UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

José Arthur de Oliveira e Britto Silveira

TPVis: Um sistema de análise visual para explorar métodos de priorização de casos de teste

Recife

2024

José Arthur de Oliveira e Britto Silveira

TPVis: Um sistema de análise visual para explorar métodos de priorização de casos de teste

Trabalho apresentado ao Programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Mídia e Interação

Orientador (a): Nivan Roberto Ferreira Junior

Recife

2024

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Silveira, José Arthur de Oliveira e Britto.
TPVis: um sistema de análise visual para explorar métodos de priorização de casos de teste / José Arthur de Oliveira e Britto Silveira. - Recife, 2024.
70f.: il.

Dissertação (Mestrado) - Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2024.
Orientação: Nivan Roberto Ferreira Junior.
Inclui referências.

1. Teste de software; 2. Análise visual; 3. Priorização de casos de teste. I. Ferreira Junior, Nivan Roberto. II. Título.

UFPE-Biblioteca Central

Folha de aprovação: Inserir a folha de aprovação enviada pela Secretaria do curso de Pós-Graduação. A folha deve conter a **data de aprovação**, estar **sem assinaturas** e em formato **PDF**.

A minha família e amigos próximos que me ajudaram nesta caminhada.

AGRADECIMENTOS

A realização desta pesquisa não seria possível sem o apoio e a contribuição de diversas pessoas às quais devo meu sincero agradecimento. Em primeiro lugar, gostaria de expressar minha gratidão ao meu orientador, Prof. Nivan Ferreira, por sua orientação incansável, paciência e críticas construtivas ao longo de todo este processo. Sua expertise e apoio foram essenciais para esta jornada. Agradeço também aos membros da banca examinadora, por dedicarem seu tempo para avaliar minha dissertação.

Aos meus amigos, que me acompanharam nesta jornada acadêmica, oferecendo suporte, compartilhando experiências e colaborando nos momentos de dificuldade, deixo meu sincero reconhecimento. Também sou imensamente grato à minha família, especialmente aos meus pais, Marceline e Eugênio, por seu amor incondicional, encorajamento inicial e constante e acima de tudo, por acreditarem em meu potencial, mesmo nos momentos mais difíceis. A Ana Beatriz, meu sincero agradecimento por seu carinho, paciência e apoio inabalável. Sua presença constante e incentivo foram essenciais para que eu pudesse superar os desafios e alcançar meus objetivos.

Também é de grande importância manifestar os devidos agradecimentos para Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), que auxiliou o trabalho com financiamento por meio do processo 88887.683727/2022-00.

"We can not test quality directly, but we can test related factors to make quality visible." (PAN, 1999).

RESUMO

À medida que a complexidade de um software aumenta, também cresce a necessidade de garantir que ele se comporta conforme o esperado. O teste de software é uma ferramenta vital para garantir a qualidade e a confiabilidade dos projetos de software desenvolvidos. No entanto, as suítes de teste são frequentemente grandes, o que torna o processo de testar o software durante suas atualizações um processo demorado. Nesse contexto, os métodos de priorização de casos de teste (TCP) desempenham um papel importante ao classificar os casos de teste a fim de permitir a detecção precoce de falhas e, assim, possibilitar correções mais rápidas dos problemas. No entanto, há uma infinidade de métodos de TCP propostos na literatura que variam em termos de dados usados e critérios de priorização. A avaliação de tais métodos é um problema difícil, devido à variedade dos métodos e objetivos. Por essa razão, essa avaliação é frequentemente limitada a algumas métricas de desempenho, que não ilustram nem capturam o comportamento complexo dos métodos de TCP. Para resolver essa questão, introduzimos o TPVis, um conjunto de ferramentas para visualização de TCP projetado em colaboração com especialistas em testes de software. Nossa solução é uma aplicação web amigável que fornece uma variedade de ferramentas analíticas para auxiliar na exploração de muitos aspectos das suítes de testes e dos algoritmos de priorização. Ilustramos a utilidade do TPVis por meio de uma série de casos de uso e também por meio de feedback obtido de especialistas da área.

Palavras-chaves: Teste de Software; Análise Visual; Priorização de Casos de Teste.

ABSTRACT

As software complexity is continuously growing, so is the need to ensure that it behaves as expected. Software testing is a vital tool to ensure the quality and trustworthiness of the pieces of software produced. However, test suites are often large, which makes the process of testing software throughout its updates a time-consuming process. In this context, test case prioritization (TCP) methods play an important role by ranking test cases in order to enable early fault detection and, hence, enable quicker problem fixes. However, there are a plethora of TCP techniques proposed in the literature which vary in terms of data used and prioritization criteria. The evaluation of such methods is a difficult problem, due to the variety of the methods and objectives. For this reason, this evaluation is often limited to a few performance metrics, which do not illustrate or capture the complex behavior of TCP methods. To address this issue we introduce TPVis, a toolkit for TCP visualization designed in collaboration with experts in software testing. Our solution is a user-friendly web application that provides a variety of analytical tools to assist in the exploration of many aspects of test suites and prioritization algorithms. We illustrate the usefulness of TPVis through a series of use cases and also via feedback obtained from domain experts.

Keywords: Software Testing; Visual Analytics; Test Case Prioritization.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

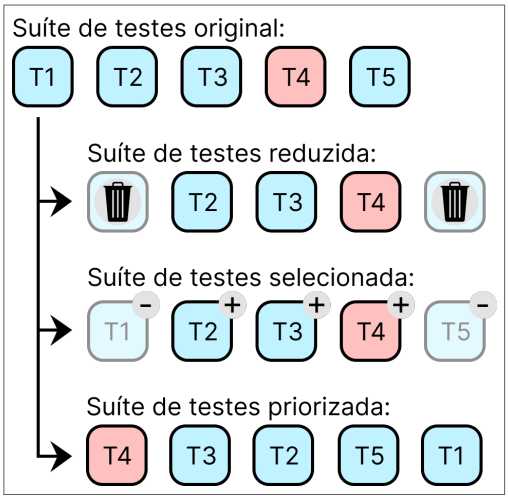
1 INTRODUÇÃO

1.1 CONTEXTO GERAL

À medida que a complexidade do software cresce, também cresce a necessidade de garantir que ele se comporta como esperado. Os testes de software são uma ferramenta vital para garantir a qualidade e confiabilidade dos programas produzidos. No entanto, os conjuntos de testes geralmente são grandes, tornando o processo de teste do software ao longo de suas atualizações demorado. Isso é especialmente relevante no cenário de teste de regressão, quando software em constante mudança é retestado para garantir que novas adições não introduzem novos bugs. De fato, foi relatado que os testes de regressão podem representar 80% do orçamento de testes e também até 50% do custo da manutenção de software (HARROLD, 2009).

Com a crescente demanda por desenvolvimento de software e a adoção de metodologias ágeis, os testes de regressão se tornaram essenciais. À medida que o software cresce em complexidade, os conjuntos de testes também tendem a ficar maiores. Portanto, a execução de tais conjuntos de testes pode se tornar computacionalmente ou custosamente inviável. É importante notar também que este não é um problema novo; Em 2003, alguns conjuntos de testes precisavam de até 7 semanas para serem executados em sua totalidade (ROTHERMEL; ELBAUM, 2003).

Figura 1 – Possíveis soluções para remediar o problema das grandes suítes de teste.



Fonte: O autor (2024)

Como observado na figura 1, existem três abordagens principais para resolver o problema

da execução de grandes conjuntos de testes: (**I**) Redução de casos de teste: Remover testes redundantes ou irrelevantes do conjunto de execução de testes; (**II**) Seleção de casos de teste: Executar apenas testes que testam as alterações de código do software na versão atual; (**III**) Priorização de casos de teste: Executar todos os testes, mas priorizá-los de forma a alcançar algum objetivo (por exemplo, melhorar a taxa de detecção de falhas ou atingir uma maior cobertura mais rapidamente) (DUGGAL; SURI, 2008; LOU et al., 2019).

O problema de priorização de casos de teste (TCP) recebeu muita atenção na comunidade de pesquisa e também alcançou sucesso prático. Por exemplo, o Google relatou que a aplicação da priorização de casos de teste pode reduzir significativamente o tempo para encontrar testes com falhas (ELBAUM; ROTHERMEL; PENIX, 2014). Existem diversos algoritmos para priorização de casos de teste. No entanto, geralmente diferem em entrada de dados e abordagem. Essas abordagens variam de heurísticas mais simples (por exemplo, ordenar os testes por cobertura de código-fonte) a questões mais avançadas, como modelos de aprendizado de máquina (por exemplo, Learn To Rank) (ZHAI; JIANG; CHAN, 2014; MUKHERJEE; PATNAIK, 2021; KHATIBSYARBINI et al., 2018; OMRI; SINZ, 2023). Cada algoritmo pode se comportar de maneira diferente dependendo do projeto de software ou entrada de dados, então escolher um algoritmo TCP adequado torna-se de grande importância para obter resultados satisfatórios (ELBAUM et al., 2004).

Para solucionar o desafio em questão, propomos o TPVis (do inglês, *Test Prioritization Visualization*), um conjunto de ferramentas de análise visual para inspecionar metadados de casos de teste e comparar métodos TCP. O TPVis foi desenvolvido em colaboração com profissionais e pesquisadores de teste de software e consiste em um sistema interativo com 12 ferramentas analíticas para facilitar a visualização e comparação de algoritmos de priorização. Até onde sabemos, este é o primeiro sistema de análise visual que tem como objetivo permitir a comparação de algoritmos TCP. Essas ferramentas podem abranger uma ampla gama de tarefas de análise e podem ser usadas para construir visualizações de diversas formas.

Demonstramos a eficácia de nossa ferramenta em dois casos de uso e também relatamos feedbacks recebidos por especialistas no assunto. Além disso, apesar da ferramenta ser demonstrada com um dataset específico, é importante observar que um conjunto de dados genérico pode ser passado para o sistema, permitindo que ele seja usado em diferentes projetos de software e integrado a pipelines de Integração Contínua e Entrega (CI/CD).

Sumarizando, neste trabalho, temos o objetivo de tratar do problema da escolha e inspeção de técnicas e algoritmos de priorização de casos de testes, visto que existe uma grande gama

destes na comunidade. Afim de atingir este objetivo, trazemos as seguintes contribuições específicas:

- Propomos o TPVis: Um conjunto de ferramentas construídas com tecnologias web que permite engenheiros de software, engenheiros de teste e outras pessoas de interesse relacionadas a software explorar e inspecionar como diferentes abordagens TCP podem se comportar em seu projeto.
- Introduzimos 12 ferramentas analíticas dentro do TPVis que ajudam a inspecionar o comportamento e desempenho de métodos de TCP. Tais ferramentas abrangem diferentes casos de uso, desde análise de metadados de teste até comparação de priorizações e evolução da cobertura de código-fonte.
- Enfatizamos a utilidade do TPVis explorando dois estudos de caso relacionados a projetos do banco de dados Defects4j (JUST; JALALI; ERNST, 2014).

1.2 ORGANIZAÇÃO DESTE TRABALHO

Esta seção descreve objetivamente como os capítulos estão estruturados.

- **Capítulo 2:** Traz uma referência teórica e conceitos essenciais à plena compreensão desta dissertação. Discutiremos mais aprofundadamente sobre métodos de priorização de testes e como tais são comumente avaliados na literatura. Também serão explorados os trabalhos no segmento de visualização de dados referentes a áreas correlacionadas.
- **Capítulo 3:** São discutidas tarefas relevantes que conseguimos elencar durante diversas reuniões com especialistas na área de teste de software e definimos os requisitos funcionais que precisam ser cumpridos pelo TPVis para atingirmos o objetivo do trabalho.
- **Capítulo 4:** Detalha a solução proposta quanto a seu pré-processamento de dados, sua usabilidade e todas as ferramentas analíticas disponíveis que dão suporte aos requisitos elencados no capítulo 3.
- **Capítulo 5:** São explorados dois casos de uso que dão suporte a utilidade do TPVis e analisam como diferentes ferramentas podem complementar-se a fim de realizar análises com diferentes propósitos.

- **Capítulo 6:** Discute sobre opiniões coletadas com os especialistas, bem como explora mais características e limitações da ferramenta proposta.
- **Capítulo 7:** Conclui nosso trabalho e traz possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 TESTE DE SOFTWARE

Podemos definir teste como o processo avaliativo para validar que um sistema testado está de acordo e corresponde às expectativas dos requerimentos levantados. Com isto, é esperado que esta atividade obtenha resultados diferentes do antecipado. Logo, podemos caracterizar o processo de teste como uma atividade de risco, visto que os profissionais responsáveis têm a tarefa de tomar a decisão do que deve ou não deve ser testado, necessária para manter a quantidade de testes gerenciável. Portanto, é necessário encontrar um equilíbrio onde o gasto com testes e a qualidade atingida são ambos satisfatórios (JAMIL et al., 2016). Mais formalmente, teste de software pode ser definido como uma verificação dinâmica (pois um programa pode responder de maneira diferente dado uma mesma entrada de dados) de que um sistema se comporta de forma esperada em um conjunto finito de casos de teste, selecionado a partir do usualmente infinito domínio de execução do sistema (BOURQUE; FAIRLEY; SOCIETY, 2014).

Na literatura, podemos observar que existem duas técnicas principais de teste. São elas a de caixa branca e a de caixa preta. Os testes de caixa branca são caracterizados principalmente pelo acesso a como o sistema funciona internamente (Ex.: Acesso ao código fonte). Isto o torna altamente efetivo em detectar e resolver problemas, pois estes podem ser detectados antes de atingirem maiores proporções e o testador tem excelente conhecimento do sistema. Por outro lado, os testes de caixa preta são feitos nos casos onde o testador não tem conhecimento do funcionamento interno do sistema.

Testes de software podem ser performados em 4 diferentes níveis. (**I**) Teste unitário, onde é verificado o funcionamento de elementos que podem ser testados de maneira isolada. Uma das principais vantagens de testes nesse nível é que suas execuções podem ser facilmente paralelizadas; (**II**) Testes de integração, que seriam o processo de fazer a verificação da interação entre diferentes componentes do sistema, dessa forma, engenheiros podem começar a abstrair-se do baixo nível do código e começar a focar em avaliar se a solução implementada é capaz de cumprir com os requerimentos do projeto. (**III**) Os testes de sistema focam em testar o comportamento geral do sistema como um todo. Neste nível, os testes unitários e de integração já foram capazes de detectar grande parte das falhas. No geral, testes de sistema costumam validar principalmente requisitos não funcionais do sistema (como performance e

segurança) e também interfaces com outras aplicações e dispositivos de hardware. (**IV**) Testes de aceitação (também conhecidos como testes de QA ou de validação) são utilizados para validar que o sistema está de acordo com o especificado pelo cliente. Apesar de comumente ser desempenhado por usuários, a equipe de desenvolvimento acompanha o processo.

Casos de testes também podem ser classificados por tipo, são eles: (**I**) Testes funcionais, onde observamos se o sistema está de acordo com os requisitos funcionais elicitados. Estes testes podem estar presentes em todos os níveis e podemos citar os testes de regressão como exemplo (**II**) Testes não funcionais são os que cobrem as características indiretas do sistema, as quais são geralmente esperadas independentemente de seus requisitos funcionais, como por exemplo, testes de segurança ou os testes de disponibilidade; (**III**) Testes Estruturais, que validam a estrutura do sistema internamente e é mais recomendadas para os níveis unitários e de integração.

À medida que o software envelhece, o custo de mantê-lo torna-se mais alto do que o de construí-lo. Com isso, os testes de regressão surgem como uma solução para validar que novas mudanças não quebram casos de uso anteriores. Um bom processo de teste de regressão deve ajudar a reduzir os custos de manutenção e melhorar a confiabilidade do software (WAHL, 1999). Existem dois tipos de testes de regressão: (**I**) Corretivo, onde a especificação do software não muda e os casos de teste podem ser reutilizados; (**II**) Progressivo, quando a especificação do software muda para acomodar uma nova funcionalidade ou melhoria e novos testes devem ser feitos. Em casos de reutilização da suíte de testes, pode ser empregada a abordagem *retest-all*, que vai testar todos os casos de teste do sistema, ou a abordagem *selective-retest*, que utiliza um subconjunto dos casos de teste para reduzir o custo de execução.

2.2 REDUÇÃO E SELEÇÃO DE CASOS DE TESTE

Uma abordagem para resolver o problema das grandes suítes de teste é a redução do número de casos de teste. Num cenário ideal, onde seria obtida uma redução máxima, seria necessário encontrar o menor subconjunto representativo dos casos de teste. Tal tarefa pode ser considerada desafiadora, visto que o problema é *NP-complete* e a maioria das técnicas depende de heurísticas, nem sempre produzindo um resultado perfeito (ROTHERMEL et al., 2002; ZHANG et al., 2011).

Como visto no trabalho de Khan et al. (2016), técnicas de redução costumam adotar diferentes tipos de abordagem, podemos citar por exemplo: (**I**) Baseadas em cobertura de código,

onde as técnicas selecionam ambiciosamente testes com a maior cobertura; (**II**) Baseadas em algoritmo de buscas, como os algoritmos genéticos; (**III**) Baseadas em programação linear inteira e (**IV**) Baseadas em similaridade.

O processo de seleção de casos de teste utiliza fatores como código fonte modificado e histórico de falhas (KAZMI et al., 2017). Neste âmbito, diversas abordagens foram propostas e incluem diferença textual no código fonte, fatiamento dinâmico e análise de fluxo de dados.

2.3 PRIORIZAÇÃO DE CASOS DE TESTES

Como já discutido, a priorização de casos de teste (TCP) não é um campo de pesquisa novo e possui uma vasta gama de literatura sobre o tópico. Uma abordagem são os algoritmos gananciosos, que encontram o próximo melhor teste para adicionar ao conjunto de testes priorizados. Esses são mais fáceis de implementar e menos custosos de executar (ROTHERMEL et al., 2001; ZHANG et al., 2013; LI; HARMAN; HIERONS, 2007). Outra abordagem é usar algoritmos genéticos, que tentam iterar gerações com o objetivo de melhorar uma função de *fitness* (que retorna quão boa é a priorização em alguma restrição, como tempo de execução ou taxa de detecção de falhas) (BAJAJ; SANGWAN, 2019). Modelos de aprendizado de máquina também foram explorados no âmbito priorização de casos de teste (TONELLA; AVESANI; SUSI, 2006; BUSJAEGER; XIE, 2016; OMRI; SINZ, 2023).

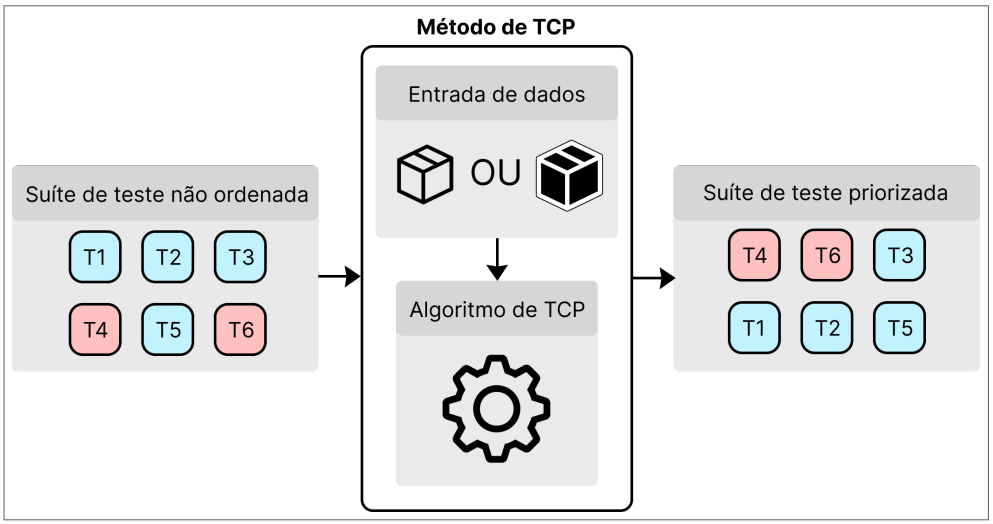
Além disso, as abordagens também podem variar quanto ao uso de diferentes tipos de dados (YOO; HARMAN, 2012). Entre as estratégias mais amplamente adotadas temos as abordagens *coverage-based*, que priorizam testes com base na cobertura de código, onde a premissa é de que a maximização rápida da cobertura estrutural do sistema também aumentará a chance de detecção de falhas antecipadamente (ROTHERMEL et al., 1999); As *history-based*, que utilizam dados históricos de falhas e execuções anteriores para identificar testes com maior probabilidade de detectar defeitos, possibilitando assim, sua priorização (KIM; PORTER, 2002); e as *cost-aware*, que consideram o custo e o tempo de execução dos testes, priorizando aqueles com maior relação custo-benefício para serem priorizados (ELBAUM et al., 2004). Além dessas mais comuns, também observamos outras na literatura, como a *human-based*, que incorpora o julgamento de especialistas para determinar a prioridade dos testes, que pode ser complementar a outras estratégias (SAPNA; MOHANTY, 2009); E a *requirement-based*, que foca em requisitos específicos do sistema para determinar a ordem de execução dos testes. Cada um desses métodos oferece uma abordagem distinta para otimizar a detecção de falhas e melhorar a

eficiência dos processos de teste (SRIKANTH; WILLIAMS, 2005).

Os métodos de priorização de casos de teste classificam os casos de teste com base em objetivos específicos. Um objetivo comum é detectar falhas nos testes o mais cedo possível. Como não é possível saber antecipadamente quais testes irão falhar, *proxies* podem ser utilizados para avaliar essa meta sem ter de executar os testes. Um exemplo de *proxy* seria a cobertura de código, que indica a quantidade de código executado por cada teste. Outra possibilidade de objetivo seria a redução de custos, onde os testes são priorizados com o objetivo de detectar versões falhas a um menor custo.

Os tipos de teste a serem priorizados podem variar e serem automatizados (por exemplo, unitários) ou manuais. No caso de testes automatizados, os algoritmos de priorização geralmente têm acesso a informações do código-fonte como entrada. Por essa razão, eles são classificados como *caixa branca*. Alguns exemplos de métodos de priorização que podem ser utilizados em contextos de caixa branca são os algoritmos gananciosos total e adicional (ROTHERMEL et al., 2001), que dão mais prioridade aos testes que cobrem maiores porções do software.

Figura 2 – Funcionamento geral de um método de priorização de testes.



Fonte: O autor (2024)

Ao lidar com testes manuais, o código-fonte geralmente não está disponível ou não é viável para os engenheiros de teste utilizarem. Esses casos dependem de técnicas caixa-preta, que podem usar resultados de testes anteriores como entrada ou apenas os metadados do teste (por exemplo, passos e descrição). Em relação a essa classe de algoritmos de TCP, podemos mencionar como exemplos I-TSD (FELDT et al., 2016) e STR (LEDRU et al., 2012). O I-TSD usa a distância de compressão normalizada para medir a diferença entre conjuntos de testes.

O STR, por outro lado, usa apenas strings de teste como entrada e seleciona o teste mais distante daqueles já selecionados. A literatura também traz diversas outras abordagens para este tipo de problema (QU et al., 2007; XU; DING, 2010; HETTIARACHCHI; DO; CHOI, 2016; ARAFEEN; DO, 2013).

Neste material, iremos nos referir à junção entre o algoritmo e o tipo de dado de entrada como um "método de TCP", ou simplesmente "método", representando assim, uma possibilidade de utilização de um algoritmo específico. A figura 2 traz um breve resumo de como um método TCP é aplicado.

2.4 AVALIAÇÃO DE MÉTODOS DE PRIORIZAÇÃO DE TESTES

A métrica mais comumente usada para avaliar a priorização de testes é a *taxa média de detecção de falhas* (APFD). A Fórmula 2.1 mostra como calcular o valor de APFD para uma suíte de testes priorizada contendo n testes e cobrindo m falhas; TF_j representa a posição na priorização do primeiro teste a revelar a falha j.

$$\text{APFD} = 1 - \frac{\sum_{j=1}^{m} TF_j}{nm} + \frac{1}{2n} \tag{2.1}$$

Os valores de APFD variam de 0 a 1; quanto maior o valor, mais rápido uma suíte de testes priorizada encontra falhas. Para ilustrar melhor este conceito, demonstramos o cálculo do APFD em um exemplo de uma suíte de testes hipotética contendo 5 testes e que cobre 5 falhas de software, mostrado na Tabela 1.

Tabela 1 – Exemplo de cálculo do APFD em uma suíte de testes hipotética. Um X em uma célula indica que o caso de teste (linha) detecta a falha correspondente (coluna).

Teste	**Falha**				
	F1	F2	F3	F4	F5
A	X				X
B	X			X	X
C	X	X	X	X	X
D					X
E					

Dada a sequência estipulada de casos de teste, denotada como ABCDE, pode-se calcular a métrica APFD considerando o primeiro teste a revelar cada falha (TF_j). Especificamente, TF_1 é atribuído o valor de 1, refletindo o cenário em que teste inicial descobre a falha F1.

Subsequentemente, TF_2 recebe o valor de 3, indicando a posição do caso de teste que identifica a segunda falha, e assim por diante. Considerando que n e m seriam iguais a 5, nosso valor de APFD seria 0.7.

Embora a métrica APFD seja comumente usada, ela possui várias limitações. Por exemplo, não levar em consideração a gravidade das falhas, tratando-as de forma igual, independentemente de seu impacto. Em cenários do mundo real, detectar bugs críticos cedo é mais crucial do que encontrar falhas menores, mas o APFD não considera esse aspecto. Além disso, o APFD também desconsidera o custo de execução dos casos de teste, que pode variar devido ao consumo de diferentes recursos e tempo de execução. Por fim, o APFD foca apenas na detecção de falhas e não considera outros fatores de qualidade, como o diagnóstico de falhas.

$$\mathrm{APFD}_C = \frac{\sum_{j=1}^{m} \left(f_i * \left(\sum_{i=TF_j}^{n} t_i - \frac{1}{2} * t_{TF_j} \right) \right)}{\sum_{j=1}^{n} t_j * \sum_{j=1}^{m} f_j} \tag{2.2}$$

$$\mathrm{APFD}_C(\text{Simplificada}) = \frac{\sum_{j=1}^{m} \left(\sum_{i=TF_j}^{n} t_i - \frac{1}{2} * t_{TF_j} \right)}{\sum_{j=1}^{n} t_j * m} \tag{2.3}$$

Várias métricas tentam resolver esses problemas, como o APFDc (ELBAUM; MALISHEVSKY; ROTHERMEL, 2001), que leva em conta o custo de execução dos testes. Esta métrica é uma adaptação da APFD e pode ser calculada pela fórmula 2.2. Nela, *fi* refere-se a severidade da *ith* falha e *tj* refere-se ao custo do *jth* teste priorizado. Porém, na prática, é difícil determinar a severidade exata da falha de antemão. Então a APFDc simplificada trata todas as falhas com a mesma severidade e está representada na fórmula 2.3 (EPITROPAKIS et al., 2015).

Ainda existe a necessidade de avaliar um algoritmo de priorização sem precisar executar a suíte de testes. Para isso, foi proposta a APXC (LI; HARMAN; HIERONS, 2007; HAO et al., 2016). Sua fórmula é a igual a 2.1 porém *tfj* representa o primeiro teste que cobre unidades estruturais, *j* e *m* referem-se ao número total de unidades estruturais cobertas pela suíte de testes. A principal diferença entre APFD e APXC é que em vez de avaliar o quão rápido as falhas são detectadas, é avaliado o quão rápido a cobertura do código cresce. Com base na APXC, podemos ter a APBC (o quão rápido uma priorização cobre blocos) e a APSC (o quão rápido uma priorização cobre instruções).

Tendo em vista que APFD caracteriza o quão rápido falhas são detectadas, surge o problema de definir o que seria "rápido" no contexto da priorização de testes. Com base nisso, (LV; YIN; CAI, 2013) propôs a *weighted gain of faults detected* (WGFD), representada na fórmula 2.4. *n* representa o número de testes, *r(i)* representa a taxa de detecção de falhas do teste *i*

e *w(i)* é o peso dado a taxa de detecção de falhas *r(i)*. A ideia é que dado que a detecção de casos de teste em diferentes instantes pode influenciar de maneiras distintas a avaliação de desempenho de uma técnica de priorização, é necessário atribuir pesos variados às taxas de detecção de falhas dos diferentes casos de teste.

$$\mathrm{WGFD} = \sum_{i=1}^{n} w(i) * r(i) \tag{2.4}$$

Por conta da grande gama de restrições possíveis no problema de priorização de testes, nem sempre todas as falhas podem ser reveladas pela suíte de testes. Como já vimos, algumas técnicas eliminam testes específicos a serem executados. Por isso, em (WALCOTT et al., 2006) foi proposta a ideia de atribuir penalidades às falhas não reveladas. Além disso, também é possível que entre versões sejam executadas diferentes quantidades de testes. Para resolver ambos problemas, foi proposta a *APFD normalizada* (NAPFD) (QU; COHEN; WOOLF, 2007) apresentada na fórmula 2.5, onde p representa o número de falhas detectadas pelo conjunto priorizado dividido pelo número de falhas total.

$$\mathrm{NAPFD} = p - \frac{\sum_{j=1}^{m} TF_j}{nm} + \frac{p}{2n} \tag{2.5}$$

Além da NAPFD, autores desenvolveram também a RAPFD (WANG, 2015), que considera a restrição de recursos de teste (que determina a quantidade de testes que podem ser executados). Também foram propostos modelos para realizar avaliações de metodologias de regressão (inclusive priorização de testes) (DO; ROTHERMEL, 2006; DO; ROTHERMEL, 2008).

Dada a dimensão e complexidade das suítes de teste, bem como a variedade de métodos de priorização, argumentamos que o uso exclusivo de métricas não é suficiente para que as pessoas de interesse tomem decisões informadas sobre a abordagem apropriada para seus casos de uso. O objetivo do TPVis é tornar esse processo mais eficiente, permitindo a avaliação de diferentes aspectos dos métodos de TCP.

2.5 VISUALIZAÇÃO DE DADOS EM ÁREAS RELACIONADAS

Visualização de dados na área de engenharia de software não é uma novidade. Trabalhos encontrados na literatura utilizam de visualização de dados para avaliar testes em diferentes aspectos, não exclusivamente no ramo de priorização. É possível utilizar a visualização de dados para, por exemplo, explorar diversidade de suítes de teste (MUDY; MANSSON, 2021).

A diversidade de testes utiliza geralmente de alguma medida de distância, como distância de *Jaccard* para identificar testes diferentes uns dos outros. Tal diversidade é a base para alguns algoritmos de priorização (como observado no trabalho de Yang e Chen (2022)), visto que ordenar testes numa ordem que maximize a diferença entre consecutivos testes pode auxiliar a cobrir a maior parte do software executando menos testes.

Também existem revisões literárias que tratam de visualização de software. Caserta e Zendra (CASERTA; ZENDRA, 2011) exploram diversas técnicas e é possível observar que estas variam desde representações até o que tentam transmitir. Algumas são em 3D e fazem analogias a cidades, outras fazem abordagens utilizando de diagramas em 2D. O trabalho observa que as visualizações operam a níveis de linha de código, classe e arquitetura e são focadas ou na evolução do software ou num ponto específico de tempo. Já em trabalhos como o de Strandberg (2017) é feita uma análise mais restrita buscando trabalhos que utilizem especificamente mapas de calor.

Eriksson e Örneholm (2021) realizaram um estudo com usuários para analisar visualizações de testes encontradas por meio de uma revisão de literatura. Os entrevistados puderam avaliar o quão útil achavam cada uma das visualizações. No geral, conclui-se que a maioria das pessoas acredita que existe espaço para visualização na área, mas ainda é necessária mais investigação.

Também foi feita uma avaliação da evolução de cobertura de código por meio da utilização de visualizações baseadas em matrizes (DREEF; PALEPU; JONES, 2023). O sistema proposto, denominado Morpheus, tem a capacidade de construir matrizes que podem representar cobertura de código e falhas reveladas por tais coberturas numa série temporal. Dessa forma, possibilitando a análise da evolução da suíte de testes. O trabalho se saiu bem em possibilitar com que desenvolvedores entendessem melhor o papel de testes e a compreender a suíte como um todo.

Strandberg, Afzal e Sundmark (2018) demonstraram que visualizar dados de resultados de testes pode auxiliar nos processos de tomada de decisão dos stakeholders. Além disso, também propuseram o Tim (STRANDBERG; AFZAL; SUNDMARK, 2022), um sistema projetado para a visualização de dados de resultados de testes. Embora as características do TPVis estejam alinhadas com as do Tim, nosso sistema foca na inspeção da TCP, em vez de se concentrar apenas nos resultados dos testes. No trabalho de Cornelissen et al. (2007), as visualizações foram baseadas em testes JUnit e diagramas de sequência UML, que se mostraram eficazes para uma melhor compreensão do funcionamento interno do projeto. Hammad et al. (2020) também propuseram outra visualização para auxiliar na compreensão de casos de teste falhos,

que pode gerar visualizações em múltiplos níveis (software, pacote e classe) e foi validada através de um estudo de usuários. Alguns outros trabalhos também visualizam suítes de testes a nível de código-fonte (JONES; HARROLD; STASKO, 2002; BRANDT; ZAIDMAN, 2022). Apesar de encontrarmos trabalhos anteriores relacionados à visualização de suítes de testes, não encontramos literatura relacionada à visualização da priorização de casos de teste.

A visualização de testes é um tópico relacionado, dado que algoritmos de priorização são apenas, diferentes formas de ordenar casos de teste. O Rankexplorer considera mudanças de classificação ao longo do tempo e propõe uma visualização para rastrear as mudanças e diferentes atributos (SHI et al., 2012). Outra abordagem, LineUp (GRATZL et al., 2013), permite a visualização de múltiplas classificações lado a lado, exibindo múltiplos atributos heterogêneos. Mylavarapu et al. (2019) compararam sete outras abordagens em um estudo coletivo com 180 participantes, concluindo que cada método tem diferentes vantagens e desvantagens, destacando que mais de uma visualização pode ser necessária para resolver o problema.

2.6 INTEGRAÇÃO E ENTREGA CONTÍNUA DE SISTEMAS

Pesquisas demonstraram que um dos principais culpados pelos prazos perdidos são as ineficiências operacionais, com o *DevOps* (operações de desenvolvimento) emergindo como uma solução para automatizar processos operacionais. Essa automação é crítica para atender às crescentes demandas por entregas rápidas e contínuas de software, tornando o CI/CD (integração e entrega contínuas) cada vez mais popular na indústria de TI.

A integração contínua, especificamente, envolve um conjunto de práticas de engenharia de software destinadas a acelerar a entrega de software, reduzindo os tempos de integração (STOLBERG, 2009). Isso é alcançado por meio de um *framework* de integração contínua que geralmente detecta alterações no repositório de código-fonte automaticamente e suporta a execução de etapas-chave, incluindo compilações de aplicativos e execuções de casos de teste. Além disso, processos de otimização de suíte de testes, como redução ou priorização, são executados nesta fase. Se a compilação do software ou os testes falharem, o processo é interrompido para evitar a transição para o processo de entrega contínua, que só pode começar após todas as etapas de integração serem concluídas com sucesso.

A entrega contínua automatiza o processo de provisionamento de hardware, tipicamente usando recursos em nuvem para implantar o aplicativo em diversos ambientes (VIRMANI, 2015). Alguns benefícios de automatizar esses processos incluem: (**I**) Tempo economizado, pois não

é mais necessário dedicar tempo humano aos processos de execução de testes e implantação da aplicação; (**II**) A implantação torna-se um processo consistente e reproduzível e (**III**) Permite o acesso dos desenvolvedores a ambientes mais próximos da produção (por exemplo, homologação e garantia de qualidade, que operam em *hardware* similar ou igual ao da produção).

3 PLANEJAMENTO DO SISTEMA TPVIS

Para construir qualquer tipo de software, em especial de visualização de dados, é necessário compreender quais serão as principais tarefas a serem realizadas pelos usuários finais ou pesquisadores. No TPVis, seguimos uma metodologia iterativa, durante a qual tivemos múltiplas reuniões com três especialistas na área (um profissional de engenharia de software e dois pesquisadores) que tinham experiência em testes de software. Durante algumas dessas reuniões, conseguimos elicitar colaborativamente as tarefas e requisitos funcionais do sistema.

1. **Explorar os dados de diferentes métodos:** Poder visualizar lado a lado diversas opções de priorização é essencial na tarefa de encontrar a mais adequada para o projeto em questão. Além disso, é fundamental que todo o conjunto de dados esteja organizado de maneira simples.

2. **Inspecionar metadados dos testes**: Afim de investigar e comparar métodos, é necessário compreender os dados dos testes em questão. Métricas como cobertura de código ou até mesmo o nome de classes de teste ajudam a compreender melhor que áreas do software são testadas ao longo da execução priorizada da suíte de testes.

3. **Avaliar performance histórica de métodos**: Avaliar como métodos de priorizações performam com o avanço do desenvolvimento do software pode revelar mudanças de performance de priorização em marcos específicos, como introdução de um novo conjunto de grandes funcionalidades ou a introdução de novos testes.

4. **Comparar métodos de priorização além de uma única métrica:** Tentar identificar características nas priorizações pode fornecer percepções sobre como encontrar uma priorização ideal, mesmo que seja feita manualmente por um engenheiro utilizando como base observações feitas.

3.1 REQUISITOS FUNCIONAIS

Um importante passo no desenvolvimento de qualquer sistema de software é o levantamento de requisitos funcionais. Tais requisitos são essenciais para que o projeto seja desenvolvido de acordo com os tipos de tarefas que softwares almejam auxiliar.

[R1] Permitir a avaliação efetiva do desempenho da priorização: o sistema deve suportar diferentes resumos visuais para permitir a avaliação do desempenho de cada método de priorização a partir de múltiplas perspectivas, inclusive os metadados dos testes;

[R2] Permitir a avaliação histórica dos métodos de priorização: projetos de software são entidades em evolução. Alguns métodos de TCP podem se tornar mais eficientes à medida que novas versões de um projeto são criadas e testadas. Portanto, observar como diferentes métodos de TCP desempenham em diferentes versões do software é importante;

[R3] Permitir a comparação efetiva de diferentes métodos de TCP: as visualizações e interações no sistema devem ser flexíveis para permitir efetivamente a comparação de múltiplos métodos de priorização em um dado projeto de software;

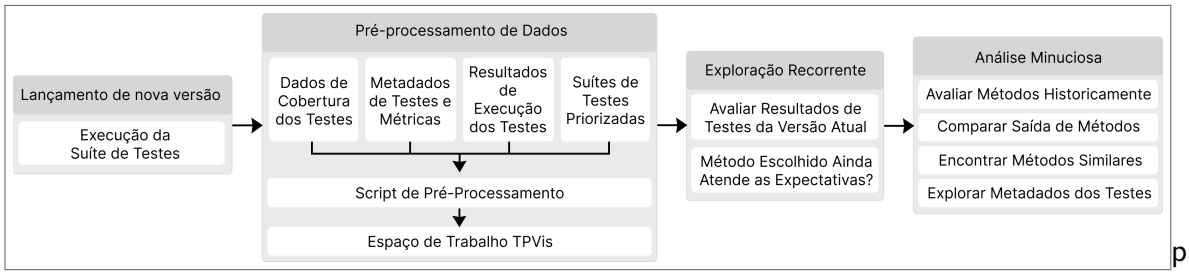
[R4] Permitir a análise de como diferentes métodos de TCP cobrem o código-fonte: A cobertura de código-fonte é um aspecto importante do teste de software. Por essa razão, é importante entender o comportamento da cobertura do código-fonte ao executar os testes seguindo a ordem dada por diferentes métodos de TCP. Além disso, um projeto de software é frequentemente organizado em uma estrutura hierárquica, e é importante poder entender como as priorizações cobrem diferentes níveis dessa hierarquia, como classes e pacotes;

[R5] Suportar diferentes estratégias e objetivos analíticos: Os tomadores de decisão do sistema incluem testadores de software profissionais, bem como pesquisadores de engenharia de software. Diferentes usuários têm diferentes objetivos e podem usar um conjunto diversificado de sequências analíticas na exploração dos dados. Portanto, o sistema deve se adaptar ao fluxo analítico do usuário.

4 O SISTEMA TPVIS

Neste capítulo apresentamos o sistema TPVis, idealizado e desenvolvido para permitir a análise de métodos de priorização de testes aplicados a reais projetos de software. Como será descrito a seguir, o sistema consome dados numa estrutura flexível e padronizada, visando permitir sua utilização e integração nos mais diversos contextos.

Figura 3 – Visão geral do sistema TPVis e como ele comumente pode ser utilizado.



Fonte: O autor (2024)

O TPVis é utilizado para fazer a análise de versões do software (que podem ser, por exemplo, commits ou qualquer outro tipo de entregável) com seus resultados de teste conhecidos e priorizações já previamente computadas. Em geral, a Figura 3 mostra o processo analítico usual como uma série de 4 etapas principais. Após cada versão candidata do sistema, é necessário executar a suíte de testes para garantir que o sistema funciona conforme o esperado. Em seguida, cada projeto deve implementar e executar um *script* de pré-processamento (que deve ser implementado para cada projeto de software e levará em consideração as particularidades das pilhas tecnológicas utilizadas) para reunir todas as informações necessárias e construir ou atualizar o espaço de trabalho do TPVis (conjunto de dados). Forneceremos informações detalhadas sobre esta etapa de pré-processamento na Seção 4.1. Essas duas etapas anteriores podem ser realizadas em um pipeline de CI/CD, técnica amplamente utilizada na indústria que automatiza os processos de integração e entrega de software, onde existirá um passo específico para a execução desse script que deve também enviar o conjunto de dados para um servidor de aplicação do TPVis. As duas etapas seguintes, exploração recorrente e análise aprofundada, ocorrem dentro da aplicação TPVis. A análise aprofundada permanece opcional, sendo necessária apenas se o método de priorização não atender às expectativas ou se for necessário analisar outros aspectos da suíte de testes. Vamos nos aprofundar na aplicação TPVis nas seções seguintes.

4.1 PRÉ-PROCESSAMENTO E ENTRADA DE DADOS

TPVis foi desenvolvido com flexibilidade em mente para que possa ser adaptável a diferentes fluxos de trabalho e tecnologias. Logo, tem como entrada um conjunto de dados genérico e depende de pré-processamento. O conjunto de dados deve ter a seguinte estrutura de arquivos:

```
/ (Raiz do espaço de trabalho)
├── prioritizations/
├── testsets/
├── call-graphs/
└── workspace.json
```

O arquivo workspace.json, estruturalmente apresentado na tabela 2, representa todos os metadados do workspace e é responsável pela árvore de priorizações. Seus principais componentes são: *subjects*, *prioritizations* e *folders*. Na pasta de *prioritizations*, temos diversos arquivos, cada um deles corresponde uma priorização única representada por uma lista em formato json de ids de testes ordenados. Na pasta *testsets* estão presentes os conjuntos de testes, incluindo ids, data de criação (*createdAt*) e quaisquer metadados relacionados a testes específicos. Na pasta call-graphs temos arquivos com dados de chamada de testes para cada *testset*. O arquivo de definição do workspace é composto em sua raiz esquema apresentado na tabela 2.

Tabela 2 – Estrutura de dados raiz do arquivo workspace.json

Estrutura de dados raiz do TPVis		
Chave	**Tipo**	**Função**
name	*string*	Nome do espaço de trabalho.
metrics	*"apfd"[]*	Métricas utilizadas no espaço de trabalho
data	*WorkspaceTreeNode[]*	Filhos raiz.

WorkspaceTreeNodes podem ser pastas, versões (*subject*) ou priorizações. Os nós de tipo pasta tem a finalidade exclusiva de auxiliar na organização de *subjects* e priorizações. Para este tipo de nó, deve ser utilizada a estrutura apresentada na tabela 3. *Prioritization subjects* representam objetos que devem ser priorizados, ou seja, versões do software. Logo, são responsáveis pelo conjunto de testes e o conjunto de falhas. Sua definição está representada na tabela 4.

Tabela 3 – Estrutura de um *WorkspaceTreeNode* de tipo pasta.

Pasta		
Chave	**Tipo**	**Função**
id	*string*	Identificador único da pasta.
name	*string*	Nome da pasta
type	"*subject*" | "*folder*" | "*prioritization*"	Tipo do nó da árvore. No caso de pasta, sempre o valor "*folder*"
markAs	"*project*" | "*algorithm*"	Representação visual na árvore (ícone)
data	*WorkspaceTreeNode[]*	Filhos da pasta

Tabela 4 – Estrutura de um *WorkspaceTreeNode* de tipo *subject*

Versão do software (*Prioritization Subject*)		
Chave	**Tipo**	**Função**
id	*string*	Identificador único do subject
name	*string*	Nome do *subject*
type	"*subject*" | "*folder*" | "*prioritization*"	Tipo do nó da árvore. No caso de versões, sempre o valor "*subject*".
testSetPath	*string*	Nome do arquivo de *testeset* disponível para esse subject na pasta *testsets*.
testKey	*string*	Nome do atributo que representa o id do teste no *testset*.
parentId	*string*	ID do nó pai deste subject
callGraphPath	*string*	Nome do arquivo de *testeset* disponível para esse *subject* na pasta *call_graphs*.
buildNumber	*int*	Número da construção desta versão. Utilizado para ordenação em algumas ferramentas.
treePath	*string*	Representa o caminha na árvore até chegar a esse nó. É apenas exibido visualmente como referência.
failures	*string*[]	Ids de teste que falharam nesta versão.
data	*WorkspaceTreeNode*[]	Filhos do *subject*

Tabela 5 – Estrutura de um *WorkspaceTreeNode* de tipo *prioritization*

Priorização		
Chave	**Tipo**	**Função**
id	*string*	Identificador único da priorização.
name	*string*	Nome da priorização
type	"*subject*" | "*folder*" | "*prioritization*"	Tipo do nó da árvore. No caso de priorização, sempre o valor "*prioritization*".
path	*string*	No do arquivo de priorização na pasta *prioritizations*.
testSetPath	*string*	Nome do arquivo de *testset* do *subject* pai desta priorização. Deve estar disponível na pasta *testsets*.

4.2 CONJUNTO DE DADOS DE EXEMPLO

Geralmente, projetos de software de grande escala utilizam de mecanismos de integração e entrega contínua (CI/CD). Tais metodologias facilitam a cooperação das equipes de desenvolvimento e operações por meio de um processo automatizado. CI/CD garante que cada alteração no código seja submetida a um processo automatizado de teste, construção e implantação. Tal metodologia acelera a transição do desenvolvimento para a produção e garante que os lançamentos de novas versões do software sejam consistentes e confiáveis (SHAHIN; BABAR; ZHU, 2017), evitando trabalhos manuais que são propensos a erros. Nesta seção, descreveremos como o espaço de trabalho utilizado para avaliação do TPVis foi construído em mais detalhes. No mundo real, tal processo idealmente deverá ser desencadeado a partir de uma automação.

Nosso conjunto de dados de exemplo foi construído a partir da junção dos dados fornecido por Miranda et al. (2018) (onde aqui chamaremos de conjunto de dados FAST), Defects4j (JUST; JALALI; ERNST, 2014) e Defects4j+M (MAHDIEH et al., 2020). O FAST é um algoritmo de priorização de testes que foi comparado com outros algoritmos comumente encontrados na literatura em projetos na linguagem de programação C, por meio de falhas falsas propositalmente plantadas, e projetos Java oriundos do Defects4j com falhas reais. No conjunto de dados FAST temos priorizações de testes produzidas por 37 diferentes métodos, cada um executado 50 vezes para tratar de suas naturezas não determinísticas. O conjunto de dados em questão é composto por 10 projetos de software, dos quais, utilizamos apenas os 5 com

falhas reais.

O Defects4j é um banco de dados abrangente de bugs reproduzíveis e suas correções correspondentes composto no momento por 17 projetos Java. Este *framework* simplifica o acesso a essas versões defeituosas e suas soluções ao alavancar a utilização dos sistemas de controle de versão para isolar *bugs* e ajustes. Embora a priorização de casos de teste seja uma das aplicações deste conjunto de dados, ele tem inúmeros usos potenciais no campo da engenharia de software, como a mineração e análise de bugs. Por fim, o Defects4j+M nos trás diversas métricas pré-calculadas para cada um dos casos de teste presentes no Defects4j, como por exemplo, a cobertura de cada teste.

Para construir este conjunto de dados agregado, foi desenvolvido um script em Node.js que gera a estrutura específica de um *workspace* TPVis. O primeiro passo para o pré-processamento de um dos projetos do nosso conjunto de dados de exemplo é realizar a leitura da matriz de falhas do FAST. Tal matriz de falha nos dá informações de que testes falharam em cada uma das versões. Em seguida, é feito o *checkout* do projeto em questão utilizando a interface de linha de comando do Defects4j na versão 1b. Este *checkout* é constante na versão 1b pois o FAST priorizou apenas até versões em que a falha pudesse ser encontrada pelo suíte de testes da primeira falha do software.

```
defects4j checkout -p <project> -v 1b -w /tmp/fast-d4j-project
```

Tendo o projeto disponível, é possível utilizar novamente a CLI do Defects4j para obter a lista de testes através do comando a seguir, que à retorna em stdout.

```
defects4j export -p tests.all
```

Em seguida, obtemos as métricas do Defects4j+M em formato CSV.

```
unzip -o <d4j+m-root>/<project>/1/Metrics.zip -d /tmp/metrics
```

Com isso, é criado um *prioritization subject* para a falha dentro de um projeto específico (pasta com o atributo *markAs* com o valor *project*). Além disso, é salvo o *testset* para esse *subject*. O processo para salvar o *testset* envolve além dos metadados disponíveis no Defects4j e Defects4j+m, a data de criação de cada teste, utilizado na ferramenta de idade de testes. Para obtenção da data de criação, primeiro é encontrado o caminho para todos os arquivos java no projeto:

```
find . -type f -name *.java
```

Em seguida, a lista é filtrada para conter apenas classes de teste e para cada classe de teste é executado:

```
git log –follow –format=%ad –date unix <caminho-de-cada-teste>
```

ou, se o projeto for versionado com SVN:

```
svn log –stop-on-copy –quiet <caminho-de-cada-teste>
```

Em seguida é salvo o *unix timestamp* no campo *creationTime*. Em resumo, o arquivo de *testset* é uma lista de objetos *json* que incluem ids de teste, *creationTime* e todos os outros metadados. Junto aos *testsets*, é obtido também os dados de cobertura dos testes. Para isso, é executado o comando de *coverage* do Defects4j e a saída do arquivo *coverage.xml* é processada para identificar que classes são chamadas por um determinado teste. Em seguida, um arquivo de chamada de classes é salvo dentro da pasta *call-graphs*. Com os *subjects* criados, são processadas as priorizações, sendo criado para cada uma delas um arquivo na pasta *prioritization* com uma *array json* dos ids de testes ordenados.

4.3 INTERFACE E COMPONENTES

A interface do TPVis (ver Figura 4) é composta por múltiplos componentes projetados para tornar seu uso fácil e intuitivo. Além disso, oferece uma gama de ferramentas analíticas para suportar os requisitos de design listados na Seção 3.1.

4.3.1 Componentes principais

Na árvore de priorizações apresentada na figura 4 (a), o *workspace* é exibido de acordo com as pastas, falhas, e priorizações definidas no workspace.json. Assim, permitindo uma visualização global e resumida dos dados disponíveis. Sua organização é totalmente flexível de acordo com a estrutura definida no estágio de pré processamento. Além da estrutura em si, é exibido um resumo numérico de cada priorização (Ex.: APFD) e a cor global de cada uma delas, que pode ser customizada individualmente pelo usuário em algumas das ferramentas analíticas e pode ser utilizadas para identificar priorizações específicas em múltiplas ferramentas simultaneamente.

Figura 4 – Uma visão geral da interface do TPVis. O sistema está organizado em várias visualizações, incluindo a (A) Árvore de Priorização e o (B) Painel de Ferramentas. Na área de trabalho, é possível instanciar as diferentes ferramentas analíticas apresentadas em (B) arrastando e soltando (F) elas em um espaço vazio. Neste exemplo, as ferramentas (C) Matriz de priorização, (D) Pilha de testes e (E) Tabela estão instanciadas. Os dados podem ser carregados nas diferentes ferramentas arrastando e soltando (G) os nós correspondentes para a sobreposição de gerenciamento de dados.

Fonte: O autor (2024)

Esse componente pode ser considerado um dos principais, visto que é o "ponto de partida" exploratório do usuário. Todos os dados disponíveis estão presentes de forma estruturada e podem ser arrastados aos seus devidos espaços nas ferramentas do sistema. No topo, é exibido um menu suspenso onde o usuário pode determinar o funcionamento do mecanismo de "arrastar e soltar". Dessa forma, possibilitando com que ele selecione se deseja arrastar todas as priorizações ao arrastar um grupo de nós ou apenas a melhor de cada método. Também é disponibilizada uma caixa de texto de busca, que filtra o nome dos nós a partir de expressões regulares.

A árvore de priorizações também mostra a métrica para cada nó (absoluta se for uma priorização ou média do grupo se for um grupo), indica uma cor customizada dada pelo usuário, e o tipo de nó por meio dos ícones (definidos no conjunto de dados por meio do atributo "*markAs*" do nó).

O *painel de ferramentas* (Figura 4 (b)) exibe todas as ferramentas analíticas disponíveis em nosso sistema. O usuário pode instanciar uma ferramenta arrastando e soltando a ferramenta no dashboard (canvas principal do sistema). Múltiplas ferramentas podem ser instanciadas ao mesmo tempo para formar dashboards, como mostrado na Figura 4 (c), (d) e (e). Os dados e parâmetros para cada ferramenta/visualização podem ser acessados através da *sobreposição de gerenciamento de dados* (ver figura 5), que pode ser aberta clicando no botão "*Manage*

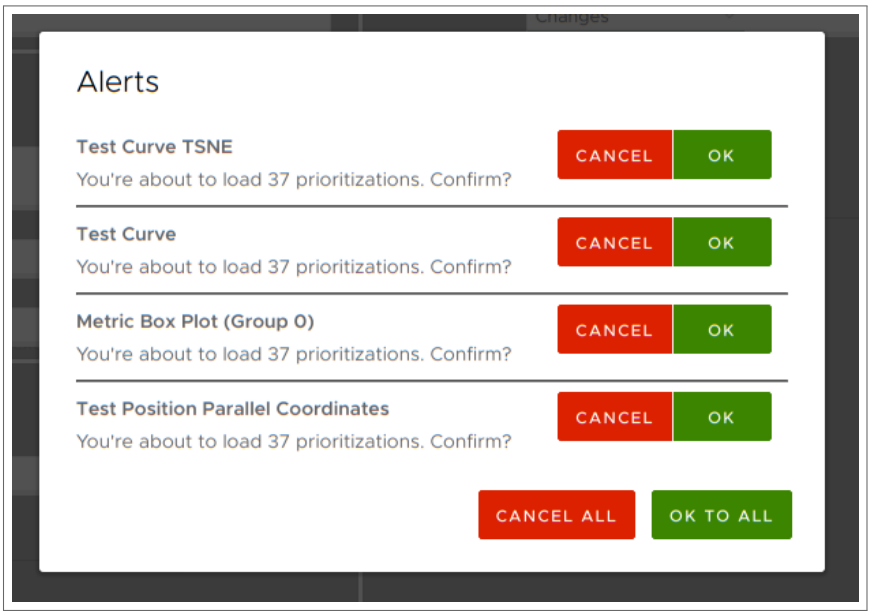
Figura 5 – Exemplo da interface de sobreposição de gerenciamento de dados

Fonte: O autor (2024)

Figura 6 – Opção de selecionar nós da árvore globalmente. Quando utilizado, tentará adicionar os nós arrastados em todas as ferramentas instanciadas no momento.

Fonte: O autor (2024)

Figura 7 – Alerta de confirmação para carregamento de dados.



Fonte: O autor (2024)

Data" no canto superior esquerdo de cada ferramenta. Graças a este componente, os usuários podem soltar nós (versões do software ou priorizações) da *árvore de priorização* em ferramentas analíticas. A depender da ferramenta são exibidas diferentes entradas de dados. Algumas ferramentas adicionam outras configurações nesta sobreposição, em tais casos, entraremos em mais detalhes sobre estas especificidades na seção 4.3.2. Também é possível selecionar uma priorização globalmente numa área que aparecerá quando o usuário está arrastando um dos nós de árvore de priorização (ver figura 6). Como os dados são carregados sob demanda, caso o usuário arraste muitas priorizações de uma vez só, também é exibido um aviso de confirmação representado na figura 7.

Finalmente, alguns *dashboards*, feitos com auxílio dos especialistas da área, foram considerados particularmente úteis em muitas aplicações e estão disponíveis através do botão "*View Presets*".

4.3.2 Ferramentas analíticas

Tabela 6 – Visão geral das ferramentas analíticas disponíveis no *TPVis*, quantas versões com falhas e priorizações cada uma pode considerar simultaneamente e também os requisitos funcionais cobertos por cada uma delas.

Ferramenta	**Dados de entrada**		**Atende os requisitos**
	Falhas	**Priorizações**	
Curva de Teste	1	n	**R1, R3**
Histórico de Falhas	n	Ø	**R5**
Pilha de Testes	1	Ø	**R1, R3**
Box Plot de Métricas	Ø	n	**R1, R2, R3**
Posições de Teste Paralelas	1	n	**R1, R3**
TSNE de Curva de Testes	1	n	**R3**
Linha de Evolução de Métricas	n	Ø	**R1, R2, R3**
Matriz de Evolução de Métricas	n	Ø	**R1, R2, R3**
Evolução da Cobertura	1	2	**R1, R3, R4**
Análise de Idade e Falhas de Testes	n	Ø	**R5**
Matriz de Priorizações	1	2	**R1, R3**
Tabela	1	1	**R1, R3, R4**

TPVis dispõe de diversas ferramentas para que o usuário possa realizar a análise de diferentes pontos de vista. Nesta subseção entraremos nos detalhes e utilidades de cada uma das ferramentas. A Tabela 6 mostra os dados de entrada esperados de cada ferramenta e sua

relação com os requisitos do projeto.

Curva de Testes: Esta ferramenta permite ao usuário selecionar uma versão com falhas e um conjunto de priorizações como entrada. Consiste em um gráfico de linhas que exibe uma linha para cada uma das priorizações de entrada. Seu eixo x representa a porcentagem da suíte de testes executada, e o eixo y representa a porcentagem de falhas detectadas. Quanto mais pontos a curva tiver localizados no canto superior esquerdo deste gráfico, mais cedo as falhas são encontradas nesta priorização. Na direita da Figura 8, podemos ver um exemplo desta ferramenta sendo usada para analisar um conjunto de priorizações. As curvas laranja e cinza representam as priorizações que encontram mais falhas mais cedo. A principal utilidade da curva de testes é avaliar em que partes de priorizações mais testes falhos estão sendo detectados, além de possibilitar a identificação de priorizações que detectam testes falhos em posições similares.

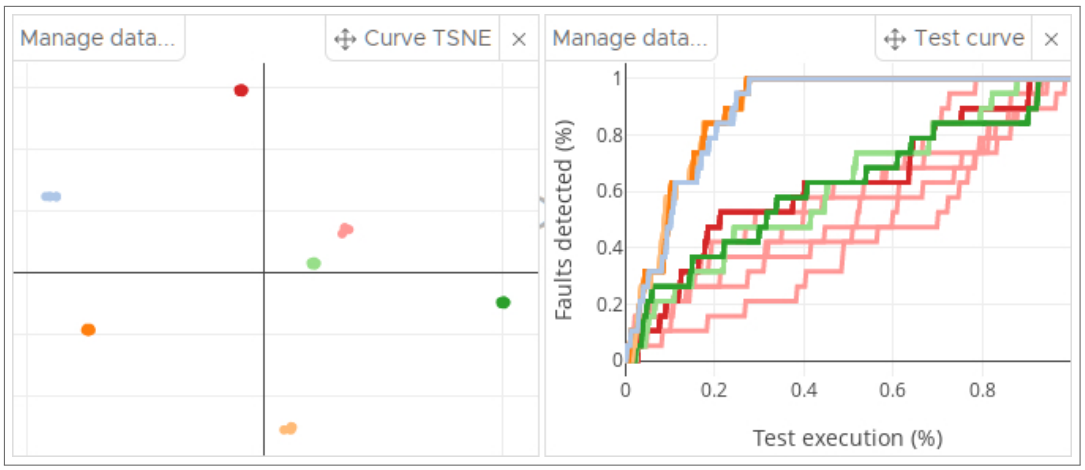
TSNE de Curva de Testes: Esta é uma ferramenta complementar a curva de testes. Consiste em um gráfico de dispersão que retrata a similaridade das curvas para múltiplas priorizações. Os pontos são posicionados usando o algoritmo TSNE (MAATEN; HINTON, 2008). Isso permite a identificação de priorizações que, apesar de diferentes, podem produzir resultados semelhantes em uma versão específica do software (curvas de teste semelhantes). Identificar priorizações com resultados semelhantes é essencial, permitindo que as partes interessadas escolham a opção que pode ser menos custosa para computar ou mais rápida para incluir nos processos do projeto. A Figura 8 destaca como essa ferramenta pode ser usada para identificar clusters. Além disso, os usuários podem selecionar pontos via *brushing*, o que filtra os dados em todas as outras ferramentas instanciadas.

Histórico de Falhas: Esta ferramenta resume um conjunto de versões de software e o número de testes falhados e passados em cada uma delas como um gráfico de barras, dessa forma, permitindo uma visão rápida de tais dados nos sujeitos de priorização carregados. A Figura 9 (b) apresenta um exemplo onde a maioria das versões com falhas possui uma única falha de teste.

Matriz de Priorizações: A ferramenta de matriz permite a comparação de duas priorizações distintas. Consiste em um gráfico de dispersão onde os eixos são as posições das priorizações. Um ponto representa cada teste, e as coordenadas consistem na posição do teste em ambas as priorizações.

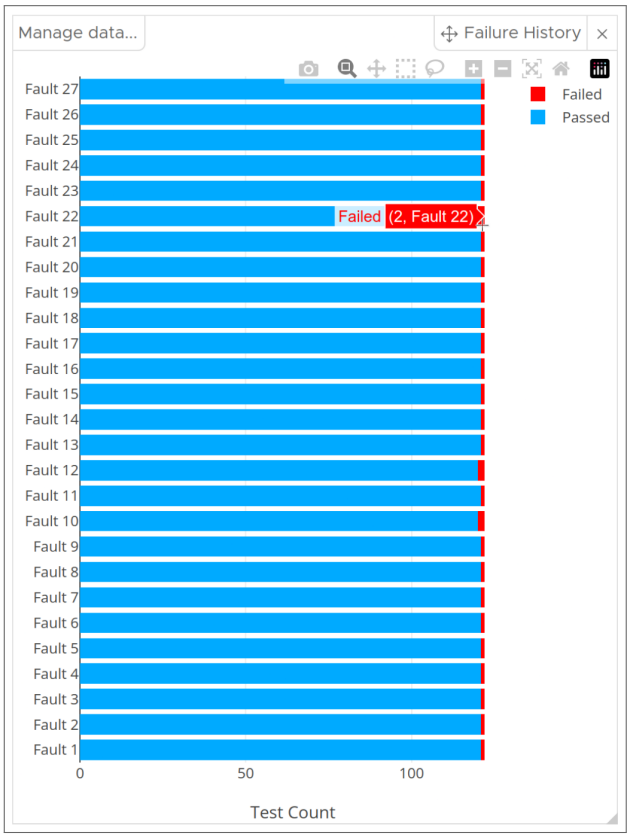
Na figura 10, podemos ver 3 casos. O primeiro, indicado pela letra "A", mostra priorizações similares nas quais é possível observar divergências mais significativas próximo ao final da exe-

Figura 8 – Ferramenta de curva de teste e ferramenta de TSNE de curva de testes carregadas com 6 priorizações para 7 métodos diferentes. Note que a ferramenta de TSNE de curva de testes agrupou as curvas de cada método juntas.



Fonte: O autor (2024)

Figura 9 – Ferramenta de histórico de falhas com 27 falhas carregadas.

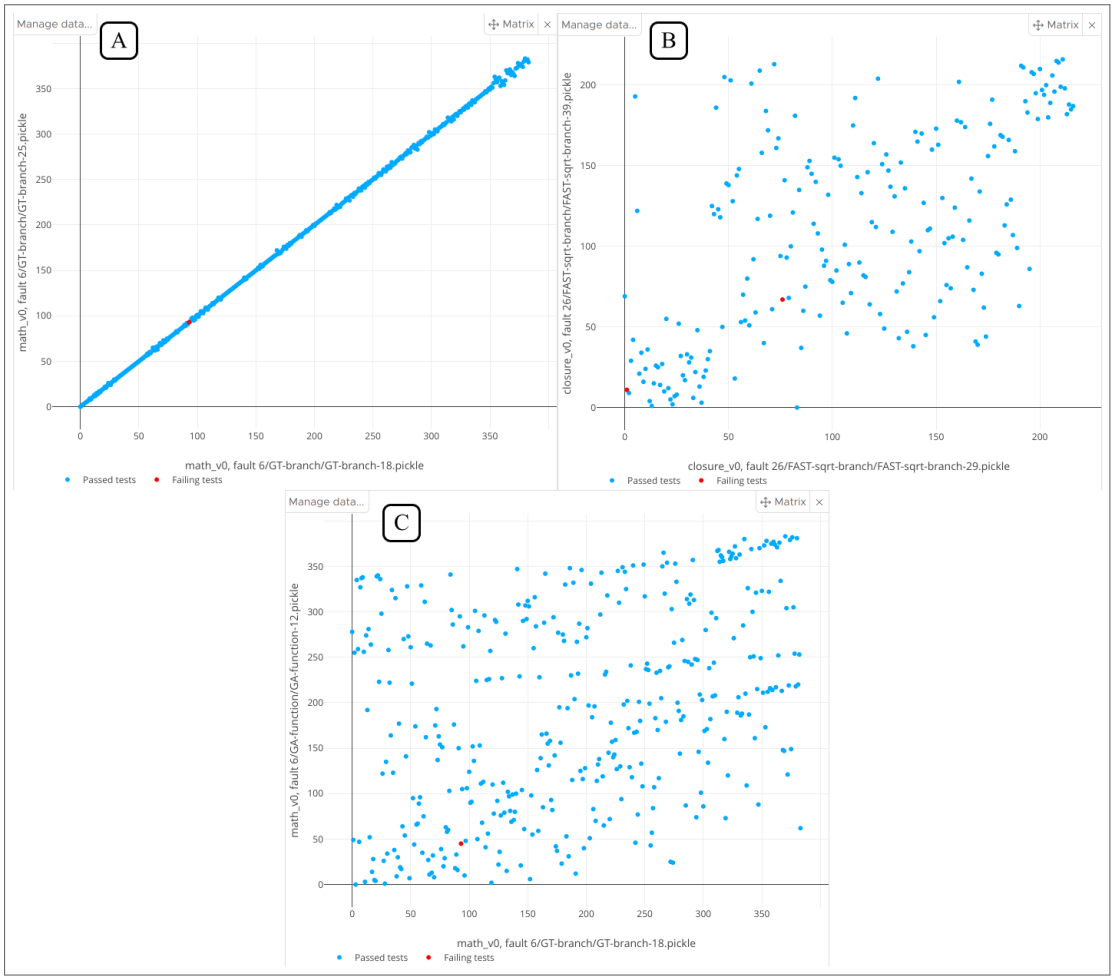


Fonte: O autor (2024)

cução de testes. No caso "B" vemos que a priorização produz três "blocos" de concordância. Ou seja, ambas priorizações chegam em concordância dos testes que devem estar no início, meio e fim. Mesmo que não concordem na posição dos testes nestes blocos. No caso "C" conseguimos ver priorizações bem diferentes. Este exemplo ilustra bem o valor entregue desta

ferramenta, que mais especificamente, seria encontrar padrões em diferentes priorizações.

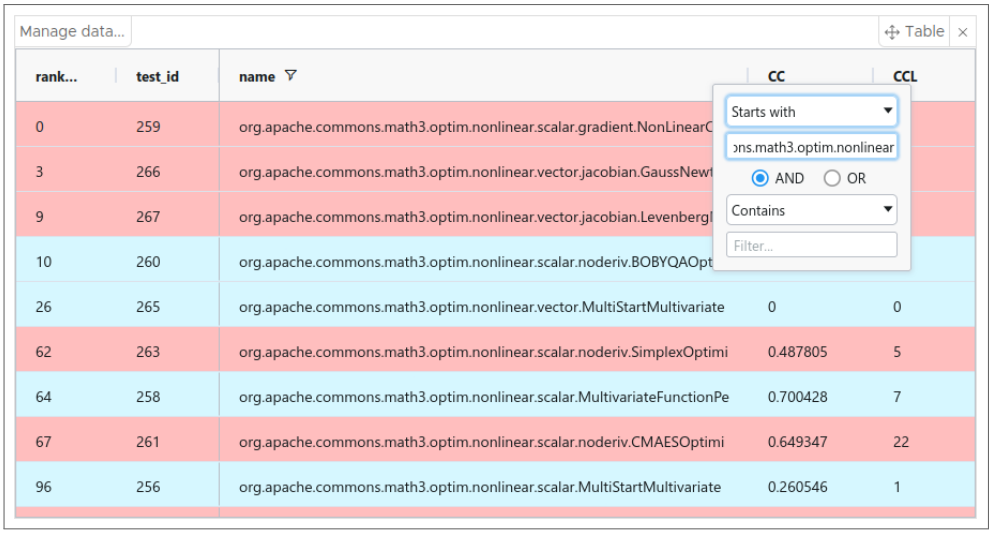
Figura 10 – Matriz de priorizações com (A) Priorizações bem similares; (B) Priorizações parcialmente similares e (C) Priorizações bem diferentes.



Fonte: O autor (2024)

Tabela: A ferramenta de tabela ilustrada na Figura 11 permite que os usuários inspecionem um conjunto de testes e seus metadados. Esses metadados devem ser definidos na etapa de pré-processamento e são exibidos em uma tabela dinâmica. IDs de testes e classificações (dada a priorização selecionada na *sobreposição de gerenciamento de dados*) são apresentados nas duas primeiras colunas fixas. Esta ferramenta é particularmente útil para inspecionar os metadados dos testes priorizados, visando encontrar padrões nos métodos de priorização. Além disso, a ferramenta permite classificar, filtrar e reordenar colunas, trazendo flexibilidade para qualquer tipo de análise. Adicionalmente, pode ajudar as pessoas de interesse a descobrir que um método está priorizando fortemente uma característica específica do programa ou é substancialmente dependente de um metadado específico do teste (por exemplo, cobertura de código).

Figura 11 – Ferramenta de tabela com sua funcionalidade de filtragem em evidência.



Fonte: O autor (2024)

Pilha de Testes: Esta ferramenta organiza os testes em um formato de pilha, com as priorizações no eixo y e os testes, representados como retângulos finos, empilhados lado a lado no eixo x, que seria o decorrer das priorizações. Retângulos azuis representam testes que passaram, e os vermelhos representam testes que falharam. Apesar de ter o mesmo propósito de curva de testes, esta ferramenta permite a identificação das posições dos testes falhados nas priorizações sem o problema de sobreposição de curvas. Por outro lado, esta abordagem não traz a noção de cumulatividade da detecção de falhas. Um exemplo pode ser visto na Figura 12.

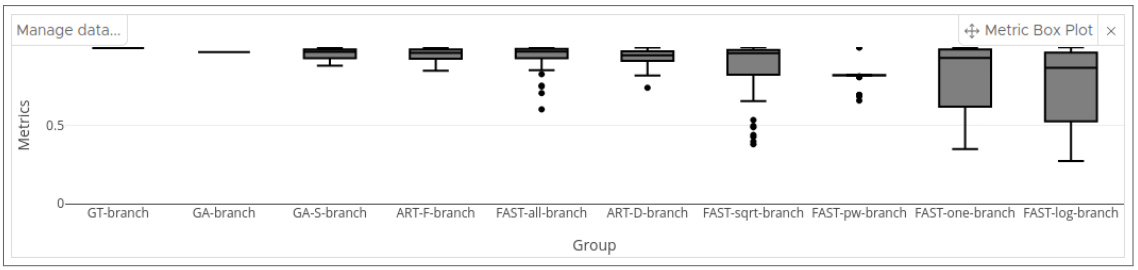
Figura 12 – Ferramenta de pilha de testes mostrando 6 priorizações. Podemos observar que a priorização azul clara detecta todos os testes com falha antes mesmo da suíte de teste atingir 30% de execução.

Fonte: O autor (2024)

Box plot de Métrica: Esta ferramenta consiste em um boxplot de uma métrica escolhida pelo usuário durante o estágio de pré-processamento (comumente o APFD) para a avaliação

das priorizações dos casos de teste. A Figura 13 mostra um exemplo. Adicionalmente, esta ferramenta permite que o usuário defina como agrupar as priorizações em cada caixa como desejar.

Figura 13 – Ferramenta de box plot de métricas com diversos métodos baseados em cobertura com critério *branch* carregados.



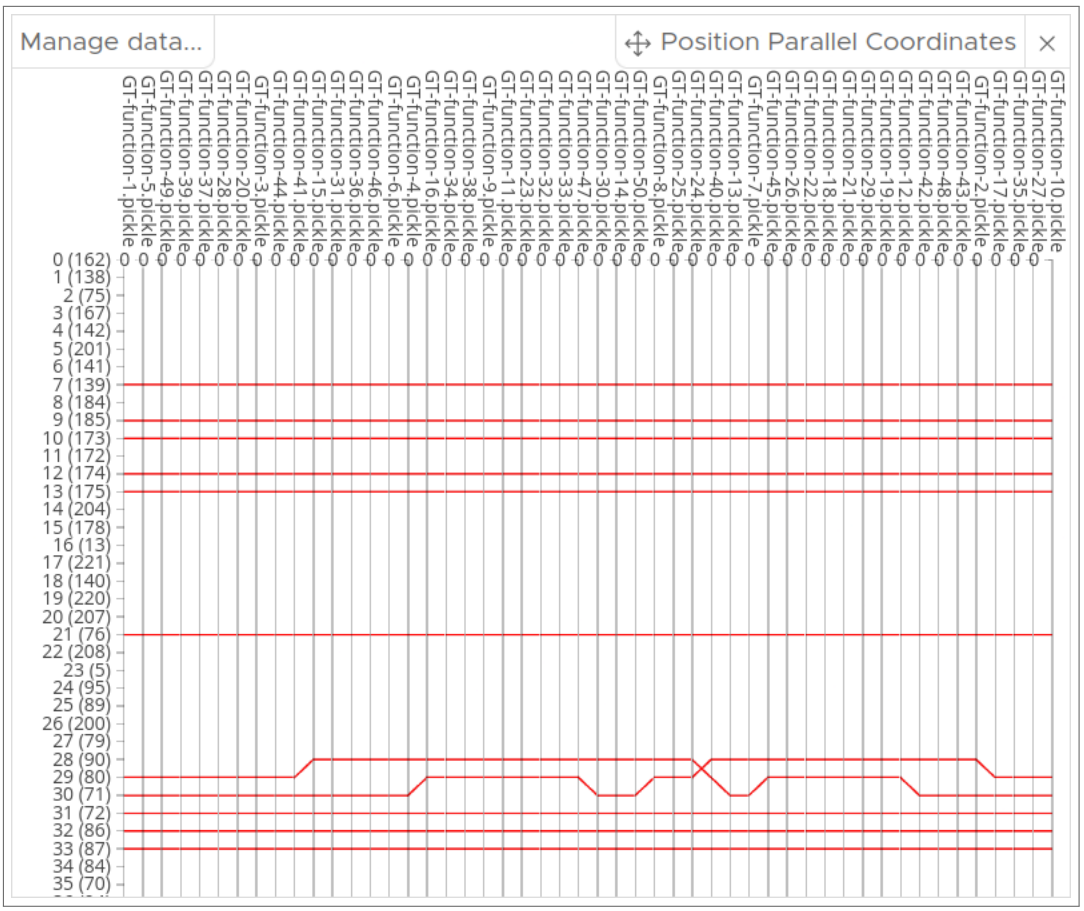
Fonte: O autor (2024)

Por exemplo, é possível plotar a distribuição do APFD de um único método em várias versões de software ou plotar múltiplos métodos TCP em uma única versão de software. Tendo em mente que métricas são a maneira mais comum de avaliar priorização de casos de teste na literatura, também é relevante dar esta possibilidade dentro do TPVis. Na forma proposta, é possibilitado com que usuários saibam de forma rápida e flexível como priorizações performam de acordo com uma métrica específica.

Posições de Teste Paralelas : Esta ferramenta consiste em um gráfico de coordenadas paralelas para comparar as classificações dos testes em múltiplas priorizações, similar à abordagem apresentada por Gratzl et al. (2013). Cada eixo corresponde a uma priorização, e cada linha corresponde a um teste. A linha conecta a classificação do teste correspondente nas diferentes priorizações. O usuário pode selecionar uma das opções de exibição: mostrar todos os testes, mostrar apenas testes que mudam de posição ou mostrar apenas falhas. Para minimizar a desordem visual, esta ferramenta usa o algoritmo de ordem de folhas otimizada (BAR-JOSEPH; GIFFORD; JAAKKOLA, 2001) para aproximar priorizações semelhantes (reordenando seus eixos). A Figura 14 mostra um exemplo onde grupos de testes são permutados em diferentes priorizações. A maioria dos testes estão suprimidos no gráfico porque a ferramenta está configurada para exibir apenas testes que falham. Esta ferramenta possibilita com que usuários possam ver rapidamente que testes variam de posição em muitas priorizações de uma vez só. Dessa forma, é possível, por exemplo, identificar diferentes priorizações de um mesmo método que priorizam testes falhos em posições muito diferentes.

Ferramentas de evolução métrica do método (linha e matriz): Como ambas as ferramentas têm o mesmo propósito (permitir observar como a performance dos métodos evoluem

Figura 14 – Ferramenta de posição de testes paralela indicando a posição de testes que falharam em diversas priorizações de uma mesma falha. É possível observar que os testes falhos 80 e 71 alternam entre si e com outros testes próximos.

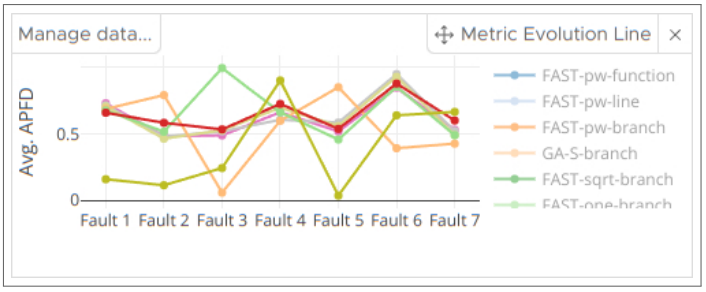


Fonte: O autor (2024)

com o passar do tempo), descreveremos as duas nesta seção. Estas ferramentas permitem a observação da evolução média da métrica do método através das versões do software (utilizam o atributo *buildNumber* dos sujeitos de priorização no conjunto de dados para ordenação). Considerando que representar dados temporais com muitas dimensões é um desafio (MORITZ; FISHER, 2018), introduzimos duas ferramentas para esse propósito. Uma usa um gráfico de linha, e a outra usa uma matriz (onde as linhas são ordenadas usando o algoritmo de ordem de folhas otimizada). Quando o conjunto de dados do usuário tem menos métodos de priorização, o gráfico de linha pode ser considerado mais adequado, visto que perceber a posição de pontos é mais fácil do que diferenciar cores. A Figura 15 exibe um exemplo da evolução métrica em linha, enquanto a Figura 16 exibe um exemplo da matriz de evolução métrica. Note que a versão em matriz tem o benefício adicional de agrupar métodos com desempenho semelhante (em relação à métrica média).

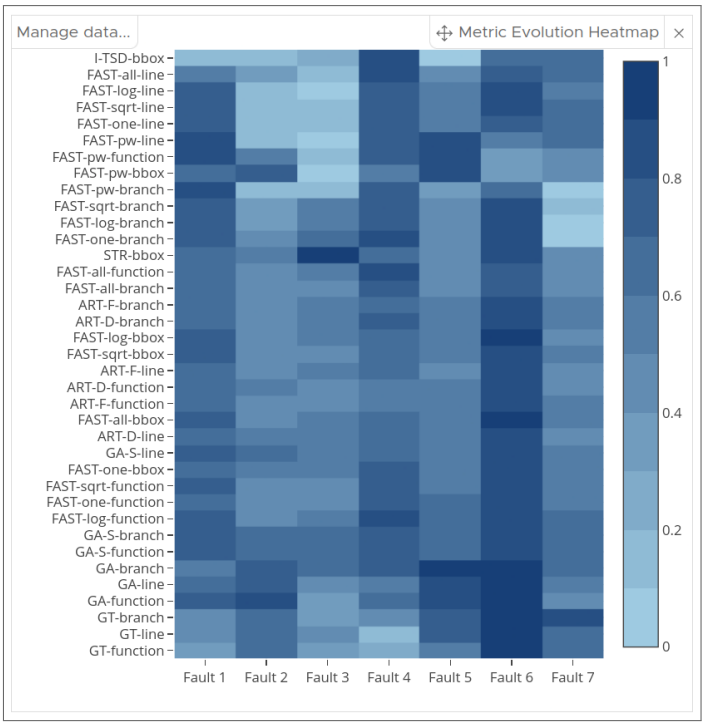
Análise de Idade e Falhas de Testes: Esta ferramenta não se relaciona diretamente com

Figura 15 – Linha de Evolução Métrica, filtrada para exibir apenas 5 métodos.



Fonte: O autor (2024)

Figura 16 – Ferramenta de matriz de evolução de métrica. Neste exemplo, fica evidente o quão simples versões com comportamentos atípicos podem ser identificados, como a falha 6 neste caso.

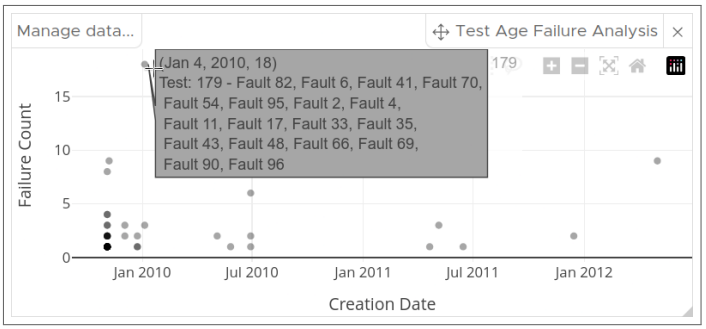


Fonte: O autor (2024)

priorizações e aceita apenas versões de software como entradas. Consiste em um diagrama de dispersão apresentado na Figura 17. O eixo X consiste na data de criação do teste (atributo de "*createdAt*" no "*testset*"), e o eixo Y consiste no número de vezes que o teste falhou. A análise de falhas por idade dos testes pode ser um aliado muito relevante para o engenheiro de testes, visto que exibe testes antigos que falham muito ou continuam falhando nas versões recentes do software. Pode-se argumentar que testes que se enquadram nesse critério devem ser priorizados em detrimento de outros.

Evolução da Cobertura: A cobertura de código desempenha um papel indispensável em inúmeras estratégias de priorização de casos de teste de caixa branca. Esta ferramenta espe-

Figura 17 – Exemplo da Ferramenta de análise de idade e falhas de testes, note que neste caso o teste 179 chega a ter o dobro de falhas (18) dos outros dois testes em segundo colocados (9).



Fonte: O autor (2024)

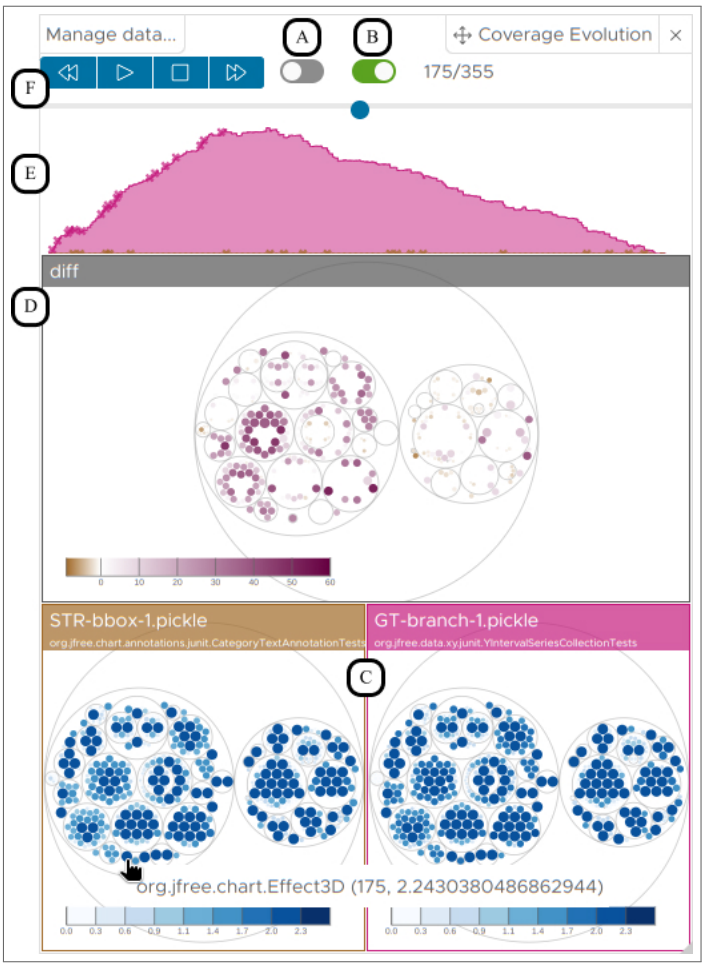
cializada foi desenvolvida para avaliar como a cobertura evoluí durante a execução das suítes de teste priorizadas. A visualização da evolução da cobertura suporta a análise de pares de priorizações, utilizando um layout de círculos empacotados que reflete a assinatura do pacote associada a cada classe (círculos coloridos) do programa. Graças a esta ferramenta, o usuário pode acompanhar que partes do seu sistema estão sendo mais interagidas a partir dos testes em diferentes pontos da execução da suíte de testes em duas diferentes priorizações.

A Figura 18 indica as seções da ferramenta de evolução da cobertura. O usuário pode configurar a ferramenta para exibir os dados em nível de classe ou pacote (a). No nível de pacote, as classes subjacentes são todas agregadas, enquanto cada classe é exibida individualmente no nível de classe. Também é possível alternar entre os modos absoluto ou incremental (b); O modo absoluto pinta uma classe em uma das priorizações se ela foi alguma vez interagida por um teste executado. No modo incremental, a cor depende da frequência com que a classe foi interagida.

O funcionamento da ferramenta é baseado nos controles de reprodução (F), onde o usuário pode optar por ver uma animação dos testes sendo exibidos sequencialmente, ou avançar e retroceder conforme julgar necessário.

Na seção inferior da ferramenta, destacada como (c) na figura, o usuário é apresentado representações visuais das priorizações carregadas, incluindo um indicador do teste sendo executado no ponto da priorização que está sendo exibido. Enquanto isso, a seção superior, indicada por (d), destaca a diferença absoluta na cobertura entre as priorizações no instante de tempo selecionado. As visualizações na parte inferior usam uma escala logarítmica para melhorar a legibilidade. O tamanho e o posicionamento da classe dentro do pacote (círculo mais claro) refletem a quantidade de interações que ela teve com os testes ao final da execução da suíte de testes.

Figura 18 – Ferramenta de evolução da cobertura com duas priorizações carregadas. (A) Alternar entre modo de classe/pacote; (B) Alternar entre modo absoluto ou incremental; (C) Inspeção individual de cobertura individual; (D) Visualização de diferenças de cobertura; (E) Resumo da cobertura das priorizações; (F) Controles de reprodução.



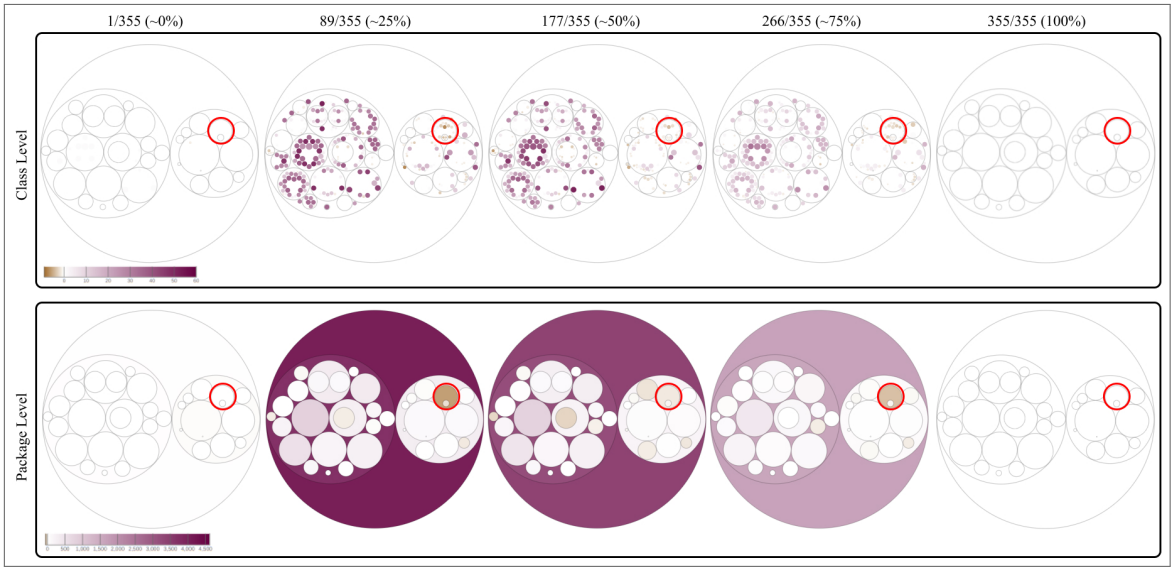
Fonte: O autor (2024)

Indicado como (e) está um gráfico de linhas representando a soma das diferenças, ajudando o usuário a saber qual priorização cobre mais do software em diferentes intervalos de tempo. Ao usar o modo absoluto, as linhas são exibidas cumulativamente e representam quanto do software foi chamado pelos testes executados.

Além disso, pode-se mapear classes para funcionalidades do programa, o que pode oferecer mais clareza semântica ao agrupar classes relacionadas sob um único componente. Para alcançar isso, a camada de gerenciamento de dados nesta ferramenta oferece funcionalidade de apelidos de classe, permitindo que os usuários mapeiem classes específicas para nomes que representam funcionalidades. Esse recurso também oferece flexibilidade para aplicar apelidos em massa ou filtrar a lista de classes. Com esse mapeamento, as classes podem ser representadas como funcionalidades, como "usuario.criar"e "usuario.editar", agrupando-as em círculos

específicos.

Figura 19 – Ferramenta de Evolução da Cobertura com dados dos métodos STR-BBOX e GT-Branch carregados. O pacote onde o STR-BBOX prioriza mais (apesar de o GT-Branch liderar em toda a priorização) é indicado em vermelho.



Fonte: O autor (2024)

Na Figura 19, trazemos um exemplo de caso onde a utilização da ferramenta de evolução de cobertura pode ser benéfica ao usuário final. Nela, podemos observar que temos a visualização da diferença de cobertura em diversos pontos da nossa priorização a nível de classes e pacotes. As duas priorizações carregadas foram produzidas por dois métodos, um caixa-preta e um caixa-branca, respectivamente. São eles o STR-BBOX e o GT-Branch. Sabendo que o algoritmo total ganancioso (GT) escolhe testes com base na cobertura do software como um todo e que o algoritmo STR prioriza testes com base em suas dissimilaridades, ao observar a imagem vemos um comportamento inusual. Como é de se esperar, a maior parte do sistema é coberto mais rapidamente pela priorização produzida pelo algoritmo GT (neste caso, indicado pela cor roxa). Isto pode ser observado mais facilmente com o uso do gráfico de linhas (Figura 18 (e)). Também é possível observar um pacote específico onde a priorização feita pelo algoritmo STR tem uma maior cobertura no decorrer de toda a execução da suíte de testes. Isso traz o entendimento de que os testes que testam as classes deste pacote específico são muito direcionados a este pacote, logo, algoritmos baseados em cobertura tendem priorizar menos estes testes. O algoritmo STR, em contrapartida, simplesmente busca testes mais diferentes dos selecionados, dando assim, uma maior chance dos testes deste pacote serem priorizados. Dessa forma, podemos visualizar que em casos onde gostaríamos de priorizar funcionalidades com testes muito específicos, algoritmos baseados em cobertura podem não ser uma boa

escolha.

4.3.3 Predefinições de espaço de trabalho

Tabela 7 – Composição de ferramentas das predefinições de visualização disponíveis.

Predefinição	**Ferramentas**
Visualização Histórica	Matriz de Evolução de Métricas
	Análise de Idade e Falhas de Testes
	Histórico de Falhas
Comparação de Métodos	TSNE de Curva de Testes
	Curva de Teste
	Box Plot de Métricas
Inspeção de Método	TSNE de Curva de Testes
	Curva de Teste
	Box Plot de Métricas
	Tabela

Compor *dashboards* pode se tornar uma tarefa repetitiva. Com isso em mente, três predefinições de visualização, que suportam três diferentes tarefas propostas por nossos especialistas do domínio, estão disponíveis e podem ser acessadas através do painel de ferramentas clicando em "*View Presets*" (Figura 4 (b)). Os três *dashboards* estão definidos junto às suas ferramentas na tabela 7. Selecionar uma predefinição irá instanciar as ferramentas correspondentes no dashboard.

A predefinição de visualização histórica pode ser considerada como um dos potenciais pontos de partida para um processo analítico do TPVis no contexto de um projeto de software. Graças a ferramenta de histórico de falhas, o usuário pode rapidamente observar a quantidade total de testes que passam e falham, dando uma intuição da saúde do sistema historicamente. Além disso, o usuário pode observar também testes que falham muito frequentemente e que métodos tem performado melhor para o projeto sendo analisado. Ou seja, com poucos cliques, o usuário pode observar diversos aspectos históricos do seu sistema.

As ferramentas dispostas na predefinição de comparação de métodos trazem algumas das mais apropriadas para tal finalidade. O intuito é possibilitar o carregamento de priorizações de diferentes métodos para que eles possam ser efetivamente comparados. Tal visão também é usualmente destinada a gestores de projetos que querem validar que métodos de priorização poderiam vir a ser mais apropriados para seus projetos.

A predefinição de inspeção de métodos traz as mesmas ferramentas da discutida anteriormente, mas agora, também com a ferramenta de tabela. Tal repetição pode ser esperada, visto que dado a natureza flexível das ferramentas do TPVis, uma mesma pode servir para diferentes propósitos analíticos. Esta *dashboard* tem o intuito de prover ferramentas para a inspeção de métodos de TCP e é direcionada a usuários que desejam desenvolver seus próprios métodos ou até mesmo, entender os pontos positivos de cada um a fim de auxiliar na escolha de um método específico.

4.4 DETALHES DA IMPLEMENTAÇÃO

O sistema é executado com dois componentes principais. Um *frontend*, construindo em Angular 16 (GOOGLE, 2024), Clarity Design System (VMWARE, 2024) e Gridster.js (DUCKSBOARD, 2024). E o *backend*, construído com Quarkus (QUARKUS, 2024) que apenas serve os dados para a aplicação, permitindo também a atualização de um *workspace* (para salvar métricas calculadas e cores customizadas). Nenhum processamento bruto é realizado pelo *backend*.

A comunicação entre o *backend* e *frontend* é feita via protocolo HTTP e consiste em apenas 5 endpoints, responsáveis por (**I**) Retornar workspace.json; (**II**) Atualizar workspace.json; (**III**) Retornar *testset*; (**IV**) Retornar priorização e (**V**) Retornar dados de cobertura.

Diversas bibliotecas de terceiros são utilizadas na construção do *frontend*. Todas as visualizações foram implementadas utilizando Plotly JS (PLOTLY, 2024), D3 (OBSERVABLE, 2024) e AG Grid (AGGRID, 2024). Além disso, quando o *workspace* é aberto, são buscadas priorizações sem métricas calculadas. Caso encontradas, são calculadas assincronamente com o uso do Pyodide (MOZILLA, 2024), um interpretador Python em WebAssembly.

A escolha de utilizar o Pyodide possibilita concentrar toda a lógica da aplicação no lado do cliente, simplificando o desenvolvimento e eliminando a necessidade de mais comunicação com o servidor *backend*. Essa abordagem permite que a lógica e o processamento sejam realizados diretamente no navegador, reduzindo a complexidade do projeto. Além disso, possibilita com que os possíveis futuros contribuidores do projeto possam focar em um conjunto de tecnologias menor. Apesar do uso do Phyton diretamente no navegador trazer limitações, principalmente, com relação à disponibilidade de bibliotecas (as que quando possuem código nativo, precisam ser compiladas especificamente para Webassembly), não julgamos que isto seria um problema pois diversas das bibliotecas mais utilizadas (como Scipy e Scikit) já foram portadas e funcionam sem problemas.

5 CASOS DE USO

Neste capítulo apresentaremos dois casos de uso elaborados de forma independente com base em necessidades que pudemos identificar a partir das reuniões com os especialistas da área e que demonstram as capacidades do TPVis. Para os exemplos, analisaremos principalmente os dados produzidos por 7 métodos diferentes: I-TSD, GA-Function, e todas as 5 variantes do FAST black box (all, sqrt, log, one, e pw).

5.1 AVALIANDO TESTES HISTORICAMENTE PROBLEMÁTICOS

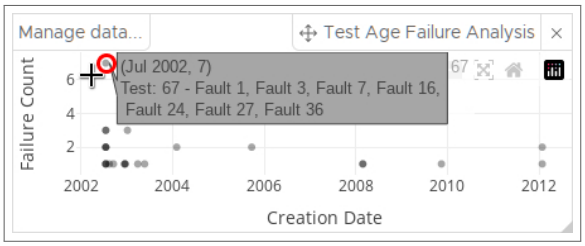
Apache commons-lang é uma biblioteca Java amplamente adotada devido à sua variedade de funções utilitárias, particularmente para tarefas como manipulação de strings e números. A biblioteca foi desenvolvida para resolver algumas deficiências da API Java, que não fornecia utilitários adequados para interagir com suas classes na visão de muitos desenvolvedores na época. A biblioteca é mantida até a escrita deste trabalho e segue com sua relevância na indústria.

Encontrar testes que historicamente falham com frequência pode ajudar a entender que tipos de testes devem ser priorizados e também que partes do sistema testado costumam apresentar mais falhas. Com base nisso, podemos iniciar a nossa análise utilizando a ferramenta de análise de idade e falhas de testes. Na Figura 20 (a), temos a ferramenta em questão com todas as falhas do Apache Commons Lang carregadas. Nela, é possível observar que o teste com ID 67 falhou 7 vezes em um grande intervalo de versões (1 até 36) e tem cerca de uma década de existência. Além de claro, ter falhado o dobro de vezes do que outros testes usuais. Se um teste falha com esta frequência significativa, executá-lo primeiro ou encontrar um método que o priorize pode ser essencial.

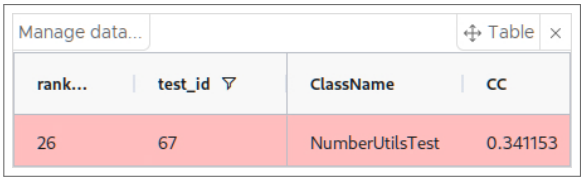
Dando continuidade com o nosso processo investigativo com a utilização do recurso de filtragem da ferramenta de tabela (Figura 20 (b)), podemos identificar que o teste problemático é o NumberUtilsTest. Que por sua vez, valida a classe NumberUtils, componente essencial da biblioteca que contém vários métodos utilitários para trabalhar com classes relacionadas a números na API Java.

Podemos investigar melhor o desempenho dos métodos de priorização nas falhas em que o teste em questão falhou adicionando-as à ferramenta de matriz de evolução de métricas (Figura

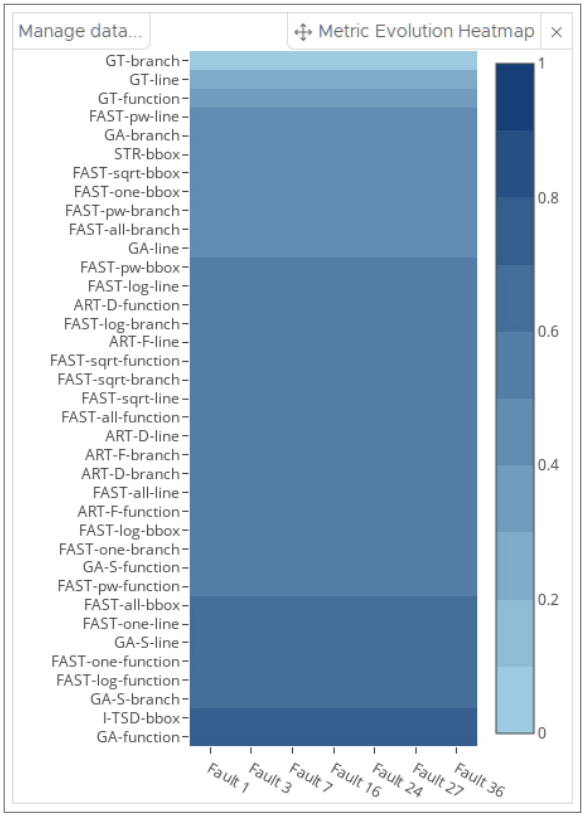
Figura 20 – Identificando um teste recorrente falho no projeto Apache commons-lang e métodos de TCP adequados para priorizá-lo.



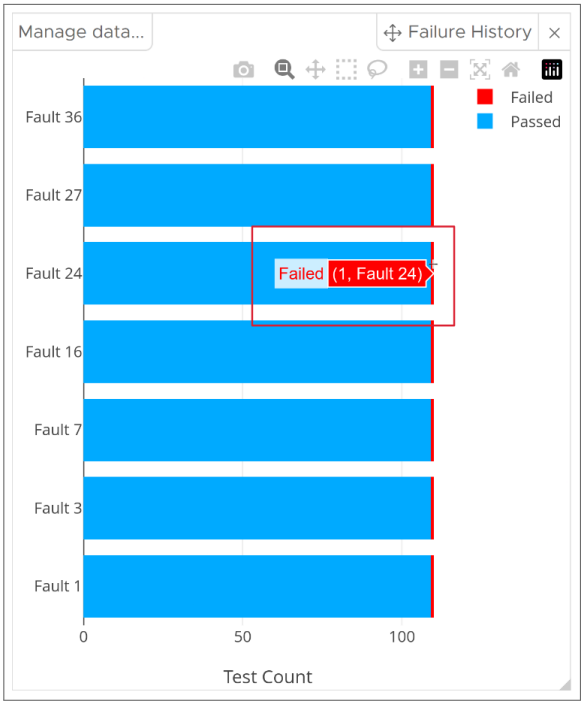
(a) Análise de falhas por idade dos testes indicando um único teste falhando em 7 versões defeituosas diferentes.



(b) Ferramenta de tabela usada para identificar o teste problemático.



(c) Matriz de evolução de métricas comparando como os métodos do conjunto de dados se comportam nas versões afetadas.



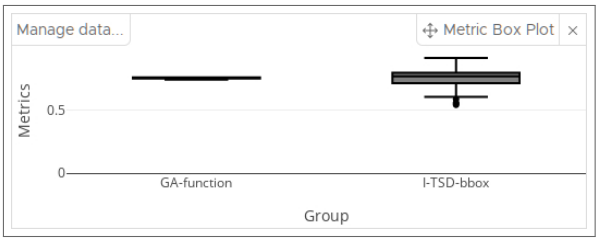
(d) Ferramenta de histórico de falhas revela que o testas 67 é o único teste que falha em todas as versões em que o próprio falhou.

Fonte: O autor (2024)

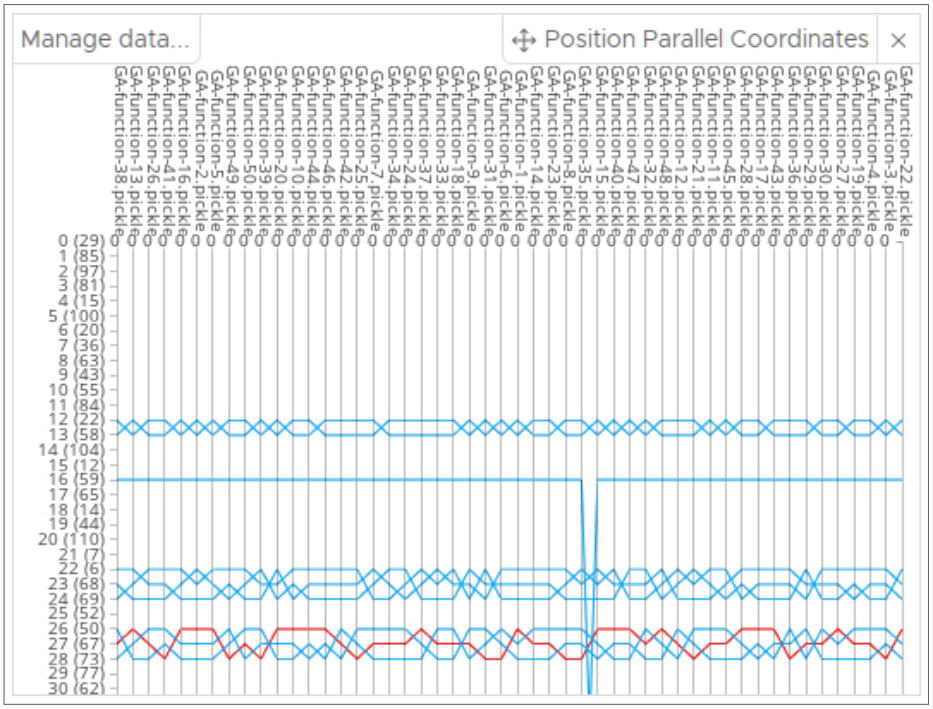
20 (c)), onde observamos que todos os métodos têm a mesma média de APFD para essas versões. Isso pode ser esperado porque, conforme confirmado pela ferramenta de histórico de falhas (Figura 20 (d)), todas compartilham o mesmo único teste falho.

Ainda observando a ferramenta de matriz de evolução de métricas, podemos verificar que ambos os métodos I-TSD-bbox e GA-function têm desempenho semelhante, com valores de APFD de 0.746 e 0.751, respectivamente. No entanto, utilizando o box plot de métrica (Figura 21 (a)), podemos notar que, embora o método I-TSD-bbox possa obter melhores resultados do

Figura 21 – Selecionando entre os métodos GA-Function e I-TSD para priorizar um teste recorrente falho no Apache Commons Lang.



(a) Boxplot de métricas com todas as priorizações para os métodos analisados.



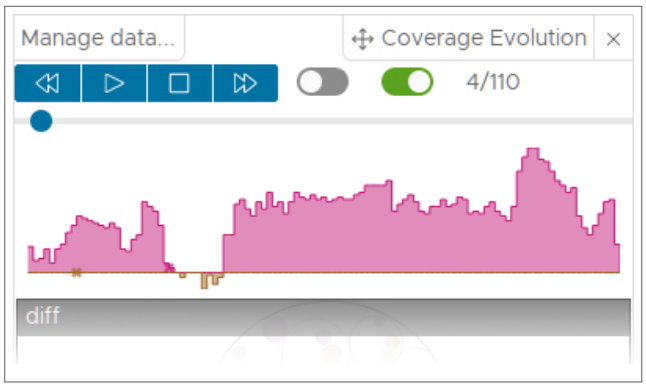
(b) Coordenadas paralelas de posição para todas as priorizações produzidas pelo GA-Function, mostrando que o teste falho varia no máximo 3 posições entre cada execução.

Fonte: O autor (2024)

que o GA-Function em alguns casos, ele também apresenta uma maior variância. Isso significa que o teste falho teve muitas posições diferentes nas 50 iterações produzidas pelo método I-TSD-bbox. Isto também pode ser validado a partir do uso da ferramenta de coordenadas paralelas de posição de testes (Figura 21 (b)), onde vemos, que por exemplo, no método GA-Function o teste 67 varia entre apenas 3 posições (26-28) em 50 iterações.

Outra vantagem do algoritmo GA é que, devido à sua natureza, ele alcança a cobertura de software mais rapidamente do que o I-TSD. Isso pode ser validado usando o gráfico de linha na ferramenta de evolução de cobertura (Figura 22). Finalmente, considerando o maior custo de execução do algoritmo I-TSD e todos os pontos discutidos, a escolha pelo GA-Function se

Figura 22 – Uma maneira rápida de identificar qual priorização cobre um software mais rápido é utilizando o gráfico de linha disponível na ferramenta de evolução de cobertura. Neste caso, GA (representado em rosa) supera o I-TSD, mantendo uma cobertura maior ao longo de toda a priorização.

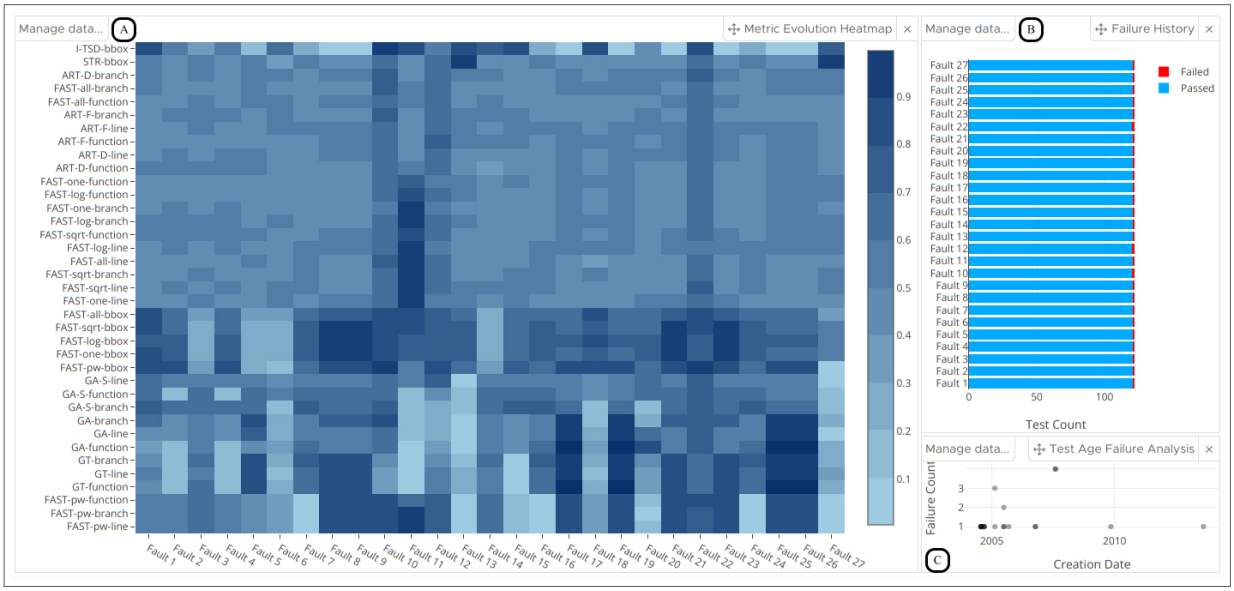


Fonte: O autor (2024)

torna evidente.

5.2 ESCOLHENDO O MÉTODO MAIS ADEQUADO PARA O PROJETO JODA TIME

Figura 23 – Predefinição de Análise Histórica com o projeto Joda-Time carregado. (A) Matriz de evolução de métricas com todas as falhas do projeto carregadas; (B) Ferramenta de histórico de falhas, onde, ao passar o mouse, é possível identificar que as falhas 22, 12 e 10 tiveram dois testes falhos, enquanto o resto teve 1; (C) Ferramenta de análise de falhas por idade dos testes.



Fonte: O autor (2024)

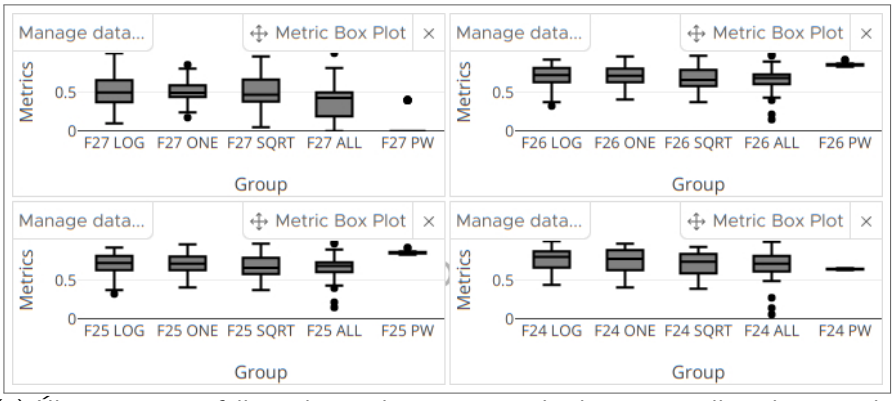
O Joda-Time é uma biblioteca Java que facilita a manipulação de datas e horários, proporcionando uma experiência mais intuitiva do que as classes de data do Java disponíveis na sua época. Apesar de ser considerada obsoleta após o advento da API ‘java.time‘ (JSR-310), a biblioteca ainda é utilizada em muitos sistemas legados e continua sendo relevante.

Podemos iniciar nossa análise com a predefinição de visualização histórica com todas as falhas do projeto em questão carregadas por meio da funcionalidade "Drop Globally", já previamente descrita (Figure 23). Utilizando a ferramenta de histórico de falhas, podemos ter uma visão geral do tamanho do conjunto de testes e com que frequência ocorrem falhas nos testes, tornando-se assim, um bom ponto de partida para análise. Neste caso, o projeto geralmente tem no máximo duas falhas, implicando que os métodos têm menos chance de priorizar um teste que revela uma versão com falha.

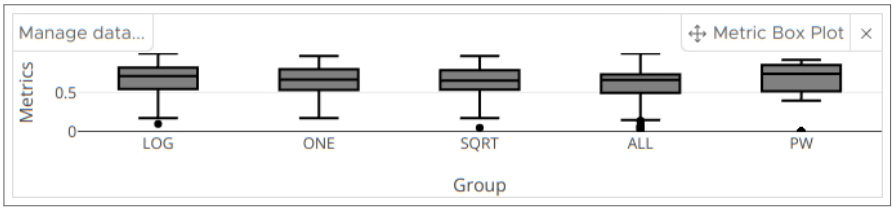
Em seguida, podemos fazer uma análise da matriz de evolução de métrica (Figura 23 (a)) onde podemos identificar que algumas falhas têm métricas APFD ligeiramente mais altas que outras na maioria dos métodos. Isto pode ser entendido analisando a ferramenta de histórico de falhas, onde podemos notar que estas são justamente as que têm mais testes falhos (10, 12 e 22, conforme visto na Figura 23 (b)). Além disso, uma inspeção mais detalhada indica

que os métodos FAST BBOX superaram os outros em grande parte das versões.

Figura 24 – Analisando métricas dos métodos de TCP com melhor desempenho para o Joda-Time.



(a) Últimas quatro falhas observadas com os métodos com melhor desempenho para o projeto Joda-Time.



(b) Métrica APFD agregada para os métodos com melhor desempenho para o projeto Joda-Time.

Fonte: O autor (2024)

Para decidir entre esses métodos que performam melhor, podemos em seguida fazer o uso da ferramenta Metric Box Plot, para que assim, possamos visualizar como esses métodos performam em suas iterações (Figura 24), uma vez que são não determinísticos, como discutimos anteriormente. Avaliando graficamente as métricas dos métodos nas últimas quatro versões (F27, F26, F25, F24), podemos observar que o FAST-PW tem a menor variação de métrica e menos *outliers*. No entanto, também vemos um desempenho consistentemente insatisfatório na falha 27 por parte dos métodos em questão. Além disso, ao *plotar* esses métodos nessas versões finais juntas (resultando em caixas com 200 priorizações e não mais 50), observamos que, de qualquer forma, o FAST-PW-BBOX produz os melhores resultados com menor variação, tornando-se assim, uma boa escolha de método TCP para o projeto.

6 DISCUSSÃO

Este capítulo discutirá as limitações na ferramenta TPVis, bem como alguns detalhes sobre o próprio processo de desenvolvimento.

6.1 PROCESSO DE INTERAÇÃO COM OS ESPECIALISTAS DO DOMÍNIO

O desenvolvimento do TPVis partiu da necessidade de uma empresa da indústria em buscar formas de resolver o problema das grandes suítes de teste. Com o apoio do especialista em testes responsável por disponibilizar esta oportunidade, conseguimos ter uma imersão no fluxo de trabalho da empresa em questão, e assim, entender as implicações práticas do problema num ambiente de produção. A partir disso, de forma iterativa, elicitamos os requisitos funcionais para a construção do TPVis.

Além disso, durante o processo de desenvolvimento do TPVis, foram realizadas várias reuniões com especialistas em testes de software conforme demandas e questionamentos foram surgindo. Durante a fase de construção, um dos especialistas levantou preocupações sobre a usabilidade central do sistema. Especificamente, ele questionou como o software poderia se tornar mais intuitivo e acessível aos usuários. Para resolver isso, criamos as predefinições de visualização. Outra preocupação levantada foi sobre a capacidade de comparar diferentes algoritmos facilmente, sem a necessidade de arrastar e soltar uma única priorização de cada um. Para resolver isso, implementamos o comportamento de arrastar e soltar "melhor priorização para o método", configurável na árvore de priorizações. Desta maneira, selecionar vários métodos trará a melhor priorização de cada.

Dois dos especialistas identificaram a falta de ferramentas de visualização histórica no sistema. Como resultado, implementamos as ferramentas de evolução de métricas, idade dos testes e histórico de falhas para abordar essas preocupações. Este segundo especialista, que era da área de TCP, também forneceu feedback sobre a apresentação do espaço de trabalho na árvore de priorização e como não era trivial saber o que cada nó representava (requerendo uma sobreposição do mouse). Resolvemos esse feedback adicionando um atributo "*markAs*" aos nós para permitir uma identificação mais fácil de diferentes tipos de nós. Isso permite, por exemplo, que uma versão com falhas seja exibida com um ícone de bug em vez de um ícone de pasta genérico. Este especialista também sugeriu a criação de uma ferramenta para avaliar

como a priorização cobre diferentes partes do software. Para resolver isso, a ferramenta de evolução de cobertura foi implementada.

6.2 PERFORMANCE E ESCALABILIDADE COMPUTACIONAL

O TPVis foi testado com um conjunto de dados grande, consistindo em 5 projetos, 198 versões com falhas, 37 métodos de priorização e 366.300 priorizações individuais. O maior conjunto de testes em nosso conjunto de dados é do projeto "math", composto por 384 testes (em comparação, Luo, Moran e Poshyvanyk (2016) utilizaram conjuntos de até 122 testes em sua avaliação empírica). Graças à abordagem de carregamento preguiçoso adotada, não houveram problemas em relação à quantidade de priorizações ou à quantidade de métodos TCP no espaço de trabalho. No entanto, ao carregar muitas priorizações em uma ferramenta, pode demorar um pouco para que elas sejam buscadas do *backend* Quarkus (poderia ser melhorado paralelizando as solicitações). Em nossa implementação atual, leva cerca de 2 segundos para carregar 100 priorizações.

Algumas ferramentas, como a evolução da cobertura, realizam uma quantidade considerável de processamento durante o uso, e outras, como as coordenadas paralelas de posição, exigem um esforço significativo para renderizar inicialmente. Web workers poderiam ser usados para descarregar o processamento do núcleo principal. A ferramenta de TSNE de curvas de teste tem seu processamento escrito em Python (executado com Pyodide) e roda em um web worker. Isso valida que a abordagem de web worker resolveria esse problema, visto que a aplicação é responsiva mesmo ao computar centenas de priorizações nesta ferramenta.

Algoritmos como o TSNE e a ordenação ótima de folhas podem exigir esforços computacionais elevados para grandes conjuntos de dados. Por exemplo, em nossos experimentos, observamos um tempo de cerca de 3 segundos para processar 100 priorizações com TSNE. Embora isso tenha sido considerado aceitável por alguns dos nossos colaboradores de domínio, é reconhecido que tais tempos de processamento podem prejudicar o processo de análise de dados do usuário (LIU; HEER, 2014). Deixamos como trabalho futuro a implementação de otimizações de performance, assim como, uma solução robusta para avaliar o TPVis, que trará performance e escalabilidade como um ponto chave a ser analisado.

6.3 LIMITAÇÕES ANALÍTICAS E DE USABILIDADE

Certas ferramentas do TPVis impõem limitações ao número de falhas ou priorizações que podem ser carregadas. No entanto, estamos seguros de que nossa solução atual é capaz de lidar com a maioria das situações do mundo real. Enquanto algumas ferramentas podem não se beneficiar da entrada de mais priorizações, outras poderiam. Por exemplo, seria útil poder monitorar a evolução da cobertura para múltiplas priorizações dentro de uma única ferramenta em vez de apenas duas. Embora seja possível utilizar várias instâncias da ferramenta simultaneamente, ter essa função em uma única instância seria mais amigável ao usuário.

Apesar de predefinições de visualização estarem disponíveis e cobrirem grande parte dos casos de uso possíveis, usuários poderiam beneficiar-se de uma gestão mais avançada do espaço de trabalho. Acreditamos que funcionalidades como abas e salvar predefinições customizadas poderiam ser de grande utilidade. Além disso, pequenas mudanças como a adição da possibilidade de ordenar a árvore por diferentes chaves ou marcar nós como mais relevantes também poderiam trazer uma melhor experiência.

No TPVis, cores são utilizadas para representar os diferentes métodos de TCP e priorizações. Embora isso limite o número de métodos que podem ser carregados efetivamente em nosso sistema ao mesmo tempo, raramente é o caso de que todos precisem ser considerados ao mesmo tempo. Além disso, o usuário está sujeito ao problema de *overplotting*, pois a criação das análises está por conta dele. Ambas estas limitações poderiam ser amenizadas permitindo que os diferentes métodos sejam agrupados hierarquicamente em classes e, seguindo o mantra do InfoVis (SHNEIDERMAN, 2003), possibilitando que o usuário inspecione "resumos gerais", mas também, "detalhes sob demanda".

6.4 NECESSIDADE DE UM *SCRIPT* DE PRÉ-PROCESSAMENTO

Embora seja uma tarefa simples, implementar o TPVis em um projeto de software exige um script de pré-processamento. Apesar de isso poder ser considerado um aspecto positivo, pois faz com que TPVis possa ser integrado em projetos em qualquer pilha tecnológica (até mesmo em projetos não relacionados a software e com casos de teste manuais), alguns usuários podem vir a enxergar isto como uma limitação.

Enxergamos que o processo de criação de espaços de trabalho poderia ser facilitado de duas maneiras. A primeira, seria com o desenvolvimento e implementação de bibliotecas para este

propósito. Tais bibliotecas poderiam ser implementadas para as linguagens mais comuns (Ex.: Javascript ou Python) e poderia conter representações dos tipos de nó do TPVis, como classes para pastas e priorizações. A API poderia conter também métodos utilitários para encapsular e abstrair do usuário o trabalho direto com ferramentas de teste ou versionamento de código (Ex.: Método para obter data de criação de uma classe de teste do sistema ou obter classes testadas por um determinado teste).

Outra abordagem seria implementar um mecanismo padronizado para fazer o pré processamento. Este mecanismo teria de dar suporte a grande parte dos executores automatizados de teste (Ex.: JUnit e Karma). Tal objetivo seria definitivamente bem mais ambicioso e deixaria o processo mais amigável ao usuário final, podendo ser implementado, como por exemplo, por meio de um *Maven Plugin*. Todavia, caso o projeto utilize uma tecnologia não suportada pelo TPVis, o usuário precisaria reverter a utilizar scripts de pré-processamento, portanto, mesmo que esta solução seja implementada futuramente, deve ser optativa a fim de manter a natureza flexível da ferramenta.

6.5 INSPIRAÇÃO EM SISTEMAS PREEXISTENTES

Como mencionado anteriormente, não encontramos outra ferramenta especificamente projetada para visualizar a priorização de casos de teste. Considerando isso, nos inspiramos em outras ferramentas de desenvolvimento para nos auxiliar a preencher a lacuna entre o mundo da indústria e acadêmico, afim de tornar as soluções para problemas de otimização de suítes de testes mais acessíveis. Um exemplo de sistema relacionado é o SonarQube (SONARSOURCE, 2024), uma ferramenta de análise estática que auxilia os desenvolvedores a prevenir bugs, vulnerabilidades de segurança e práticas prejudiciais.

Semelhante ao TPVis, o SonarQube foi projetado para ser integrado a um pipeline de CI/CD e utiliza um software chamado "Sonar Scanner" para coletar todos os metadados do código-fonte e escanear em busca de maus cheiros de código, cobertura de código e outros indicadores de qualidade do código-fonte. O Sonar Scanner é compatível com várias ferramentas de construção (Ex.: Gradle e Maven) e suporta mais de 30 linguagens de programação. Além disso, está disponível como uma ferramenta de interface de linha de comando para casos em que não há *plugin* de ferramenta de construção disponível para a plataforma de desenvolvimento que está sendo analisada.

O Sonar Scanner é o componente SonarQube executado durante o processo de CI. Fun-

cionalmente, ele é semelhante ao script de pré-processamento do TPVis, pois coleta todos os dados necessários em diferentes ambientes para construir um conjunto de dados padronizado. Além disso, assim como o TPVis, o SonarQube é executado em um ambiente web e é responsável por apresentar visualmente os resultados gerados pelo Sonar Scanner. Ele também permite que o usuário final navegue pelo código-fonte.

Acreditamos que, assim como o SonarQube tornou as análises de código estático mais acessíveis, o TPVis pode ajudar mais equipes de software a otimizar suas execuções de suíte de testes.

7 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho, apresentamos o TPVis, um sistema de análise visual de priorização de testes com ênfase na seleção e comparação de métodos de priorização e exploração de suítes de testes composto por diversas ferramentas analíticas.

Durante o desenvolvimento do TPVis, encontramos diversos desafios técnicos a serem contornados. Para possibilitar a utilização do TPVis na indústria, arquitetamos uma estrutura de dados genérica o bastante para se enquadrar em qualquer projeto de software e aqui, descrevemos a mesma em detalhes. A fim de lidar com a grande quantidade de dados que podem vir a ser produzidos em grandes projetos de software, adotamos uma estratégia de carregamento preguiçoso e mecanismos assíncronos para viabilizar a utilização da ferramenta em qualquer contexto.

Quando tratamos da interface gráfica, conseguimos chegar ao que acreditamos ser um bom balanço entre flexibilidade e experiência do usuário. Trazendo um mecanismo para criação de *dashboards* que podem ser construídos da maneira que o usuário julgar mais apropriada, tendo em vista a análise a ser realizada.

Nosso trabalho explora as características únicas de cada ferramenta e demonstra dois casos de uso que mostram como elas podem fornecer inspeções valiosas de forma intuitiva. Além disso, demonstramos como o TPVis pode ser integrado em cenários do mundo real e apresentamos os resultados de nossa análise usando métodos de priorização e dados de teste reais. Em conclusão, o TPVis oferece uma solução flexível e abrangente para avaliar métodos de priorização, bem como analisar e inspecionar casos de teste em diferentes ambientes de software.

Durante a pesquisa e o desenvolvimento do TPVis, houve uma colaboração com especialistas do domínio de testes de software. Estes especialistas nos auxiliaram a entender as reais necessidades relacionadas ao problema de priorização de teste e nos guiaram a trazer a usabilidade ao estado que encontramos atualmente.

Pretendemos melhorar o TPVis adicionando mais funcionalidades no futuro. Tais funcionalidades podem ser distribuídas em três categorias principais. São elas: (**I**) Melhorias de usabilidade; (**II**) Melhorias analíticas e (**III**) Melhorias de escalabilidade.

Quanto a melhorias de usabilidade, pretendemos disponibilizar uma maneira dos usuários se autenticarem na ferramenta, visto que como a ferramenta será executada num servidor

web, deve existir algum tipo de controle de acesso. Além disso, almejamos implementar algumas melhorias já previamente discutidas, como suporte a abas e a criação de predefinições de visualização customizadas. Visto que os usuários também são responsáveis por desenvolver um script de pré-processamento, gostaríamos também de implementar uma API para o desenvolvimento ainda mais simplificado do mesmo. Também temos interesse em remediar este problema desenvolvendo um *Maven Plugin*, como discutido anteriormente, que poderia substituir o script de pré-processamento inteiro, facilitando a integração do TPVis em projetos Java.

Quando tratamos das melhorias analíticas, podemos abordar principalmente as limitações de entrada de dados impostas por algumas das ferramentas e incluir diferentes técnicas para agregar um maior número de falhas e priorizações carregadas. Uma possível abordagem seria utilizar a versão do software da priorização carregada, e não depender especificamente do usuário para a seleção de versões em algumas ferramentas. Além disso, gostaríamos de avaliar como explorar mais a temática da evolução da cobertura de software durante a execução de uma suíte de testes priorizadas, considerando que este é um dos principais pilares de alguns métodos de priorização, como vimos no trabalho.

Com relação a escalabilidade, apesar do TPVis ter se comportado bem com uma quantidade de dados significativa, podemos ver que ainda há espaço para melhorias. Podemos melhorar o tempo de resposta para obtenção de dados do *backend* criando mecanismos de armazenamento em memória, evitando carregamentos diretamente do disco. Outra possível abordagem seria a criação de *web workers* para ferramentas mais computacionalmente demandantes.

Também planejamos como trabalho futuro um estudo formal de usuários em um ambiente de produção em colaboração com uma equipe de testes de software. Neste estudo, queremos avaliar a usabilidade do TPVis por meio de uma adaptação do *System Usability Scale* (SUS) (BROOKE, 1996), sua performance, sua curva aprendizado e também o impacto a longo prazo do uso do nosso sistema em um ambiente de produção.

Por fim, TPVis foi disponibilizado como um projeto de código aberto e está disponível em um repositório no Github[1]. O projeto é licenciado com a *GNU General Public License*, desta forma, não só é possível, como encorajamos a contribuição da comunidade no projeto, criando novas funcionalidades e ferramentas analíticas.

[1] <https://github.com/vixe-cin-ufpe/TPVis>

REFERÊNCIAS

AGGRID. *AgGrid*. 2024. Disponível em: <https://ag-grid.com/angular-data-grid>.

ARAFEEN, M. J.; DO, H. Test case prioritization using requirements-based clustering. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. [s.n.], 2013. p. 312–321. Disponível em: <https://doi.org/10.1109/ICST.2013.12>.

BAJAJ, A.; SANGWAN, O. P. A systematic literature review of test case prioritization using genetic algorithms. *IEEE Access*, v. 7, p. 126355–126375, 2019. Disponível em: <https://doi.org/10.1109/ACCESS.2019.2938260>.

BAR-JOSEPH, Z.; GIFFORD, D. K.; JAAKKOLA, T. S. Fast optimal leaf ordering for hierarchical clustering . *Bioinformatics*, v. 17, n. suppl_1, p. S22–S29, 06 2001. ISSN 1367-4803. Disponível em: <https://academic.oup.com/bioinformatics/article/17/suppl_1/S22/261423?login=false>.

BOURQUE, P.; FAIRLEY, R. E.; SOCIETY, I. C. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. ed. Washington, DC, USA: IEEE Computer Society Press, 2014. ISBN 0769551661.

BRANDT, C.; ZAIDMAN, A. How does this new developer test fit in? a visualization to understand amplified test cases. In: *2022 Working Conference on Software Visualization (VISSOFT)*. [s.n.], 2022. p. 17–28. Disponível em: <https://doi.org/10.1109/VISSOFT55257.2022.00011>.

BROOKE, J. *"SUS-A quick and dirty usability scale."Usability evaluation in industry*. CRC Press, 1996. ISBN: 9780748404605. Disponível em: <https://www.crcpress.com/product/isbn/9780748404605>.

BUSJAEGER, B.; XIE, T. Learning for test prioritization: an industrial case study. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 975–980. ISBN 9781450342186. Disponível em: <https://doi.org/10.1145/2950290.2983954>.

CASERTA, P.; ZENDRA, O. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, v. 17, n. 7, p. 913–933, July 2011. ISSN 1941-0506. Disponível em: <https://doi.org/10.1109/TVCG.2010.110>.

CORNELISSEN, B.; DEURSEN, A. van; MOONEN, L.; ZAIDMAN, A. Visualizing testsuites to aid in software understanding. In: *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. [s.n.], 2007. p. 213–222. Disponível em: <https://ieeexplore.ieee.org/document/4145039>.

DO, H.; ROTHERMEL, G. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2006. (SIGSOFT '06/FSE-14), p. 141–151. ISBN 1595934685. Disponível em: <https://doi.org/10.1145/1181775.1181793>.

DO, H.; ROTHERMEL, G. Using sensitivity analysis to create simplified economic models for regression testing. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2008. (ISSTA '08), p. 51–62. ISBN 9781605580500. Disponível em: <https://doi.org/10.1145/1390630.1390639>.

DREEF, K.; PALEPU, V. K.; JONES, J. A. Exploring granular test coverage and its evolution with matrix visualizations. *Information and Software Technology*, v. 155, p. 107085, 2023. ISSN 0950-5849. Disponível em: <https://doi.org/10.1016/j.infsof.2022.107085>.

DUCKSBOARD. *Gridster.js*. 2024. Disponível em: <https://dsmorse.github.io/gridster.js>.

DUGGAL, G.; SURI, B. Understanding regression testing techniques. In: *Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology*. [S.l.: s.n.], 2008. v. 1, p. 8.

ELBAUM, S.; MALISHEVSKY, A.; ROTHERMEL, G. Incorporating varying test costs and fault severities into test case prioritization. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. [s.n.], 2001. p. 329–338. Disponível em: <https://ieeexplore.ieee.org/document/919106>.

ELBAUM, S.; ROTHERMEL, G.; KANDURI, S.; MALISHEVSKY, A. G. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, v. 12, n. 3, p. 185–210, Sep 2004. ISSN 1573-1367. Disponível em: <https://doi.org/10.1023/B:SQJO.0000034708.84524.22>.

ELBAUM, S.; ROTHERMEL, G.; PENIX, J. Techniques for improving regression testing in continuous integration development environments. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 235–245. ISBN 9781450330565. Disponível em: <https://doi.org/10.1145/2635868.2635910>.

EPITROPAKIS, M.; YOO, S.; HARMAN, M.; BURKE, E. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In: . [s.n.], 2015. p. 234–245. Disponível em: <http://dx.doi.org/10.1145/2771783.2771788>.

ERIKSSON, N.; ÖRNEHOLM, M. *Understanding the role of visual analytics for software testing*. 2021. Disponível em: <https://www.diva-portal.org/smash/get/diva2:1575024/FULLTEXT02>.

FELDT, R.; POULDING, S.; CLARK, D.; YOO, S. Test set diameter: Quantifying the diversity of sets of test cases. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [s.n.], 2016. p. 223–233. Disponível em: <http://dx.doi.org/10.1109/ICST.2016.33>.

GOOGLE. *Angular Framework*. 2024. Disponível em: <https://angular.io/>.

GRATZL, S.; LEX, A.; GEHLENBORG, N.; PFISTER, H.; STREIT, M. Lineup: Visual analysis of multi-attribute rankings. *IEEE Transactions on Visualization and Computer Graphics*, v. 19, n. 12, p. 2277–2286, 2013. Disponível em: <https://doi.org/10.1109/TVCG.2013.173>.

HAMMAD, M.; OTOOM, A. F.; HAMMAD, M.; AL-JAWABREH, N.; SEINI, R. A. Multiview visualization of software testing results. *International Journal of Computing and Digital Systems*, v. 9, n. 1, 2020. Disponível em: <http://dx.doi.org/10.12785/ijcds/090105>.

HAO, D.; ZHANG, L.; ZANG, L.; WANG, Y.; WU, X.; XIE, T. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering*, v. 42, n. 5, p. 490–505, 2016. Disponível em: <https://doi.org/10.1109/TSE.2015.2496939>.

HARROLD, M. J. Reduce, reuse, recycle, recover: Techniques for improved regression testing. In: *2009 IEEE International Conference on Software Maintenance*. [s.n.], 2009. p. 5–5. Disponível em: <https://doi.org/10.1109/ICSM.2009.5306347>.

HETTIARACHCHI, C.; DO, H.; CHOI, B. Risk-based test case prioritization using a fuzzy expert system. *Information and Software Technology*, v. 69, p. 1–15, 2016. ISSN 0950-5849. Disponível em: <https://doi.org/10.1016/j.infsof.2015.08.008>.

JAMIL, M. A.; ARIF, M.; ABUBAKAR, N. S. A.; AHMAD, A. Software testing techniques: A literature review. In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. [s.n.], 2016. p. 177–182. Disponível em: <https://doi.org/10.1109/ICT4M.2016.045>.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2002. (ICSE '02), p. 467–477. ISBN 158113472X. Disponível em: <https://doi.org/10.1145/581339.581397>.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2014. (ISSTA 2014), p. 437–440. ISBN 9781450326452. Disponível em: <https://doi.org/10.1145/2610384.2628055>.

KAZMI, R.; JAWAWI, D. N. A.; MOHAMAD, R.; GHANI, I. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 50, n. 2, may 2017. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3057269>.

KHAN, S. U. R.; LEE, S. P.; AHMAD, R. W.; AKHUNZADA, A.; CHANG, V. A survey on test suite reduction frameworks and tools. *International Journal of Information Management*, v. 36, n. 6, Part A, p. 963–975, 2016. ISSN 0268-4012. Disponível em: <https://doi.org/10.1016/j.ijinfomgt.2016.05.025>.

KHATIBSYARBINI, M.; ISA, M. A.; JAWAWI, D. N.; TUMENG, R. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, v. 93, p. 74–93, 2018. ISSN 0950-5849. Disponível em: <https://doi.org/10.1016/j.infsof.2017.08.014>.

KIM, J.-M.; PORTER, A. A history-based test prioritization technique for regression testing in resource constrained environments. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2002. (ICSE '02), p. 119–129. ISBN 158113472X. Disponível em: <https://doi.org/10.1145/581339.581357>.

LEDRU, Y.; PETRENKO, A.; BORODAY, S.; MANDRAN, N. Prioritizing test cases with string distances. *Automated Software Engineering*, Springer, v. 19, p. 65–95, 2012. Disponível em: <https://doi.org/10.1007/s10515-011-0093-0>.

LI, Z.; HARMAN, M.; HIERONS, R. M. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, v. 33, n. 4, p. 225–237, 2007. Disponível em: <https://doi.org/10.1109/TSE.2007.38>.

LIU, Z.; HEER, J. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, IEEE, v. 20, n. 12, p. 2122–2131, 2014.

LOU, Y.; CHEN, J.; ZHANG, L.; HAO, D. Chapter one - a survey on regression test-case prioritization. In: MEMON, A. M. (Ed.). Elsevier, 2019, (Advances in Computers, v. 113). p. 1–46. Disponível em: <https://doi.org/10.1016/bs.adcom.2018.10.001>.

LUO, Q.; MORAN, K.; POSHYVANYK, D. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 559–570. ISBN 9781450342186. Disponível em: <https://doi.org/10.1145/2950290.2950344>.

LV, J.; YIN, B.; CAI, K.-Y. On the gain of measuring test case prioritization. In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. [s.n.], 2013. p. 627–632. Disponível em: <https://doi.org/10.1109/COMPSAC.2013.101>.

MAATEN, L. van der; HINTON, G. Visualizing data using t-sne. *Journal of Machine Learning Research*, v. 9, n. 86, p. 2579–2605, 2008. Disponível em: <http://jmlr.org/papers/v9/vandermaaten08a.html>.

MAHDIEH, M.; MIRIAN-HOSSEINABADI, S.-H.; ETEMADI, K.; NOSRATI, A.; JALALI, S. Incorporating fault-proneness estimations into coverage-based test case prioritization methods. *Information and Software Technology*, v. 121, p. 106269, 2020. ISSN 0950-5849. Disponível em: <https://doi.org/10.1016/j.infsof.2020.106269>.

MIRANDA, B.; CRUCIANI, E.; VERDECCHIA, R.; BERTOLINO, A. Fast approaches to scalable similarity-based test case prioritization. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [s.n.], 2018. p. 222–232. Disponível em: <https://doi.org/10.1145/3180155.3180210>.

MORITZ, D.; FISHER, D. Visualizing a million time series with the density line chart. *arXiv preprint arXiv:1808.06019*, 2018. Disponível em: <https://doi.org/10.48550/arXiv.1808.06019>.

MOZILLA. *Pyodide*. 2024. Disponível em: <https://pyodide.org>.

MUDY, K.; MANSSON, A. An empirical study on the trade-off of diversity types and measures on test visualisation. 2021. Disponível em: <https://odr.chalmers.se/items/865beda4-b9ee-4820-95c8-2a23e907ebd0>.

MUKHERJEE, R.; PATNAIK, K. S. A survey on different approaches for software test case prioritization. *Journal of King Saud University - Computer and Information Sciences*, v. 33, n. 9, p. 1041–1054, 2021. ISSN 1319-1578. Disponível em: <https://doi.org/10.1016/j.jksuci.2018.09.005>.

MYLAVARAPU, P.; YALCIN, A.; GREGG, X.; ELMQVIST, N. Ranked-list visualization: A graphical perception study. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing

Machinery, 2019. (CHI '19), p. 1–12. ISBN 9781450359702. Disponível em: <https://doi.org/10.1145/3290605.3300422>.

OBSERVABLE. *D3*. 2024. Disponível em: <https://d3js.org>.

OMRI, S.; SINZ, C. Learning to rank for test case prioritization. In: *Proceedings of the 15th Workshop on Search-Based Software Testing*. New York, NY, USA: Association for Computing Machinery, 2023. (SBST '22), p. 16–24. ISBN 9781450393188. Disponível em: <https://doi.org/10.1145/3526072.3527525>.

PAN, J. Software testing. *Dependable Embedded Systems*, Citeseer, v. 5, n. 2006, p. 1, 1999.

PLOTLY. *Plotly JS*. 2024. Disponível em: <https://plotly.com/javascript>.

QU, B.; NIE, C.; XU, B.; ZHANG, X. Test case prioritization for black box testing. In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. [s.n.], 2007. v. 1, p. 465–474. Disponível em: <https://doi.org/10.1109/COMPSAC.2007.209>.

QU, X.; COHEN, M.; WOOLF, K. Combinatorial interaction regression testing: A study of test case generation and prioritization. In: . [s.n.], 2007. p. 255 – 264. ISBN 978-1-4244-1256-3. Disponível em: <https://doi.org/10.1109/ICSM.2007.4362638>.

QUARKUS. *Quarkus Framework*. 2024. Disponível em: <https://quarkus.io>.

ROTHERMEL, G.; ELBAUM, S. Putting your best tests forward. *IEEE Software*, v. 20, n. 5, p. 74–77, 2003. Disponível em: <https://doi.org/10.1109/MS.2003.1231157>.

ROTHERMEL, G.; HARROLD, M. J.; RONNE, J. von; HONG, C. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, v. 12, n. 4, p. 219–249, 2002. Disponível em: <https://doi.org/10.1002/stvr.256>.

ROTHERMEL, G.; UNTCH, R.; CHU, C.; HARROLD, M. Test case prioritization: an empirical study. In: *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. [S.l.: s.n.], 1999. p. 179–188.

ROTHERMEL, G.; UNTCH, R.; CHU, C.; HARROLD, M. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, v. 27, n. 10, p. 929–948, 2001. Disponível em: <https://doi.org/10.1109/32.962562>.

SAPNA, P.; MOHANTY, H. Prioritizing use cases to aid ordering of scenarios. In: *2009 Third UKSim European Symposium on Computer Modeling and Simulation*. [S.l.: s.n.], 2009. p. 136–141.

SHAHIN, M.; BABAR, M. A.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, v. 5, p. 3909–3943, 2017. Disponível em: <https://doi.org/10.1109/ACCESS.2017.2685629>.

SHI, C.; CUI, W.; LIU, S.; XU, P.; CHEN, W.; QU, H. Rankexplorer: Visualization of ranking changes in large time series data. *IEEE Transactions on Visualization and Computer Graphics*, v. 18, n. 12, p. 2669–2678, 2012. Disponível em: <https://doi.org/10.1109/TVCG.2012.253>.

SHNEIDERMAN, B. The eyes have it: A task by data type taxonomy for information visualizations. In: BEDERSON, B. B.; SHNEIDERMAN, B. (Ed.). *The Craft of Information Visualization*. San Francisco: Morgan Kaufmann, 2003, (Interactive Technologies). p. 364–371. ISBN 978-1-55860-915-0. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9781558609150500469>.

SONARSOURCE. *SonarQube*. 2024. Disponível em: <https://www.sonarsource.com/products/sonarqube/>.

SRIKANTH, H.; WILLIAMS, L. On the economics of requirements-based test case prioritization. In: *Proceedings of the Seventh International Workshop on Economics-Driven Software Engineering Research*. New York, NY, USA: Association for Computing Machinery, 2005. (EDSER '05), p. 1–3. ISBN 159593118X. Disponível em: <https://doi.org/10.1145/1083091.1083100>.

STOLBERG, S. Enabling agile testing through continuous integration. In: *2009 Agile Conference*. [s.n.], 2009. p. 369–374. Disponível em: <https://doi.org/10.1109/AGILE.2009.16>.

STRANDBERG, P. E. Software test data visualization with heatmaps–an initial survey. *Report, no. MDH-MRTC318/2017-1-SE*, 2017. Disponível em: <http://mdh.diva-portal.org/smash/record.jsf?pid=diva2%3A1104419&dswid=-2312>.

STRANDBERG, P. E.; AFZAL, W.; SUNDMARK, D. Decision making and visualizations based on test results. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: Association for Computing Machinery, 2018. (ESEM '18). ISBN 9781450358231. Disponível em: <https://doi.org/10.1145/3239235.3268921>.

STRANDBERG, P. E.; AFZAL, W.; SUNDMARK, D. Software test results exploration and visualization with continuous integration and nightly testing. *International Journal on Software Tools for Technology Transfer*, v. 24, n. 2, p. 261–285, Apr 2022. ISSN 1433-2787. Disponível em: <https://doi.org/10.1007/s10009-022-00647-1>.

TONELLA, P.; AVESANI, P.; SUSI, A. Using the case-based ranking methodology for test case prioritization. In: *2006 22nd IEEE International Conference on Software Maintenance*. [s.n.], 2006. p. 123–133. Disponível em: <https://doi.org/10.1109/ICSM.2006.74>.

VIRMANI, M. Understanding devops & bridging the gap from continuous integration to continuous delivery. In: *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*. [s.n.], 2015. p. 78–82. Disponível em: <https://doi.org/10.1109/INTECH.2015.7173368>.

VMWARE. *Clarity Design System*. 2024. Disponível em: <https://clarity.design>.

WAHL, N. J. An overview of regression testing. *SIGSOFT Softw. Eng. Notes*, Association for Computing Machinery, New York, NY, USA, v. 24, n. 1, p. 69–73, jan 1999. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/308769.308790>.

WALCOTT, K. R.; SOFFA, M. L.; KAPFHAMMER, G. M.; ROOS, R. S. Timeaware test suite prioritization. In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2006. (ISSTA '06), p. 1–12. ISBN 1595932631. Disponível em: <https://doi.org/10.1145/1146238.1146240>.

WANG, Z. Improved metrics for non-classic test prioritization problems. In: *International Conference on Software Engineering and Knowledge Engineering*. [s.n.], 2015. Disponível em: <https://doi.org/10.18293/seke2015-230>.

XU, D.; DING, J. Prioritizing state-based aspect tests. In: *2010 Third International Conference on Software Testing, Verification and Validation*. [s.n.], 2010. p. 265–274. Disponível em: <https://doi.org/10.1109/ICST.2010.14>.

YANG, Y.; CHEN, X. Crowdsourced test report prioritization based on text classification. *IEEE Access*, v. 10, p. 92692–92705, 2022. Disponível em: <https://doi.org/10.1109/ACCESS.2021.3128726>.

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., GBR, v. 22, n. 2, p. 67–120, mar 2012. ISSN 0960-0833. Disponível em: <https://doi.org/10.1002/stv.430>.

ZHAI, K.; JIANG, B.; CHAN, W. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *IEEE Transactions on Services Computing*, v. 7, n. 1, p. 54–67, 2014. Disponível em: <https://doi.org/10.1109/TSC.2012.40>.

ZHANG, L.; HAO, D.; ZHANG, L.; ROTHERMEL, G.; MEI, H. Bridging the gap between the total and additional test-case prioritization strategies. In: *2013 35th International Conference on Software Engineering (ICSE)*. [s.n.], 2013. p. 192–201. Disponível em: <https://doi.org/10.1109/ICSE.2013.6606565>.

ZHANG, L.; MARINOV, D.; ZHANG, L.; KHURSHID, S. An empirical study of junit test-suite reduction. In: *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. [s.n.], 2011. p. 170–179. Disponível em: <https://doi.org/10.1109/ISSRE.2011.26>.