0.2), pequena (|d|<0.5), média (|d|<0.8), larga (|d|>=0.8).

Como é possível observar, a Tabela 12 apresenta os resultados das linguagens analisadas considerando o número de *commits*. Observando os valores de *p-value*, podemos verificar que para todas as linguagens a diferença de números de *commits* são estatisticamente diferentes. Mesmo consideradas pequenas ao aplicar o *Cohen's D*, pode-se afirmar que para todas as linguagens, os repositórios com forte co-evolução possuem mais *commits*.

Na Tabela 13 encontramos os resultados das linguagens analisadas considerando o número de colaboradores. As diferenças são consideradas média para as linguagens Javascript, Java e Ruby, enquanto para Python é considerada pequena e para PHP a diferença não é estatisticamente diferente. No geral, a diferença é considerada pequena. Com isso, para a maioria das linguagens analisadas a diferença de colaboradores são

estatisticamente diferentes e significativas, somente para PHP o número de colaboradores é similar entre os repositórios com forte co-evolução e os com fraca co-evolução. Assim, no geral, os repositórios com forte co-evolução possuem mais colaboradores.

Também podemos verificar na Tabela 14 os resultados das linguagens analisadas considerando o número de *forks*. Observando as medianas por linguagem percebe-se que repositórios com forte co-evolução em Javascript, Python e Ruby possuem mais *forks*. Para Java são similares. Para PHP é o inverso. Contudo, os valores de *p-value* são estatisticamente diferentes apenas para Javascript. No geral, os repositórios com forte co-evolução possuem mais *forks*.

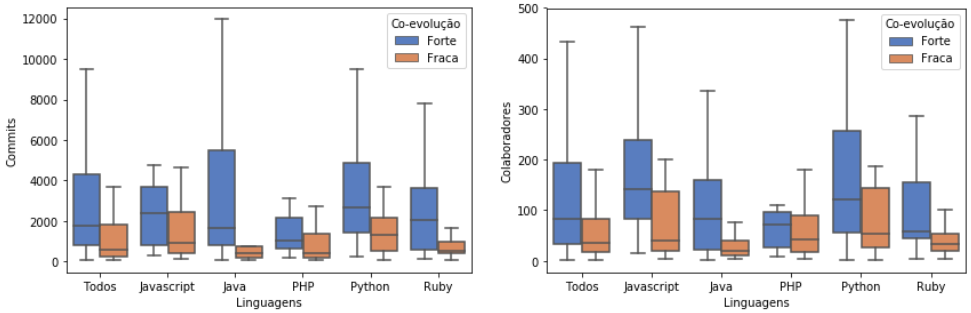
Por fim, a Tabela 15 apresenta os resultados das linguagens analisadas considerando o número de *issues*. As diferenças são consideradas média para Java, enquanto para Javascript e Python é considerada pequena. Para PHP e Ruby a diferença não é estatisticamente diferente. No geral, a diferença é considerada pequena. Com isso, para a maioria das linguagens analisadas a diferença de *issues* são estatisticamente diferentes e significativas, somente para PHP e Ruby o número de *issues* é similar entre os repositórios com forte co-evolução e os com fraca co-evolução. Portanto, no geral, os repositórios com forte co-evolução possuem mais *issues*.

Observando as medianas das distribuições do grupo *Todos*, os repositórios com forte co-evolução de teste possuem mais *commits*, mais colaboradores, mais *forks* e mais *issues*. Contudo ao aplicar os testes *Wilcoxon/Mann-Whitney* e *Cohen's D*, os resultados mostram que repositórios com forte co-evolução apresentam diferenças entre as distribuições, porém consideradas pequenas.

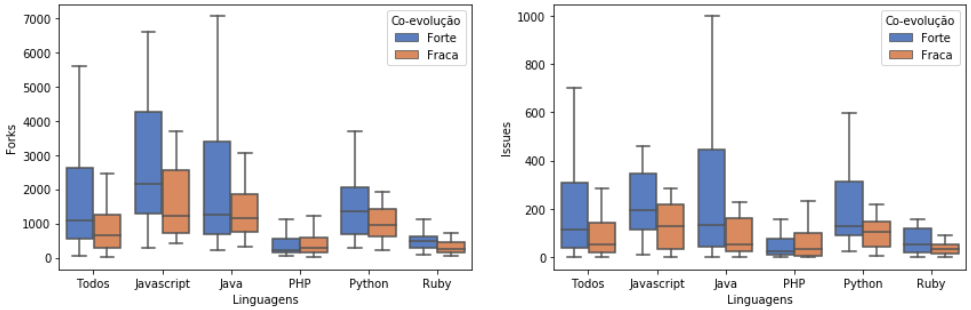
Resumo da QP3:
A comparação dos dois grupos de repositórios demonstrou a existência de diferenças significativas entre repositórios com forte co-evolução e os com fraca. Repositórios identificados com forte co-evolução possuem mais *commits*, colaboradores, *forks* e *issues*.

Tabela 12 – Resultado do teste estatístico para *Commits*

Grupo	*Wilcoxon/Mann-Whitney (p-value)*	Estatisticamente diferentes	*Cohen's D*	Tamanho diferença
Todos	0.000	Sim	0.231201	pequena
Javascript	0.000	Sim	0.32112	pequena
Java	0.000	Sim	0.43222	pequena
Python	0.004	Sim	0.30412	pequena
PHP	0.011	Sim	0.26444	pequena
Ruby	0.023	Sim	-0.230689	pequena



(a) *Commits* (b) Colaboradores



(c) *Forks* (d) *Issues*

Figura 21 – Distribuições entre projetos com co-evolução por número de *commits*, colaboradores, *forks*, *issues*.

Tabela 13 – Resultado do teste estatístico para Colaboradores

Grupo	*Wilcoxon/Mann-Whitney (p-value)*	Estatisticamente diferentes	*Cohen's D*	Tamanho diferença
Todos	0.000	Sim	0.377192	pequena
Javascript	0.000	Sim	0.504043	média
Java	0.001	Sim	0.553034	média
Python	0.008	Sim	0.373607	pequena
PHP	0.031	Não		
Ruby	0.023	Sim	0.66173	média

Tabela 14 – Resultado do teste estatístico para *Forks*

Grupo	*Wilcoxon/Mann-Whitney (p-value)*	Estatisticamente diferentes	*Cohen's D*	Tamanho diferença
Todos	0.000	Sim	0.377367	pequena
Javascript	0.000	Sim	0.60552	média
Java	0.296	Não		
Python	0.135	Não		
PHP	0.483	Não		
Ruby	0.052	Não		

Tabela 15 – Resultado do teste estatístico para *Issues*

Grupo	*Wilcoxon/Mann-Whitney (p-value)*	Estatisticamente diferentes	*Cohen's D*	Tamanho diferença
Todos	0.000	Sim	0.410043	pequena
Javascript	0.000	Sim	0.21734	pequena
Java	0.015	Sim	0.577404	média
Python	0.032	Sim	0.31844	pequena
PHP	0.300	Não		
Ruby	0.177	Não		

5.2 Discussão

Nossos resultados mostram que 63,8% dos repositórios possuem arquivos de teste. Conforme visto na Figura 14, 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 corresponde a 20%. Esses resultados demonstram que algumas linguagens 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 evoluíram. Na Figura 18, é observado que os repositórios com teste são agrupados em cinco padrões de crescimento. Os *clusters I* e *II* 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 forte co-evolução nos repositórios desses *clusters*, sendo confirmado ao observarmos que esses dois *clusters* possuem a maior parte dos repositórios identificados com forte co-evolução.

Em relação a clusterização, podemos observar que a forte co-evolução está altamente relacionada à evolução equilibrada entre código de teste e de produção. O *cluster I* representa o crescimento equilibrado entre código de teste e de produção (Conforme visto na Fig. 18a), a esta característica denominamos de Teste estáveis desde o início. O *cluster II* representa uma combinação dos *clusters I e III*, ou seja, existe uma priorização em relação ao código de teste, mas que ocorre em uma intensidade mais devagar, a esta característica denominamos de Testes estáveis. O *cluster III* representa o crescimento contínuo de código de teste, a esta característica denominamos de Crescimento contínuo dos testes. Em seguida, o *cluster IV* representa uma combinação dos *clusters III e V*, caracterizando uma implementação atrasada do código de teste em relação ao de produção, a esta característica denominamos de Testes tardios e em crescimento intensivo. Por fim, o *cluster V* representa uma implementação muito tardia ou vagarosa do código de teste em relação ao de produção, a esta característica denominamos de Testes tardios e em crescimento Teste.

Analisando as Tabelas 12, 13, 14, 15 em termos de números absolutos, temos as *Issues* como a característica mais impactada pela co-evolução de teste, possuindo *Cohen's D* com valor de maior que 0.41. Isto pode indicar que em projetos com forte co-evolução de teste é mais fácil utilizar *Issues* para rastrear o tipo de atividade de manutenção a que se referem os *commits*, ou seja, se são alterações para correções de erros (corretivas), para mudanças nas regras de negócio (adaptativas) ou para melhorar o software (perfectivas). Esta é uma importante informação para estudos que tem por objetivo medir a qualidade do software. Em contrapartida, temos *commits* com apenas 0.23, indicando que, das características analisadas, *commits* é a que possui menor impacto.

De forma geral, os resultados da comparação das diferenças entre os repositórios baseado no nível de co-evolução de teste mostram que repositórios com forte co-evolução possuem mais *commits*, colaboradores, *forks* e *issues*. Esse fato indica que a forte 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*, e que utilizam mais *issues* como ferramenta de relacionamento da tarefa com a atividade de desenvolvimento. 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.

5.3 Ameaças à Validade

Os resultados mostrados nessa seção apresentam algumas ameaças à validade da abordagem desta pesquisa. Durante as etapas da abordagem de uma pesquisa existem riscos relacionados à coleta de dados, controle dos processos executados, generalização dos resultados e conclusões realizadas. Portanto, a seguir são listadas algumas ameaças à validade identificadas.

- **Seleção do *dataset*:** A seleção de projetos através de estrelas pode ter deixado projetos de engenharia de software com teste fora do *dataset* desta pesquisa. Contudo, compreender a relação de popularidade e projetos de engenharia de software não é o objetivo deste trabalho;
- **Projetos classificados como não sendo de teste:** Dos três mil projetos baixados, não foi encontrada a presença de arquivos de teste em um mil e oitenta e quatro projetos (36,20%). A não detecção de arquivos de teste pode representar uma falha na abordagem de extração, coleta de dados dos repositórios de software ou do algoritmo de classificação de teste. Para minimizar essa ameaça, selecionamos alguns repositórios para realizar a verificação manual;
- **Seleção dos repositórios com co-evolução:** A abordagem para a seleção dos repositórios com co-evolução foi o percentual de teste. A escolha do percentual de teste mostra que a variação de código de produção e de teste são similares, ou seja, há co-evolução;
- **Padrões de crescimento de teste:** A classificação dos padrões de crescimento foi realizada utilizando uma lista de um mil, novecentos e catorze séries temporais. Cada série temporal com tamanho dez, sendo representada por dez *commits* do projeto. O tamanho da série pode impactar no resultado da classificação de *clusters*.

Pretende-se realizar a extração de mais dados para comparar os novos resultados com os já existentes;

5.4 Considerações finais

Neste capítulo foi apresentado as respostas para as questões de pesquisas que nortearam este trabalho, como também os resultados da análise do impacto da co-evolução de teste no desenvolvimento de software. A co-evolução de teste pode contribuir para representar a maturidade do projeto, uma vez que ele irá possuir mais pessoas envolvidas no desenvolvimento (colaboradores e *forks*), possuir maior uso de *issues* para a organização de tarefas e, por fim, possuir mais alterações realizadas (*commits*).

6 Conclusão

Este capítulo apresenta os principais pontos do estudo da co-evolução de teste em 3.000 projetos de software reais hospedados no GitHub. O foco da investigação foi entender como ocorre a co-evolução de teste e identificar o seu impacto nas características dos projetos.

6.1 Resumo

O Teste de Software é uma atividade no desenvolvimento de software, e a evolução do software deve ser acompanhada por testes. Dessa forma é importante para os gerentes de projeto a informação dos impactos da co-evolução de teste. 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. Após a seleção e extração de dados, foi realizada a seleção de dez recortes para representar o histórico de desenvolvimento, e em seguida, para cada recorte, foi realizada a classificação de arquivos de teste e a coleta de LOC dos arquivos para poder calcular a proporção de teste dos repositórios. Para a identificação de padrões de crescimento de teste foi utilizado o KSC para realizar a clusterização da proporção de teste. O objetivo da clusterização é agrupar os repositórios com proporção de teste semelhantes, permitindo identificar comportamentos comuns e padrões de crescimento. O LOC de teste e produção foi utilizado também para identificar os níveis de co-evolução dos repositórios, para isso foi feita a distribuição do Coeficiente de *Pearson*, que mediu a correlação existente entre a evolução de teste e a evolução do projeto.

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 320 (26,60%) possuem forte 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*), utilizam mais *issues* como ferramenta de relacionamento da tarefa com a atividade de manutenção e possuem maior interesse da comunidade em contribuir (número de colaboradores e *forks*).

6.2 Contribuições

As principais contribuições deste trabalho são:

1. Construção e disponibilização pública de um grande *dataset* para estudo de testes de softwares, composto por repositórios desenvolvidos em cinco diferentes linguagens de programação;
2. Investigação da evolução e co-evolução de testes em um grande conjunto de sistemas reais, o qual, dentre outras, permitiu a identificação de padrões comportamentais comuns no desenvolvimento de testes em projetos de software. Estes padrões podem, por exemplo, serem utilizados para avaliar outros ambientes de desenvolvimento de software verificando quão maduro/saudáveis são tais ambientes quanto ao uso de testes;
3. Definição de uma metodologia para identificação de co-evolução de testes em projetos de desenvolvimento de software, baseada na análise do histórico de mudanças desses projetos;
4. Investigação do impacto da co-evolução de testes em projetos de desenvolvimento de software, provendo indícios da influência da co-evolução em aspectos diversos do projeto.
5. Publicação de Artigo no Workshop on Software Visualization, Evolution and Maintenance (VEM 2021). O VEM é um importante fórum na área de Engenharia de Software que permite a pesquisadores e profissionais da academia e da indústria exporem seus trabalhos e trocar ideias.

6.3 Limitações

Atualmente a abordagem possui limitações, em especial na identificação dos repositórios com co-evolução e nos resultados.

- *Extração da proporção de teste* - O estudo é limitado a dez recortes da proporção de teste. Apesar dos dados e análises produzidos indicarem resultados significativos. Uma extensão deste trabalho poderia envolver mais recortes do histórico de desenvolvimento.
- *Identificação dos Repositórios com forte co-evolução* - O estudo utiliza o Coeficiente de *Pearson* para determinar o nível de co-evolução dos repositórios. Apesar de o coeficiente ser utilizado em outros estudos de correlação entre duas variáveis, há muitas outras possibilidades de abordagens que podem ser úteis, como por exemplo, o uso de técnicas de inteligência artificial nessa classificação.

- *Resultados* - O estudo apresenta resultados de caráter quantitativo. No entanto, é necessário realizar uma análise de viés qualitativo, coletando e apresentando o *feedback* dos desenvolvedores envolvidos no desenvolvimento dos projetos;

6.4 Trabalhos Futuros

Este trabalho pode ser estendido por meio de análises com mais recortes do histórico de desenvolvimento. Como também, pode-se comparar ou realizar outras abordagens para identificação da co-evolução de teste. Outra possibilidade é realizar um *survey* com os desenvolvedores sobre os resultados deste trabalho (Como dito nas limitações).

Além disso, temos o objetivo de verificar os defeitos pós-release dos projetos, pretende-se investigar atividades de correção de *bugs* e projetos de software com forte co-evolução de teste. Dessa maneira, será possível relacionar a co-evolução de teste com a qualidade do software.

6.5 Publicações

Este trabalho resultou em uma publicação no Workshop on Software Visualization, Evolution and Maintenance (VEM 2021) que foi realizado no dia 21 de Setembro de 2021 de forma online. O artigo está disponível em <https://sol.sbc.org.br/index.php/vem/article/view/17215/17053>.

6.6 Artefatos da Pesquisa

Os dados utilizados na publicação estão publicamente disponíveis em <https://zenodo.org/record/5222965#.YR55lUBv-Uk>.

Referências

AGARWAL, B.; TAYAL, S.; GUPTA, M. *Software engineering and testing*. [S.l.]: Jones & Bartlett Learning, 2010. 8

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. *Engenharia de Software Magazine*, v. 1, n. 3, p. 54–57, 2008. 10

BERNER, S.; WEBER, R.; KELLER, R. K. Observations and lessons learned from automated testing. In: *Proceedings of the 27th international conference on Software engineering*. [S.l.: s.n.], 2005. p. 571–579. 12

BORGES, H.; HORA, A.; VALENTE, M. T. Understanding the factors that impact the popularity of github repositories. In: IEEE. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2016. p. 334–344. 15, 29, 30

BORGES, H. S. Characterizing and predicting the popularity of github projects. Universidade Federal de Minas Gerais, 2018. 20

CHACON, S.; STRAUB, B. *Pro git*. [S.l.]: Springer Nature, 2014. 12

CLEGG, B. S.; ROJAS, J. M.; FRASER, G. Teaching software testing concepts using a mutation testing game. In: IEEE. *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. [S.l.], 2017. p. 33–36. 2

COHEN, J. *Statistical power analysis for the behavioral sciences*. [S.l.]: Academic press, 2013. 31

COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Pearson Education, 2010. 11

CRESPO, A. N. et al. Uma metodologia para teste de software no contexto da melhoria de processo. *Simpósio Brasileiro de Qualidade de Software*, p. 271–285, 2004. 7

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013. 1

FIGUEIREDO, F. On the prediction of popularity of trends and hits for user generated videos. In: *Proceedings of the sixth ACM international conference on Web search and data mining*. [S.l.: s.n.], 2013. p. 741–746. 29

FIRESMITH, D. G. *Common system and software testing pitfalls: how to prevent and mitigate them: descriptions, symptoms, consequences, causes, and recommendations*. [S.l.]: Addison-Wesley Professional, 2014. 7

GONZALEZ, D. et al. A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 391–401. 15, 17, 19, 22, 23, 24, 27

KUMAR, D.; MISHRA, K. K. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, Elsevier, v. 79, p. 8–15, 2016. 1

LEHMAN, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, Elsevier, v. 1, p. 213–221, 1979. 12

LEHMAN, M. M. Laws of software evolution revisited. In: SPRINGER. *European Workshop on Software Process Technology*. [S.l.], 1996. p. 108–124. 9

LEHMANN, J. et al. Dynamical classes of collective attention in twitter. In: *Proceedings of the 21st international conference on World Wide Web*. [S.l.: s.n.], 2012. p. 251–260. 29

LEVIN, S.; YEHUDAI, A. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In: IEEE. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2017. p. 35–46. 2, 3, 12, 15, 16

MARSAVINA, C.; ROMANO, D.; ZAIDMAN, A. Studying fine-grained co-evolution patterns of production and test code. In: IEEE. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. [S.l.], 2014. p. 195–204. 2, 3, 10, 12, 15, 16

MCKNIGHT, P. E.; NAJAB, J. Mann-whitney u test. *The Corsini encyclopedia of psychology*, Wiley Online Library, p. 1–1, 2010. 31

MENASCE, D. A.; ALMEIDA, V. *Capacity Planning for Web Services: metrics, models, and methods*. [S.l.]: Prentice Hall PTR, 2001. 29, 34

MENS, T. et al. Challenges in software evolution. In: IEEE. *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. [S.l.], 2005. p. 13–22. 2, 12

MYERS, G. J. et al. *The art of software testing*. [S.l.]: Wiley Online Library, 2004. v. 2. 2

OLUWOLE, D. Agile methodology is not all about exploratory testing. *Scrum Alliance*, 2013. 11

PETTICHORD, B.; BACH, J.; KANER, C. *Lessons Learned in Software Testing: A Context-Driven Approach*. [S.l.]: Wiley, 2013. 2

PLANNING, S. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002. 8

POL, M.; TEUNISSEN, R.; VEENENDAAL, E. V. *Software testing: A guide to the TMap approach*. [S.l.]: Pearson Education, 2002. 8

PRESSMAN, R. *Engenharia de Software. 6ª edição.[Sl]: Ed*. [S.l.]: Mc Graw Hill, 2007. 1, 9

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave macmillan, 2005. 8

RIOS, E.; MOREIRA, T. *Teste de software*. [S.l.]: Alta Books Editora, 2006. 7

SEDGWICK, P. Pearson's correlation coefficient. *Bmj*, British Medical Journal Publishing Group, v. 345, 2012. 4

SOMMERVILLE, I. Software engineering 9th edition. *ISBN-10*, v. 137035152, p. 18, 2011. 15, 17, 1, 7, 8, 9, 10

SPINELLIS, D. Version control systems. *IEEE Software*, IEEE, v. 22, n. 5, p. 108–109, 2005. 12

STEFINKO, Y.; PISKOZUB, A.; BANAKH, R. Manual and automated penetration testing. benefits and drawbacks. modern tendency. In: IEEE. *2016 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*. [S.l.], 2016. p. 488–491. 17, 11

VIDÁCS, L.; PINZGER, M. Co-evolution analysis of production and test code by learning association rules of changes. In: IEEE. *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. [S.l.], 2018. p. 31–36. 2, 3, 7, 15, 16

VIEIRA, L. S. et al. Test automation in a test factory: an experience report. In: *Proceedings of the XIV Brazilian Symposium on Information Systems*. [S.l.: s.n.], 2018. p. 1–8. 2

WANGENHEIM, C. G. V.; SILVA, D. A. Qual conhecimento de engenharia de software é importante para um profissional de software? *Proceedings of the Fórum de Educação em Engenharia de Software*, v. 2, p. 1–8, 2009. 2

WHITE, R.; KRINKE, J.; TAN, R. Establishing multilevel test-to-code traceability links. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. [S.l.: s.n.], 2020. p. 861–872. 7

WONG, W. E. et al. Teaching software testing: Experiences, lessons learned and the path forward. In: IEEE. *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*. [S.l.], 2011. p. 530–534. 2

YANG, J.; LESKOVEC, J. Patterns of temporal variation in online media. In: *Proceedings of the fourth ACM international conference on Web search and data mining*. [S.l.: s.n.], 2011. p. 177–186. 19, 29

ZAIDMAN, A. et al. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, Springer, v. 16, n. 3, p. 325–364, 2011. 2, 3, 12, 15, 16, 17