Universidade Federal de Pernambuco
Centro de Informática

FAST-GRADLE-PLUGIN: UM PLUGIN PARA PRIORIZAÇÃO DE CASOS DE TESTE UTILIZANDO O FAST

Luiz André de Jesus Silva

Recife
Março, 2024

Universidade Federal de Pernambuco
Centro de Informática

Luiz André de Jesus Silva

FAST-GRADLE-PLUGIN: UM PLUGIN PARA PRIORIZAÇÃO DE CASOS DE TESTE UTILIZANDO O FAST

Trabalho de graduação apresentado para o programa de Bacharelado em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco em cumprimento parcial dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Prof. Dr. Breno Alexandro Ferreira de Miranda*

Recife
Março, 2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Luiz André de Jesus.
FAST-gradle-plugin: Um plugin para priorização de casos de teste utilizando o FAST / Luiz André de Jesus Silva. - Recife, 2024.
33 p. : il., tab.

Orientador(a): Breno Alexandro Ferreira de Miranda
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2024.

1. Testes de Software. 2. Priorização de Casos de Teste. 3. FAST. I. Miranda, Breno Alexandro Ferreira de. (Orientação). II. Título.

000 CDD (22.ed.)

Agradecimentos

Dedico este trabalho à minha esposa, que sempre esteve ao meu lado, me apoiando e incentivando em todos os momentos da minha vida. Agradeço por todo o amor, carinho e compreensão que ela me proporciona, e por ser a minha maior fonte de inspiração e motivação.

Dedico também à minha mãe e ao meu pai, que sempre me apoiaram e incentivaram a buscar os meus sonhos e objetivos. Agradeço por todo o suporte e amor incondicional que recebi ao longo da minha vida.

Agradeço ao meu orientador, que me orientou e guiou durante todo o processo de elaboração deste trabalho. Agradeço por compartilhar seus conhecimentos e experiências, e por me incentivar a buscar sempre o melhor.

Por fim, dedico este trabalho a todos os profissionais da área de software, que trabalham incansavelmente para garantir a qualidade dos sistemas de software e contribuem para o avanço da tecnologia.

Abstract

The software testing process is essential for developing high-quality software. However, testing activities can be quite costly, consuming a significant portion of the entire development cycle. One strategy to optimize the time spent on test execution is test case prioritization, aiming to identify failures as quickly as possible. The *FAST Approaches to Scalable Similarity-based Test Case Prioritization* is an application that prioritizes test cases based on similarity. This project aims to implement a plugin for Gradle that enables the application of test case prioritization in the software development lifecycle.

Keywords: software testing; test case priorization; similarity; gradle

Resumo

O processo de testes de software é essencial para o desenvolvimento de software com qualidade. Entretanto as atividades de teste são bastante custosas, tomando boa parte do ciclo completo de desenvolvimento. Uma das estratégias para otimizar o tempo gasto com a execução dos testes é a de priorização de casos de testes, para que as falhas sejam identificadas o mais breve possível. O *FAST Approaches to Scalable Similarity- based Test Case Prioritization* é uma aplicação que faz a priorização dos casos de teste com base em similaridade. Este trabalho visa a implementação de um plugin para o *Gradle* que possibilite a aplicação da priorização dos casos de teste no ciclo de vida do software.

Palavras-chave: teste de software; priorização de casos de teste; similaridade; gradle

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AFPD Average Percentage of Faults Detected.

BB Caixa preta.

CD Continous Delivery.

CI Continous Integration.

CT Caso de teste.

FAST FAST *Approaches to Scalable Similarity-based Test Case Prioritization*.

IDE Integrated Development Environment.

RTS Regression Test Selection.

TCP Test Case Priorization.

TSM Test Suit Minimization.

WB Caixa branca.

CAPÍTULO 1

Introdução

Construir software com qualidade, eficiência, manutenibilidade, segurança e portabilidade, respeitando custos e prazos é um dos grandes desafios da Engenharia de Software. O Standish Group (2018) faz periodicamente, desde 1994, um relatório que apresenta a taxa de sucesso de projetos de software. O relatório, denominado CHAOS Report, do ano de 2015 avaliou 50.000 projetos em todo o mundo, e apenas 29% destes projetos foram concluídos com sucesso. Dos projetos que foram concluídos com sucesso, somente 8% eram considerados grandes ou muito grandes, enquanto 62% eram considerados pequenos. Esses dados sugerem que quanto maior e mais complexo um projeto, menor é a sua taxa de sucesso [1].

Segundo Pressman [2], "a atividade de teste de software é um elemento crítico na garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação". Desta forma, o processo de testes de software é um grande aliado para superar esse desafio, seja alinhando as expectativas, descobrindo erros antes do cliente, detectando problemas de design, mantendo o sistema estável, melhorando a manutenibilidade do sistema, e até mesmo garantindo a entrega.

Entretanto, as atividades de teste são bastante custosas, tomando boa parte do ciclo completo de desenvolvimento [3]. A atividade de teste é um elemento crítico da garantia de qualidade de software e pode assumir até 40% do esforço despendido no processo de desenvolvimento, como afirma Pressman [2]. Somente o Google, em seus mais de 13 mil projetos, executa 150 milhões de casos de testes diariamente [4].

Diante das razões supracitadas, o teste de software tornou-se, pouco a pouco, um tema de grande importância com a necessidade de adaptação de métodos práticos que assegurem a qualidade dos produtos finais, a fim de torná-los confiáveis e de fácil manutenção. Mesmo com o custo elevado, esse processo é fundamental para avaliar a preservação do comportamento esperado para o sistema [5].

De acordo com Myers & Sandler [6], quanto mais tardiamente um defeito é encontrado em um sistema sendo desenvolvido ou testado, maior é o custo de sua correção. Esse grande número de testes é típico dos testes de regressão, pois eles visam garantir a integralidade do software como um todo após modificações. Entretanto, quanto mais testes são adicionados ao ciclo de regressão, mais demorado e, consequentemente, custoso ele vem a se tornar [3]. O processo mais comum existente para garantir que nenhuma funcionalidade foi impactada com as mudanças realizadas é o refazer todos os testes a cada versão do software.

Então, nesse contexto, faz-se essencial a utilização de estratégias que possam tornar este processo mais rápido e menos custoso. Existem algumas formas de reduzir o custo dos testes de regressão, como: redução de grupo de testes, seleção de testes de regressão e priorização de casos de teste. Neste trabalho, nosso foco será a priorização de casos de teste, que é, das metodologias citadas, a única que garante a não eliminação de nenhum caso de teste do conjunto gerado.

A priorização de casos de teste consiste na ideia de ordenar os casos de teste de acordo com critérios estabelecidos, dando prioridade aos testes que têm maior possibilidade de falha,

fazendo com que estas sejam identificadas o mais brevemente possível, facilitando assim a correção dos problemas.

Existem diversas abordagens de priorização. O FAST Approaches to Scalable Similarity-based Test Case Prioritization é uma dessas. Atualmente, o FAST conta com os testes processados de dez projetos reais para que sejam executadas as operações de priorização. Porém, o projeto não se encontra disponível como ferramenta, de forma que ela venha a suportar a realização da priorização com os casos de testes de outros softwares. Esta implementação pode facilitar e difundir o seu uso, além de dar margem para a criação de novas aplicações para o projeto [3].

Este trabalho busca um aprimoramento deste projeto, com a implementação de um plugin na ferramenta de geração de build Gradle, para facilitar a utilização do projeto para os desenvolvedores das mais diversas linguagens de programação, de forma que seja possível integrar a execução do FAST no ciclo de vida do software de forma mais simples.

1.1 Motivação

A aplicação da priorização dos casos de teste é fundamental para diminuir o custo de aplicação dos testes de regressão, fazendo com que os erros sejam descobertos o mais rápido possível e melhorando o desenvolvimento do software como um todo. Porém, o uso do FAST ainda é restrito à ferramenta de geração de build Maven. Diante disto, surgiu a proposta de tornar o FAST disponível no Gradle, que é bastante utilizada nos mais diversos projetos ao redor do mundo, sendo possível realizar a execução dos casos de testes do software de forma priorizada durante o processo de compilação ou geração de uma nova versão de alguma aplicação. E também dará a possibilidade de execução dos testes de maneira priorizada a qualquer momento do processo de desenvolvimento do projeto [3].

1.2 Objetivos

O objetivo geral desse trabalho é construir uma ferramenta que possibilite a utilização do FAST, aplicando a priorização dos casos de teste à execução dos testes do software para os projetos que utilizam o Gradle como ferramenta de geração de build.

Como objetivos específicos, podemos citar a criação de um plugin no Gradle que possibilite a integração com o FAST, além de um estudo comparativo dos resultados sem e com a aplicação do FAST aos testes de um sistema nos diversos tipos de algoritmos suportados pelo FAST.

CAPÍTULO 2

Conceitos Básicos

Neste capítulo é apresentado o embasamento teórico necessário para o entendimento da solução proposta neste trabalho. Este capítulo será composto de seis seções. Iniciaremos por uma introdução aos conceitos da Engenharia de Software (Seção 2.1), após isso, abordaremos conceitos e técnicas de teste (Seção 2.2). Na Seção 2.3, exploraremos com mais profundidade os Testes de Regressão, que é o tipo de teste tema deste trabalho. Na Seção 2.4 entraremos nas técnicas de priorização de testes, passando pelas ferramentas de automação de compilação na Seção 2.5.

2.1 Engenharia de Software

Segundo Tian, a expectativa das pessoas em relação à qualidade do software pode ser definida em duas vertentes: o software deve fazer o que se propõe a fazer e deve realizar tarefas específicas corretamente [7]. Assim, temos que, para os usuários de software, qualidade se refere ao software executar o que foi definido e de forma correta e satisfatória [8].

A Engenharia de Software atua nesse contexto, fornecendo técnicas, métodos, metodologias e ferramentas de apoio, e visa contribuir para o desenvolvimento de produtos de software de alta qualidade, maximizando o uso dos recursos e minimizando os custos de desenvolvimento [8].

O processo de engenharia de software pode ser dividido em uma série de passos ordenados que envolvem atividades, restrições e recursos, que irão produzir uma saída desejada. Esse processo envolve um conjunto de técnicas e ferramentas que são realizadas com o objetivo de desenvolver, manter e gerenciar softwares [3]. Segundo Sommerville, Engenharia de Software é uma disciplina da engenharia que está relacionada a todos os aspectos da produção de software, desde os primeiros estágios da especificação do sistema até a sua manutenção [9]. Ainda segundo Sommerville [9], mesmo que existam diversos processos de software diferentes, existem atividades comuns a todas elas. Estas são:

- Especificação do software: essa atividade visa definir as funcionalidades do software e suas restrições sobre operação;
- Design e implementação do software: garante que o software deve ser produzido para atender às especificações definidas anteriormente;
- Verificação e validação do software: visa garantir a qualidade do produto, certificando-se de que atenda de forma efetiva aos requisitos e às expectativas dos clientes.
- Evolução do software: garante que o software deve evoluir para atender às necessidades de mudança dos clientes.

No contexto deste trabalho, concentraremos nossa atenção na fase de verificação de software, que busca examinar se o software em desenvolvimento cumpre suas especificações sem apresentar erros ou falhas, proporcionando a funcionalidade esperada pelos usuários ou stakeholders. Esses procedimentos iniciam-se assim que os requisitos são estabelecidos e continuam

ao longo de todas as etapas do desenvolvimento. Uma das práticas mais comuns de verificação de software é a realização de testes de software, que desempenham um papel crucial na garantia da qualidade do produto final.

2.2 Testes de Software

Segundo Weiszflog [10], a qualidade é definida como: "Atributo, condição natural, propriedade pela qual algo ou alguém se individualiza, distinguindo-se dos demais; maneira de ser, essência, natureza. Excelência, virtude, talento". Com isso, é perceptível que a qualidade de software é uma área da Engenharia de Software que tem como objetivo principal garantir a qualidade do produto de software [5].

Nesse contexto, os testes de software são uma das atividades que visam garantir a qualidade de um software, tendo como finalidade validar se o produto em desenvolvimento está em conformidade com a especificação.

Segundo Sommerville, o processo de teste visa garantir que o software foi desenvolvido de acordo com os requisitos definidos, além de descobrir situações em que o comportamento do software não está dentro do esperado [9]. Nesse contexto, Dijkstra et al. [11] conseguiram chegar à conclusão de que os testes só podem mostrar a presença de defeitos, mas não a sua ausência.

Teste é uma atividade muito importante dentro do ciclo de vida de desenvolvimento de um software. Testar um software consiste em executá-lo usando dados variados. Dessa forma, é possível analisar os resultados dos testes em busca de erros ou anomalias [9].

Para que um processo de teste seja bem-sucedido, alguns pontos devem ser analisados:

- O processo de teste deve ser tratado pelas organizações como um processo: Verifica-se que a maioria das empresas desenvolvedoras de software não dá a devida importância à atividade de teste, sendo esta uma fase informal, conduzida sem metodologia e com funções não definidas, se confundindo com a própria gestão do processo de desenvolvimento.

- Os testes devem abranger o sistema completamente: É evidente que, se os testes forem incompletos durante o desenvolvimento, a probabilidade de ocorrerem problemas após sua implantação é notória.

- A abordagem de testes deve ser adequada a novas tecnologias: Investir na reciclagem e/ou treinamento do pessoal técnico de testes, de forma a adequar os procedimentos de testes às novas tecnologias.

- A estrutura organizacional para testes deve ser modificada: Em geral, quase todas as etapas do processo de teste são realizadas pelos desenvolvedores. É interessante ressaltar que essa característica não deve ser necessariamente eliminada, porém, em alguns estágios do processo de testes, há a necessidade de que pessoas qualificadas e com o perfil adequado à execução dos testes passem a avaliar o produto.

- Ferramentas de automação de testes devem ser usadas: A automação agiliza o processo de testes e diminui os custos na etapa de manutenção. Além disso, há alguns tipos de testes, de desempenho por exemplo, que são inviáveis de serem executados manualmente.

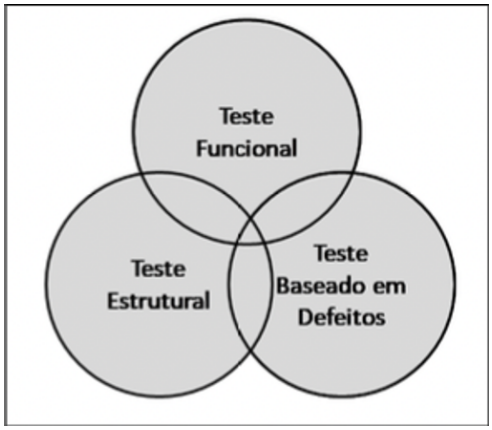

Figura 2.1: Combinação das técnicas de teste

- Os artefatos produzidos durante o processo de testes devem ser documentados: Cada fase do processo de teste deve ser devidamente documentada, pois, além de facilitar a futura automação das atividades de teste, estreita a relação entre os processos de teste e de desenvolvimento e ainda, fornece estrutura para organizar, conduzir e gerenciar o processo de teste. Além disso, as informações dos testes são de grande ajuda para a realização de outras atividades, tais como a atividade de depuração.

Existem diversos tipos de testes, desde os manuais, que envolvem uma abordagem presencial, exigindo a interação física com o aplicativo ou software, até os automatizados, conduzidos por máquinas que executam scripts de teste previamente elaborados. Os testes manuais demandam a presença de um testador para configurar ambientes e realizar os testes pessoalmente. Contudo, essa abordagem pode resultar em custos elevados e está sujeita a erros humanos, como erros ortográficos ou omissão de etapas nos scripts de teste.

Por outro lado, os testes automatizados são executados por máquinas seguindo scripts programados com previamente. Esses testes abrangem uma ampla gama de complexidades, desde a verificação de métodos individuais em uma classe até a garantia de que uma sequência complexa de ações na interface do usuário produza resultados consistentes. Em comparação com os testes manuais, os automatizados são mais robustos e confiáveis. No entanto, a qualidade desses testes automatizados depende da precisão com que os scripts foram desenvolvidos.

2.2.1 Técnicas e Critérios de Teste

Para que o processo de teste seja viável e ainda garanta alta qualidade no software, é preciso utilizar técnicas de teste. São elas: técnica estrutural, técnica funcional ou comportamental e técnica baseada em defeitos [12]. Segundo Myers, Sandler e Badgett [13], para elaborar um rigoroso processo de teste, é recomendado que se use uma combinação das técnicas, ou, se possível, todas elas, uma vez que cada uma tem suas vantagens e desvantagens.

A figura 2.1 mostra uma representação da combinação das técnicas de teste.

A técnica de teste funcional utiliza informações da especificação do programa para a geração de casos de teste; por esse motivo, o teste funcional é um tipo de teste normalmente conhecido como Teste Caixa-Preta. Em decorrência de não se conhecer o código do programa, os casos de teste são projetados a partir da especificação do produto de software. Nesse sentido, a partir da especificação, é definido o conjunto de valores que podem ser utilizados para executar o programa em teste, bem como o conjunto de casos de teste. Em seguida, o programa é testado,

e o conjunto de resultados produzidos pelo programa é comparado com a especificação do programa [14].

Já a técnica de teste estrutural baseia-se na estrutura interna do programa ou software. Nesse sentido, a pessoa que conduz a atividade de teste deve levar em consideração os dados de entrada, a semântica e sintaxe do programa, o programa em execução e os dados de saída para elaborar os casos de teste. Assim, enquanto se faz a analogia do teste funcional a uma "Caixa-Preta", pode-se analogamente relacionar o teste estrutural com uma "Caixa-Branca", em contradição ao funcional [14].

A técnica baseada em defeito é constituída por critérios que buscam definir requisitos de testes levando em consideração os defeitos típicos do processo de implementação do software [15]. Essa técnica coloca em ênfase os defeitos que normalmente são cometidos pelo programador durante o processo de implementação do código. Nela, modelos de defeitos são utilizados para criar hipóteses sobre defeitos que podem estar presentes no programa ou software e elaborar ou avaliar o conjunto de casos de teste com base na capacidade que os mesmos possuem em revelar os defeitos que foram modelados por meio das hipóteses [16].

2.3 Testes de Regressão

As modificações, em especial a adição e modificação de funcionalidades, têm um grande custo e esforço para serem realizadas. Durante o processo de evolução de software, defeitos podem ser introduzidos [5]. Nesse contexto, o teste de regressão é uma estratégia de teste de software que tem como objetivo garantir que alterações ou correções feitas em partes do software não alterem o funcionamento ou propaguem efeitos indesejados para outras partes [17], [18].

Formalmente, testes de regressão podem ser definidos da seguinte maneira: dado um programa Pi e um conjunto de casos de teste Ti; quando Pi sofre alteração de versão para Pi+1, deve ser gerado um novo conjunto de casos de teste Ti+1 que contém um subconjunto de Ti e novos casos de teste, capazes de testar as alterações em Pi+1 [19]. Para Elbaum [20], o teste de regressão é um processo de teste caro usado para validar modificações de software e na detecção de novos defeitos introduzidos em um código testado anteriormente. Esta técnica é utilizada a cada adição ou modificação de funcionalidades e defende que todos os casos de teste devem ser executados a fim de garantir que nenhuma funcionalidade tenha sido comprometida. Mesmo com o custo elevado, pois consome bastante tempo e recurso, esse processo garante qualidade e resultados [5].

Casos de teste de regressão devem ser reutilizados na nova situação do sistema para garantir que qualquer alteração realizada não afete a já alcançada estabilidade de funcionamento [21]. De nada adianta promover inovações e atender às exigências de clientes para novas funcionalidades se a introdução desses fatores no sistema também afetar negativamente as funcionalidades que já vinham sendo executadas de maneira satisfatória [22].

Devido aos testes de regressão serem extremamente custosos, foram desenvolvidas diversas técnicas para otimizá-los. Existem três técnicas que são comumente usadas para otimizar os testes de regressão: seleção de testes de regressão (RTS, do inglês Regression Test Selection), minimização do conjunto de teste (TSM, do inglês Test Suite Minimization) [23] e priorização de casos de teste (TCP, do inglês Test Case Prioritization) [1].

A Figura 2.2 mostra um esquema que exemplifica a aplicação das técnicas de seleção de testes de regressão, priorização de casos de teste e minimização do conjunto de teste. Do lado esquerdo está o programa Pi e o conjunto de casos de teste Ti. Do lado direito está o programa que sofreu alterações Pi+1 e o conjunto de casos de teste de regressão Ti+1 que deverá ser

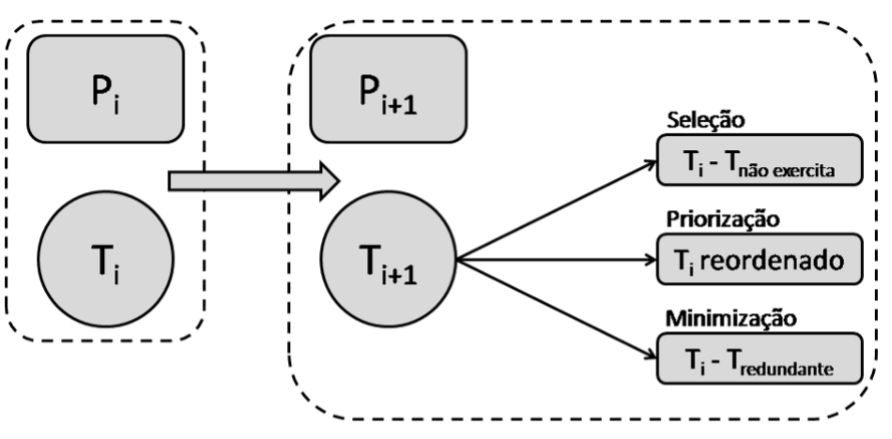

Figura 2.2: Aplicando as técnicas de RTS, TCP e TSM nos testes de regressão

executado. Esse novo conjunto pode ser obtido por seleção, o qual seria equivalente ao conjunto Ti menos o conjunto de casos de teste que não exercitam as partes alteradas do código (Ti - Tnão exercita); por priorização, o qual seria equivalente ao conjunto Ti reordenado por algum critério (Ti reordenado); ou por minimização, o qual seria equivalente ao conjunto Ti menos o conjunto de casos de teste redundantes (Ti - Tredundante) [1].

Neste trabalho, o foco será a priorização dos casos de teste ou TCP, pois das estratégias acima citadas, é a única que não descarta nenhum caso de teste. Nos aprofundaremos na TCP na seção posterior.

2.4 Priorização do Casos de Testes

A priorização dos casos de teste é uma das técnicas mais utilizadas para otimizar os testes de regressão. As técnicas de TCP ajudam a revelar defeitos logo no início dos testes, e um grande diferencial é que todo o conjunto de casos de teste pode ser executado. Com isso, é possível evitar problemas relacionados à omissão de casos de teste[24].

Esta metodologia engloba técnicas que ordenam os CT de acordo com determinados critérios, de modo que os mais prioritários sejam executados primeiro durante a execução dos TR [20]. Por exemplo, os CT podem estar ordenados para atingir uma completa cobertura de código o mais rápido possível; ou de acordo com o histórico de execuções anteriores [25]; ou fazer uso das funcionalidades de acordo com a expectativa de frequência de uso das mesmas [26]; ou aumentar a possibilidade de detecção de defeitos no início dos testes [27].

A priorização dos casos de teste é feita de acordo com algum critério. Por exemplo, um determinado critério pode definir uma ordem de casos de teste que seja eficaz para atingir um objetivo, mas que seja ineficaz para atingir outro objetivo [28].

Além disso, Rothermel et al. [29] fazem distinção de dois tipos de priorização de casos de teste: geral e específica de versão. Na priorização geral, a suíte de testes priorizada pode ser utilizada para *n* versões de código, já que ela será semelhante para todas. Em contrapartida, na priorização específica de versão, a suíte de testes priorizada é válida apenas para a versão gerada.

Existem diversas técnicas de priorização de casos de teste, que podem ser divididas em grandes grupos [30] [31]:

- Técnicas baseadas em cobertura: prioriza os casos de teste de acordo com a cobertura

$$APFD = 1 - \frac{(TF1 + TF2 + ... + TFn)}{nm} + \frac{1}{2n}$$

Figura 2.3: Fórmula de Average Percentage of Faults Detected (APFD)

de código (testes estruturais); casos de teste com maior cobertura podem aumentar a detecção de defeitos;

- Técnicas baseadas em distribuição: prioriza os casos de teste de acordo com seus perfis; casos de teste com o mesmo perfil são agrupados para evitar redundância;
- Técnicas baseadas em fatores humanos: prioriza os casos de teste de acordo com fatores que os testadores julgam mais importantes;
- Técnicas baseadas em probabilidade: prioriza os casos de teste de acordo com teorias probabilísticas; uma técnica bastante conhecida é a baseada em rede Bayesiana;
- Técnicas baseadas em histórico: prioriza os casos de teste de acordo com informações de histórico de execução dos casos de teste e mudanças no código;
- Técnicas baseadas nos requisitos: prioriza os casos de teste de acordo com informações extraídas dos requisitos como a importância do requisito para o cliente, a volatilidade do requisito, a complexidade de implementação do requisito;
- Técnicas baseadas em modelo: prioriza os casos de teste de acordo com informações extraídas de modelos UML (diagramas de sequência ou de atividades);
- Técnicas baseadas no custo dos casos de teste: prioriza os casos de teste de acordo com os custos definidos para cada caso de teste;
- Outras técnicas: outras técnicas conhecidas, mas que não se encaixam em nenhum dos casos anteriores como priorização usando algoritmos genéticos, priorização dos casos de teste das partes de relevância, entre outros.

Apesar dos vários tipos de técnicas de priorização e da grande quantidade de variações que os autores apresentam dentro de cada tipo, grande parte das técnicas utiliza a mesma métrica para avaliar os resultados e a sua eficácia. Essa métrica, chamada de Average Percentage of Faults Detected (APFD), foi proposta por Elbaum, Malishevsky e Rothermel [20]. Seus valores vão de 0 a 100 e quanto maior o APFD, melhor a taxa de detecção de defeitos do conjunto de casos de teste. Consequentemente, mais rápido os defeitos são revelados por aquele conjunto. A fórmula para calcular o APFD está demonstrada na figura 2.3.

Na equação da métrica APFD, n representa o número de casos de teste do conjunto; m representa o número de defeitos revelados pelo conjunto; e TFi é o primeiro caso de teste da sequência que revelou o defeito i.

2.4.1 FAST Approaches to Scalable Similarity-based Test Case Prioritization

São muitos e com as mais distintas abordagens os algoritmos para a priorização dos casos de teste. Neste trabalho nos aprofundaremos no FAST Approaches to Scalable Similarity-based

```
Input  : Coded test suite info T; (optional) selection function f.
Output : Prioritized test suite P.
1  P ← EmptyList()
2  I ← GetTestCaseIDs(T)
3  M ← MHSignatures(T)                          ▷ No need of T from here on
4  B ← LSHBuckets(M)
        ▷ M(v): Cumulative signature of so-far-ordered test cases
5  M(v) ← MHSignature(∅)
6  while |P| ≠ |I| do
7      C_s ← LSHCandidates(B, M(v))
8      if C_s = ∅ then
9          M(v) ← MHSignature(∅)
10         C_s ← LSHCandidates(B, M(v))
11     C_d ← (I − P − C_s)                      ▷ Complement of C_s
12     s ← Select(M(v), M, C_d, f)
13     M(v) ← UpdateMHSignature(M(v), M, s)
14     M ← Remove(M, s)
15     P ← Append(P, s)
16 return P
17 function Select(M(v), M, C, f)
18     if no f then                                              ▷ FAST-pw
19         return arg max_{c ∈ C} { EstimateJD(M(v), M(c)) }
20     else                                                      ▷ FAST-f
21         return RandomSample(C, f)
```

Figura 2.4: Algoritmo do FAST

Test Case Prioritization, ou apenas FAST. Ele é uma família de algoritmos para priorização de casos de teste que utilizam técnicas de big data e de mineração de dados, como algoritmos de hashing sensível à localidade e minhashing, para encontrar casos de teste diversos dentro de um grande conjunto e realizar o processo de TCP. A família é composta pelos algoritmos FAST-pw, FAST-one, FAST-log, FAST-sqrt e FAST-all [3].

Segundo Miranda [32], um estudo de simulação de escalabilidade uma das técnicas do FAST conseguiu priorizar mais de um milhão de casos de teste em menos de vinte minutos. Isso demonstra a escalabilidade do FAST, algo não encontrado em outras técnicas existentes [32]. Ademais, o FAST possui priorização escalonável tanto para casos de teste de caixa preta quanto para de caixa branca.

Na figura 2.4 é demonstrado em pseudocódigo o algoritmo do FAST.

As linhas 2 e 3 são linhas executam a preparação dos dados. A entrada do algoritmo, que são os casos de teste são transformados em assinaturas minhash M. Nos testes de caixa branca, as informações de cobertura de código podem ser representadas diretamente como conjuntos (independentemente do critério de cobertura), já nos testes de caixa preta representação em string dos casos de teste precisa ser pré-processada em k-shingles. Observe que esta é a única operação em que o FAST trata as entradas BB e WB de maneira diferente. Foi utilizado k = 5 para ter uma representação de conjunto adequada. Uma vez que as assinaturas minhash são computadas, T não é mais necessário e apenas M é usado durante o processo de priorização. Usamos um número de funções hash h = 10, o que garante um erro esperado não maior que 0,32 na estimativa da similaridade (e distância) de Jaccard entre duas assinaturas. Mesmo que o erro na estimativa seja alto, a escolha do próximo caso de teste é realizada sobre um subconjunto de testes que são todos diferentes dos até então priorizados [32].

Na Linha 4, a coleção de buckets LSH é calculada. A variável B será preenchida com b

buckets, e cada um tem controle de todos os casos de teste que estão colidindo. Foi definido o número de bandas b = 10 e linhas r = 1 de modo que o número de linhas na matriz de assinatura seja igual ao tamanho da assinatura, ou seja, h = r · b. Esses valores garantem um limiar de similaridade s próxima a 0,1 para o conjunto candidato. Observe que, enquanto para encontrar os itens mais semelhantes um limite de similaridade mais alto seria melhor, para o contexto de STP queremos selecionar os casos de teste que são diferentes dos até agora priorizados. Intuitivamente, com um limiar de similaridade s = 0,1 o conjunto candidato conterá quase todos os casos de teste, exceto aqueles que são diferentes de M(v) (ou seja, com distância de Jaccard maior que 0,9). O conjunto candidato real Cd usado pelo FAST é calculado na Linha 11, como o complemento de Cs, excluindo os casos de teste priorizados até agora.

Entre as linhas 6 e 15 é onde a priorização de fato acontece. Os conjuntos candidatos são criados dentro do while. M(v) é dividido em b bandas e em cada banda é aplicado um hash. Caso exista uma colisão com o bucket correspondente em B, os casos de teste desse bucket são adicionados ao conjunto candidato Cs. M(v) é inicializado na Linha 5 e atualizado na Linha 13, sempre que novos casos de teste são selecionados para acompanhar a assinatura cumulativa dos casos de teste até então ordenados. Entre as Linhas 8 e 10, sempre que Cs estiver vazio, redefinimos M(v) e recalculamos Cs [32].

Na Linha 17, a função Select é onde as diferentes abordagens do FAST são diferenciadas. A abordagem FAST-pw calcula a distância Jaccard estimada entre M(v) e cada caso de teste no conjunto de candidatos Cd usando assinaturas minhash e seleciona o candidato que está mais distante de M(v). As outras abordagens usam uma função f que é fornecida como entrada para o algoritmo para selecionar um subconjunto aleatório de Cd de tamanho f(|Cd|). Na Linha 15, os casos de teste selecionados são anexados ao conjunto de testes priorizado P. Para os experimentos deste trabalho, consideramos as seguintes funções que aumentam progressivamente a eficiência da priorização: one, log, sqrt, all. Em geral, f é uma função genérica que pode ser ajustada para alcançar o equilíbrio certo entre a eficiência e a precisão exigidas em um contexto específico [32].

2.5 Ferramentas de Automação de Build

O processo de build de um software é responsável por verificar se todos os componentes do nosso código fonte estão sendo integrados corretamente. Antigamente, esse processo era manual e, com o passar do tempo, tornou-se uma tarefa repetitiva e propensa a erros. A necessidade de resolver esse problema levou à ideia de automatizar esse processo, tornando-o mais rápido, completo e acessível a qualquer pessoa.

A automação de compilação é o processo de automatizar a geração do código fonte de um software, verificando a correta integração de todos os componentes. Dentre as ações que fazem parte da automação, incluem-se a compilação em código binário, a execução de testes automatizados, a publicação em um repositório compartilhado e centralizado, a realização do deploy, a verificação do código, a compatibilidade entre IDEs, o empacotamento e a distribuição da aplicação, a adição de bibliotecas, entre outras.

Os benefícios da utilização de ferramentas de automação de build são enormes, destacando-se:

- Aumento da produtividade: A automação de construção garante um feedback rápido, permitindo que os desenvolvedores aumentem a produtividade. Eles gastarão menos tempo lidando com ferramentas e processos e mais tempo entregando valor.

- Aceleração da entrega: A automação de construção ajuda a acelerar a entrega, eliminando tarefas redundantes e garantindo que problemas sejam identificados mais rapidamente, possibilitando uma liberação mais rápida.
- Melhoria da qualidade: A automação de construção ajuda a equipe a se movimentar mais rapidamente, encontrando problemas mais rapidamente e resolvendo-os para melhorar a qualidade geral do produto e evitar construções inadequadas.
- Manutenção de um histórico completo: A automação de compilação mantém um histórico completo de arquivos e alterações, permitindo rastrear os problemas até a origem.
- Economia de tempo e dinheiro: A automação de construção economiza tempo e dinheiro, configurando para CI/CD, aumentando a produtividade, acelerando a entrega e melhorando a qualidade.

Existem diversas opções de ferramentas de automação de build, incluindo o Maven, o Ant e o Gradle. Este último é o foco deste trabalho.

2.5.1 Gradle

O Gradle é uma ferramenta de compilação open source, implementada na linguagem Groovy, com foco na automação de compilação e suporte para desenvolvimento. Se você está construindo, testando, publicando e implantando software em qualquer plataforma, o Gradle oferece um modelo flexível que pode suportar todo o ciclo de vida de desenvolvimento, desde a compilação e empacotamento do código até a publicação de sites. O Gradle foi projetado para dar suporte à automação de compilação em várias linguagens e plataformas, incluindo Java, Scala, Android, Kotlin, C/C++ e Groovy, e está intimamente integrado a ferramentas de desenvolvimento e servidores de integração contínua, como Eclipse, IntelliJ e Jenkins.

Dentre as ferramentas citadas, o Gradle é a mais recente e, por se tratar de uma ferramenta open source, a quantidade de updates e upgrades tende a ser maior do que nas demais ferramentas, aumentando a probabilidade de ser uma ferramenta cada vez mais moderna.

A ideia do Gradle é permitir configurações baseadas em tasks, ou seja, quando queremos um novo comportamento durante o build, basta criar uma task. Uma task é basicamente uma estrutura que executa alguma ação, dada uma certa entrada. Além disso, o Gradle possui diversas tasks predefinidas para facilitar a configuração do projeto.

CAPÍTULO 3

Metodologia

O capítulo que se inicia tem por finalidade descrever a metodologia de pesquisa utilizada e o método desenvolvido. Todo o embasamento teórico necessário para esta etapa foi discutido no Capítulo 2 – Conceitos Básicos, deixando para o Capítulo 3 a missão de descrever o método propriamente dito. A aplicação do método e os resultados obtidos serão abordados no Capítulo 4 – Experimentos e Análise de Resultados.

3.1 Contextualização

A metodologia em um trabalho de graduação refere-se à descrição detalhada dos métodos e procedimentos utilizados para abordar seu tema de estudo, desempenhando assim um papel fundamental na estruturação e na compreensão deste trabalho, pois fornece informações sobre como foram coletados e analisados os dados, como chegou às conclusões e como foram garantidas a validade e a confiabilidade desta pesquisa.

3.2 Etapas

Neste trabalho, a metodologia está dividida em seis tópicos: 3.2.1 - Desenvolvimento, Estrutura - 3.2.2, Instalação - 3.2.3, Utilização - 3.2.4 e Arquivos Gerados - 3.2.5.

3.2.1 Desenvolvimento

O desenvolvimento do projeto se deu em duas etapas: parametrização do código do maven-FAST para funcionamento com o JUnit 4 e implementação do plugin para o Gradle. Na parametrização do código do maven-FAST para funcionamento com o JUnit 4, foi necessário fazer algumas atualizações no código para dar suporte às annotations do JUnit 4, tornando-o capaz de lidar com projetos que utilizem esta versão. Para esta etapa, foi utilizado um sistema operacional MacOS, além do Visual Studio Code como IDE. O código fonte está disponibilizado no repositório maven-FAST.

Na implementação do plugin para o Gradle, foram utilizados alguns tutoriais, como o do próprio Gradle, do Baeldung, além de um artigo de Tom Gregory. Além disso, foram consultados alguns repositórios de plugins do Gradle e alguns links do Stack Overflow para sanar algumas dúvidas. Para esta etapa, foi utilizado um sistema operacional MacOS, além do IntelliJ IDEA como IDE. O código fonte está disponibilizado no repositório FAST-gradle-plugin.

3.2.2 Estrutura

Como abordado anteriormente, este projeto pode ser dividido em duas partes. O projeto FAST-Maven é composto pelo diretório /py, onde são armazenados os arquivos fast.py, lsh.py e pri-

oritize.py, responsáveis pela função central do projeto, a priorização de casos de testes. Além disso, a pasta também possui o arquivo FASTWatchdog.py, utilizado para monitorar alterações que podem ter sido feitas no projeto a ser testado. Ele também é composto pelo diretório /tests, que possui os testes desenvolvidos para o projeto. Por fim, a pasta /tools possui ferramentas auxiliares que foram desenvolvidas para facilitar a sua utilização. Ele é composto pelos arquivos clean-project.py, que realiza a exclusão dos arquivos gerados pelo projeto, e pelo arquivo project-instrumentation.py que pode adicionar ou remover a instrumentação necessária nas classes de teste do projeto [3].

A integração entre o FAST-gradle-plugin com o projeto FAST-maven é realizada através da execução de comandos via CMD feitos pelo plugin internamente. Para esta integração também é necessário que o repositório do FAST-maven seja clonado localmente. Tal processo é realizado internamente através do plugin e clona o repositório no diretório local do Maven (normalmente o diretório /.m2/repository), dentro da pasta do plugin (/.m2/repository/br/ufpe/cin/fast-tool/fast-maven-plugin).

O processo de clonagem inicial do repositório é realizado apenas uma única vez, durante a compilação do projeto após a adição do plugin a ele. Este procedimento adiciona um tempo extra que irá variar de acordo com a largura de banda da conexão à internet do usuário. Nos testes realizados, o tempo inicial ficou entre um e três minutos. Além disso, todo o processo de clonagem do repositório é transparente para o usuário, sendo exibido no terminal o processo [3]. O processo de clonagem do repositório do FAST-maven não é repetido quando o usuário tenta integrá-lo a outros projetos Java, depois do primeiro uso no ambiente. Para essas próximas utilizações é usado o repositório que foi clonado inicialmente, e é exibida ao usuário uma mensagem de que o repositório não necessita ser clonado [3].

3.2.3 Instalação

Esta seção tem por objetivo explicar a utilização do plugin em projetos que utilizam o Gradle como ferramenta de build. Inicialmente, é necessário fazer o clone do repositório do projeto e executar os comandos *gradle build* e *gradle publishToMavenLocal* para executar o build e publicar no repositório maven local (pasta .m2). O Gradle utiliza os repositórios do Maven em seus plugins. No cenário ideal, o plugin já estará publicado em um repositório remoto, como o Maven Central Repository, não sendo necessário este passo. Será necessário apenas referenciar o plugin nos arquivos de configuração do Gradle. Este passo será explicado mais à frente. Devido a algumas questões externas, a publicação em um repositório externo está fora do escopo deste trabalho.

3.2.4 Utilização

Após a instalação explicada na seção 3.2.3, o plugin fica disponível para utilização no Gradle. Para isso, deverão ser feitas alterações nas configurações nos arquivos do Gradle *settings.gradle* e *build.gradle*.

No arquivo settings.gradle, será necessário inserir as configurações de acordo com a figura 3.1. Na linha 3, é permitido o acesso do projeto ao repositório local do Maven para tornar o plugin visível ao projeto. Na linha 4 é permitido o acesso aos plugins remotos do Gradle que são utilizados pelo projeto.

Já no arquivo build.gradle, será preciso importar a dependência do plugin, conforme linha 8 da figura 3.2. Além disso, será preciso definir a parametrização da task prioritize do plugin, conforme mostrado na figura 3.3.

```
pluginManagement {
    repositories {
        mavenLocal()
        gradlePluginPortal()
    }
}
```

Figura 3.1: Arquivo settings.gradle

```
plugins {
    id 'java-library'
    id 'maven-publish'
    id 'br.ufpe.cin.fastgradle' version '0.0.1-SNAPSHOT'
}
```

Figura 3.2: Instanciando o plugin no arquivo build.gradle

Além dos passos supracitados, é necessário definir as tasks para priorização, conforme as figuras 3.4 e 3.5, onde definimos as seguintes tasks:

- testPrioritizedBuild: possibilita a execução dos passos do build padrão do Gradle, adicionando a task prioritize para efetuar a priorização.
- testPrioritized: possibilita a execução dos testes de acordo com a priorização do plugin.
- test: possibilita apenas a execução dos testes sem priorização.

Seguindo os passos acima descritos, é possível a utilização da priorização do FAST em qualquer projeto que utiliza o Gradle como ferramenta de build.

3.2.5 Arquivos Gerados

Durante o processo de priorização dos casos de teste, são criados alguns arquivos no diretório do projeto, sendo a pasta /.fast um deles. Esta pasta contém os arquivos resultantes da priorização dos casos de teste, incluindo assinaturas dos casos de teste, os resultados da priorização, os caminhos de cada caso de teste identificado no projeto, e outros documentos relevantes. Além disso, é produzido o arquivo FASTPrioritizedSuite.java (conforme exibido na figura 3.6), uma classe Java que armazena os casos de teste ordenados com base nos resultados do processo de priorização do FAST.

```
prioritize {
    projectPath = "/Users/luizandre/Repositories/TCC/projects/commons-cli"
    algorithm = "FAST-pw"
    repetitions = "1"
}
```

Figura 3.3: Utilizando a task Prioritize do plugin no arquivo build.gradle

```
53  tasks.register('testPrioritizedBuild', GradleBuild) { GradleBuild it ->
54      tasks = ['clean',
55               'compileJava',
56               'processResources',
57               'classes',
58               'jar',
59               'assemble',
60               'compileTestJava',
61               'processTestResources',
62               'testClasses',
63               'prioritize',
64               'test',
65               'check',
66               'build']
67  }
```

Figura 3.4: Registranto task testPrioritizedBuild no arquivo *build.gradle*

```
80  test {
81      filter {
82          excludeTestsMatching '*.FASTPrioritizedSuite'
83      }
84  }
85
86  tasks.register('testPrioritized', Test) { Test it ->
87      include '**/fast/**'
88  }
```

Figura 3.5: Definindo instrumentação para executar os testes priorizados ou não de acordo com a task

```
207 @RunWith(Suite.class)
208 @Suite.SuiteClasses({
209     AbstractIteratorTest.class,
210     UnmodifiableOrderedBidiMapTest.class,
211     CollectionUtilsTest.class,
212     TestUtils.class,
213     ArrayStackTest.class,
214     SetUtilsTest.class,
215     PredicatedMapTest.class,
216     DefaultKeyValueTest.class,
217     TiedMapEntryTest.class,
218     MultiKeyTest.class,
219     DefaultMapEntryTest.class,
220     UnmodifiableMapEntryTest.class,
221     AbstractMapEntryTest.class,
222     UnmodifiableMultiValuedMapTest.class,
223     AbstractCollectionTest.class,
224     AbstractAnyAllOnePredicateTest.class,
225     AbstractMapIteratorTest.class,
226     AllPredicateTest.class,
227     AbstractMapTest.class,
228     AbstractCompositePredicateTest.class,
229     AbstractListTest.class,
230     AbstractQueueTest.class,
231     AbstractBagTest.class,
232 })
233 public class FASTPrioritizedSuite{}
```

Figura 3.6: Conteúdo do arquivo FASTPrioritizedSuite.java do Commons CLI

CAPÍTULO 4

Experimentos e Análise de Resultados

Nesta seção, serão apresentados os resultados dos experimentos realizados com o FAST-gradle-plugin com o intuito de avaliar os resultados obtidos, analisando o tempo adicional de execução do plugin para a priorização dos testes, comparando com a execução sem a devida priorização.

Nos testes, foram utilizados projetos da família Commons devido à quantidade relevante de testes que cada projeto possui. No entanto, considerando que todos os projetos utilizados neste trabalho utilizam o Maven como ferramenta de build, foi necessário convertê-los para o Gradle. Utilizando a task *gradle init*, foi possível migrar boa parte das configurações, mas, após isso, ainda foi necessário corrigir versões de bibliotecas e trocar o escopo de dependências.

Nos testes do projeto, foram metricadas as quantidades de métodos de teste. Como resultados, foram avaliados os tempos de execução com e sem o plugin e o percentual de tempo adicionado após a execução dos testes com o plugin. Os experimentos e resultados estão descritos adiante.

4.1 Commons CSV

O Commons CSV é uma biblioteca que fornece muitos recursos úteis para criar e ler arquivos CSV. Arquivos CSV (valores separados por vírgula) são amplamente utilizados para a troca de dados entre aplicativos. No entanto, as operações com arquivos CSV podem ser complicadas e demoradas.

Os resultados obtidos nos experimentos estão demonstrados na tabela 4.1.

4.2 Commons Collections

O Commons Collections é uma biblioteca que fornece uma série de classes, interfaces e métodos para facilitar as operações com coleções de dados.

Os resultados obtidos nos experimentos estão demonstrados na tabela 4.2.

Quantidade de testes	305 testes
Tempo de execução dos testes sem plugin	17,349 segundos
Tempo de execução dos testes com plugin	18,291 segundos
Overhead de tempo do plugin em segundos	0,942 segundos
Overhead percentual de tempo do plugin	5,43%

Tabela 4.1: Resultados da execução do plugin no Commons CSV

Quantidade de testes	16190 testes
Tempo de execução dos testes sem plugin	14,993 segundos
Tempo de execução dos testes com plugin	16,124 segundos
Overhead de tempo do plugin em segundos	1,131 segundos
Overhead percentual de tempo do plugin	7,5%

Tabela 4.2: Resultados da execução do plugin no Commons Collections

Quantidade de testes	803 testes
Tempo de execução dos testes sem plugin	13,889 segundos
Tempo de execução dos testes com plugin	14,207 segundos
Overhead de tempo do plugin em segundos	0,318 segundos
Overhead percentual de tempo do plugin	2,24%

Tabela 4.3: Resultados da execução do plugin no Commons Codec

4.3 Commons Codec

O Commons Codec é uma biblioteca que contém codificadores e decodificadores simples para vários formatos, como Base64 e Hexadecimal. Além desses codificadores e decodificadores amplamente utilizados, o pacote de codecs também mantém uma coleção de utilitários de codificação fonética.

Os resultados obtidos nos experimentos estão demonstrados na tabela 4.3.

4.4 Commons CLI

O Commons CLI é uma biblioteca que fornece suporte para operações que utilizam a linha de comando, como execução de programas, passagem de argumentos, além de comandos de ajuda.

Os resultados obtidos nos experimentos estão demonstrados na tabela 4.4.

4.5 Resultados

De acordo com os resultados descritos nas seções anteriores deste capítulo, foi possível avaliar a execução dos testes com o FAST-gradle-plugin. Nos projetos avaliados, a priorização do plugin gerou um overhead computacional entre 2,24% e 7,5% para a execução completa da suíte de testes. É importante ressaltar que, embora exista um overhead, esses números são considerados bastante favoráveis, pois não compromete significativamente o desempenho global, considerando os benefícios proporcionados pelo uso eficaz do plugin. O ganho na eficiência

Quantidade de testes	408 testes
Tempo de execução dos testes sem plugin	59ms
Tempo de execução dos testes com plugin	63ms
Overhead de tempo do plugin em segundos	4ms
Overhead percentual de tempo do plugin	6,78%

Tabela 4.4: Resultados da execução do plugin no Commons CLI

do processo de teste devido a priorização dos casos é um aspecto crucial que pode superar esse pequeno *overhead*, reforçando a utilidade e a eficácia do FAST-gradle-plugin na prática.

CAPÍTULO 5

Conclusão e Trabalhos Futuros

A priorização de casos de teste é uma técnica importante para otimizar o processo de testes de software. Com a crescente complexidade dos sistemas de software, a execução de todos os casos de teste pode se tornar inviável em termos de tempo e recursos. Nesse contexto, a utilização de técnicas de priorização de casos de teste pode ajudar a identificar os casos de teste mais relevantes e reduzir o tempo de execução dos testes.

Este trabalho apresentou o FAST-gradle-plugin, uma ferramenta que possibilita a utilização do FAST (Approaches to Scalable Similarity-based Test Case Prioritization) para priorização de casos de teste em projetos que utilizam o Gradle como ferramenta de automação de build. O objetivo geral deste trabalho foi construir uma ferramenta que possibilite a utilização do FAST para priorização de casos de teste e analisar o ganho que essa técnica trará em uma execução de uma quantidade massiva de testes.

Para atingir esse objetivo, foram definidos objetivos específicos, como a criação de um plugin no Gradle que possibilite a integração com o FAST e um estudo comparativo dos resultados com e sem a aplicação do FAST aos testes de alguns projetos nos diversos tipos de algoritmos suportados pelo FAST.

A revisão bibliográfica realizada neste trabalho mostrou que a priorização de casos de teste é uma técnica amplamente utilizada na indústria de software. Diversas abordagens foram propostas para a priorização de casos de teste, como a utilização de métricas de código, a análise de dependências entre os casos de teste e a utilização de técnicas de aprendizado de máquina. O FAST é uma abordagem que se destaca por sua eficiência e escalabilidade, sendo capaz de lidar com grandes conjuntos de casos de teste em um curto espaço de tempo.

A implementação do FAST-gradle-plugin mostrou-se simples e de fácil utilização. O plugin foi desenvolvido em Java e utiliza o FAST para realizar a priorização dos casos de teste. A integração com o Gradle foi realizada através da criação de um novo plugin, que pode ser facilmente adicionado ao arquivo de configuração do Gradle. O plugin permite a configuração de diversos parâmetros, como o número máximo de casos de teste a serem priorizados e o tipo de algoritmo a ser utilizado pelo FAST.

Os resultados obtidos mostraram que a utilização do FAST-gradle-plugin pode trazer benefícios significativos para o processo de testes de software. O estudo comparativo realizado neste trabalho mostrou que a utilização do plugin gera um overhead no tempo de execução dos testes, porém esses números são considerados bastante favoráveis, pois não comprometem significativamente o desempenho global. Então, através da análise de métricas de código e da priorização dos casos de teste mais com maior chance de falha, é possível aumentar a eficiência do processo de teste como um todo.

Além dos benefícios em termos de tempo de execução dos testes, a utilização do FAST-gradle-plugin também pode contribuir para a melhoria da qualidade do software. A priorização dos casos de teste mais relevantes permite identificar os defeitos mais críticos do software, aumentando a efetividade dos testes e reduzindo o risco de falhas em produção.

Como trabalhos futuros, sugere-se a realização de estudos mais aprofundados sobre a apli-

cação do FAST em diferentes contextos e a implementação de novas funcionalidades no FAST-gradle-plugin. Também é importante destacar que a utilização de técnicas de priorização de casos de teste deve ser combinada com outras técnicas de teste, como testes exploratórios e testes de integração, para garantir a qualidade do software.

Em resumo, este trabalho apresentou uma solução prática e eficiente para a utilização do FAST em projetos de desenvolvimento de software. Através da utilização do FAST-gradle-plugin, é possível priorizar os casos de teste mais relevantes e reduzir o tempo de execução dos testes, contribuindo para a melhoria da qualidade do software e para a otimização do processo de testes. A utilização do FAST em conjunto com outras técnicas de teste pode trazer benefícios ainda maiores para o processo de desenvolvimento de software, aumentando a efetividade dos testes e reduzindo o risco de falhas em produção.

Referências Bibliográficas

[1] G. G. Malimpensa, "Uma abordagem para a priorização de casos de teste de regressão baseada em rastreabilidade," 2018 (cit. on pp. 1, 6, 7).

[2] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005 (cit. on p. 1).

[3] S. A. d. Barros, "FAST-MAVEN-PLUGIN: Um Plugin para a Priorização de Casos de Testes Baseada em Similaridade," *Trabalho de Graduação*, 2021 (cit. on pp. 1–3, 9, 13).

[4] A. Memon, Z. Gao, B. Nguyen, *et al.*, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, IEEE, 2017, pp. 233–242 (cit. on p. 1).

[5] B. É. S. Cavalcante *et al.*, "Uma técnica de priorização de casos de teste para múltiplas mudanças agregadas.," 2016 (cit. on pp. 1, 4, 6).

[6] G. J. Myers, *The art of software testing*. John Wiley & Sons, 2006 (cit. on p. 1).

[7] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005 (cit. on p. 3).

[8] A. R. Vidal *et al.*, "Teste funcional sistemático estendido: Uma contribuição na aplicação de critérios de teste caixa-preta," 2011 (cit. on p. 3).

[9] I. Sommerville, *Software Engineering, 9/E*. Pearson Education India, 2011 (cit. on pp. 3, 4).

[10] W. WEISZFLOG, "Dicionário online-dicionários michaelis-uol," *Editora Melhoramentos Ltda*, 2009 (cit. on p. 4).

[11] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming*. Academic Press Ltd., 1972 (cit. on p. 4).

[12] J. C. Maldonado, E. F. Barbosa, A. M. Vincenzi, M. E. Delamaro, S. d. R. S. Souza, and M. Jino, "Introducao ao teste de software (versão 2004-01)," 2004 (cit. on p. 5).

[13] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2 (cit. on p. 5).

[14] L. N. Paschoal, "Contribuições ao ensino de teste de software com o modelo flipped classroom e um agente conversacional," Ph.D. dissertation, Universidade de São Paulo, 2019 (cit. on p. 6).

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978 (cit. on p. 6).

[16] S. A. d. Andrade and M. E. Delamaro, "Execução paralela de programas como suporte ao teste de mutação," *Anais*, 2014 (cit. on p. 6).

[17] R. Abid and A. Nadeem, "A novel approach to multiple criteria based test case prioritization," in *2017 13th International Conference on Emerging Technologies (ICET)*, IEEE, 2017, pp. 1–6 (cit. on p. 6).

[18] R. Greca, B. Miranda, and A. Bertolino, "State of practical applicability of regression testing research: A live systematic literature review," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–36, 2023 (cit. on p. 6).

[19] H. Srikanth, M. Cashman, and M. B. Cohen, "Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study," *Journal of Systems and Software*, vol. 119, pp. 122–135, 2016 (cit. on p. 6).

[20] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002 (cit. on pp. 6–8).

[21] S. R. Rakitin, *Software verification and validation: a practitioner's guide*. Artech House, Inc., 1997 (cit. on p. 6).

[22] D. Elfriede, "Automate regression tests when feasible," *Automated Testing: Selected Best Practices, Pearson Education*, 2003 (cit. on p. 6).

[23] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 419–429 (cit. on p. 6).

[24] H. Do, "Recent advances in regression testing techniques," *Advances in computers*, vol. 103, pp. 53–77, 2016 (cit. on p. 7).

[25] V. Siqueira and B. Miranda, "Investigating the adoption of history-based prioritization in the context of manual testing in a real industrial setting," in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2022, pp. 141–148 (cit. on p. 7).

[26] B. Miranda and A. Bertolino, "Scope-aided test prioritization, selection and minimization for software reuse," *Journal of Systems and Software*, vol. 131, pp. 528–549, 2017 (cit. on p. 7).

[27] C. SIMONS, "Priorização de casos de testes de regressão usando amostragem por perseguição de defeitos," Ph.D. dissertation, Pontifícia Universidade Católica do Paraná, 2010 (cit. on p. 7).

[28] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "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)*, IEEE, 1999, pp. 179–188 (cit. on p. 7).

[29] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001 (cit. on p. 7).

[30] C. Catal and D. Mishra, "Test case prioritization: A systematic mapping study," *Software Quality Journal*, vol. 21, pp. 445–478, 2013 (cit. on p. 7).

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012 (cit. on p. 7).

[32] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, “Fast approaches to scalable similarity-based test case prioritization,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 222–232 (cit. on pp. 9, 10).