Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Uma técnica de priorização de casos de teste para múltiplas mudanças agregadas

Berg Élisson Sampaio Cavalcante

Campina Grande, Paraíba, Brasil

20/08/2016

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma técnica de priorização de casos de teste para múltiplas mudanças agregadas

Berg Élisson Sampaio Cavalcante

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software e Teste de Software

Tiago Lima Massoni (Orientador)
Patrícia Duarte de Lima Machado (Coorientador)

Campina Grande, Paraíba, Brasil

Um simples ato de carinho cria uma onda sem fim. (Autor desconhecido)

Resumo

É evidente hoje o grande investimento em qualidade de software. Assim, para submeter um produto com qualidade aceitável, é necessário determinar a sua testabilidade, uma propriedade que indica a facilidade e precisão na avaliação dos resultados de um teste. Teste de Regressão é um processo custoso, que demanda esforço considerável para detectar defeitos introduzidos em um código testado anteriormente. A fim de aumentar a custo-efetividade deste processo, são aplicadas técnicas de priorização de casos de teste (CTs), que tem por objetivo reordenar o conjunto de testes seguindo algum critério de ordenação. Em particular, a técnica Changed Blocks realiza priorização baseada em mudanças. Segundo estudos realizados neste trabalho, essa técnica apresenta algumas limitações, como: i. os resultados não obtém cobertura máxima de defeitos no topo da lista ordenada; ii. CTs com mesmo número de mudanças cobertas são ordenados aleatoriamente, sem seguir uma regra de importância específica; iii. CTs que revelam mudanças inéditas, mas que apresentam baixa cobertura de mudanças são desfavorecidos. Este trabalho propõe a implementação de duas técnicas baseadas na Changed Blocks, para que mudanças múltiplas agregadas em uma mesma versão do sistema em teste sejam melhor consideradas, não resultando em perdas aos benefícios oferecidos pela técnica original. Várias métricas foram utilizadas na análise, são elas: APFD; F-measure; F-spreading; Group-measure; Group-spreading; e Tempo de Execução. Através de análise experimental, avaliou-se a eficácia das técnicas propostas utilizando uma variedade de versões mutantes de quatro projetos *open sources*. Os resultados indicam que não houve perda estatística significante na aplicação da melhoria e, na antecipação de CTs em cenários de múltiplas mudanças, em média, foi superior.

Palavras-Chave. Teste de Regressão, Caso de Teste, Técnicas de Priorização, *Changed Blocks*, *PriorJ*.

Abstract

The investiment on software quality has grown. To ensure acceptable quality in a product, one needs to determine its testability, a property that indicates the feasibilityand accuracy in test results evaluation. Regression testing is an expensive technique to detect faults introduced in a previously tested code. In order to increase its cost-effectiveness, test case prioritization techniques may be used. One of the proeminent techniques is based on changes, called Changed Blocks. According to previous studies, this technique presents limitations, such as: i. Test Cases (TCs) with significant impact on the final result end up in undesired positions in the queue; ii. TCs with same number of covered changes are randomly ordered, without following a specific rule; iii. TCs revealing undetected changes, with low coverage are disadvantaged. This work proposes techniques to improve Changed Blocks by grouping multiple changes in a version of the system under test, with no losses in technical benefits. Several metrics were used as follow: i. APFD ; ii. F-measure ; iii. F-spreading ; iv. Group-measure ; v. Group-spreading ; and vi. Execution Time . We carried out an experimental study to evaluate the efficacy of the proposed techniques using a variety of mutant versions of four open sources Java projects. The results indicate the proposed techniques performed better in the contexts they intend to improve, while presenting no statistically significant loss in contexts common to the original Change Blocks technique.

Keywords. Regression Testing, Test Case, Prioritization Techniques, *Changed Blocks*, *PriorJ*.

Agradecimentos

A Deus, por proporcionar a realização deste trabalho.

Aos meus pais Rosângela, Rosemberg. Aos meus avós, especiais em minha vida. Pela força, dedicação e pelo amor eterno. Por me darem as bases para que meus esforços fossem transformados em sucesso.

A minha querida irmã Émilly, meu cunhado Wellington e meu sobrinho Eduardo por estarem sempre presentes alegrando minha vida em todos os momentos.

Ao meu tio Reginaldo por toda força dada.

À toda minha família.

Aos meus orientadores, Prof. Tiago Lima Massoni e Prof.ªPatrícia Duarte de Lima Machado, pela paciência, oportunidade e conhecimento transmitidos ao longo desse trabalho. O apoio de vocês foi substancial para que eu chegasse até aqui.

A minha namorada Laís Patrícia por todo amor, ajuda, incentivo, dedicação e paciência durante esses últimos 2 anos. E aos meus sogros Edvânio e Cláudia que me aceitaram da melhor forma possível em Campina Grande e na família.

A todos os meus amigos por me apoiarem na luta de estudar longe de casa e nunca se afastaram por conta de tempo e distância, em especial Liniker Weyne, Talles Figueiredo, Vitor Nicodemos e Ramon Gomes.

Ao pessoal do Embedded, em especial ao pessoal da Compal, pela ajuda e conhecimentos transmitidos. Aos meus colegas do mestrado pela amizade e momentos de alegria.

A todos que fazem parte da Universidade Federal de Campina Grande.

À Capes pelo apoio financeiro.

Conteúdo

Lista de Símbolos

APFD - *Average of the Percentage of faults detected*

BMAT - *Binary Matching Tool*

CB - *Changed Blocks*

EA - *Estratégia Additional*

ET - *Estratégia Total*

KLoC - *Kilo Lines of Code*

SIR - *Software-artifact Infrastructure Repository*

SUT - *System Under Test*

AMC - *Additional Method Coverage*

ASC - *Additional Statement Coverage*

TDD - *Test-Driven Development*

RND - *Random*

PTR - *Priorização de Testes de Regressão*

TMC - *Total Method Coverage*

TSC - *Total Statement Coverage*

WCS-A - *Additional Weighted Changed Statement*

WCS-T - *Total Weighted Changed Statement*

V&V - *Verificação e Validação*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

A indústria de *software* precisa dar ênfase a qualidade. Segundo Pressman [28] "a atividade de teste software é um elemento crítico na garantia de qualidade de *software* e representa a última revisão de especificação, projeto e codificação". No contexto de testes, como instrumento de verificação de qualidade, uma técnica bastante utilizada é o teste de regressão.

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 [29]. Kanellopoulos et. al. [23] mostra que existem empresas que investem de 50% a 70% de seus recursos disponíveis nas fases de testes e qualquer iniciativa de diminuir os defeitos é extremamente bem-vinda. Também é fato que, quanto mais tardiamente um defeito é encontrado em um sistema sendo desenvolvido ou testado, maior é o custo de sua correção, de acordo com Myers & Sandler [27]; e que de nada adianta entregar novas e inovadoras funcionalidades aos clientes, se estas alterações afetam a qualidade do sistema já utilizado. Por estes motivos, 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.

De acordo com Elbaum et. al. [10] "teste de regressão é um processo caro usado para validar modificações de *software* e na detecção de novos defeitos introduzidos em um código testado anteriormente". Essa técnica é utilizada a cada adição ou modificação de funcionalidades. Ela tem por objetivo principal que todos os casos de testes (CTs) devem ser execu-

tados a fim de garantir que nenhuma funcionalidade tenha sido comprometida. Atualmente, o processo mais simples existente para garantir que nenhuma funcionalidade foi impactada com as mudanças realizadas é o Refazer Todos os Testes (do inglês, *Retest-all*). Esse processo consiste na execução de todos os testes de regressão existentes para a nova versão do sistema. Isso pode adicionar um custo alto para a manutenção da qualidade do sistema, já que Rothermel et. al. [36] afirma que existem registros de teste de regressão que precisam de sete semanas para serem executados por completo. Visivelmente, esse problema tende a se agravar.

1.1 Priorização de casos de teste

Existem várias metodologias 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. A redução de testes consiste na inserção de novos CTs ao grupo de teste de regressão e a consequente exclusão de CTs redundantes. A Seleção de testes de regressão consiste em selecionar um subgrupo do conjunto completo da suíte a fim de testar uma parcela do sistema. Tomando como foco a priorização de casos de teste, várias técnicas de priorização propostas tem o objetivo de ordenar CTs de acordo com determinados critérios, de modo que os mais prioritários sejam executados primeiro durante a execução dos testes de regressão [10; 11; 17; 20; 33; 34; 35; 36; 40]. O objetivo é melhorar a taxa de detecção de defeitos. O motivo da priorização de casos de teste ser o alvo da presente pesquisa é que, das metodologias citadas, ela é a única que garante a não eliminação de nenhum CT do conjunto gerado. Logo, não há redução da capacidade de revelar defeitos do conjunto de CTs [19; 22], pois as metodologias de redução de grupos de testes oferecem uma desvantagem: a potencial possibilidade da remoção permanente de CTs; e isso pode afetar a capacidade de detecção de defeitos de uma suíte de testes.

As técnicas de priorização podem assumir diferentes categorias, como: Técnicas Baseadas em Cobertura (TBC) [35], Técnicas Baseadas em Distribuição (TBD) [25] e Técnicas Baseadas em Mudanças (TBM) [40]. As TBCs baseiam-se na execução de uma suíte de testes sobre a estrutura completa de um sistema; a priorização se dá dos CTs mais abrangentes (que cobrem mais elementos diferentes de software), para os menos abrangentes (que

cobrem menos elementos de software). Segundo Elbaum et. al. [10], a principal vantagem das técnicas baseadas em cobertura é que elas dependem exclusivamente dos dados recolhidos sobre a versão original de um programa (antes da alteração) em suas priorizações. De acordo com Leon & Podgurski [25], apesar de apresentarem um resultado similar, as TBCs são inferiores as TBDs para sistemas maiores.

Já as TBDs são técnicas que se baseiam no uso de uma função que, para cada par de versões, resulta em um número real representando o grau de dissimilaridade entre eles. A partir desse número é possível medir a similaridade entre as definições dos CTs, podendo construir agrupamentos de CTs relativamente similares capazes de indicar redundâncias para o cenário em questão. As vantagens dessa abordagem são a redução de custo e tempo empregado com as atividades de teste, aumento na qualidade dos testes gerados, entre outros [41]. De acordo com Utting & Legeard [41], Teste Baseado em Modelo exige maior experiência do testador, pois um modelo de boa qualidade necessita que ele conheça bem o sistema e o modele de forma adequada, além de acertar a granularidade desejada do modelo.

Por fim, as Técnicas Baseadas em Modelos também baseiam-se na execução de uma suíte de testes sobre a estrutura completa do sistema. A diferença das TBC é apenas que as mudanças entre versões do sistema são levadas em consideração, ao invés da cobertura geral. Essas técnicas apresentam uma vantagem em relação às demais, pois elas consideram CTs que executam o bloco de código modificado. A principal desvantagem é que para cada versão de código é necessário a geração de uma priorização específica.

Segundo Rothermel et al. [36], as técnicas de priorização, além de assumir quaisquer das três categorias supracitadas, podem ser classificadas em dois tipos distintos: 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 que foi gerada, pois ela se baseia em duas versões do sistema: uma versão base e a outra modificada. As técnicas específicas de versão devem se basear em algum critério entre as versões do código a fim de realizar a ordenação da suíte.

Existem ainda duas estratégias de priorização de casos de teste que podem seguidas por quaisquer técnicas, são elas: *Total* e *Additional* [36]. A estratégia *Total* tem como principal característica priorizar CTs a partir da contagem do número total de linhas de código que

cada CT possui em sua lista de cobertura. Assim, torna-se possível a reordenação da suíte de testes de forma decrescente. A estratégia *Additional* tem como objetivo principal selecionar iterativamente um CT que possui a maior cobertura de código-fonte e atualizar os dados dos CTs não analisados ainda. Essa atualização consiste em marcar as linhas de código já cobertas em toda a suíte, o que evita que partes do código-fonte já analisadas por CTs anteriores apresentem maior prioridade perante CTs com linhas de código inéditas. Este passo se repete até que a suíte de testes seja totalmente analisada. Caso vários CTs possuírem um número igual de linhas de código cobertas inéditas, a ordenação será realizada de forma aleatória.

Nesse contexto, Srivastava & Thiagarajan [40] afirmam que "novos defeitos introduzidos recentemente no sistema tem a maior probabilidade de aparecer em mudanças recentes. Assim, uma estratégia efetiva é focar o esforço de testes nas partes do sistema afetadas pelas mudanças". Eles propuseram o *Echelon* [40], um sistema de priorização de teste de regressão baseado na cobertura de mudanças entre versões de um mesmo projeto. Essa ferramenta contém uma técnica de priorização que apresenta o seguinte padrão na ordenação: no topo da lista CTs mais genéricos (que mais cobrem mudanças), e no final os mais específicos (que menos cobrem mudanças). O *Echelon* se utiliza de várias ferramentas exclusivas da *Microsoft*, o que impede a contribuição de pesquisadores externos. Alves et. al. [32] implementaram uma ferramenta *open source* chamada de *PriorJ*, que dá suporte a várias técnicas de priorização de casos de teste. Nela existe uma versão da técnica usada no *Echelon*, batizada de *Changed Blocks (CB)*, que é usada neste trabalho.

1.2 Changed Blocks

Esta técnica é baseada no trabalho realizado por Srivastava and Thiagarajan em [40] e batizada de *Changed Blocks (CB)*, por conta das mudanças executadas em sua implementação. Em tese, essa é uma técnica de priorização de casos de teste que leva em consideração as mudanças realizadas de uma versão para outra do código.

A CB resulta em uma lista de casos de teste priorizados. Segundo Srivastava and Thiagarajan [40], essa lista se inicia com a menor quantidade de casos de teste, proveniente da entrada, que cobrem a maior quantidade de partes afetadas do programa. A Figura 1.1 mos-

tra um fluxograma representando o funcionamento da técnica, que recebe como entrada duas versões de código-fonte, informações de cobertura da versão antiga de código e uma lista de casos de teste para priorizar. Primeiro, compara-se as duas versões do código que resulta nas mudanças existentes. Segundo há uma análise de cobertura reunindo as informações de cobertura da versão antiga com as mudanças realizadas. Por fim, é realizada a priorização dos casos de teste.

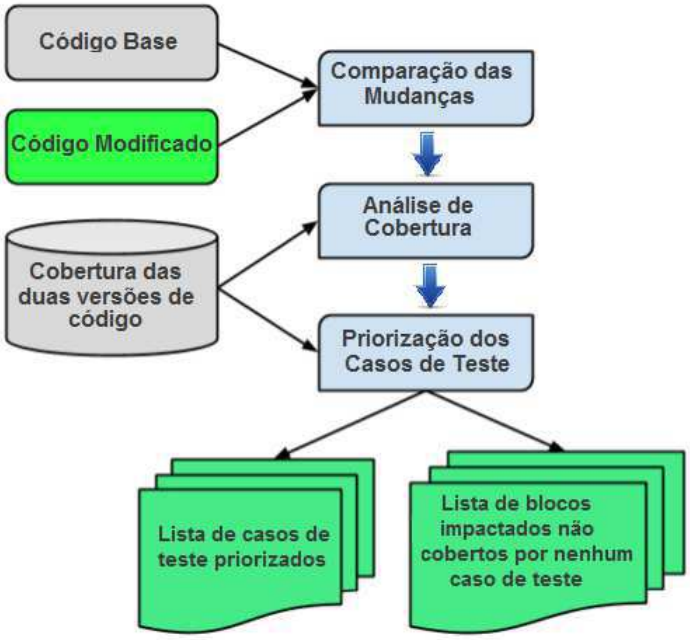

Figura 1.1: Funcionamento da técnica *Changed Blocks*.

1.3 Exemplo Motivante

Imaginemos um cenário onde mudanças múltiplas foram realizadas em um sistema qualquer. Os Códigos Fonte 1.1 e 1.2 representam as mudanças inseridas no código hipotético, onde a condição de um *statement if-else* é modificada na linha 3.

```
1 class OperacoesMatematicas {
2     static int menor(final int a, final int b) {
3         if (a < b) { return a; }
4         else { return b; }
```

```
5       }
6       // ...
7   }
```

Código Fonte 1.1: SUT Hipotético - versão base e modificada.

```
1   class OperacoesMatematicas {
2       static int menor(final int a, final int b) {
3           /* Mudança no Operador Lógico */
4           if (a > b) { return a; }
5           else { return b; }
6       }
7       // ...
8   }
```

Código Fonte 1.2: SUT Hipotético versão modificada.

Assumindo que um total de oito mudanças foram inseridas no código (M01 - M08) e que a suíte de testes para esse sistema contém cinco casos de teste (CT01 - CT05), a Figura 1.2 informa as mudanças cobertas por cada CT da suíte.

	CT01	CT02	CT03	CT04	CT05
M01		X	X		
M02		X	X	X	
M03		X	X	X	
M04	X		X	X	
M05	X				
M06	X				
M07					X
M08					X

Figura 1.2: Cobertura de cada caso de teste da suíte.

Como apresentado na Seção 1.2, a técnica Changed Blocks realiza a priorização dos casos de teste baseando-se na quantidade de mudanças cobertas por cada CT da suíte. Como observa-se na Figura 1.3, o CT03 cobre 4 mudanças, os CTs 01, 02 e 04 cobrem 3 mudanças e o CT 05 cobre 2 mudanças. A CB realiza a priorização através da quantidade de mudanças

cobertas pelo CT. Dessa forma, a Figura 1.1 mostra a priorização dos CTs com base nesta técnica. O CT03 assume a primeira posição da lista por cobrir 4 mudanças; os CTs 01, 02 e 04 são priorizados de forma aleatória entre eles, disputando as posições 2, 3 e 4 da lista ordenada, pois eles cobrem a mesma quantidade de mudanças; o CT05 assume a última posição da lista ordenada por cobrir o menor número de mudanças, apenas duas.

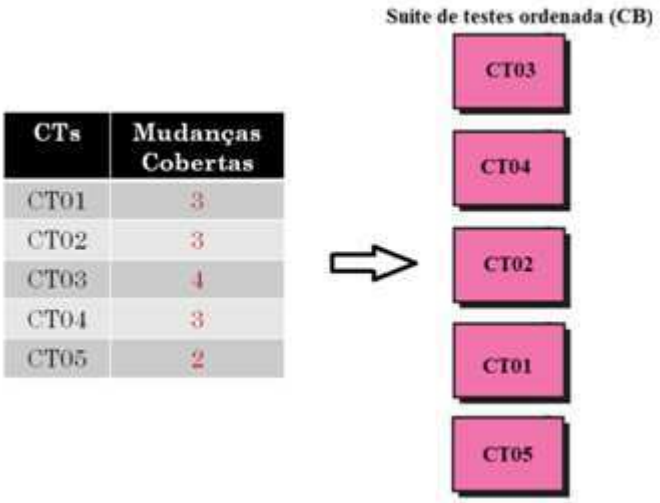

Figura 1.3: Suíte de testes ordenada pela técnica Changed Blocks.

Através do funcionamento da técnica, percebe-se que os resultados da Changed Blocks não obtém a melhor cobertura de defeitos no topo da lista ordenada. Na suíte ordenada da Figura 1.1, visualiza-se que os casos de teste que ocupam as posições subseqüentes à primeira cobrem maior número de mudanças, porém estas são instruções já cobertas pelo CT03 que ocupa a primeira posição. Assim, seria mais interessante a priorização dos CTs 01 e 05, que cobrem as mudanças inéditas (M05 - M08), como podemos observar na Tabela 1.2.

1.4 Problema

A técnica *Changed Blocks* pode não apresentar bons resultados para contextos em que várias mudanças são realizadas simultaneamente no código. Levando em consideração ambientes de desenvolvimento real, em que cada *commit* (envio do código para o repositório) produz uma nova versão no repositório e pode conter inúmeras modificações no código, é perceptível a utilidade dessa técnica, já que ela necessita de duas versões para gerar a priorização

	CT01	CT02	CT03	CT04	CT05
M01		X	X		
M02		X	X	X	
M03		X	X	X	
M04	X		X	X	
M05	X				
M06	X				
M07					X
M08					X

Figura 1.4: Casos de teste que cobrem mudanças inéditas.

baseadas nas mudanças realizadas.

Apesar de se encaixar no cenário supracitado, a *CB* apresenta uma grande limitação: por realizar a priorização baseada na quantidade de mudanças cobertas pelo caso de teste, CTs menos abrangentes que incluem mudanças inéditas em sua cobertura podem ser negligenciados na priorização. Levando em conta os resultados da técnica *Changed Blocks* e as considerações apresentadas acima, o problema pode se resumir a: *os resultados da técnica de priorização de testes Changed Blocks pode não obter a melhor cobertura de mudanças no topo da lista ordenada, com mudanças múltiplas agregadas em uma mesma evolução (commit)*.

1.5 Objetivos

O objetivo é implementar duas extensões da técnica *Changed Blocks*, através da técnica *Weighted Changed Statement (WCS)* proposta neste trabalho para aprimorar o desempenho em contextos de mudanças múltiplas, a fim de aumentar o rastreamento de defeitos ocasionadas por múltiplas modificações no código. A ideia é que CTs que cubram mudanças inéditas tenham prioridade sobre CTs que cubram mudanças previamente cobertas por casos já priorizados. A consequência é o aumento da percentagem de mudanças (possíveis defeitos) cobertas pelo topo da lista priorizada.

Nesse trabalho vamos propor, implementar e avaliar duas extensões da técnica de priorização de casos de testes *Changed Blocks*: uma seguindo a estratégia de priorização *Total*

e outra *Additional*; a fim de comparar a custo-efetividade entre a técnica original e as duas propostas. O foco principal são as suítes de teste implementadas na linguagem de programação *Java*. Nesse contexto será aprimorada também a ferramenta *PriorJ* [32], que provê um ambiente de execução estável para as técnicas utilizadas nesse trabalho.

1.5.1 Objetivos Específicos

Neste trabalho de mestrado, pretende-se atingir os seguintes objetivos específicos:

- Realizar uma pesquisa científica sobre a priorização de CTs em um contexto de mudanças.
- Atualizar funcionalidades necessárias da ferramenta *PriorJ* [32] para a obtenção da melhor execução da priorização de CTs.
- Implementar duas extensões da técnica *Changed Blocks*.
- Realizar um estudo experimental para avaliar as técnicas de priorização propostas.

1.6 Relevância

Como foi citado anteriormente, a realização da regressão dos casos de teste é um processo caro para sistemas de larga escala que se utilizam de um bom processo de verificação. Geralmente, a quantidade de CTs é diretamente proporcional ao tamanho do sistema, na medida em que um cresce, os CTs também. Nesse contexto, torna-se perceptível a necessidade de uma extensão na detecção de defeitos na técnica *CB*. Pois, considerando um projeto que utiliza metodologias ágeis e integração contínua, muitas modificações ocorrem durante o desenvolvimento do produto, que podem não ser testada de forma completa por limitações de tempo.

Essa proposta tem como objetivo contribuir para tratar deste problema de forma que nossa técnica possa rastrear as mudanças realizadas no código e priorizar os CTs impactados por elas, a fim de capturar defeitos o quanto antes sem a necessidade de executar todos os CTs que fazem parte do sistema. A presença de inúmeras mudanças são comuns em sistemas implementados por um time, onde vários *commits* acontecem diariamente.

Um dos experimentos realizados por Ren et. al. [30] mostra uma porcentagem média de CTs que detectam defeitos provenientes das mudanças realizadas. Seus estudos empíricos confirmaram que após a edição do programa, 52% dos testes foram afetados pela mudança, direta ou indiretamente. Esse número mostra a grandeza do impacto com relação à porcentagem da mudança, já que tal impacto foi consequência da mudança de apenas 3.95% de todo o código.

1.7 Estrutura da Dissertação

O restante desta dissertação está estruturado da seguinte forma: no **Capítulo 2** é apresentada a Fundamentação teórica sobre os temas abordados neste trabalho; no **Capítulo 3** está a Descrição da abordagem proposta; no **Capítulo 4** a Apresentação e Discução dos resultados obtidos; e no **Capítulo 5** encontram-se as Considerações finais, conclusões e sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados os conceitos básicos necessários para o entendimento da solução proposta nesta dissertação. Primeiramente realiza-se uma introdução dos conceitos básicos de teste de software (Seção 2.1). Em seguida apresenta-se os conceitos de evolução de *software*, priorização e seleção de testes. E, por fim, explica-se algumas das técnicas de priorização utilizadas como base nos estudos e experimentos que serão realizados (Seção 2.2.1).

2.1 Teste de *Software*

Segundo Weiszflog [42], 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.

De acordo com Sommerville [39], existem várias atividades importantes em um processo de desenvolvimento *software* e uma das etapas mais importantes é a *Verificação e Validação de Software (V&V)*. Por definição, a verificação analisa se as funcionalidades do produto de *software* foram implementadas de forma correta e a validação analisa se o que foi implementado satisfaz as necessidades do cliente. Sommerville [39] afirma que "o teste se propõe a mostrar que um programa faz o que ele se propunha a fazer e encontra defeitos antes do sistema entrar em uso. Para torná-lo possível, o teste se utiliza de dados artificiais para simular

uma ação real". Segundo Pressman [28] "a atividade de teste software é um elemento crítico na garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação". Nesse contexto, Dijkstra et al. [8] conseguiram chegar a conclusão de que os testes só podem mostrar a presença de defeitos, mas não a sua ausência.

2.1.1 Casos de Teste

De acordo com Craig e Jaskiel [6], um Caso de Teste (CT) é um artefato que descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado. A Figura 2.1 apresenta uma representação dos casos de teste que são responsáveis por verificar a corretude de determinado módulo. O teste de unidade tem por objetivo a verificação da menor unidade do *software*, assim como mostra a representação abaixo, o módulo referencia a menor unidade do System Under Test (SUT).

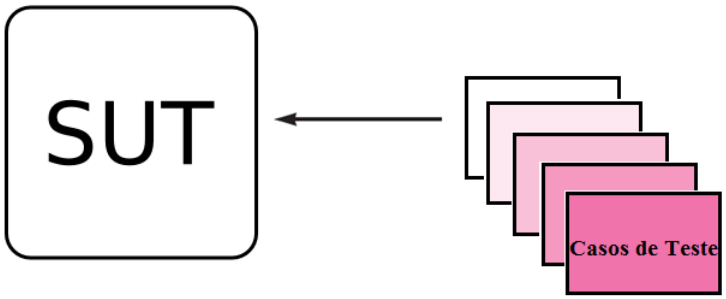

Figura 2.1: Representação de Testes de Unidade.

O caso de teste é uma descrição do comportamento de um determinado requisito e é sempre acompanhado por um procedimento de teste, que representa os passos que devem ser seguidos a fim de que o teste seja executado da forma correta. Uma suíte de testes é um conjunto de casos de teste referentes a um módulo, que pode incluir mais de um requisito. A unidade básica de toda a atividade de teste de software é o caso de teste [28]. Um caso de teste é composto pelos seguintes requisitos que envolvem o procedimento de teste:

Entradas:

Tabela 2.1: Exemplo de um caso de teste.

ID	CT001
Caso de Teste	Efetuar login no sistema.
Funcionalidade	Login
Pré-Condição:	1. A aplicação deve estar executando. 2. Possuir usuário válido para efetuar o login.
Passos:	1. Acessar o sistema. 2. Inserir usuário e senha. 3. Clicar em "Login".
Resultado esperado:	1. Login do usuário efetuado com sucesso.
Pós-condições:	

- *Pré-condições*: condições necessárias para o início da execução do caso de teste;
- *Passos*: ações que devem ser desempenhadas no sistema sob teste.

Saídas:

- *Resultados esperados*: resultados que devem ser retornados pelo *SUT*, desde que os passos (entrada) sejam executados corretamente;
- *Pós-condição*: condições que devem ser verificadas e satisfeitas ao final da execução do caso de teste.

A Tabela 2.1 demonstra um exemplo de um caso de teste real verificando a ação de login de uma aplicação. A Figura 2.2 mostra a representação de um *SUT* sob a ação de uma suíte de testes que contém três casos de teste (CT001, CT002 e CT003).

O diagrama de classes apresentado na Figura 2.3 representa casos de testes automatizados reunidos por uma suite de testes. Esta é a representação de testes que podem ser implementados pelo framework *JUnit*. *JUnit* é um framework de código fonte aberto criado por Kent Beck e Erich Gamma em 1995 para criação de testes automatizados na linguagem de programação *Java* a fim de facilitar a implementação e execução de testes unitários, gratuita e orientada a objetos [4]. A classe *TesteLogin* e *TesteCamposDoLogin* apresentadas na Figura 2.3 representam um conjunto de testes unitários de um requisito em diagrama de classes UML [37], no caso, o *login* em alguma aplicação. Todos os métodos de teste são marcados pela anotação @Test [4]. A classe *SuiteDeTestes* representa a suite que reunirá

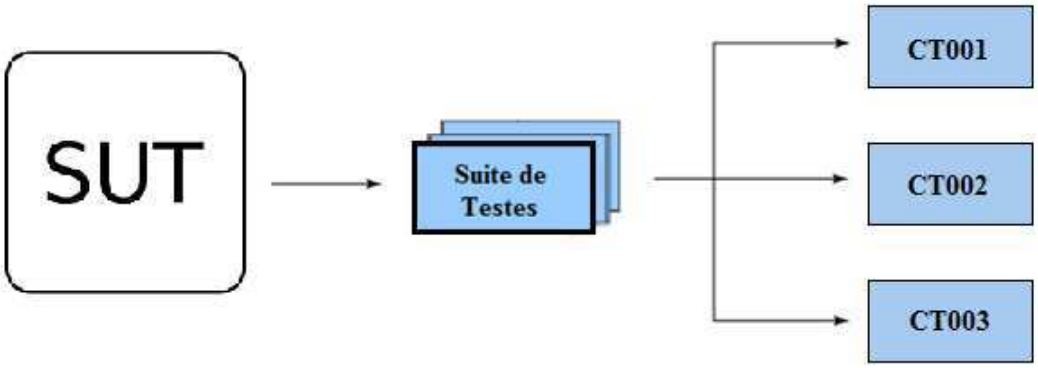

Figura 2.2: Representação de uma suite de testes com três casos de teste (CT001, CT002 e CT003).

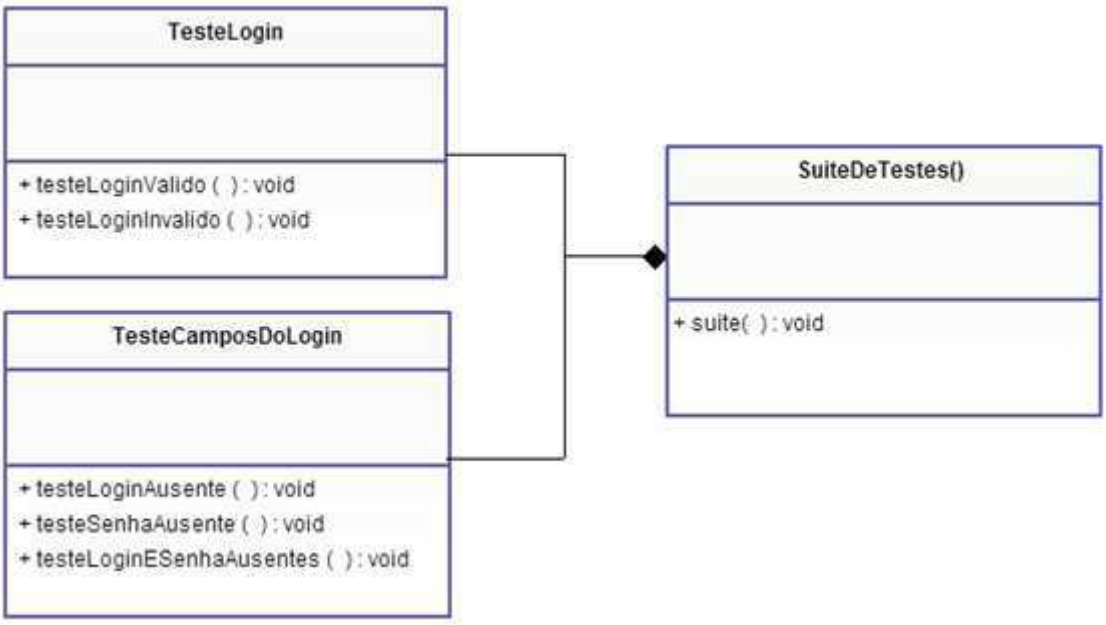

Figura 2.3: Demonstração de casos de teste e suite de teste.

essas duas classes de testes unitários em uma única execução.

```
1 public class TesteLogin {
2     SUT sut = new SUT();
3     @Before
4     public void preCondicoes() {
5         sut.abrirPaginaInicial();
6     }
```

```
7       @Test
8       public void testeLoginValido() {
9           String loginValido = "admin";
10          String senhaValida = "admin";
11          boolean loginResult = sut.fazerLogin(loginValido,
                senhaValida);
12          assertTrue("O Usuário admin deveria ter acesso com esta
                senha!", loginResult);
13      }
14      @Test
15      public void testeLoginInvalido() {
16          String loginValido = "admin";
17          String senhaValida = "senha";
18          boolean loginResult = sut.fazerLogin(loginValido,
                senhaValida);
19          assertTrue("O Usuário admin não deveria ter acesso com esta
                 senha!", loginResult);
20      }
21      @After
22      public void posCondicoes() {
23          sut.fecharSistema();
24      }
25  }
```

Código Fonte 2.1: Implementação da classe TesteLogin com a utilização do framework junit, apresentada na Figura 2.3.

O Código Fonte 2.1 mostra a implementação do grupo de testes *TesteLogin*, apresentado na Figura 2.3, realizada na linguagem *Java* e com a utilização do framework Junit. A ideia de automação de testes é utilizar um script ou uma ferramenta de software (no caso, uma classe Java) que facilite a execução dos testes. Isso acarretará em uma enorme economia de tempo, já que esse processo envolve a automação dos testes que seriam realizados manuais, tornando a sua execução consideravelmente mais rápida.

2.1.2 Tipos de Teste

Segundo Sommerville [39], durante o desenvolvimento, os testes podem ser divididos em três níveis de granularidade:

- *Teste de unidade*: nesse nível são testadas unidades individuais ou classes de objetos. O foco principal desse tipo de teste são as funcionalidades dos objetos ou métodos.
- *Teste de componente*: nesse nível várias unidades individuais são integradas para criar componentes compostos. Esses testes devem focar nas interfaces dos componenetes.
- *Teste de sistema*: nesse nível alguns ou todos os componentes do sistema são integrados e o sistema é testado como um todo. Esse tipo de teste deve focar nas interações dos componentes.

Os testes automáticos trazem grande vantagem de custo-benefício na execução, porém não estão isentos de problemas. O principal deles é a manutenção dos testes durante a evolução de *software*, pois nela ocorrem mudanças no código que podem acarretar em grandes impactos aos testes. Estas mudanças podem mudar funcionalidades fazendo com que o teste não funcione corretamente e necessite de uma atualização. Porém, em alguns casos o caso de teste pode até se tornar obsoleto. Um dos principais problemas existentes atualmente é como rastrear o impacto dessas modificações.

2.1.3 Evolução de Software

Um produto de software só tem um final definitivo em caso de cancelamento ou em caso de não satisfação das necessidades dos clientes. Assim, pode-se afirmar que um produto de software vive em constante evolução, seja para garantir a manutenibilidade do sistema, ou para modificar/inserir comportamentos ao mesmo. Com base em pesquisas informais na indústria, Erlikh [12] sugere que entre 85-90$\%$ dos custos de software das organizações são custos de evolução. Com isso, é perceptível a importância da evolução de software para as organizações, já que grande parte do orçamento é relacionada a ela.

Sommerville [39] afirma que existem três tipos de manutenção de software e apresenta a distribuição de esforço da manutenção de software no gráfico apresentado na Figura 2.5:

1. *Reparação de defeitos*: correção de erros (que relativamente não custa caro); correção da arquitetura (que custa caro, pois envolve mudanças de componentes); e correção de requisitos (custo muito alto).

2. *Adaptação do ambiente*: mudanças na infra-estrutura que acarretam em adaptações no sistema.

3. *Adição ou modificação de funcionalidade*: a verificação analisa se as funcionalidades do produto de software foram implementadas de forma correta e a validação analisa se o que foi implementado satisfaz as necessidades do cliente.

A Figura 2.4 mostra o processo de evolução de *software* e como a análise de impacto se encaixa no seu fluxo.

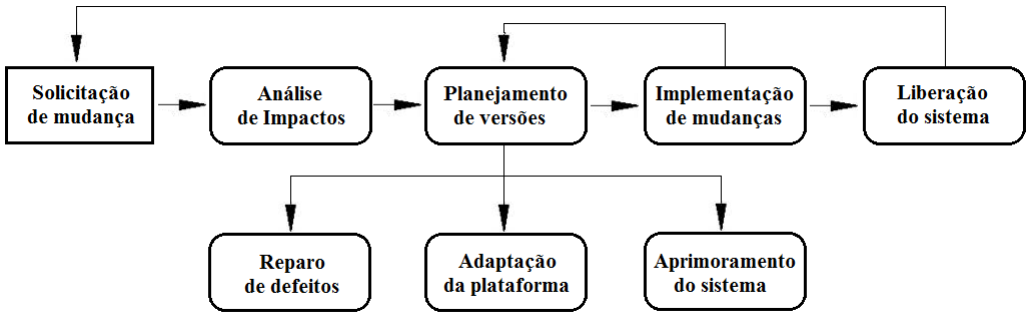

Figura 2.4: Processo de Evolução de *Software* (Baseado em Sommerville, 2011).

A manutenção de software necessita de grande esforço por conta dos impactos causados ao restante do código. Muitas vezes pequenas mudanças podem acarretar em grandes impactos cuja rastreabilidade demanda a utilização de grande parte dos recursos do projeto. Sommerville [39] afirma que 17% desses esforços são voltados para a reparação de defeitos, 18% para a adaptação do ambiente e, em maior escala, 65% para a adição ou modificação de funcionalidades. O gráfico com esses dados podem ser visualizados na Figura 2.5.

2.1.4 Teste 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. Em seu trabalho, Srivastava and Thiagarajan [40] afirmam que "Se

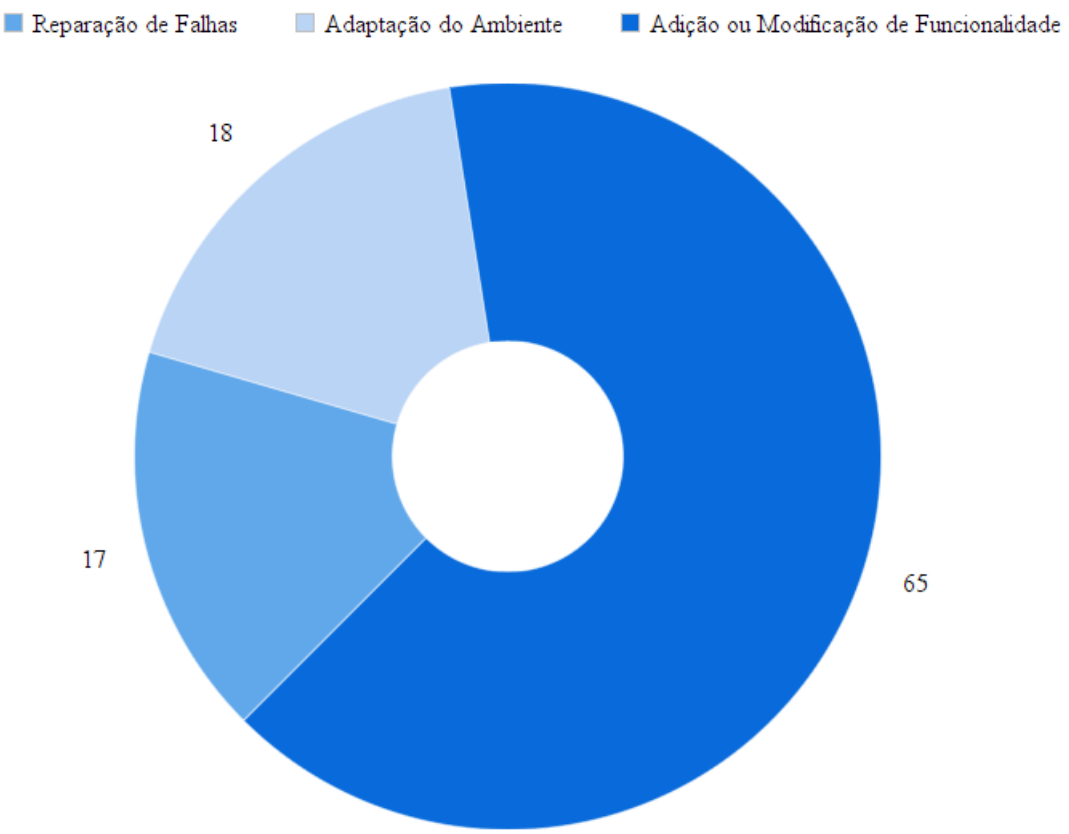

Figura 2.5: Distribuição de esforço da manutenção. (Baseado em Sommerville, 2011).

os desenvolvedores introduzem novos defeitos ao *software*, eles devem ser detectados tão cedo e com custo baixo quanto possível no ciclo de desenvolvimento". A fim de atingir tais objetivos é necessária a utilização das técnicas introduzidas a seguir, como: teste de regressão, seleção e priorização de casos de teste.

Segundo Elbaum et al. [10] "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.

Existem várias maneiras de redução de custo, uma delas é a abordagem de técnicas de seleção de teste de regressão. Ela tem o foco principal na redução dos custos do teste de regressão a partir da seleção de um subconjunto da suíte de testes que foi executada durante

o desenvolvimento a fim de testar o sistema modificado [33].

O mercado atual é marcado pelo imediatismo - o cliente sempre necessita dos resultados o mais rápido possível. Estes pedidos de mudança, muitas vezes, ocorrem em um período muito próximo do prazo de entrega acordado, fazendo com que a equipe se depare com a restrição tripla existente entre tempo, custo e escopo. É de conhecimento comum que essas restrições têm impacto direto com a qualidade do produto.

Uma solução para esse problema é encontrar uma forma de aumentar a taxa de detecção de defeitos, pois um aumento nessa taxa ocasiona em uma maior rapidez na apresentação do resultado de defeitos encontradas no *SUT*. Uma das principais formas de reduzir essa taxa é a utilização de técnicas de priorização de casos de teste, que Rothermel et al. [36] define como sendo "uma ordenação nos casos de teste de uma forma que o objetivo de detectar defeitos o quanto antes se torne possível".

2.2 Priorização de Testes

O objetivo da priorização de testes é posicionar os CTs que satisfazem um determinado requisito de interesse mais próximo do início da ordem de execução, a fim de melhorar a taxa de detecção de defeitos de um sistema.

2.2.1 Técnicas de Priorização de Casos de Teste

A definição do problema de priorização de casos de teste é a seguinte [32]:
Dado:

- T, uma suíte de testes;
- PT, o conjunto de permutações de T;
- $f{:}PT \rightarrow \mathbb{R}$, a função que mapeia a permutação PT em números Reais.

Problema:

- Encontrar: $T' \in PT$ tal que $\forall(T''$ e $PT)$ com $(T'' \neq T')$ e $[f(T') \geq f(T'')]$.

Tabela 2.2: Técnicas de priorização de casos de teste organizadas por grupo.

Grupos	**Legenda**	**Nomenclatura**
Grupo 01	T1	*Random*
Grupo 02	T2	*Total Statement Coverage Prioritization*
	T3	*Additional Statement Coverage Prioritization*
	T4	*Changed Blocks*
Grupo 03	T5	*Total Method Coverage Prioritization*
	T6	*Addition Method Coverage Prioritization*

Neste âmbito, *PT* representa o conjunto de todas as possibilidades de ordens de priorização de T e f é a função que, aplicada em qualquer ordem, tem como resultado um único valor de ordenação.

Rothermel et al. [36] faz 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.

A instrumentação de *software* tem o objetivo de monitorar e mensurar o nível de desempenho do sistema através do rastreamento de informações. O rastreamento envolve a adição de código extra na aplicação. As técnicas de priorização são classificadas em três grupos baseados em sua granularidade. A Tabela 2.2 apresenta as técnicas organizadas por grupo: o primeiro grupo é o de controle, contendo apenas uma técnica que será usada para comparações futuras; o segundo grupo refere-se às técnicas de granularidade fina, ou seja, técnicas que se utilizam da instrumentação, análise da cobertura e priorização em nível de linhas de código; o terceiro grupo refere-se às técnicas de granularidade grossa, ou seja, que consideram partes maiores do sistema na instrumentação, análise de cobertura e priorização.

Grupo 01: Técnica de controle

T1: Ordenação *Random*. A "técnica"de priorização de ordenação aleatória terá a utilidade de exercer o papel de grupo de controle experimental nesse trabalho. Ela pode ser a ordem

natural de execução da suíte de testes ou a aplicação de uma ordenação aleatória no conjunto de testes.

Grupo 02: Técnicas de granularidade fina

T2: *Total Statement Coverage Prioritization*. É possível mensurar o nível de cobertura de instruções de código-fonte (statements) a partir dos casos de testes automatizados com a utilização da instrumentação de código. Com os resultados é possível ordenar os casos de teste através da quantidade de linhas de código-fonte coberta por cada caso de teste. Sendo assim, os casos que cobrem a maior quantidade de instruções ocuparão os primeiros lugares na fila de ordenação que resulta dessa técnica.

Segundo Elbaum et al. [10], "dada uma suite de testes com *m* casos de teste e um programa com n linhas de código, a técnica apresentada requer um custo de tempo de O(mn + mlog m). Normalmente, n é maior que m, fazendo com que o tempo fique equivalente a O(mn)".

É possível observar na Figura 2.2 um *SUT* sob uma bateria de testes com três casos de testes automatizados: CT001, CT002 e CT003. Para fins de exemplificação, será considerado que o *SUT* contém vinte linhas de código-fonte e que os casos de teste CT001, CT002 e CT003 cobrem, respectivamente, quatro, dez e seis linhas de código-fonte. A Figura 2.6 mostra a ordem natural de execução e a ordem de execução com a aplicação da técnica T2:

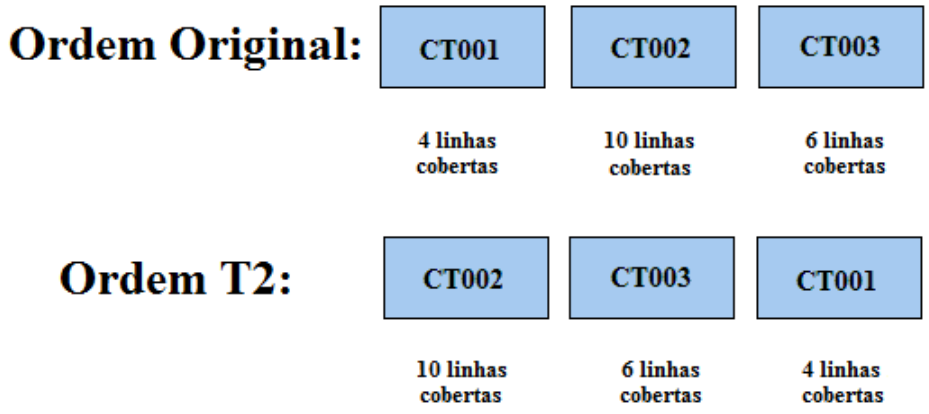

Figura 2.6: Ordem Natural de execução e ordem com a aplicação da técnica T2.

T3: *Additional Statement Coverage Prioritization*. Esta técnica é uma evolução da T2.

O que a torna melhor (em contextos específicos) é a sua necessidade de utilização da cobertura de código alcançada até o momento, a fim de escolher casos de teste que apresentem cobertura em partes do código-fonte ainda desconhecidas pelos casos de teste executados anteriormente.

Para tornar isso possível, a técnica seleciona os casos de teste com os maiores valores de cobertura e os que não foram selecionados são marcados como "Não cobertos ainda". Feito isso, ocorre uma iteração sobre esses casos de teste até que a cobertura de um atinja linhas de código-fonte que não foram cobertas ainda. O restante dos casos será marcado como "Não cobertos"e esse procedimento se repete até que não sobre nenhum caso de teste. Segundo Elbaum et al. [10], "dada uma suite de testes com m casos de teste e um programa com n linhas de código, respectivamente, a técnica apresentada requer um custo de tempo de $O(m^2n)$".

Considere um *SUT* com vinte linhas de código, onde: o CT001 cobre linhas 1 a 10 do *SUT*; o CT002 cobre da linha 1 à 7 do *SUT*; e o CT003 cobre da linha 11 à 14 do *SUT*. Com essas pré-condições, apresenta-se na Figura 2.7 o resultado simulado da aplicação dessa técnica. Observa-se que o CT001 é priorizado por ter uma cobertura com o valor mais alto, no caso dez linhas de código. Percebe-se também que o CT003 (cobertura = 4), apesar de ter uma cobertura menor que a do CT002 (cobertura = 7), teve prioridade. Isso ocorre pelo fato de que essa técnica se utiliza do histórico de cobertura e dá prioridade a casos de teste que cobrem partes do código que ainda não foram executadas.

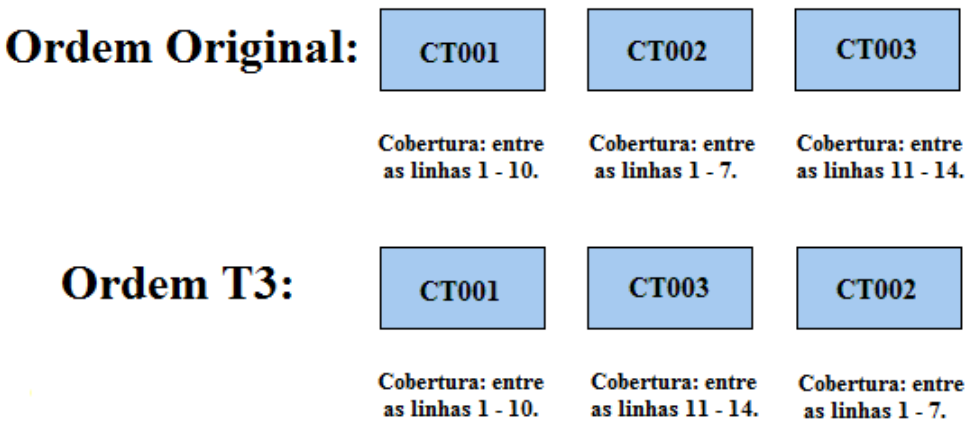

Figura 2.7: Ordem Natural de execução e ordem com a aplicação da técnica T3.

T4: *Changed Blocks*. Como foi apresentado na Introdução, ela é uma técnica de prio-

rização de casos de teste que leva em consideração as mudanças realizadas de uma versão para outra do código, logo, é perceptível que ela é classificada como técnica de priorização de versão específica.

Imaginemos um cenário onde foram realizadas mudanças no código e que apenas o caso de teste CT001 contém chamadas de métodos que executam a parte do código modificada,. Em contrapartida, os casos CT002 e CT003 não executam partes do código modificadas. A Figura 2.8 mostra a representação do resultado da aplicação da técnica *Changed Blocks*, onde CT001 é priorizado por executar partes do código que foram modificadas, porém, os outros casos de teste podem receber quaisquer das outras posições na fila de priorização, já que eles serão priorizados de forma aleatória.

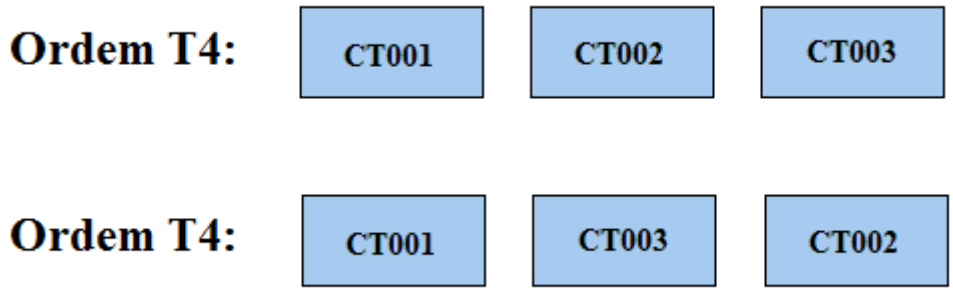

Figura 2.8: Possíveis ordens de priorização da técnica *Changed Blocks*.

Grupo 03: Técnicas de maior granularidade

T5: *Total Method Coverage Prioritization*. Análoga à técnica total statement coverage prioritization, porém, ao invés de varrer a cobertura no nível de linhas de código-fonte, opera no nível de métodos. A técnica também tem o seu pior caso análogo, O(mn + mlog m) para uma suite de testes contendo m casos de teste e um programa contendo n métodos. Como o total method coverage prioritization se apresenta com uma granularidade grossa, ela promete ser menos custosa, pois não será necessário varrer todas as linhas de código-fonte, apenas as chamadas de métodos serão contabilizadas.

T6: *Additional Method Coverage Prioritization*. Esta técnica é análoga à T3, porém, ao invés de varrer a cobertura no nível de linhas de código-fonte, opera no nível de métodos. A técnica também tem o seu pior caso análogo, O(m^2n) para uma suite de testes com m casos

de teste e um programa com n métodos. É perceptível que a utilização apenas da análise de cobertura dos casos de teste ou mesmo inserindo a comparação entre versões ainda não é suficiente para certos cenários. Pensando nisso, talvez uma rastreabilidade nos impactos causados por cada mudança possa ajudar de alguma forma na seleção e priorização de casos de teste.

2.3 Metodologias Ágeis

Sommerville [38] demonstra que a melhoria no processo de desenvolvimento de software aperfeiçoa a qualidade do produto final; e como destaca Côrtes [7] a "preocupação com a qualidade deixou de ser um diferencial competitivo e passou a ser um pré-requisito básico para participação no mercado". Atualmente existem dois tipos de metodologias, as tradicionais e as ágeis. A primeira valoriza uma quantidade excessiva de documentação, enquanto a segunda por um *software* funcionando com o mínimo de documentação possível.

A partir da década de 90 começaram a surgir novos métodos sugerindo uma abordagem de desenvolvimento ágil, em que os processos adotados tentam se adaptar às mudanças, apoiando a equipe de desenvolvimento em seu trabalho [13]. As metodologias ágeis surgiram como uma reação às metodologias tradicionais [15] e tiveram como principal motivação criar alternativas para o Modelo em Cascata. De acordo com Highsmith [18], os valores do Manifesto Ágil são: i. indivíduos e interações valem mais que processos e ferramentas; ii. um software funcionando vale mais que documentação extensa; iii. a colaboração do cliente vale mais que a negociação de contrato; e iv. responder a mudanças vale mais que seguir um plano.

Uma prática de bastante utilidade na implementação dos valores do Manifesto Ágil é a Integração Contínua (IC). Segundo Fowler [15], IC é uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um *build* automatizado (construção da aplicação e execução da suíte de testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente. É perceptível que as

técnicas específicas de versão, mais especificamente a *Changed Blocks*, tem o funcionamento que se encaixa perfeitamente na prática de IC, garantindo assim a qualidade de cada versão.

2.4 Considerações Finais

Neste capítulo apresentamos um breve resumo dos fundamentos básicos necessários relacionados com o nosso trabalho de mestrado. Acreditamos que os leitores serão capazes de compreender os próximos capítulos presentes neste documento, desde que tenham entendido os conceitos aqui apresentados. No próximo capítulo apresentamos o comportamento da ferramenta *PriorJ* e a implementação das técnicas propostas neste trabalho de mestrado.

Capítulo 3

Técnica de Priorização Proposta: Weighted Changed Statement

Esse capítulo apresenta a proposta de duas extensões da técnica de priorização de casos de teste *Changed Blocks (CB)*. Esta técnica foi proposta com o intuito de implementar uma melhoria que não resulte em impactos negativos no custo, que surgiu a partir da percepção da negligência com alguns casos de teste (CTs) apresentada no resultado da *CB*, em que CTs menos abrangentes afetados por mudanças inéditas ocupam posições desfavoráveis na lista priorizada, quando comparado a CTs mais abrangentes.

Nas seções a seguir, é apresentada toda a explicação necessária para o correto entendimento da ferramenta *PriorJ*, da técnica *CB*, das técnicas propostas, que foram nomeadas de *Weighted Changed Statements Total (WCS-T) e Weighted Changed Statements Additional (WCS-A)*, além de exemplos para facilitar o entendimento das técnicas.

3.1 Visão Geral da Abordagem

Como foi apresentado na Seção 2.2.1, Alves et. al. [1] implementaram uma versão da técnica idealizada em [40], dando o nome de *Changed Blocks*. Lembrando que, em um ambiente real, onde muitas mudanças são realizadas no código diariamente, mudanças inéditas que são executadas por CTs com pouca cobertura são facilmente desfavorecidas na priorização da técnica *CB*. Nosso objetivo é melhorar seu resultado, através da sua extensão com uma heurística adicional, com foco em contextos em que múltiplas mudanças sejam inseridas

em uma versão de código de um sistema qualquer. Isso resulta em CTs que cobrem várias mudanças sejam privilegiados e negligencia CTs que apresentam poucas mudanças em sua cobertura.

As técnicas propostas na Figura 3.1 aparecem cinco atividades, em que cada uma é responsável por gerar um artefato específico; cada atividade recebe um artefato na entrada e gera outro na saída que servirá de entrada para a próxima atividade. Nas seções a seguir são apresentadas as atividades de forma detalhada, com suas respectivas entradas e saídas. A Figura 3.1 oferece uma visão geral da abordagem proposta.

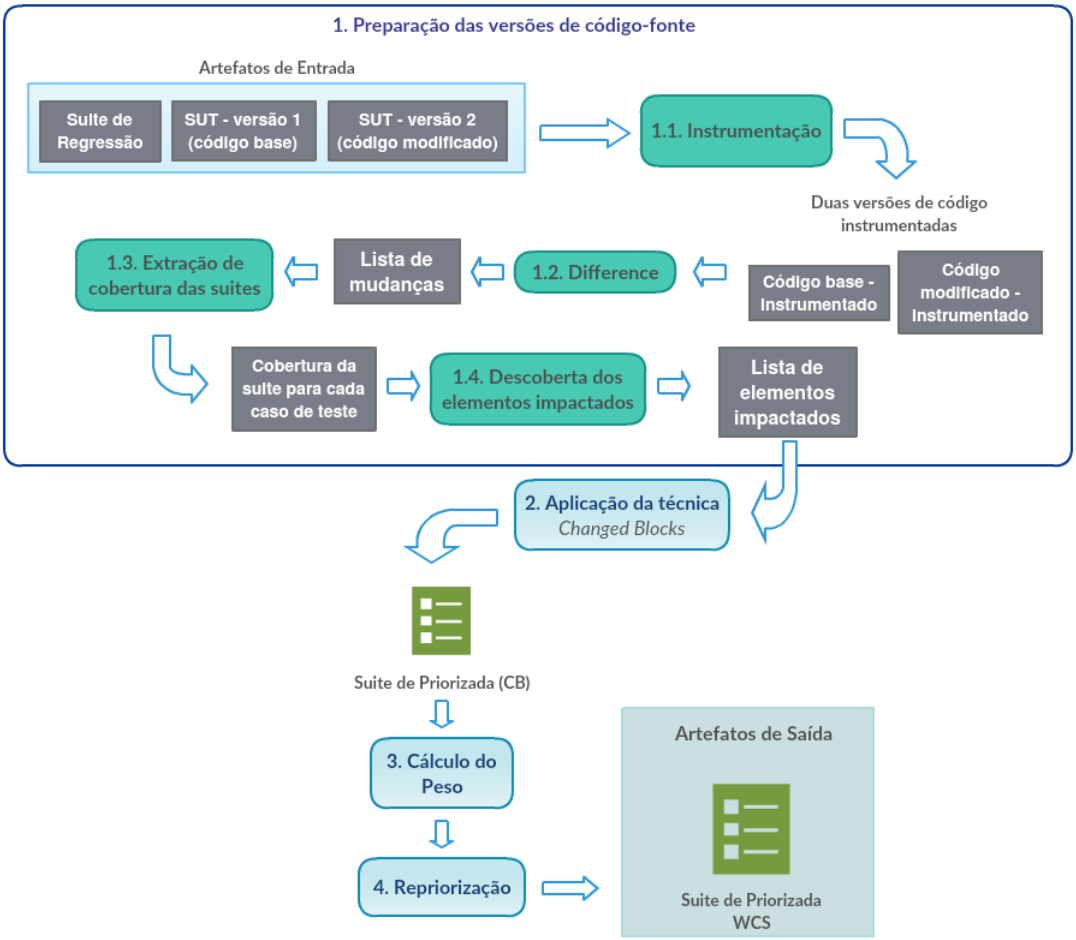

Figura 3.1: Fluxograma das atividades gerais para a aplicação das técnicas.

3.2 Atividade 1: Preparação das versões de código-fonte

Esta atividade tem como objetivo extrair informações dos artefatos de entrada a fim de gerar artefatos de saída necessários para a realização da priorização. Os requisitos de entrada para o correto funcionamento da ferramenta são: i. Duas versões de código de um programa qualquer. A primeira é a versão estável do sistema sob testes (do inglês, *System Under Test - SUT*). Ela é conhecida como versão base (ou versão antiga), que já deve ter passado por testes de regressão. A segunda versão é o código modificado pelos desenvolvedores, resultado de uma evolução qualquer no sistema (manutenção adaptativa, corretiva, evolutiva ou preventiva) e que possivelmente contém defeitos causados pela evolução. Novos testes podem ter sido incluídos, mas não farão parte da priorização, pois não fazem parte da regressão; e ii. uma suíte de testes unitários para o *SUT*. Esta atividade é subdivida em quatro, com os objetivos a seguir:

- Instrumentar o código-fonte;
- Realizar a comparação entre as duas versões;
- Extrair a cobertura da suíte de testes;
- Descobrir quais elementos foram impactados.

Atividade 1.1: Instrumentação

O objetivo desta atividade é processar cada classe *Java* recebida como entrada e adicionar linhas de código próprias da instrumentação, necessárias para o rastreamento de código, coletando os dados da cobertura para cada CT. Estas linhas de código adicionais são atribuições a uma variável chamada *watchPriorJApp* [32], que resulta em um código-fonte instrumentado e pronto para ser analisado na atividade de análise de cobertura.

```
1 private boolean isElementPresent(final String element) {
2     watchPriorJApp = watchPriorJApp;
3     return this.element.contains(element);
4 }
5
```

```
6 static boolean watchPriorJApp;
```

Código Fonte 3.1: Exemplo de instrumentação de código-fonte.

Atividade 1.2: Comparação entre Versões

Logo após a instrumentação, é realizada uma comparação entre as duas versões de código instrumentadas pela Atividade 1.1. Esse processo resulta em uma lista de modificações existentes entre as versões submetidas, composta por linhas de código (*statements*) que foram modificados, seguindo a estrutura: *CaminhoDoPacote.NomeDaClasse.AssinaturaDoMetodo.NumeroDaLinha*. Como é possível perceber no exemplo, os Códigos 3.2 e 3.3 mostram duas versões de código (base e modificada, respectivamente). Como mostra o Código 3.3, foi aplicada uma modificação do operador matemático na Linha 3. Dessa forma, o rastreamento dessa linha no *log* é: *br.org.splab.math.api.OperacoesMatematicas.soma(int, int).3*.

```
1 class OperacoesMatematicas {
2     static int soma(final int a, final int b) {
3         int soma = a + b;
4         return soma;
5     }
6     // ...
7 }
```

Código Fonte 3.2: SUT Hipotético versão base.

```
1 class OperacoesMatematicas {
2     static int soma(final int a, final int b) {
3         // mudança no operador matemático
4         int soma = a - b;
5         return soma;
6     }
7     // ...
8 }
```

Código Fonte 3.3: SUT Hipotético versão modificada.

Atividade 1.3: Extração de Cobertura das Suítes

Esta atividade é responsável por extrair de cobertura de código para cada CT, coletando informações necessárias automaticamente no momento em que o usuário executa da suíte de testes da versão instrumentada modificada do SUT. No ato da execução será criada uma lista de cobertura para cada CT dessa versão. É importante enfatizar que novos testes podem ter sido incluídos, mas não farão parte da priorização, pois não fazem parte da regressão. Ao fim, temos como saída a cobertura da versão modificada do código-fonte. Esta Atividade utiliza o paradigma de programação orientado a aspectos [24], mais especificamente, *AspectJ*[1]. Para extrair as informações de cobertura e rastrear as mudanças, a ferramenta utiliza as variáveis *watchPriorJApp* inseridas na atividade de Instrumentação[32].

Atividade 1.4: Descoberta dos Elementos Cobertos

Esta atividade é realizada pelo analisador de cobertura e utiliza a lista de mudanças geradas na Atividade 3.2; e o *log* de cobertura gerado no ato da execução dos CTs pela Atividade de Extração da cobertura. Assim, para cada CT, existe uma lista que contém todas as linhas de código que foram executadas de alguma forma por ele. O resultado dessa atividade é um *log*, que é preenchido com a interseção entre os dois artefatos de entrada desta atividade.

Para exemplificar, considere um SUT hipotético que serve de abstração para operações matemáticas e contém uma única classe com vinte métodos, sendo que dez deles contém uma única mudança cada. A suíte de testes utilizada nesse SUT contém vinte CTs (CT01, CT02,..., CT20) e a lista de cobertura de cada caso de teste contém as mais variadas mudanças em sua composição. Os Códigos 3.4 e 3.5 mostram de forma detalhada a versão base e a modificada. O código-fonte 3.6 apresenta parte da suíte de testes para os Códigos 3.4 e 3.5.

```
1 class OperacoesMatematicas {
```

[1] https://eclipse.org/aspectj/

```
2       static int soma(final int a, final int b) {
3           int soma = a + b;
4           return soma;
5       }
6       static int produto(final int a, final int b) {
7           int produto = a * b;
8           return produto;
9       }
10      static int divisao(final int a, final int b) {
11          int divisao = a / b;
12          return divisao;
13      }
14  }
```

Código Fonte 3.4: SUT Hipotético versão base.

```
1   class OperacoesMatematicas {
2       static int soma(final int a, final int b) {
3           // Mudança no Operador Matemático
4           int soma = a - b;
5           return soma;
6       }
7       static int produto(final int a, final int b) {
8           //Mudança no Operador Matemático
9           int produto = a + b;
10          return produto;
11      }
12      static int divisao(final int a, final int b) {
13          // Mudança no Operador Matemático
14          int divisao = a - b;
15          return divisao;
16      }
17  }
```

Código Fonte 3.5: SUT Hipotético versão modificada.

```
1  public class OperacoesMatematicasTests {
2      @Test
3      public void CT01() {
4          assertEquals("O resultado deveria ser 4.", 4,
               OperacoesMatematicas.soma(2, 2));
5          assertEquals("O resultado deveria ser 1.", 1,
               OperacoesMatematicas.divisao(2, 2));
6      }
7      @Test
8      public void CT02() {
9          assertEquals("O resultado deveria ser 4.", 4,
               OperacoesMatematicas.soma(2, 2));
10         assertEquals("O resultado deveria ser 6.", 6,
               OperacoesMatematicas.produto(2, 3));
11     }
12 }
```

Código Fonte 3.6: Parte da suíte de testes para o SUT Hipotético.

Na Figura 3.2 é possível visualizar os arquivos de *log* das entradas e saídas desta Atividade para os códigos-fonte 3.4, 3.5 e 3.6.

3.3 Atividade 2: Aplicação da Técnica *Changed Blocks (CB)*

A técnica *CB* recebe como entrada os *logs* de cobertura de cada teste automatizado. Dando sequência ao exemplo, os artefatos de entrada dessa etapa seriam as saídas apresentadas na Figura 3.2 e o *Log* de mudanças para tornar possível a priorização.

Segundo Srivastava & Thiagarajan [40], a técnica recebe como entrada duas versões de código-fonte, informações de cobertura da versão base de código e uma lista de CTs a

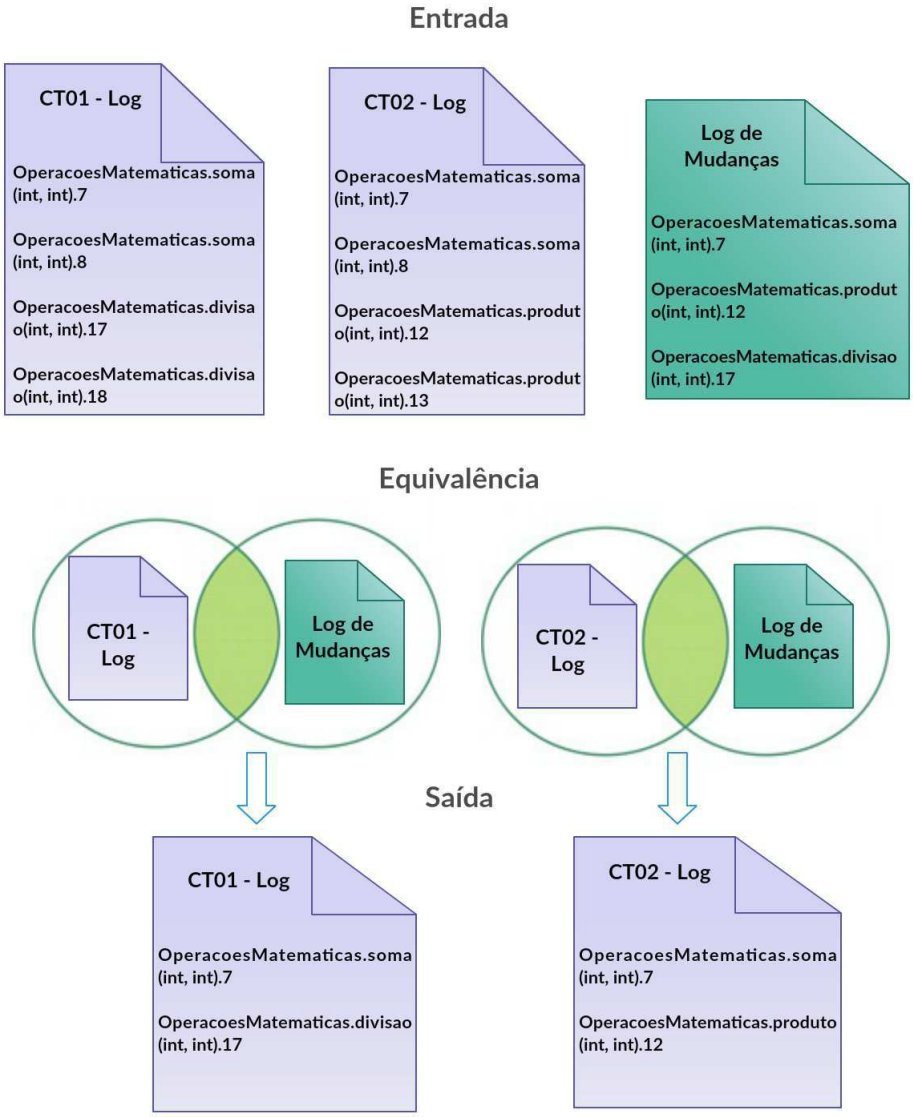

Figura 3.2: Entrada, processo de equivalência e saída da Atividade de Descoberta dos Elementos Impactados.

serem priorizados. Primeiro é realizada uma comparação entre as duas versões do código que resulta nas mudanças existentes. Na sequência é realizada uma análise de cobertura reunindo as informações de cobertura da versão base com as mudanças realizadas. Por fim, é realizada a priorização da suíte de testes.

A técnica *CB* se baseia na simples contagem de mudanças cobertas por cada CT; a quantidade final dessa contagem vai definir a posição do CT na lista priorizada. O CT com a quantidade maior assumirá a primeira posição da lista, caso existam empates, não é possível afirmar qual será a posição entre eles, assumirão posições aleatória dentro do grupo de

CTs empatados. A Figura 3.3 mostra um fluxograma simples de como funciona a técnica *Changed Blocks*.

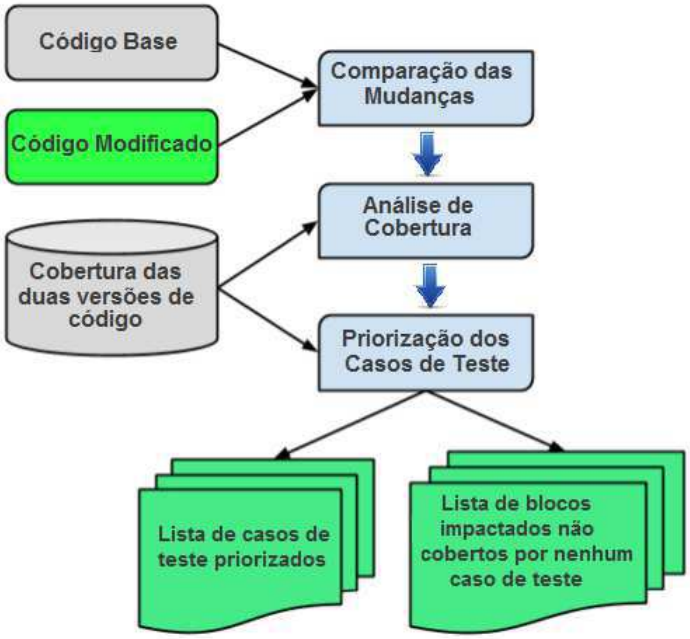

Figura 3.3: Fluxograma da técnica *Changed Blocks*, baseado em Srivastava and Thiagarajan.

Levando em consideração a heurística da *Changed Blocks*, a Figura 3.4 apresenta cinco CTs (CT01 - CT05) e suas respectivas mudanças cobertas (M01 - M08). A suíte ordenada demonstrada na Figura 3.4 segue o passo a passo apresentado no algoritmo apresentado no fluxograma da Figura 3.3. Através da contagem de mudanças cobertas por cada CT, temos que o CT03 ocupa a primeira posição por cobrir quatro mudanças (M01 - M04); os CT01, CT02 e CT04 são ordenados aleatoriamente na ocupação da segunda, terceira e quarta posições, pois cobrem a mesma quantidade de mudanças; e na última posição temos o CT05 que cobre apenas duas mudanças (M07 - M08).

3.4 Atividade 3: Aplicação da Técnica *Weighted Changed Statements (WCS)*

A técnica CB contém algumas desvantagens no seu comportamento. A primeira delas refere-se a CTs que executam a mesma quantidade de mudanças, independente de serem inéditas, serão priorizadas de forma aleatória entre o grupo empatado. CTs com pouca cobertura

	CT01	CT02	CT03	CT04	CT05
M01		X	X		
M02		X	X	X	
M03		X	X	X	
M04	X		X	X	
M05	X				
M06	X				
M07					X
M08					X

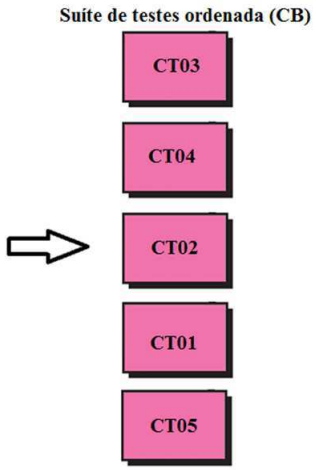

Figura 3.4: Exemplo para o funcionamento da técnica *WCS-Total*.

que executam mudanças inéditas também são desfavorecidos na lista priorizada, quando comparados a CTs com alto grau de cobertura. Dessa forma, torna-se evidente a necessidade de se ter outro critério que solucione tais problemas.

Visando solucionar as desvantagens apontadas, propõe-se uma técnica que implemente um novo critério de priorização aplicado ao resultado da técnica *CB*. Para essa atividade tem-se como entrada: i. a lista priorizada do *Changed Blocks*, pois nela já está aplicada a priorização por contagem de mudanças executadas; ii. O *log* de mudanças; e iii. O *log* de cobertura de cada CT. Assim, é possível aplicar algum critério a partir da análise por ordem crescente na lista resultante da *CB*, onde todos os CTs que executaram alguma mudança farão parte desse subconjunto sob análise.

O fluxograma da Figura 3.5 mostra os passos necessários para a implementação do algoritmo *WCS-Total*. De acordo com o objetivo inicial, essa abordagem contém um novo critério de priorização que baseia-se no cálculo de um *peso (W)*, que é totalmente dependente dos valores do *incremento (inc)* e *decremento (dec)*, que serão responsáveis pelo desempate de CTs com o mesmo número de cobertura e pela valorização de CTs que executam mudanças inéditas. Estes valores foram definidos empiricamente, através de vários testes manuais, de uma forma que a técnica *WCS-T* mantivesse a ideia principal da *CB* de privilegiar CTs com alto grau de cobertura de mudanças.

O Algoritmo 1 representa a implementação da técnica *WCS*. Ele mostra que além de varrer a lista priorizada de casos de teste da *CB*, também varre a lista de mudanças que

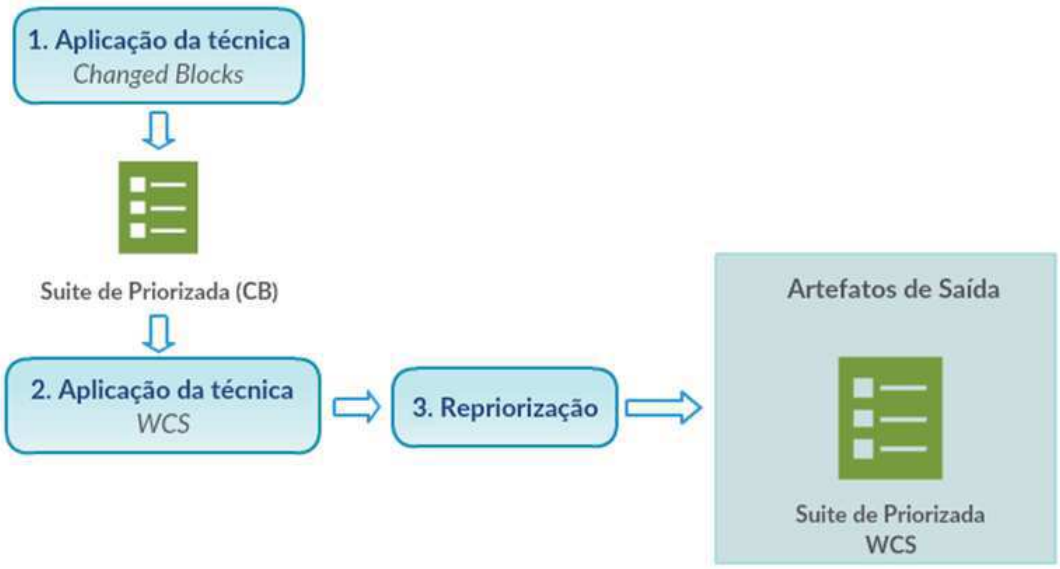

Figura 3.5: Fluxograma da técnica *WCS-Total*.

afetam cada CT. Logo, para cada CT, valores de incremento e decremento são calculados, e, para cada mudança, existe a aplicação desses fatores ao *Peso*. Esse cálculo pode ser observado nas linhas 8 e 9 do algoritmo. O objetivo do incremento é favorecer casos de teste que cobrem mudanças inéditas (ou seja, linhas de código que não foram cobertas por nenhum caso de teste com maior prioridade no subconjunto proveniente da *Changed Blocks*). A linha 8 do Algoritmo 1 representa a fórmula que caracteriza a bonificação do peso de um CT apresentada abaixo.

$$Inc : \frac{1}{n}$$

onde *n* é a quantidade de instruções cobertas pelo CT.

O valor do decremento objetiva evitar que mudanças que já foram cobertas por CTs que estão no topo da lista priorizada sejam repetidamente priorizadas nas posições subsequentes, enquanto que algumas mudanças inéditas que apresentam taxas de coberturas baixas se encontrem em posições menos prioritárias. A linha 9 do Algoritmo 1 representa a fórmula que caracteriza a penalização do peso de um CT apresentada abaixo. Após vários testes realizados, tornou-se perceptível que o valor do decremento não poderia ser maior ou igual ao incremento, pois casos de testes que cobrem mudanças analisadas por vários CTs seriam bastante penalizados a ponto de ocuparem posições desfavorecidas no resultado final.

$$Dec : \frac{1}{2n}$$

onde *n* é a quantidade de instruções cobertas pelo CT.

O valor do peso objetiva melhorar a priorização realizada pela *CB* e é totalmente dependente do número de mudanças cobertas, do incremento e decremento. A cada iteração das linhas de código do laço entre as linhas 4 - 27 do Algoritmo 1, um peso é calculado para cada CT seguindo a Equação 3.4 apresentada abaixo.

$$W : n + (mi * inc) - (mc * dec)$$

onde *n* é a quantidade de instruções cobertas pelo CT, *mi* é o número de mudanças inéditas cobertas pelo CT e *mc* é o número de mudanças previamente cobertas pelo CT.

Todas essas equações são implementadas na técnica *WCS-T* e demonstradas no Algoritmo 1. Ele vai varrer a lista previamente priorizada pela técnica *CB* (linha 4). Para cada CT, o *peso* é inicializado com a quantidade de mudanças cobertas calculado pela *CB* (linhas 5, 6 e 7) e, baseado na quantidade de mudanças, tem-se o valor do incremento e decremento (linhas 8 e 9). Para o primeiro CT da lista, soma-se ao *peso* a quantidade de mudanças multiplicada pelo incremento, pois por ser o primeiro elemento analisado, a lista de controle não contém nenhuma mudança, então todas são inéditas (linhas 10 - 14). Se for do segundo CT em diante, o algoritmo varre sua lista de mudanças, aplica o incremento e/ou decremento e atualiza a lista de controle, se necessário (linhas 16 - 24). Feito isso, o peso do CT é atualizado, a lista final é ordenada pelo peso de forma decrescente (linha 28) e resultado final retornado.

O algoritmo apresenta duas estruturas de repetição aninhadas, resultando no custo assintótico $O(nm)$, onde *n* é a quantidade de casos de teste e *m* é a quantidade de mudanças cobertas por cada CT.

Utilizando os mesmos dados da priorização da técnica *CB* apresentados na Figura 3.6 (cinco CTs e oito mudanças), será realizada uma priorização com a técnica *WCS-T* seguindo o fluxograma da Figura 3.5 e o Algoritmo 1. A suíte ordenada da técnica *CB* presente na Figura 3.4 é a entrada para a técnica *WCS-T*, assim como as mudanças e a cobertura dos

Algorithm 1 Weighted Changed Statement Total Algorithm

1: **função** APLICAWCST($listaCB$) ▷ Aplicação da técnica WCS-T
2: $controle \leftarrow criaLista()$
3: $listaWCST \leftarrow criaLista()$
4: **para** i ← 1 **até** tamanho(listaCB) **faça** ▷ Varre a lista priorizada CB a partir do topo
5: $ct \leftarrow listaCB[i]$
6: $mudancasCobertas \leftarrow buscaMudancasCobertas(ct)$
7: $peso \leftarrow tamanho(mudancasCobertas)$
8: $incremento \leftarrow 1/peso$
9: $decremento \leftarrow 1/(2 * peso)$
10: **se** $i == 1$ **então** ▷ Primeiro elemento da lista priorizada CB
11: $peso \leftarrow peso * incremento$
12: $atualizaListaDeControle(mudancasCobertas)$
13: $teste.atualizaPeso(peso)$
14: $continue$ ▷ Comando para passar para a próxima iteração do laço
15: **senão**
16: **para** i ← 1 **até** tamanho(mudancasCobertas) **faça** ▷ Cálculo do peso
17: $mudanca \leftarrow mudancasCobertas[i]$
18: **se** $controle.contem(mudanca)$ **então**
19: $peso \leftarrow peso - decremento$ ▷ Aplicação do decremento
20: **senão**
21: $peso \leftarrow peso + incremento$ ▷ Aplicação do incremento
22: $atualizaListaDeControle(mudanca)$ ▷ Atualização da lista de controle
23: **fim se**
24: **fim para**
25: $teste.atualizaPeso(peso)$ ▷ Aplicação do Peso no CT
26: **fim se**
27: **fim para**
28: $ordenaListaPeloPeso(listaWCST, decrescente)$
29: **return** $listaWCST$ ▷ Lista priorizada
30: **fim função**

casos de teste. Como os cinco CTs cobrem pelo menos uma mudanças, a *WCS-T* executará com cinco iterações, uma para cada CT, analisando na mesma sequência em que se encontra a suíte de testes ordenada da *CB* (CT03 - CT04 - CT02 - CT01 - CT05).

	CT01	CT02	CT03	CT04	CT05
M01		X	X		
M02		X	X	X	
M03		X	X	X	
M04	X		X	X	
M05	X				
M06	X				
M07					X
M08					X

Figura 3.6: Dados para a priorização da técnica *WCS-Total*.

A Tabela 3.1 representa os atributos da primeira iteração do *WCS-T* aplicada ao resultado da aplicação da técnica *CB*. Como apresentado na Figura 3.2, CT03 cobre quatro mudanças (M01 - M04). Aplicando as fórmulas do incremento e decremento, temos que $n = 4$ (quantidade de mudanças cobertas pelo CT), logo o valor do incremento será 1/4 = 0.25 e o do decremento será 1/8 = 0.125.

Além disso, ao visualizar o item (b) da Tabela 3.1, percebemos que a lista de controle está vazia. Logo, o CT03 cobre quatro mudanças inéditas e nenhuma mudança coberta previamente, então o cálculo do seu peso será *4 + (4 * 0.25) - (0 * 0.125) = 5*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT03 são inseridas na lista de controle.

Na segunda iteração, o CT que ocupa a segunda posição da suíte ordenada *CB* é o CT04. Segundo a Figura 3.6, ele cobre as mudanças M02, M03 e M04, todas elas previamente cobertas pelo CT03. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.2. Logo, o CT04 cobre três mudanças cobertas previamente, então o cálculo do seu peso será *3 + (0 * 1/3) - (3 * 1/6) = 2,5*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT04 são inseridas na lista de controle, como todas já estão presentes na lista de controle, ela não é atualizada.

Na terceira iteração, o CT que ocupa a terceira posição da suíte ordenada *CB* é o CT03. Segundo a Figura 3.6, ele cobre as mudanças M01, M02 e M03, todas elas previamente

Tabela 3.1: Atributos da primeira iteração do *WCS-T*

(a)

Atributo	**Valor**
Caso de Teste	CT03
n	4
Iteração	1
Inc	0.25
Dec	0.125
W	5

(b)

Lista de Controle

Tabela 3.2: Atributos da segunda iteração do *WCS-T*

(a)

Atributo	**Valor**
Caso de Teste	CT04
n	3
Iteração	2
Inc	1/3
Dec	1/6
W	2,5

(b)

Lista de Controle
M01
M02
M03
M04

cobertas pelos CT03 e CT04. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.3. Logo, o CT02 cobre três mudanças cobertas previamente, então o cálculo do seu peso será *3 + (0 * 1/3) - (3 * 1/6) = 2,5*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT03 são inseridas na lista de controle, como todas já estão presentes na lista de controle, ela não é atualizada.

Na quarta iteração, o CT que ocupa a quarta posição da suíte ordenada *CB* é o CT01. Segundo a Figura 3.6, ele cobre as mudanças M04, M05 e M06, onde M04 é previamente coberta pelos CT03 e CT04 e M05 e M06 são inéditas. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.4. Logo, o CT01 cobre uma

Tabela 3.3: Atributos da terceira iteração do *WCS-T*

(a)

Atributo	Valor
Caso de Teste	CT02
n	3
Iteração	3
Inc	1/3
Dec	1/6
W	2,5

(b)

Lista de Controle
M01
M02
M03
M04

mudança coberta previamente e duas inéditas, então o cálculo do seu peso será *3 + (2 * 1/3) - (1 * 1/6) = 3,333*. Ao final do cálculo do *peso* as mudanças inéditas cobertas pelo CT01 são inseridas na lista de controle, no caso, M05 e M06 serão inseridas na lista de controle.

Tabela 3.4: Atributos da quarta iteração do *WCS-T*

(a)

Atributo	Valor
Caso de Teste	CT01
n	3
Iteração	4
Inc	1/3
Dec	1/6
W	3,333

(b)

Lista de Controle
M01
M02
M03
M04
M05
M06

Na quinta e última iteração, o CT que ocupa a quinta posição da suíte ordenada *CB* é o CT05. Segundo a Figura 3.6, ele cobre as mudanças M07 e M08, ambas inéditas. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.5. Logo, o CT05 cobre duas mudanças inéditas, então o cálculo do seu peso será *2 + (2 * 1/2) - (0 * 1/4) = 3*. Ao final do cálculo do *peso* as mudanças inéditas cobertas pelo CT05 são inseridas na lista de controle, no caso, M07 e M08 serão inseridas na lista de controle.

Tabela 3.5: Atributos da quinta iteração do *WCS-T*

(a)

Atributo	Valor
Caso de Teste	CT05
n	2
Iteração	5
Inc	1/2
Dec	1/4
W	3

(b)

Lista de Controle
M01
M02
M03
M04
M05
M06

(c)

Lista de Controle
M01
M02
M03
M04
M05
M06
M07
M08

CTs	*n*	*Inc*	*Dec*	Cálculo do Peso	*W*
CT01	3	1/3	1/6	3 + (2*1/3) – (1*1/6)	3,333
CT02	3	1/3	1/6	3 + (0*1/3) – (3*1/6)	2,5
CT03	4	1/4	1/8	4 + (4*1/4) – (0*1/8)	5
CT04	3	1/3	1/6	3 + (0*1/3) – (3*1/6)	2,5
CT05	2	1/2	1/4	2 + (2*1/2) – (0*1/4)	3

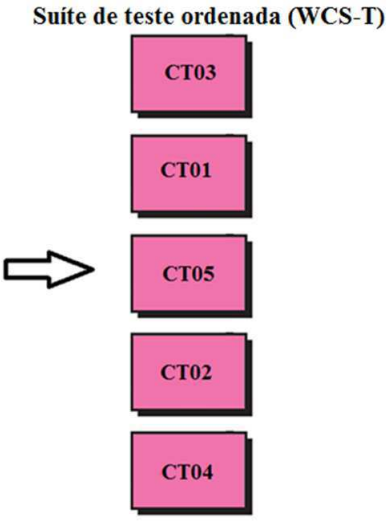

Figura 3.7: Reordenação da suíte de testes baseando-se no peso - *WCS-Total*.

Com a finalização das iterações para o cálculo do *peso*, temos que todas as mudanças se encontram na Lista de Controle, demonstrada no item (c) da Tabela 3.5. Assim, resta realizar uma reordenação na suíte de testes, como mostra a Figura 3.7. Como podemos perceber, o CT03 manteve a sua posição original, pois é o CT que cobre mais mudanças. Em contrapartida, os CT02 e CT04, que cobriam apenas mudanças previamente cobertas pelo CT03 foram penalizadas e perderam posições no resultado final. Os CT01 e CT05, que cobrem mudanças inéditas, foram privilegiados no resultado final da priorização da técnica

WCS-T.

3.4.1 *WCS-Additional*

Como foi explicado brevemente no Capítulo 2, duas estratégias de priorização de casos de teste foram estudadas e utilizadas para a realização desse trabalho, são elas: *Total* e *Additional*. Esta última, geralmente, apresenta resultados superiores para CTs que executam instruções inéditas. O intuito da implementação dessa abordagem é aumentar ainda mais o privilégio para CTs que cobrem mudanças inéditas e realizar uma comparação entre as técnicas, a fim de descobrir quais apresentam os melhores resultados. A estratégia *Total* tem como principal característica priorizar CTs a partir da contagem do número total de linhas de código que cada CT possui em sua lista de cobertura. Assim, torna-se possível a reordenação da suíte de testes de forma decrescente. Caso vários CTs possuírem um número igual de linhas de código cobertas, a ordenação será realizada de forma aleatória.

Com uma abordagem diferente da estratégia *Total*, a *Additional* tem como objetivo principal selecionar iterativamente um CT que possui a maior cobertura de código-fonte e atualiza os dados dos CTs não analisados ainda. Essa atualização consiste em marcar as linhas de código já cobertas em toda a suíte e realizar o desempate entre CTs com a mesma quantidade de mudanças cobertas. CTs empatados que cobrem mudanças inéditas são privilegiados sobre os que cobrem mudanças cobertas previamente. Isso evita que partes do código-fonte já analisadas por CTs anteriores apresentem maior prioridade perante CTs com linhas de código inéditas. Este passo se repete até que a suíte de testes seja totalmente analisada. Caso vários CTs possuírem um número igual de linhas de código cobertas inéditas, a ordenação será realizada de forma aleatória.

O fluxograma da Figura 3.8 mostra o comparativo entre as execuções das técnicas *WCS Total e Additional*. De acordo com o objetivo inicial, a técnica*WCS-A* contém os mesmos critérios de priorização que se baseiam nos cálculos do *peso (W)*, do *incremento (inc)* e do *decremento (dec)*, que serão responsáveis pelo desempate de CTs com o mesmo número de cobertura e pela valorização de CTs que executam mudanças inéditas. As principais diferenças entre as duas técnicas são: i. a presença da estratégia *additional* como atividade intermediária entre a aplicação da técnica *CB* e sua priorização, com o objetivo de privilegiar CTs que cobrem mudanças inéditas; ii. uma pequena mudança no cálculo do peso que, nessa

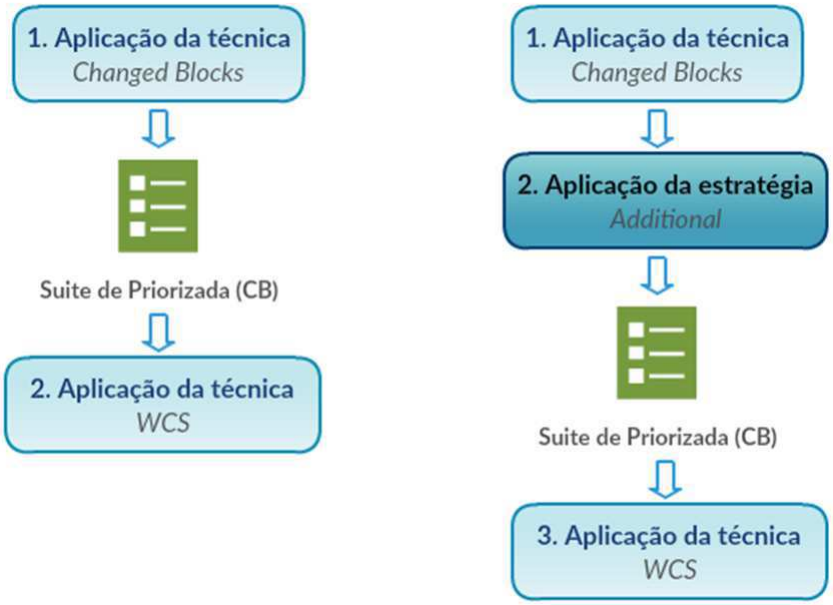

Figura 3.8: Fluxograma comparativo entre as técnicas *WCS-T e WCS-A*.

abordagem, ao invés de utilizar a quantidade de mudanças no cálculo (*n*), utilizará uma porcentagem com relação ao total de mudanças existentes, como mostra a Equação 3.4.1 a seguir.

$$W : n/m + (mi * inc) - (mc * dec)$$

onde *n* é a quantidade de instruções cobertas pelo CT, *m* é a quantidade de instruções modificadas, *mi* é o número de mudanças inéditas cobertas pelo CT e *mc* é o número de mudanças previamente cobertas pelo CT.

Segundo Rothermel et al. [36], cada CT não priorizado precisa atualizar as informações do restante dos CTs. Sendo assim, dada uma suíte de testes contendo *m* CTs e *n* linhas de código, a seleção de um CT e a atualização as informações de cobertura da suíte tem um custo de *O(mn)* as atualizações devem ser realizadas *O(m)* vezes. Logo, o custo total dessa estratégia consiste em $O(m^2n)$. A estratégia *Total* apresenta um custo de *O(mn)*, que em alguns casos pode chegar a $O(m^2)$. Mesmo assim a *Total* é menos custosa que a *Additional*, pois ela não precisa atualizar todos os CTs a cada iteração.

Utilizando os mesmos dados da priorização da técnica *CB* apresentados na Figura 3.6 (cinco CTs e oito mudanças), será realizada uma priorização com a técnica *WCS-A* seguindo

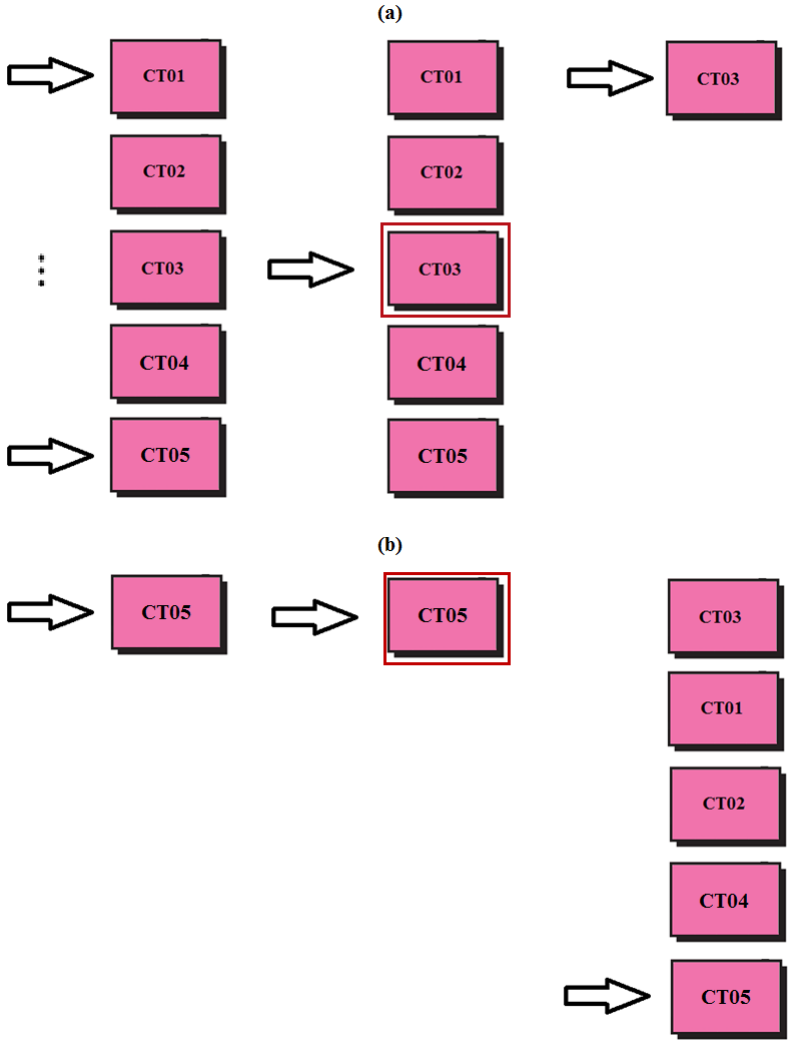

Figura 3.9: Exemplo da priorização da Changed Blocks com a estratégia Additional.

o fluxograma da Figura 3.8. A suíte ordenada da técnica *CB* presente na Figura 3.4 é a entrada para a técnica *WCS-A*, assim como as mudanças e a cobertura dos casos de teste. Como apresentado, junto a execução da técnica *CB*, ocorre a aplicação da estratégia *additional*, representada na Figura 3.9.

Na priorização da suíte de testes através da técnica *CB* com a estratégia *additional*, a cada iteração todos os CTs são analisados a fim de encontrar o que cobre a maior quantidade de mudanças. O item (a) da Figura 3.9 representa a análise de todos os casos de testes e o item (b) representa a última iteração do laço e apresenta a suíte ordenada. Como esperado da estratégia *additional*, quando ocorre empate, o CT que cobre mais mudanças inéditas será privilegiado com relação aos outros. Caso os CTs ainda assim continuarem empatados, serão

Algorithm 2 Weighted Changed Statement Additional Algorithm

```
1: função APLICAWCSA(listaCB, listaMudancas)        ▷ Aplicação da técnica WCS-A
2:     controle ← criaLista()
3:     listaWCSA ← criaLista()
4:     para i ← 1 até tamanho(listaCB) faça ▷ Varre a lista priorizada CB a partir do topo
5:         ct ← listaCB[i]
6:         totalMudancas ← tamanho(listaMudancas)
7:         mudancasCobertas ← buscaMudancasCobertas(ct)
8:         peso ← tamanho(mudancasCobertas)/totalMudancas
9:         incremento ← 1/tamanho(mudancasCobertas)
10:        decremento ← 1/(2 * tamanho(mudancasCobertas))
11:        se i == 1 então                          ▷ Primeiro elemento da lista priorizada CB
12:            peso ← peso * incremento
13:            atualizaListaDeControle(mudancasCobertas)
14:            teste.atualizaPeso(peso)
15:            continue                ▷ Comando para passar para a próxima iteração do laço
16:        senão
17:            para i ← 1 até tamanho(mudancasCobertas) faça          ▷ Cálculo do peso
18:                mudanca ← mudancasCobertas[i]
19:                se controle.contem(mudanca) então
20:                    peso ← peso − decremento                ▷ Aplicação do decremento
21:                senão
22:                    peso ← peso + incremento                ▷ Aplicação do incremento
23:                    atualizaListaDeControle(mudanca)        ▷ Atualização da lista de
   controle
24:                fim se
25:            fim para
26:            teste.atualizaPeso(peso)                        ▷ Aplicação do Peso no CT
27:        fim se
28:    fim para
29:    ordenaListaPeloPeso(listaWCSA, decrescente)
30:    return listaWCSA                                        ▷ Lista priorizada
31: fim função
```

ordenados aleatoriamente no subconjunto com o mesmo número de mudanças cobertas. Isso explica porque no resultado da priorização da *CB*, o CT01 ocupou a segunda posição na lista ordenada, pois os CT01, CT02 e CT04 cobrem três mudanças cada, mas o CT01 cobre duas mudanças inéditas. A suíte ordenada apresentada no item (b) da Figura 3.9 será a entrada do Algoritmo 2, assim como uma lista com todas as mudanças realizadas no código.

A Tabela 3.6 representa os atributos da primeira iteração do *WCS-A* aplicada ao resultado da aplicação da técnica *CB* com a estratégia *additional*. Como apresentado na Figura 3.6, CT03 cobre quatro mudanças (M01 - M04). Aplicando as fórmulas do incremento e decremento, temos que $n = 4$ (quantidade de mudanças cobertas pelo CT) e $m = 8$ (quantidade total de mudanças), logo a porcentagem de mudanças é 0.5, o valor do incremento será 1/4 = 0.25 e o do decremento será 1/8 = 0.125.

Além disso, ao visualizar o item (b) da Tabela 3.6, percebemos que a lista de controle está vazia. Logo, o CT03 cobre quatro mudanças inéditas e nenhuma mudança coberta previamente, então o cálculo do seu peso será *0.5 + (4 * 0.25) - (0 * 0.125) = 1.5*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT03 são inseridas na lista de controle.

Tabela 3.6: Atributos da primeira iteração do *WCS-A*

(a)

Atributo	**Valor**
Caso de Teste	CT03
n	4
m	8
Iteração	1
Inc	0.25
Dec	0.125
W	1.5

(b)

Lista de Controle

Na segunda iteração, o CT que ocupa a segunda posição da suíte ordenada *CB* é o CT01. Segundo a Figura 3.6, ele cobre as mudanças M04, M05 e M06, onde M04 é previamente cobertas pelo CT03 e as outras são mudanças inéditas. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.7. Logo, o cálculo do peso

para o CT01 é *0.375 + (2 * 1/3) - (1 * 1/6) = 0.7083*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT01 são inseridas na lista de controle.

Tabela 3.7: Atributos da segunda iteração do *WCS-A*

(a)

Atributo	**Valor**
Caso de Teste	CT01
n	3
m	8
Iteração	2
Inc	1/3
Dec	1/6
W	1.5

(b)

Lista de Controle
M01
M02
M03
M04

Na terceira iteração, o CT que ocupa a terceira posição da suíte ordenada *CB* é o CT02. Segundo a Figura 3.6, ele cobre as mudanças M01, M02 e M03, onde todas elas são previamente cobertas pelo CT03. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.8. Logo, o CT02 cobre três mudanças cobertas previamente, então o cálculo do seu peso será *0.3753 + (0 * 1/3) - (3 * 1/6) = -0.125*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT02 são inseridas na lista de controle, como todas já estão presentes na lista de controle, ela não é atualizada.

Na quarta iteração, o CT que ocupa a quarta posição da suíte ordenada *CB* é o CT04. Segundo a Figura 3.6, ele cobre as mudanças M02, M03 e M04, onde todas elas são previamente cobertas pelo CT03. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da Tabela 3.9. Logo, o CT04 cobre três mudanças cobertas previamente, então o cálculo do seu peso será *0.3753 + (0 * 1/3) - (3 * 1/6) = -0.125*. Ao final do cálculo do *peso* as mudanças cobertas pelo CT04 são inseridas na lista de controle, como todas já estão presentes na lista de controle, ela não é atualizada.

Na quinta e última iteração, o CT que ocupa a quinta posição da suíte ordenada *CB* é o CT05. Segundo a Figura 3.6, ele cobre as mudanças M07 e M08, ambas inéditas. Na iteração anterior a lista de controle foi atualizada, como podemos observar no item (b) da

Tabela 3.8: Atributos da terceira iteração do *WCS-A*

(a)

Atributo	Valor
Caso de Teste	CT02
n	3
m	8
Iteração	3
Inc	1/3
Dec	1/6
W	-0.125

(b)

Lista de Controle
M01
M02
M03
M04
M05
M06

Tabela 3.10. Logo, o CT05 cobre duas mudanças inéditas, então o cálculo do seu peso será *0.25 + (2 * 1/2) - (0 * 1/4) = 1.25*. Ao final do cálculo do *peso* as mudanças inéditas cobertas pelo CT05 são inseridas na lista de controle, no caso, M07 e M08 serão inseridas na lista de controle.

CTs	*n*	*Inc*	*Dec*	Cálculo do Peso	*W*
CT01	0.375	1/3	1/6	0.375 + (2*1/3) – (1*1/6)	0.7083
CT02	0.375	1/3	1/6	0.375 + (0*1/3) – (3*1/6)	-0.125
CT03	0.5	1/4	1/8	0.5 + (4*1/4) – (0*1/8)	1.5
CT04	0.375	1/3	1/6	0.375 + (0*1/3) – (3*1/6)	-0.125
CT05	0.25	1/2	1/4	0.25 + (2*1/2) – (0*1/4)	1.25

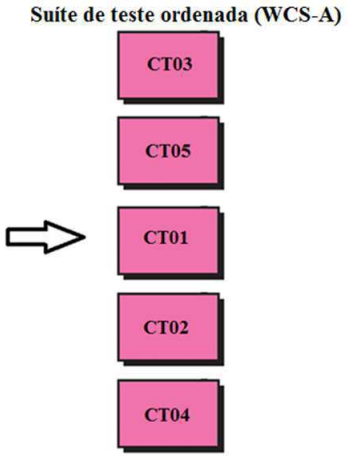

Figura 3.10: Reordenação da suíte de testes baseando-se no peso - *WCS-Additional*.

Com a finalização das iterações para o cálculo do *peso*, temos que todas as mudanças se encontram na Lista de Controle, demonstrada no item (c) da Tabela 3.10. Assim, resta realizar uma reordenação na suíte de testes, como mostra a Figura 3.10. Como podemos perceber, o CT03 manteve a sua posição original, pois é o CT que cobre mais mudanças inéditas. Em contrapartida, os CT02 e CT04, que cobriam apenas mudanças previamente

Tabela 3.9: Atributos da quarta iteração do *WCS-A*

(a)

Atributo	**Valor**
Caso de Teste	CT04
n	3
m	8
Iteração	3
Inc	1/3
Dec	1/6
W	-0.125

(b)

Lista de Controle
M01
M02
M03
M04
M05
M06

cobertas pelo CT03 foram penalizadas e perderam posições no resultado final. Os CT01 e CT05, que cobrem mudanças inéditas, foram privilegiados no resultado final da priorização da técnica *WCS-A* e ocuparam posições bem próximas do topo da lista, inclusive o CT05 foi superior ao CT01 pelo fato de cobrir apenas mudanças inéditas, apesar de cobrir menos mudanças.

3.5 Atividade 4: Priorização dos Casos de Teste

A atividade de priorização tem como função principal priorizar CTs de acordo com as técnicas escolhidas pelo usuário, realizando a reordenação da suíte sobre os pesos calculados, a percentagem de casos de testes selecionados para o preenchimento do relatório final e a versão do *JUnit*.

3.6 Implementação

As abordagens propostas foram implementadas utilizando a ferramenta já existente *PriorJ* para priorização de testes em *JUnit*. Ela analisa a cobertura de código-fonte juntamente com a comparação dos arquivos com extensão *.java*, além de ter incorporada as duas técnicas utilizadas na análise experimental deste trabalho (*RND e CB*). Para tornar este trabalho pos-

Tabela 3.10: Atributos da quinta iteração do *WCS-A*

(a)

Atributo	Valor
Caso de Teste	CT05
n	2
8	8
Iteração	5
Inc	1/2
Dec	1/4
W	1.25

(b)

Lista de Controle
M01
M02
M03
M04
M05
M06

(c)

Lista de Controle Final
M01
M02
M03
M04
M05
M06
M07
M08

sível, duas classes de priorização (*WCSTotal.java e WCSAdditional.java*) foram inseridas na ferramenta, além de ampliar as funcionalidades da ferramenta com a inserção das métricas apresentadas no Capítulo 4. O *framework* também gera *reports* que mostram a ordem das suítes priorizadas para cada técnica e gráficos visuais para a comparação dos resultados [1].

É importante enfatizar que a maioria das linguagens de programação orientadas a objeto (OO) funcionam de forma similar, sendo assim, nós temos motivos para acreditar que a nossa proposta possa ser aplicável para outras linguagens de programação OO além de *Java*, caso sejam realizadas algumas mudanças no código-fonte da ferramenta. A principal delas seria a mudança na implementação da Atividade 3.2, pois o *difference* se utiliza de bibliotecas que reconhecem palavras-chave do Java. Essa atividade deve ser generalizada para atingir uma quantidade maior de linguagens. Outra modificação seria nas Atividades Instrumentação e Extração de Cobertura, elas estão vinculadas diretamente ao *AspectJ*, que também trabalha apenas com a linguagem Java.

3.7 Observações Finais

Neste capítulo foram introduzidas as abordagens propostas neste trabalho, *WCS-Total e WCS-Additional*. Elas foram desenvolvidas a fim de acelerar o processo de detecção de

defeitos introduzidos através de mudanças realizadas no código-fonte que impactam diretamente em casos de teste e independe da percentagem de cobertura do código. Para isso, um conjunto de atividades foram definidas com o objetivo de melhorar a priorização dos casos de teste de regressão do SUT em certos contextos que a técnica *Changed Blocks* ainda deixa a desejar. No próximo capítulo apresentamos com detalhes como foi realizada a análise experimental das heurísticas propostas.

Capítulo 4

Experimento

Este capítulo apresenta uma avaliação empírica sobre as técnicas de priorização propostas comparadas a *Changed Blocks (CB) e Random (RND)*. O objetivo desta avaliação é investigar se as técnicas *Weighted Changed Statements (WCS) Total e Additional* podem melhorar a detecção de defeitos quando comparadas as técnicas *Random (RND)* e *CB*. Por resultados melhores, entende-se que as técnicas produzem um resultado onde os CTs que cobrem mudanças inéditas que, por consequência ocasionam defeitos, encontrem-se em posições privilegiadas na lista priorizada.

Na Seção a seguir, são apresentadas as Perguntas de Pesquisa na Seção 4.1, a Configuração Experimental na Seção 4.2, os Resultados e Discussão na Seção 4.3, as Ameaças à Validade na Seção 4.4 e as Considerações Finais na Seção 4.5.

4.1 Perguntas de Pesquisa

Para avaliar o comportamento das técnicas propostas (*WCS-Total* e *WCS-Additional*) em relação a técnica *CB* [40] avaliada no trabalho de Alves et. al. [1] e a técnica de controle (*RND*), foi realizada uma avaliação empírica baseada nas perguntas de pesquisa listadas a seguir. Essas perguntas são importantes para investigar que a melhoria realizada na técnica *Changed Blocks* não apresentou impactos estatísticos negativos nos seus resultados, além deles se sobressaírem com relação aos resultados obtidos em [1].

1. **PP01:** Existe diferença entre as técnicas para a métrica *APFD*?

Essa pergunta de pesquisa tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *APFD*.

2. **PP02:** Existe diferença entre as técnicas para a métrica *F-Measure*?

   Essa pergunta de pesquisa tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *F-Measure*.

3. **PP03:** Existe diferença entre as técnicas para a métrica *F-Spreading*?

   Essa pergunta de pesquisa tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas agrupam melhor casos de teste que cobrem defeitos que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *F-Spreading*.

4. **PP04:** Existe diferença entre as técnicas para a métrica *Group-Measure*?

   Essa pergunta de pesquisa tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos inéditos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *Group-Measure*.

5. **PP05:** Existe diferença entre as técnicas para a métrica *Group-Spreading*?

   Essa pergunta de pesquisa tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas agrupam melhor casos de teste que cobrem defeitos inéditos que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *Group-Spreading*.

6. **PP06:** Existe diferença entre as técnicas para a métrica *Tempo de Execução*?

Essa pergunta de pesquisa tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas apresentam um custo maior que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados do *Tempo de Execução*.

4.2 Configuração Experimental

As técnicas propostas visam melhorar os resultados da *CB* para contextos onde múltiplas mudanças são inseridas no código, sem que ocorra nenhuma perda estatística considerável no resultado final da técnica original, no que diz respeito a antecipação de CTs que cobrem mudanças inéditas. Para avaliar o desempenho das técnicas *WCS-Total* e *WCS-Additional*, o estudo experimental compara ambas com os resultados da *CB* e a técnica *(RND)* em relação a vários cenários de mudança. A RND, também muito utilizada em experimentos [20; 10] como técnica de controle, não contém nenhuma heurística, por isso também é conhecida como ordem natural. Para realizar a priorização aleatória, basta tomar o próximo caso de teste aleatoriamente até que todos sejam considerados.

4.2.1 Variáveis Independentes

Nesse trabalho foram utilizadas duas variáveis independentes apresentadas a seguir:

Técnica de Priorização: Das quatro técnicas de priorização de teste de regressão utilizadas para a experimentação, duas seguem a estratégia *Total* (CB e WCS-Total), uma segue a estratégia *Additional* (WCS-Additional) e uma que não segue nenhuma das estratégias e faz parte do grupo de controle (RND).

SUT: É de conhecimento geral que os sistemas utilizados variam de acordo com a equipe de desenvolvimento, a metodologia utilizada, o tempo planejado para a entrega e *n* fatores que influenciam no resultado final do sistema. Os sistemas utilizados neste trabalho são *open sources* e foram escolhidos de acordo com os requisitos da ferramenta *PriorJ*, implementados na linguagem Java com suíte de testes automáticos.

4.2.2 Variáveis Dependentes

As métricas usadas para a avaliação de qualidade de uma suíte de testes priorizada tem sido aplicadas em trabalhos de pesquisa similares [1; 10; 17; 20; 35; 36], são elas: a *F-Measure*, *APFD*, *F-spreading*, *Group-measure*, *Group-spreading* e Tempo de Execução.

APFD: do inglês, *Average Percentage of Fault Detection* [36; 10; 11], pode ser calculada de acordo com a Equação 4.1. Seu objetivo é medir a velocidade com que a suíte de testes priorizada detecta defeitos. Ela se baseia na posição que os primeiros casos de teste que detectam defeitos estão alocados na suíte priorizada. Grandes valores da APFD implicam em uma rápida taxa de detecção de defeitos.

$$APFD = 1 - \frac{\sum_1^m F_i}{nm} + \frac{1}{2n} \tag{4.1}$$

sendo T uma suite de testes priorizada, a APFD para T é calculada através da equação apresentada, onde *n* é o número de casos de teste, *m* é o número de defeitos encontradas, e F_i é a posição do primeiro caso de teste que detecta o defeito *i* na sequência ordenada de casos de teste.

Exemplo: supondo um *SUT* que contém uma suíte com dez casos de teste (CTs) e uma versão com 3 mudanças aplicadas, representada na Tabela 4.1. Aplicando a técnica *Changed Blocks (CB)*, O CT03 necessariamente será alocado na primeira posição, enquanto os CT06, CT07 e CT10 ocuparão, com ordem aleatória entre si, a segunda, terceira e quarta posições na lista; o restante dos CTS ocuparão o restante das posições com ordem aleatória. Então, *T* (CT03 - CT07 - CT10 - CT06 - CT01 - CT02 - CT05 - CT08 - CT09 - CT04) é um possível resultado para a priorização dessa suíte com a aplicação da técnica *CB*.

A Tabela 4.2 representa os dados coletados da suíte priorizada *T* a fim da demonstração do cálculo da *APFD*, a posição *F* que revelou um defeito. A primeira posição aparece 2 vezes no conjunto *F*, pois o CT03 revelou os dois defeitos que os CT06 e 07 capturaram. Dessa forma, aplicando a Equação 4.1, temos que a *APFD* para a suíte priorizada *T* é de 88%.

F1-Measure: Esta métrica nos dá o número de casos de teste distintos que precisam serem executados, a fim de detectar o primeiro defeito do sistema, utilizando a suíte priorizada

Tabela 4.1: SUT para servir de base de exemplificação do cálculo das métricas.

CT X Change	**#01**	**#02**	**#03**
CT01			
CT02			
CT03	X	X	
CT04			
CT05			
CT06		X	
CT07	X		
CT08			
CT09			
CT10			X

Tabela 4.2: Valores para o cálculo da *APFD*

Termos	***n***	***m***	**F**	$\sum_1^m F_i$
Valor	10	3	[1, 1, 3]	5

Tabela 4.3: Valores para o cálculo da *F-spreading*

F-spreading	***n***	***m***	***S***	$\sum_2^m (S_i - S_{i-1})$
Valor	10	4	[1, 1, 1, 3]	(3 - 1) + (1 - 1) + (1 - 1) = 2

[44].

Exemplo: Utilizando a suíte priorizada *T* (CT03 - CT07 - CT10 - CT06 - CT01 - CT02 - CT05 - CT08 - CT09 - CT04), afirma-se que o resultado da F1-measure para *T* é F_1 = 1, pois o CT03 detecta pelo menos um defeito e está alocado na primeira posição de *T*.

F-Spreading: mede o grau de espalhamento de CTs que falham em uma dada permutação da suíte. Para esta métrica, valores próximos de zero significam que CTs impactados estão agrupados. Valores próximos de um, significam que CTs impactados estão distantes entre si. Equação 4.2 formaliza a métrica, onde *n* é o número de CTs da suíte; *m* é o número de testes falhos; S é uma sequência contendo as posições dos testes que falham; e S_i é a posição do i-ésimo teste falho na suíte de testes priorizada.

$$F - Spreading = \sum_{i=2}^{m} (S_i - S_{i-1}) * \frac{1}{n} \tag{4.2}$$

Exemplo: Tomando como base a suíte priorizada *T* (CT03 - CT07 - CT10 - CT06 - CT01 - CT02 - CT05 - CT08 - CT09 - CT04), a Tabela 4.3 representa os valores para cada variável da Equação 4.2. Ao aplicar a equação, o resultado para a *F-spreading* é 0,2.

Group-Measure: Sendo uma métrica proposta neste trabalho a fim de expor as posições das mudanças inéditas detectadas por cada CT e compará-las entre os resultados das técnicas. Ela é basicamente o mesmo cálculo da *F-Measure*, totalizada para o conjunto de CTs que executam mudanças. A Equação 4.3 denota essa formalização que tem como objetivo revelar a posição do primeiro caso de teste que falha na suíte ordenada para cada mudança detectada. Logo, para cada mudança executada, tem-se a posição na lista priorizada do CT que a executou. Seja *n* o total de casos de teste da suíte; *t*, o total de mudanças no SUT; GM

Tabela 4.4: Valores para o cálculo da *Group-measure*

Termos	***n***	***t***	***M***	***GM***
Valor	10	3	[#01, #02, #03]	[1, 1, 3,]

é a sequência contendo as posições dos primeiros testes falhos para cada mudança; M é a sequência contendo as mudanças inseridas no SUT; e GM_i é a posição do i^{o} teste falho para a i^{a} mudança.

$$\begin{aligned} M &= [M_i,\ M_{i+1},\ M_{i+2},\ ...,\ M_t], \\ GM &= [GM_i,\ GM_{i+1},\ GM_{i+2},\ ...,\ GM_t] \end{aligned} \tag{4.3}$$

Exemplo: Tomando como base a suíte priorizada *T* (CT03 - CT07 - CT10 - CT06 - CT01 - CT02 - CT05 - CT08 - CT09 - CT04), a Tabela 4.4 representa os valores para cada variável da Equação 4.3. Com esses valores é possível realizar a comparação entre as posições que cada uma das mudanças foram detectadas na suíte priorizada.

Group-Spreading: métrica também proposta neste trabalho, é basicamente o mesmo cálculo da *F-Spreading*, totalizada para o conjunto da *Group-Measure*. Ela mede como os CTs falhos que capturam a mudança estão espalhados em uma suíte de teste priorizada. Equação 4.4 formaliza a métrica, onde *n* é o número de casos de teste da suite; *m* é o número de testes falhos; GS é uma sequência contendo as posições dos testes falhos para cada mudança; e GS_i é a posição do i^{o} teste que falha para a i^{a} mudança.

$$GS = \sum_{i=2}^{m} (GS_i - GS_{i-1}) * \frac{1}{n} \tag{4.4}$$

Exemplo: Baseando-se na suíte priorizada *T* (CT03 - CT07 - CT10 - CT06 - CT01 - CT02 - CT05 - CT08 - CT09 - CT04), a Tabela 4.5 representa os valores para cada variável

Tabela 4.5: Valores para o cálculo da *Group-spreading*

Termos	***n***	***m***	***GS***	$\sum_2^m (GS_i - GS_{i-1})$
Valor	10	4	[1, 1, 3]	(1 -1) + (3 - 1) = 2

da Equação 4.4. Ao aplicar a equação, o resultado para a *Group-spreading* é 0,2.

Tempo: Esta variável registra os custos gerais de tempo de execução de cada técnica de priorização. Especificamente, nós comparamos todos os custos referentes a execução das técnicas de priorização, não levando em consideração a preparação de todo o ambiente, como: aplicação do *difference*, instrumentação, etc.

4.2.3 Unidades Experimentais

Nós avaliamos as técnicas usando quatro projetos desenvolvidos em *Java*, de código aberto, que podem ser visualizados na Tabela 4.6 junto com suas respectivas versões. As unidades são aplicações ou *plugins* que abordam a área de teste de software, seja dando suporte a *Test-Driven Development (TDD)* ou a testes de unidade. Todas as unidades foram encontradas no repositório *OpenHub*[1]. Alguns critérios de escolha das amostras foram utilizados e serão listados abaixo:

1. Projetos de código aberto.
2. Projetos de pequeno (1,9 KLoc) e médio (35,8 KLoc) porte.
3. Projetos com suite de teste separadas por pacotes, evitando cenários onde todos os casos cobrem grande parte do código.
4. Indicações do *OpenHub* para projetos similares aos que foram escolhidos.

É importante enfatizar que, na *release* atual, o *PriorJ* não dá suporte a projetos que utilizam ferramentas de integração e configuração (como *Ant, Maven, Gradle*, etc). Por isto a seleção das unidades utilizadas nessa avaliação se tornou limitada a projetos que não usam tais ferramentas ou que puderam ser transformadas em aplicações *Java* sem nenhum impacto.

[1] https://www.openhub.net/

Esses *frameworks* são responsáveis por gerenciar dependências, controlar o versionamento de artefatos, gerar relatórios de produtividade, garantir a regressão dos testes, entre outras funcionalidades. Porém, o *PriorJ* não utiliza a estrutura de organização de dependências, não tornando possível a execução da priorização. Sendo assim, os projetos utilizados sofreram uma modificação simples, os que foram implementados a partir dessas ferramentas de integração, foram transformados em *Java Application*, que é a estrutura que podemos garantir a execução correta da priorização dos casos de teste.

JMock[2] é uma biblioteca que dá suporte ao desenvolvimento orientado a testes (TDD) de códigos Java com objetos fictícios (*mock* de objetos) [16]. Atualmente, o *jmock* apresenta em seu site [16] duas versões estáveis da ferramenta: *jMock1* (estável para o JDK 1.3 ou superior) e *jMock2* (estável para o JDK 1.6 ou superior). Essa ferramenta tem seu código hospedado no *GitHub*, tanto da primeira versão, quanto da segunda.

JBehave[3] é um *framework* para o Desenvolvimento Orientado a Comportamento (do inglês, *Behavior-Driven Development - BDD*). *BDD* é uma evolução do desenvolvimento orientado a testes (do inglês, *Test-Driven Development - TDD*) e design orientado a teste de aceitação, e destina-se a fazer estas práticas mais acessíveis e intuitivas para os novatos e especialistas da área.

O *XML-Unit*[4] é um projeto que fornece as funcionalidades necessárias para verificar se o XML criado é o que teria sido planejado. Ele contém funções para validar um esquema XML, fazer comparações entre valores de consultas ou comparar documentos XML face aos resultados esperados. Ele consiste de três módulos, são eles: *XMLUnit-Legacy, XMLUnit-Core* e *XMLUnit-Matches*. Dentre eles, foi escolhido o *core* do sistema para fazer parte do nosso estudo, pois ele apresenta uma boa quantidade de testes, diferentemente dos outros módulos.

XMLMatchers[5] é uma biblioteca *Java* para combinar documentos *XML* com *templates*. As suas principais características são as expressões regulares a nível de elemento, assertivas voltadas para *Javascript*, tolerância, padrões e muito mais. Foi projetado para validação automática de *APIs* que produzem dados *XML* úteis.

[2]http://www.jmock.org/

[3]http://jbehave.org/

[4]http://www.xmlunit.org/

[5]http://xmlmatcher.sourceforge.net/

Tabela 4.6: Unidades experimentais utilizadas.

Amostra	**#Pacotes**	**#Classes**	**#Métodos**	**#Linhas**	**KLoC**	**#Testes**
jbehave	51	606	2,898	35,802	26,646	540
jmock	36	315	1270	14,067	9,774	396
xml-unit	21	221	939	14,560	9,595	283
xml-matchers	20	26	138	1,914	1,202	66

4.2.4 Mutações

Para este estudo foram utilizados mutantes, modificações no código-fonte de um SUT qualquer que podem gerar ou não modificações que ocasionam falha na suíte de testes. Essa abordagem foi escolhida pela restrição do *PriorJ* a projetos do tipo *Java Application*, onde as duas versões da suíte de testes devem ser exatamente iguais. Como a grande maioria dos projetos atuais utilizam algum gerenciador de dependência (*Ant, Maven, Gradle, etc*), a busca por projetos tornou-se restrita. Além do mais, em um contexto real, uma *release* de um sistema qualquer liberada para uso não apresenta grande quantidade de defeitos detectados em suas respectivas suítes de teste, pois, geralmente, essas defeitos são resolvidos para a liberação de uma versão estável do sistema. Para tornar possível a avaliação empírica das heurísticas propostas, são geradas mutações simples (modificações atômicas no código) através da ferramenta MuJava [26], selecionadas aleatoriamente e inseridas no código-fonte de forma manual.

As mutações utilizadas seguiram as categorias apresentadas na Tabela 4.7, simulando mudanças que ocorrem em ambientes reais. Pesquisas anteriores [2; 3; 9] confirmaram que a utilização de defeitos produzidos através da mutação de código é adequada para a experimentação da priorização, por serem mais fáceis, menos custosas, e torna possível a utilização sem prejuízos. Sendo assim, essa abordagem foi utilizada nesse trabalho para a geração dos dados apresentados na Seção 4.2.5, permitindo assim a geração e análise dos resultados para várias versões. Para cada amostra (versão modificada de um sistema) gerada e selecionada para esta análise, pelo menos uma mudança inserida resultou em falha ou erro nos CTs.

Para cada amostra, foram inseridas mutações aleatórias geradas a partir do sistema μJava (MuJava) [26]. Ele é um sistema de mutação para programas implementados em *Java* que

Tabela 4.7: Categorização das mutações utilizadas na experimentação.

Tipo	**Descrição da Mutação**
MOD	Mudanças gerais (ex.: troca de valores de variáveis, mudança na sequência do código, etc...)
REM	Remoção de uma ou mais linhas de código
OPL	Mudança de operadores lógicos em quaisquer estruturas (if-else, for, while, etc...)
PAR	Mudanças em parâmetros, geralmente trocando o parâmetro por '*null*'
RET	Mudanças no retorno do método, geralmente trocando o retorno por '*null*'
STR	Mudanças em *Strings*

gera modificações automaticamente para mutação de testes unitários e mutação de testes a nível de classe. μJava pode realizar testes em classes individuais ou em pacotes com múltiplas classes. Os testes são submetidos pelo usuário como uma sequência de chamadas de métodos para o SUT [26]. μJava foi utilizado para a geração das mutações e o procedimento realizado em [9] foi utilizado para a seleção de mutações específicas. Esse procedimento consiste na utilização do *framework* μ*Java* para gerar várias mutações em todas as classes do código para cada um dos sistemas apresentados na Tabela 4.2.4. Com as mutações geradas, a criação de uma versão consiste na duplicação do projeto *Java* e inserção manual da mutação gerada pelo μ*Java*.

É importante deixar claro que o μ*Java* pode gerar mutações com erro de compilação ou de execução, no entanto estes foram descartados, pois não apresentavam nenhuma utilidade para a nossa análise experimental. Cada amostra gerada faz par com uma versão base, que não resulta em nenhuma falha na regressão de testes (o PriorJ marca essa amostra como *'OLD'*, a fim de categorizar qual será a versão base) e a outra é a versão que contém as modificações.

4.2.5 Grupos de Amostras

Como objetos de estudo, foi considerada uma quantidade de 20 versões por projeto, o que resultaria em 80 pares de versões por grupo de amostras. Essa quantidade de versões foi escolhida para tentarmos generalizar os dados de um ambiente real. Porém, por possuir um código pequeno e limitado, em alguns grupos não foi possível atingir a quantidade considerada de versões para o *XMLMatchers*. Isso não impossibilitou a análise experimental das

técnicas propostas. Para uma melhor organização, as versões foram divididas em quatro grupos que podem ser visualizadas a seguir:

- **Grupo 01:** versões com uma única mudança inserida no projeto.
- **Grupo 02:** versões com várias mudanças inseridas em uma mesma classe do projeto.
- **Grupo 03:** versões com várias mudanças inseridas em várias classes de um mesmo pacote do projeto.
- **Grupo 04:** versões com várias mudanças inseridas em várias classes de vários pacotes diferentes do projeto.
- **Grupo 05:** conjunto de todas as versões citadas acima.

O Grupo 01 contém um total de 80 mudanças, cada uma inserida em uma das 80 versões (20 para cada projeto) e seus detalhes podem ser observados na Tabela 4.9. O Grupo 02 contém 235 mudanças subdivididas em 76 versões (20 versões para cada projeto, com exceção do *XMLMatchers* que engloba 16 delas), seus detalhes podem ser visualizados na Tabela 4.10. O Grupo 03 contém 565 mudanças subdivididas em 73 versões (20 versões para cada projeto, com exceção do *XMLMatchers* que engloba 13 delas), os detalhes podem ser analisados na Tabela 4.11. Por fim, apresenta-se o Grupo 04 com um total de 748 mudanças subdivididas em 75 versões (20 para cada projeto, com exceção do *XMLMatchers* que engloba 15 delas, pois o seu código é muito pequeno e limitado), seus detalhes podem ser observados na Tabela 4.12.

4.3 Resultados e Discussão

Nessa seção, são apresentados os resultados alcançados na realização do experimento. Eles foram gerados pela ferramenta *PriorJ* [32] e analisados com a ajuda da ferramenta *RStudio*, que é um software livre de ambiente de desenvolvimento integrado para R, uma linguagem de programação para gráficos e cálculos estatísticos. Na Seção 4.3.1 serão apresentados todos os gráficos necessários para a comparação entre as técnicas, eles estão subdivididos pelos grupos de amostras (apresentados na Seção 4.2.5). A Seção 4.3.2 objetiva demonstrar os resultados do Teste de Hipóteses para as amostras utilizadas nessa experimentação.

Tabela 4.8: Grupos de Amostras

Project	#Testes	#Versões	#Mudanças
Jbehave	540	20	20
Jmock	396	20	20
XML-Unit	283	20	20
XML-Matchers	66	20	20

Tabela 4.9: Grupo 01

Project	#Testes	#Versões	#Mudanças
Jbehave	540	20	67
Jmock	396	20	63
XML-Unit	283	20	65
XML-Matchers	66	16	40

Tabela 4.10: Grupo 02

Project	#Testes	#Versões	#Mudanças
Jbehave	540	20	167
Jmock	396	19	131
XML-Unit	283	20	220
XML-Matchers	66	13	47

Tabela 4.11: Grupo 03

Project	#Testes	#Versões	#Mudanças
Jbehave	540	20	239
Jmock	396	20	131
XML-Unit	283	20	281
XML-Matchers	66	15	97

Tabela 4.12: Grupo 04

4.3.1 Resultados Gerais

Neste trabalho, será considerado que a frequência onde cada técnica antecipou de forma mais eficiente o caso de teste impactado pela mudança. Ou seja, a altura do retângulo será o número de vezes que a técnica foi mais eficiente com relação as outras.

4.3.2 Teste de Hipóteses

Para avaliar o comportamento das utilizamos o teste de hipóteses [14]. O objetivo é aplicar esse teste nos grupos de amostras apresentados na Seção 4.2.5.

Testes de hipótese podem ser paramétricos ou não paramétricos. Os testes paramétricos baseiam-se em medidas intervalares da variável dependente (um parâmetro ou caraterística quantitativa de uma população) e a utilização deste tipo de testes exige que sejam cumpridos três pressupostos: distribuição normal, homogeneidade dos dados e variáveis intervalares e contínuas. Se pelo menos um desses pressupostos forem refutados, então será realizado um teste não paramétrico. Os testes não paramétricos quando comparados com os testes paramétricos, requerem menos pressupostos para as distribuições. Baseiam-se em dados ordinais e nominais e são muito úteis para a análise de testes de hipóteses; são também úteis para a análise de amostras grandes, em que os pressupostos paramétricos não se verifiquem, assim como para as amostras muito pequenas e para as investigações que envolvam hipóteses

Tabela 4.13: Representação das hipóteses para métrica APFD

Hipótese		Descrição	Grupo de Amostras
01	$H1_0$	Amostra com distribuições iguais.	Grupo 01
	$H1_1$	Amostra com distribuições diferentes.	
02	$H2_0$	Amostra com distribuições iguais.	Grupo 02
	$H2_1$	Amostra com distribuições diferentes.	
03	$H3_0$	Amostra com distribuições iguais.	Grupo 03
	$H3_1$	Amostra com distribuições diferentes.	
04	$H4_0$	Amostra com distribuições iguais.	Grupo 04
	$H4_1$	Amostra com distribuições diferentes.	
05	$H5_0$	Amostra com distribuições iguais.	Grupo 05
	$H5_1$	Amostra com distribuições diferentes.	

cujos processos de medida sejam ordinais.

Hipóteses

A seguir são listadas as hipóteses a serem testadas em relação as populações de dados apresentados na Seção 4.2.5. As hipóteses foram categorizadas pela métrica e todas elas apresentam as hipóteses nula H_0 e a alternativa H_1. Dessa forma, a Tabela 4.13 representa as hipóteses da APFD, a Tabela 4.14 representa as hipóteses da F-Measure, a Tabela 4.15 representa as hipóteses da F-Spreading, a Tabela 4.16 representa as hipóteses da Group-Measure, a Tabela 4.17 representa as hipóteses da Group-Spreading e a Tabela 4.18 representa as hipóteses do Tempo de Execução.

Normalidade dos dados

Em estatística, os testes de normalidade são usados para determinar se um conjunto de dados de uma dada variável aleatória. Pode ser bem modelada por uma distribuição normal ou não, ou para calcular a probabilidade da variável aleatória subjacente estar normalmente distribuída.

Para analisar a normalidade das distribuições das amostras, foi utilizado o teste de nor-

Tabela 4.14: Representação das hipóteses para métrica F-Measure

Hipótese		Descrição	Grupo de Amostras
06	$H6_0$	Amostra com distribuições iguais.	Grupo 01
	$H6_1$	Amostra com distribuições diferentes.	
07	$H2_0$	Amostra com distribuições iguais.	Grupo 02
	$H7_1$	Amostra com distribuições diferentes.	
08	$H8_0$	Amostra com distribuições iguais.	Grupo 03
	$H8_1$	Amostra com distribuições diferentes.	
09	$H9_0$	Amostra com distribuições iguais.	Grupo 04
	$H9_1$	Amostra com distribuições diferentes.	
10	$H10_0$	Amostra com distribuições iguais.	Grupo 05
	$H10_1$	Amostra com distribuições diferentes.	

Tabela 4.15: Representação das hipóteses para métrica F-Spreading

Hipótese		Descrição	Grupo de Amostras
11	$H11_0$	Amostra com distribuições iguais.	Grupo 01
	$H11_1$	Amostra com distribuições diferentes.	
12	$H12_0$	Amostra com distribuições iguais.	Grupo 02
	$H12_1$	Amostra com distribuições diferentes.	
13	$H13_0$	Amostra com distribuições iguais.	Grupo 03
	$H13_1$	Amostra com distribuições diferentes.	
14	$H14_0$	Amostra com distribuições iguais.	Grupo 04
	$H14_1$	Amostra com distribuições diferentes.	
15	$H15_0$	Amostra com distribuições iguais.	Grupo 05
	$H15_1$	Amostra com distribuições diferentes.	

Tabela 4.16: Representação das hipóteses para métrica Group-Measure

Hipótese		Descrição	Grupo de Amostras
16	$H16_0$	Amostra com distribuições iguais.	Grupo 02
	$H16_1$	Amostra com distribuições diferentes.	
17	$H17_0$	Amostra com distribuições iguais.	Grupo 03
	$H17_1$	Amostra com distribuições diferentes.	
18	$H18_0$	Amostra com distribuições iguais.	Grupo 04
	$H18_1$	Amostra com distribuições diferentes.	
19	$H19_0$	Amostra com distribuições iguais.	Grupo 05
	$H19_1$	Amostra com distribuições diferentes.	

Tabela 4.17: Representação das hipóteses para métrica Group-Spreading

Hipótese		Descrição	Grupo de Amostras
20	$H20_0$	Amostra com distribuições iguais.	Grupo 02
	$H20_1$	Amostra com distribuições diferentes.	
21	$H21_0$	Amostra com distribuições iguais.	Grupo 03
	$H21_1$	Amostra com distribuições diferentes.	
22	$H22_0$	Amostra com distribuições iguais.	Grupo 04
	$H22_1$	Amostra com distribuições diferentes.	
23	$H23_0$	Amostra com distribuições iguais.	Grupo 05
	$H23_1$	Amostra com distribuições diferentes.	

Tabela 4.18: Representação das hipóteses para métrica Tempo de Execução

Hipótese		Descrição	Grupo de Amostras
24	$H24_0$	Amostra com distribuições iguais.	Grupo 01
	$H24_1$	Amostra com distribuições diferentes.	
25	$H25_0$	Amostra com distribuições iguais.	Grupo 02
	$H25_1$	Amostra com distribuições diferentes.	
26	$H26_0$	Amostra com distribuições iguais.	Grupo 03
	$H26_1$	Amostra com distribuições diferentes.	
27	$H27_0$	Amostra com distribuições iguais.	Grupo 04
	$H27_1$	Amostra com distribuições diferentes.	
28	$H28_0$	Amostra com distribuições iguais.	Grupo 05
	$H28_1$	Amostra com distribuições diferentes.	

malidade *Shapiro-Wilk*, proposto em 1965. As hipóteses nula e alternativa de cada grupo de dados são representadas por H_0 e H_1, respectivamente. Como visualizamos, a Tabelas 4.13 mostra a normalidade dos dados para métrica *APFD*; a Tabela 4.14 mostra a normalidade para a métrica *F-Measure*; a Tabela 4.15 mostra a normalidade para a métrica *F-Spreading*; a Tabela 4.16 mostra a normalidade para a métrica *Group-Measure*; a Tabela 4.17 mostra a normalidade para a métrica *Group-Spreading*; e a Tabela 4.18 mostra a normalidade para a métrica *Tempo de Execução*. A H_0 significa afirmar que os dados seguem uma distribuição normal, em contrapartida, a H_1 significa afirmar que os dados não seguem uma distribuição normal. Logo, rejeitar a H_0 é o mesmo que afirmar que os dados não são normais e vice-versa.

O valor da probabilidade *W* listado na saída é o *p-value*. O limiar escolhido é de 0,05 para a análise que foi realizada, então, se o valor resultante p é menor que 0,05, a hipótese nula de que os dados são normalmente distribuídas é rejeitada. Se o valor p é superior a 0,05, a hipótese nula não é rejeitada. Nos resultados apresentados nas Tabelas 4.19, 4.20, 4.21, 4.22, 4.23 e 4.24, a hipótese rejeitada será marcada com um 'X' vermelho. Entre os valores de *p-value* apresentados, é possível observar a presença do valor $2.2e-16$, isso significa que o valor é o menor número maior do que 0 que pode ser armazenado pelo sistema de ponto

Tabela 4.19: Teste de Normalidade Shapiro-Wilk (*p-value*) para métrica APFD

Hipótese		Rejeitada	CB	RND	WCS-T	WCS-A
01	H1_0 H1_1	X	4.276e-14	2.039e-07	1.567e-14	1.709e-14
02	H2_0 H2_1	X	5.14e-16	2.669e-08	1.203e-15	1.099e-15
03	H3_0 H3_1	X	8.332e-14	4.647e-11	3.133e-14	< 2.2e-16
04	H4_0 H4_1	X	4.823e-16	1.039e-11	< 2.2e-16	< 2.2e-16
05	H5_0 H5_1	X	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16

flutuante de um computador pessoal. A representação $e-16$ significa uma multiplicação com uma potência de 10 elevado a -16 ($*10^{-16}$).

4.3.3 Resultados do Teste de Hipóteses

Como é possível perceber nas Tabelas 4.13, 4.14, 4.15, 4.16, 4.17, 4.18 e *Tempo de Execução*, a H_0 foi rejeitada para todas as hipóteses. Logo, um dos pressupostos para a utilização de testes paramétricos foi refutada. Como consequência, faz-se necessária a utilização de um teste não paramétrico para a obtenção correta dos resultados. O teste escolhido foi o *Kruskal-Wallis (KW)*, que é uma extensão do teste de *Wilcoxon-Mann-Whitney* [43]. É um teste não paramétrico utilizado para comparar dois ou mais tratamentos. Ele é usado para testar a hipótese nula de que todas as populações possuem funções de distribuição iguais contra a hipótese alternativa de que ao menos duas das populações possuem funções de distribuição diferentes. Os resultados estatísticos (*p-value* para o teste de *Kruskall-Walis*), que são detalhados nas Tabelas 4.25, 4.26, 4.27, 4.28, 4.29, 4.30. Como é possível observar, as hipóteses nulas foram rejeitadas para todas as hipóteses. Concluímos, assim, que pelo menos uma das técnicas possui um comportamento diferente das demais.

Tabela 4.20: Teste de Normalidade Shapiro-Wilk (*p-value*) para métrica F-Measure

Hipótese		Rejeitada	CB	RND	WCS-T	WCS-A
06	H6_0 H6_1	X	3.287e-14	9.744e-09	< 2.2e-16	< 2.2e-16
07	H7_0 H7_1	X	< 2.2e-16	9.778e-11	2.83e-16	< 2.2e-16
08	H8_0 H8_1	X	4.282e-16	2.387e-09	1.873e-12	5.869e-15
09	H9_0 H9_1	X	3.723e-16	2.603e-10	< 2.2e-16	< 2.2e-16
10	H10_0 H10_1	X	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16

Tabela 4.21: Teste de Normalidade Shapiro-Wilk (*p-value*) para métrica F-Spreading

Hipótese		Rejeitada	CB	RND	WCS-T	WCS-A
11	H11_0 H11_1	X	5.709e-09	< 2.2e-16	1.677e-09	1.398e-12
12	H12_0 H12_1	X	1.163e-10	0.0001218	6.005e-10	1.174e-09
13	H13_0 H13_1	X	1.811e-07	4.153e-08	1.326e-07	2.1e-07
14	H14_0 H14_1	X	0.0001326	7.615e-09	0.001632	0.000257
15	H15_0 H15_1	X	5.323e-14	1.355e-13	2.844e-13	2.706e-13

Tabela 4.22: Teste de Normalidade Shapiro-Wilk (*p-value*) para métrica Group-Measure

Hipótese		Rejeitada	CB	RND	WCS-T	WCS-A
16	H16_0 H16_1	X	< 2.2e-16	1.28e-12	< 2.2e-16	< 2.2e-16
17	H17_0 H17_1	X	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
18	H18_0 H18_1	X	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
19	H19_0 H19_1	X	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16

Tabela 4.23: Teste de Normalidade Shapiro-Wilk (*p-value*) para métrica Group-Spreading

Hipótese		Rejeitada	CB	RND	WCS-T	WCS-A
20	H20_0 H20_1	X	1.206e-12	3.659e-06	5.59e-13	1.321e-13
21	21_0 H21_1	X	9.295e-09	0.001124	5.44e-15	1.981e-14
22	H22_0 H22_1	X	9.663e-10	2.086e-12	6.297e-08	0.005289
23	H23_0 H23_1	X	< 2.2e-16	5.506e-08	< 2.2e-16	< 2.2e-16

Tabela 4.24: Teste de Normalidade Shapiro-Wilk (*p-value*) para métrica Tempo de Execução

Hipótese		Rejeitada	CB	RND	WCS-T	WCS-A
24	H24_0 H24_1	X	5.323e-14	1.355e-13	2.844e-13	2.706e-13
25	H25_0 H25_1	X	9.272e-07	< 2.2e-16	1.492e-06	5.939e-10
26	H26_0 H26_1	X	4.799e-11	< 2.2e-16	2.499e-08	2.438e-10
27	H27_0 H27_1	X	4.888e-10	< 2.2e-16	1.939e-10	1.305e-11
28	H28_0 H28_1	X	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16

Como apresentado nas Tabelas 4.25, 4.26, 4.27, 4.28, 4.29, 4.30, o teste de *kruskal-wallis* rejeitou a hipótese nula para todas as populações de dados das 28 hipóteses. Com este resultado afirma-se que para cada hipótese existe pelo menos uma técnica (dentre *CB, RND, WCS-T e WCS-A*) cujo os resultados diferem. Após a realização da análise de variância, testes *posthoc* são necessários para identificar quais dos pares de grupos diferem. A hipótese testada é sempre a hipótese nula (a de que não há diferença entre os grupos), logo, H_0 é que não há diferença entre os grupos, enquanto que H_1 afirma que há diferença entre os grupos (para um *p-value* < 0.05). Para a realização desse teste foi utilizado o método *posthoc.kruskal.conover.test*[6] que pode ser encontrado no pacote *PMCMR* da linguagem R [7]. As Tabelas 4.31, 4.33, 4.34, 4.35, 4.36 e 4.37 mostram os resultados do teste *posthoc* de pares a fim de descobrir quais populações de dados diferem entre si.

Como é possível observar, a Tabela 4.31 mostra as comparações entre as distribuições para a métrica *APFD*; a Tabela 4.33 mostra as comparações entre as distribuições para a métrica *F-Measure*; a Tabela 4.34 mostra as comparações entre as distribuições para a métrica *F-Spreading*; a Tabela 4.35 mostra as comparações entre as distribuições para a métrica *Group-Measure*; a Tabela 4.36 mostra as comparações entre as distribuições para a métrica

[6]https://cran.r-project.org/web/packages/PMCMR/PMCMR.pdf

[7]https://www.r-project.org/

	Hipótese	Rejeitada	*Kruskal-wallis (p-value)*
01	H1_0 H1_1	X	2.881e-12
02	H2_0 H2_1	X	< 2.2e-16
03	H3_0 H3_1	X	< 2.2e-16
04	H4_0 H4_1	X	1.374e-14
05	H5_0 H5_1	X	< 2.2e-16

Tabela 4.25: Teste Estatístico Kruskal-Wallis (*p-value*) para métrica APFD

	Hipótese	Rejeitada	*Kruskal-wallis (p-value)*
11	H11_0 H11_1	X	< 2.2e-16
12	H12_0 H12_1	X	< 2.2e-16
13	H13_0 H13_1	X	6.453e-14
14	H14_0 H14_1	X	< 2.2e-16
15	H15_0 H15_1	X	< 2.2e-16

Tabela 4.27: Teste Estatístico Kruskal-Wallis para métrica F-Spreading

	Hipótese	Rejeitada	*Kruskal-wallis (p-value)*
20	H20_0 H20_1	X	1.244e-09
21	H21_0 H21_1	X	< 2.2e-16
22	H22_0 H22_1	X	< 2.2e-16
23	H23_0 H23_1	X	< 2.2e-16

Tabela 4.29: Teste Estatístico Kruskal-Wallis para métrica Group-Spreading

	Hipótese	Rejeitada	*Kruskal-wallis (p-value)*
06	H6_0 H6_1	X	< 2.2e-16
07	H7_0 H7_1	X	< 2.2e-16
08	H8_0 H8_1	X	< 2.2e-16
09	H9_0 H9_1	X	< 2.2e-16
10	H10_0 H10_1	X	< 2.2e-16

Tabela 4.26: Teste Estatístico Kruskal-Wallis (*p-value*) para métrica F-Measure

	Hipótese	Rejeitada	*Kruskal-wallis (p-value)*
16	H16_0 H16_1	X	< 2.2e-16
17	H17_0 H17_1	X	< 2.2e-16
18	H18_0 H18_1	X	< 2.2e-16
19	H19_0 H19_1	X	< 2.2e-16

Tabela 4.28: Teste Estatístico Kruskal-Wallis para métrica Group-Measure

	Hipótese	Rejeitada	*Kruskal-wallis (p-value)*
24	H24_0 H24_1	X	< 2.2e-16
25	H25_0 H25_1	X	< 2.2e-16
26	H26_0 H26_1	X	< 2.2e-16
27	H27_0 H27_1	X	< 2.2e-16
28	28_0 H28_1	X	< 2.2e-16

Tabela 4.30: Teste Estatístico Kruskal-Wallis para métrica Tempo de Execução

Group-Spreading; e a Tabela 4.37 mostra as comparações entre as distribuições para a métrica *Tempo*. Nessas tabelas, todas as células que estiverem com fundo na cor cinza, significa que a hipótese nula foi rejeitada e que os dados diferem de forma estatística.

A correção de Bonferroni é uma correção de múltipla comparação usada quando vários testes estatísticos independentes estão sendo realizadas simultaneamente (uma vez que, enquanto um determinado valor *Alpha* Valor pode ser apropriado para cada comparação individual, não é para o conjunto de todas as comparações). A fim de evitar um grande número de positivos falsos, o valor de alfa deve ser reduzido para ter em conta o número de comparações a ser executada [5].

4.3.4 Frequência de Vitórias

Os histogramas apresentados na Figura 4.1 representam a frequência com que uma mudança, capturada por um CT que falha, ocupa uma posição superior na lista ordenada, perante as demais (*RND e CB*). Em caso de empates na liderança, todas as técnicas são pontuadas. A Figura 4.1 apresenta cinco gráficos, cada um representa a frequência de vitórias para um dos grupos de amostras apresentados na Seção 4.2.5.

O gráficos *(a), (b), (c)* da Figura 4.1 apresentam uma leve diferença na frequência de vitórias. A técnica *WCS-A* se superou na frequência de vitórias, com relação as demais. O gráfico *(d)* da Figura 4.1 mostra que novamente a técnica *WCS-A* apresentou superioridade bem maior que as demonstradas anteriormente. A *WCS-T* obteve uma leve superioridade com relação a *CB*. A fim de confirmar a superioridade das técnicas propostas, o gráfico *(e)* da Figura 4.1 demonstra a quantidade de vezes que cada CT ocupou uma posição superior na lista ordenada para o Grupo 05. Esse histograma constata que as técnicas *WCS-T e WCS-A* são superiores à *CB e RND* para o conjunto de todas as amostras geradas.

4.3.5 PP01: Existe diferença entre as técnicas para a métrica *APFD*?

Para responder a pergunta de pesquisa 01 será necessário analisar os gráficos apresentados nas Figuras 4.2, 4.1 e nas Tabelas 4.19 e 4.25. Para este último, foram propostas as hipóteses 01, 02, 03, 04 e 05 apresentadas na Tabela 4.13. O objetivo principal desse questionamento é descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção

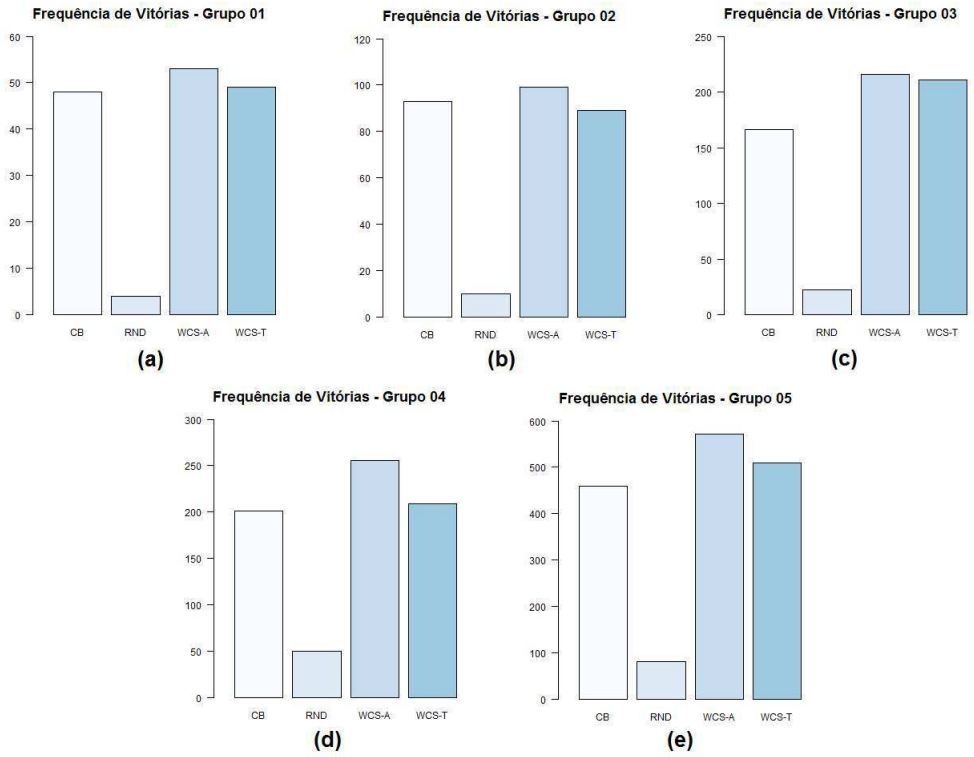

Figura 4.1: Histogramas para todos os grupos de amostras.

de defeitos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *APFD*.

A fim de descobrir quais técnicas diferem entre si através dos seus resultados, a Tabela 4.31 mostra o resultado do teste *posthoc* para as quatro técnicas. Inferindo as hipóteses 01, 02 e 03 na comparação entre as distribuições das técnicas *WCS-T e WCS-A* com a técnica de controle *RND*, o *p-value* é menor que 0.05, confirmando que as distribuições são diferentes. Observando os gráficos *(a), (b) e (c)* da Figura 4.2, percebe-se a diferença de eficiência entre as propostas e a *RND*, apontando as nossas técnicas com um comportamento superior. Comparando as distribuições dos dados da *WCS-T e WCS-A* com *CB*, para as hipóteses 01, 02 e 03, tem-se como resultado um *p-value* maior que 0.05. Dessa forma, afirma-se que não existe diferença estatística entre suas distribuições.

A fim de deduzir a hipótese 04 ao comparar as técnicas *WCS-A e CB*, é necessário analisar o gráfico *(d)* da Figura 4.2. Nota-se que a variação dos dados são menores. Por isso os *boxplots* para as técnicas propostas encontram mais achatados. Ainda observa-se que a variação da *WCS-A* é menor que a *WCS-T*. Quando se compara com a *RND*, a superioridade

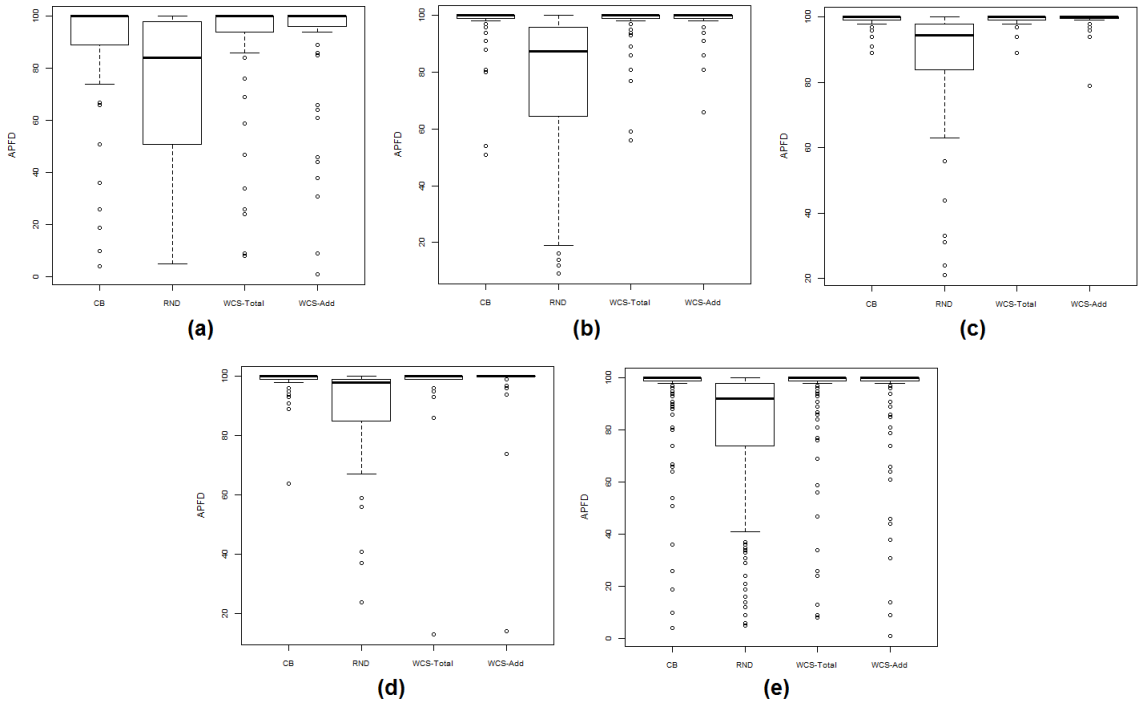

Figura 4.2: Boxplots para a métrica APFD.

é evidente. A hipótese 04 da Tabela 4.31 confirma esses dados visuais. Ao se comparar *WCS-T e WCS-A* com *CB*, afirma-se que a *WCS-T*, apesar de se mostrar superior visual, não apresenta diferença nos dados (*p-value* > *0.05*, porém, o *p-value* comprova que os dados da *WCS-A* diferem da *CB*. Assim, podemos afirmar que a estratégia *additional* da técnica *WCS* é superior.

Para inferir a hipótese 05, realiza-se a análise dos dados do grupo 05 referentes aos resultados da métrica *APFD*. Observa-se no gráfico *(e)* da Figura 4.2 que, com exceção da *RND*, todas as técnicas mantiveram uma média semelhante. Como não é possível afirmar

Tabela 4.31: Teste Estatístico *PostHoc* Kruskal-Wallis para métrica APFD

Hipótese	**WCS-T x RND**	**WCS-T x CB**	**WCS-A x RND**	**WCS-A x CB**
01	1.9e-09	1.0000	2.9e-11	1.0000
02	< 2e-16	1.0000	< 2e-16	1.0000
03	< 2e-16	1.0000	< 2e-16	0.63
04	1.1e-11	0.446	1.8e-15	0.013
05	< 2e-16	1.0000	< 2e-16	0.017

Tabela 4.32: Sumário estatístico do Grupo de amostras 05 para a métrica APFD.

Técnica	Mínimo	1º Quartil	Mediana	Média	3º Quartil	Máximo
CB	4.00	99.00	100.00	95.91	100.00	100.00
WCS-A	1.00	99.00	100.00	96.29	100.00	100.00

nada através do gráfico, a comparação entre os dados é demonstrado na hipótese 05 da Tabela 4.31. Nela, percebe-se que as técnicas *WCS-T e WCS-A* mantiveram-se com os resultados superiores a *RND*. Quando comparadas com a *CB*, afirma-se que a *WCS-T* não difere dos seus dados, pois o *p-value* é maior que 0.05; confirma-se também que os dados da *WCS-A* diferem da *CB*, pois o *p-value* é menor que 0.05, mas não podemos afirmar qual é superior. Para tanto, a Tabela 4.32 mostra que a média da *WCS-A* é superior a *CB*, constatando a sua superioridade.

4.3.6 PP02: Existe diferença entre as técnicas para a métrica *F-Measure*?

Para responder a pergunta de pesquisa 02 será necessário analisar os gráficos apresentados nas Figuras 4.1, 4.3, nas Tabelas 4.20 e 4.26. Para este último, foram propostas as hipóteses 06, 07, 08, 09 e 10 apresentadas na Tabela 4.14. O objetivo principal desse questionamento é descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *F-Measure*.

Para inferirmos as hipóteses 06, 07, 08 vamos observar os gráficos *(a), (b) e (c)* da Figura 4.3. Nelas percebemos que os *boxplots* da *WCS-T e WCS-A* se apresentam levemente mais achatados do que o da *CB*, inclusive, com uma quantidade menor *outliers*, que se encontram mais agrupados. Apesar dos resultados visuais positivos, a Tabela 4.33 mostra que as técnicas propostas, quando comparadas com a *RND*, são superiores, pois os *p-values* apresentados nas hipóteses 06, 07 e 08 são menores que o limiar de 0.05. Quando comparadas a *CB*, nota-se que os dados da *WCS-T* não se diferenciam da *CB* e, o *p-value* da *WCS-A* por pouco não se diferencia. Assim, podemos afirmar que não houve nenhum impacto negativo nas modificações aplicadas na *CB* para essas três hipóteses.

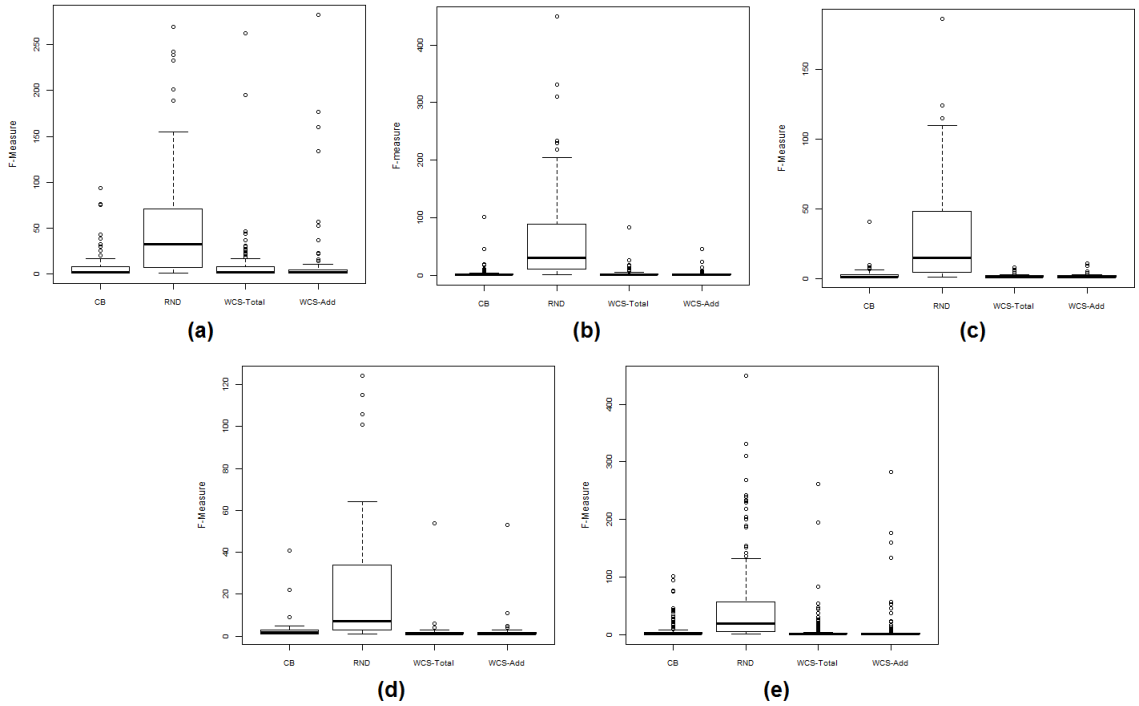

Figura 4.3: Boxplots para a métrica *F-Measure*.

A fim de deduzir a hipótese 09 ao comparar as técnicas *WCS-A e CB*, é necessário analisar o gráfico *(d)* da Figura 4.3. Nota-se que a variação dos dados são menores. Por isso os *boxplots* para as técnicas propostas são levemente mais achatados. Ainda observa-se que a variação da *WCS-A* é menor que a *WCS-T*. Quando se compara com a *RND*, a superioridade é evidente. A hipótese 09 da Tabela 4.33 confirma esses dados visuais. Ao se comparar *WCS-T e WCS-A* com *CB*, afirma-se que a *WCS-T*, apesar de se mostrar superior visual, não apresenta diferença nos dados (*p-value* > *0.05*, porém, o *p-value* comprova que os dados da *WCS-A* diferem da *CB*. Assim, podemos afirmar que a estratégia *additional* da técnica

Tabela 4.33: Teste Estatístico *PostHoc* Kruskal-Wallis para métrica F-Measure

Hipótese	WCS-T x RND	WCS-T x CB	WCS-A x RND	WCS-A x CB
06	1.9e-09	1.0000	2.9e-11	1.0000
07	< 2e-16	1.0000	< 2e-16	1.0000
08	< 2e-16	1.0000	< 2e-16	0.62
09	1e-13	0.676	< 2e-16	0.015
10	< 2e-16	1.0000	< 2e-16	0.066

WCS é superior. Acredita-se que essa superioridade da *WCS-A* se dá pelo fato de além da estratégia *additional*, também é aplicada a técnica *WCS*. Observando os *outliers* ainda do gráfico *(d)*, é notório o nível maior de agrupamento das amostras, tanto da *WCS-T*, quanto da *WCS-A*, quando comparados a *CB*. O *outlier* mais afastado ocorre pelo fato dos decrementos para CTs que cobrem mudanças previamente cobertas, que são aplicados pela técnica *WCS*.

Para inferir a hipótese 10, realiza-se a análise dos dados do grupo 05 referentes aos resultados da métrica *F-Measure*. Observa-se no gráfico *(e)* da Figura 4.3 que todas as técnicas baseadas em mudanças são superiores a *RND*, e que as técnicas propostas apresentam-se mais achatadas que a *CB*. Porém, seus *outliers* estão mais espaçados, como é esperado, baseado no comportamento da técnica *WCS*. Dessa forma, A hipótese 10 da Tabela 4.33 confirma, com todos os *p-values* comparados a *RND* menores que 0.05 e, com todos os *p-values* comparados a *CB* maiores que 0.05. Para os dados gerais, não afirma-se que as técnicas *WCS-T e WCS-A* são superiores, mas constata-se que não houve impacto negativo.

Para todos os grupos de amostras, a métrica *F-measure* baseia-se principalmente na posição do primeiro CT, como as técnicas propostas e a *CB* são semelhantes, a expectativa para os resultados dessas métricas é que sejam equivalentes. Como é possível observar nos *boxplots*, as técnicas propostas se apresentam um pouco mais achatados que as técnicas concorrentes. Assim, é impossível afirmar que o resultado obtido é superior, porém, a alegação de que a mudança aplicada à técnica não obteve nenhum impacto estatístico negativo é verdadeira. O fato de os *boxplots* da Figura 4.3 estarem mais achatados do que a *CB e RND*, significa que os CTs que falham e cobrem as mudanças encontram-se mais agrupados. Nos gráficos *(d) e (e)* da Figura 4.3 também observa-se que a *WCS-T* tem dois *outliers* bem distantes, se comparado a *CB*. Isso se dá pelo fato de CTs que cobrem mudanças previamente cobertas por outros CTs são desfavorecidos na lista priorizada. O mesmo acontece na técnica *WCS-A*.

4.3.7 PP03: Existe diferença entre as técnicas para a métrica *F-Spreading*?

Para responder a pergunta de pesquisa 03 será necessário analisar os gráficos apresentados nas Figuras 4.1, 4.4, nas Tabelas 4.21 e 4.27. Para este último, foram propostas as hipóteses 11, 12, 13, 14 e 15 apresentadas na Tabela 4.15. O objetivo principal desse questionamento é

descobrir se as abordagens *total* e *additional* das técnicas propostas agrupam melhor casos de teste que cobrem defeitos que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *F-Spreading*.

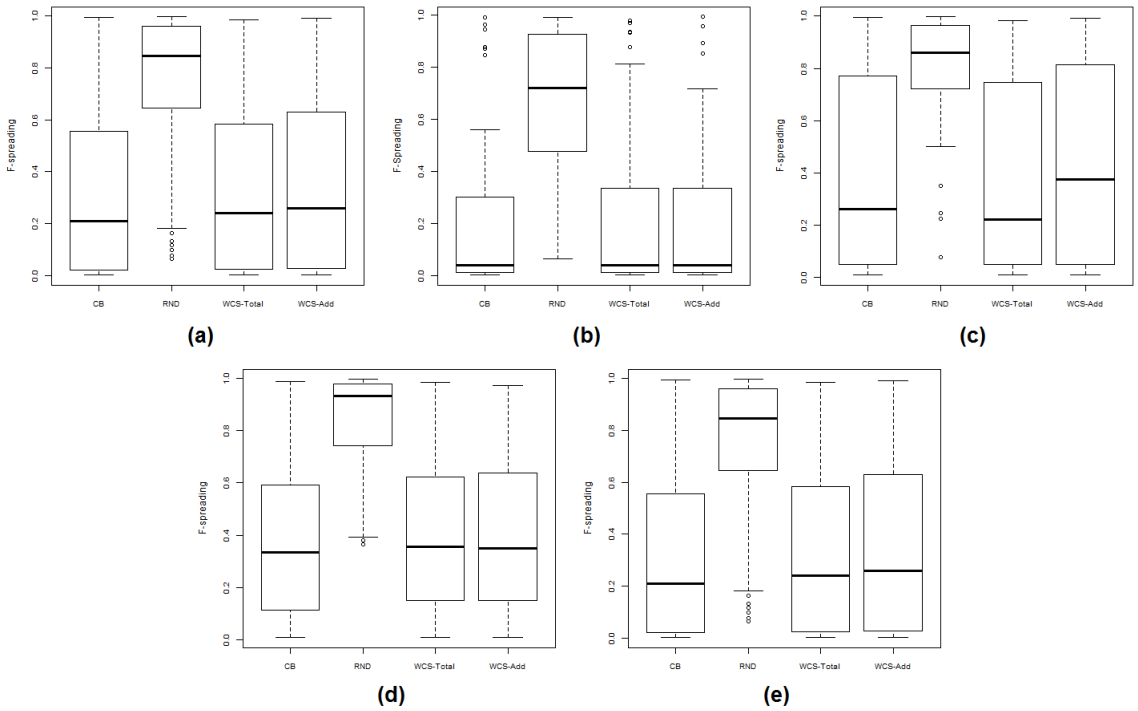

Figura 4.4: Boxplots para a métrica *F-Spreading*.

O gráfico *(a)* da Figura 4.4 mostra que os CTs priorizados pelas técnicas propostas mantiveram o nível de agrupamento, quando comparados a *CB* e foram superiores a técnica *RND*, o que é comprovado nos resultados da hipótese 11 da Tabela 4.33.

Com relação a métrica hipótese 12, afirma-se que os resultados para as propostas também se mantiveram com a mesma média da *CB*, porém, elas obtiveram uma variação maior, o que explica os *boxplots* apresentados no gráfico *(b)* da Figura 4.4. Quando comparado a *RND*, a superioridade é visível. Esses dados são comprovados na Tabela 4.33, hipótese 12. Essa variação é esperada para as 4 técnicas, pois existem CTs que falham, mas não capturam nenhuma mudança, fazendo com que os mesmos sejam priorizados de forma aleatória. Isso explica o porque das caixas se encontrarem mais amplas.

No gráfico *(c)* da Figura 4.3, espera-se que os CTs que falham encontrem-se com um nível de agrupamento menor, já que a técnica *WCS* tende a espalhar os CTs que cobrem mudanças iguais. Como observa-se, o nível de agrupamento manteve-se com a mesma média

Tabela 4.34: Teste Estatístico *PostHoc* Kruskal-Wallis para métrica F-Spreading

Hipótese	WCS-T x RND	WCS-T x CB	WCS-A x RND	WCS-A x CB
11	4.6e-14	1.0000	7.3e-16	1.0000
12	< 2e-16	1.0000	< 2e-16	1.0000
13	2.9e-12	1.0000	9.7e-11	1.0000
14	< 2e-16	1.0000	< 2e-16	1.0000
15	< 2e-16	1.0000	< 2e-16	1.0000

para as técnicas robustas e todas superiores à *RND*. A hipótese 13 da Tabela 4.33 comprova exatamente os resultados apresentados no gráfico, as distribuições das propostas são superiores à técnica *RND* ($p\text{-}value < 0.05$) e semelhantes à *CB* ($p\text{-}value > 0.05$).

Segundo o gráfico *(d)* da Figura 4.3, o agrupamento dos CTs que falham mantiveram a mesma média. Além do mais, todas as técnicas baseadas em mudanças apresentam-se superiores a *RND*. Certificando os diagramas visuais, a hipótese 14 da Tabela 4.33 mostra a comparação entre as técnicas e que os dados diferem apenas da *RND*.

Como visualiza-se no gráfico *(e)* da Figura 4.3, os resultados para a hipótese 15 são equivalentes aos resultados da hipótese 14 discutidos no parágrafo anterior. Como constatado na hipótese 15 da Tabela 4.33, as distribuições das técnicas propostas não diferem da *CB*, mas são superiores à técnica *RND*.

4.3.8 PP04: Existe diferença entre as técnicas para a métrica *Group-Measure*?

Para responder a pergunta de pesquisa 04 será necessário analisar os gráficos apresentados nas Figuras 4.1, 4.5, nas Tabelas 4.22 e 4.28. Para este último, foram propostas as hipóteses 16, 17, 18 e 19 apresentadas na Tabela 4.16. O objetivo principal desse questionamento descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos inéditos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *Group-Measure*.

Equivalente a *F-measure*, temos os resultados para a métrica *Group-measure*. Ela mostra os resultados aplicados apenas aos casos de testes que capturaram as mudanças. Como

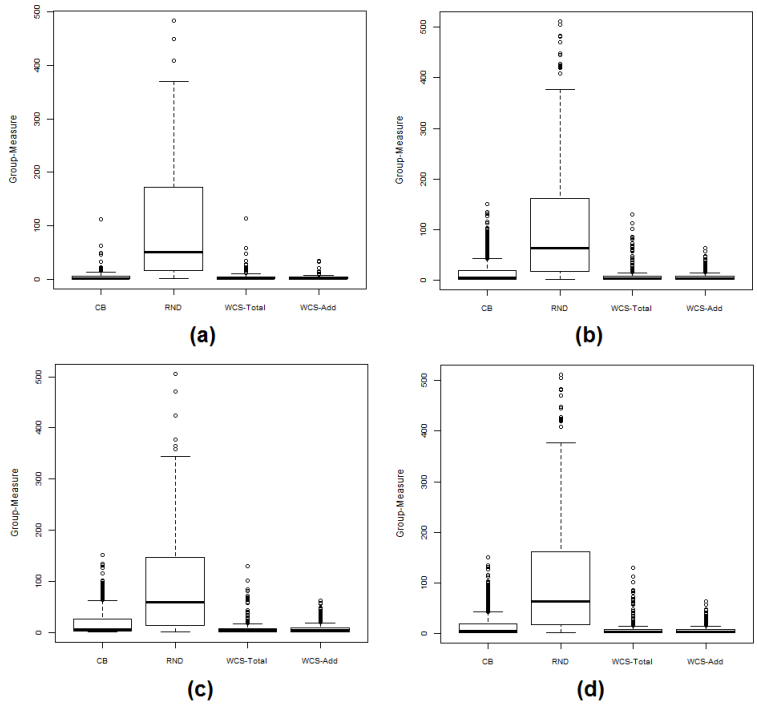

Figura 4.5: Boxplots para a métrica *Group-Measure*.

mostra o gráfico (a) da Figura 4.5, houve um pequeno ganho na distribuição das técnicas propostas, principalmente quando observamos o agrupamento dos *outliers*, que se mostra superior, com relação as técnicas *RND e CB*. Porém, como mostram os resultados da Tabela 4.35 para a hipótese 16, esse pequeno ganho tem nenhuma diferença estatística quando comparado à *CB*, porém, se mostra superior à *RND*.

Analisando o gráfico *(b)* da Figura 4.5, podemos verificar que as caixas para as técnicas propostas também se mostraram mais achatadas do que a *CB e RND*, inclusive, os *outliers* estão com um nível de agrupamento maior. Isso acontece pelo fato de a técnica *WCS* estar sendo aplicada para o contexto em que foi previamente idealizado, de várias mutações inseridas no código. Como esperado, a hipótese 17 comprova que os dados da *WCS-T e WCS-A* diferem da *RND* ($p\text{-values} < 0.05$) e, reafirmando o que as distribuições também diferem da *CB* ($p\text{-values} < 0.05$).

Observando o gráfico *(c)* da Figura 4.5, afirma-se claramente que as técnicas *WCS-T e WCS-A* são superiores a *CB e RND*. Inclusive, os *outliers* para as duas técnicas propostas apresentam-se com um nível de agrupamento maior que a *CB*, afirmando que os CTs que cobrem mudanças inéditas estão apresentando uma percentagem maior de prioridade. Com-

Tabela 4.35: Teste Estatístico *PostHoc* Kruskal-Wallis para métrica Group-Measure

Hipótese	WCS-T x RND	WCS-T x CB	WCS-A x RND	WCS-A x CB
16	< 2e-16	1.0000	< 2e-16	1.0000
17	< 2e-16	2.4e-07	< 2e-16	1.5e-06
18	< 2e-16	3.8e-16	< 2e-16	4.9e-14
19	< 2e-16	< 2e-16	< 2e-16	< 2e-16

parando as técnicas na hipótese 18 da Tabela 4.35, as propostas obtiveram *p-value < 0.05* em todas as comparações, reafirmando que as distribuições dos dados se diferem estatisticamente. Dessa forma, afirma-se que as propostas superaram a técnica *CB* para um contexto de inúmeras mutações em classes de pacotes diferentes (Grupo 04).

Mesmo para o conjunto total de dados (Grupo 05), é notória a diferença entre os resultados visuais apresentados no gráfico *(d)* 4.5, tanto da *WCS-T*, quanto da *WCS-A*, com relação as técnicas *CB e RND*. Confirmando essa diferença, a hipótese 19 da Tabela 4.35 prova que as distribuições dos dados diferem para todas as comparações, já que o *p-value* é menor que 0.05 para todos. Dessa forma, podemos garantir que na lista priorizada das técnicas propostas, os CTs que cobrem mudanças ocupam as mesmas posições, ou até melhores, que as técnicas *RND e CB*.

4.3.9 PP05: Existe diferença entre as técnicas para a métrica *Group-Spreading*?

Para responder a pergunta de pesquisa 05 será necessário analisar os gráficos apresentados nas Figuras 4.1, 4.6, nas Tabelas 4.23 e 4.29. Para este último, foram propostas as hipóteses 20, 21, 22 e 23 apresentadas na Tabela 4.17. O objetivo principal desse questionamento é descobrir se as abordagens *total* e *additional* das técnicas propostas agrupam melhor casos de teste que cobrem defeitos que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *Group-Spreading*.

No gráfico *(a)* da Figura 4.6, os *outliers* se mantiveram aproximadamente no mesmo padrão. Com isso, observa-se que o agrupamento dos CTs que cobrem as mudanças se mantiveram semelhantes para todas as técnicas, com exceção da *RND*. Essa afirmação é

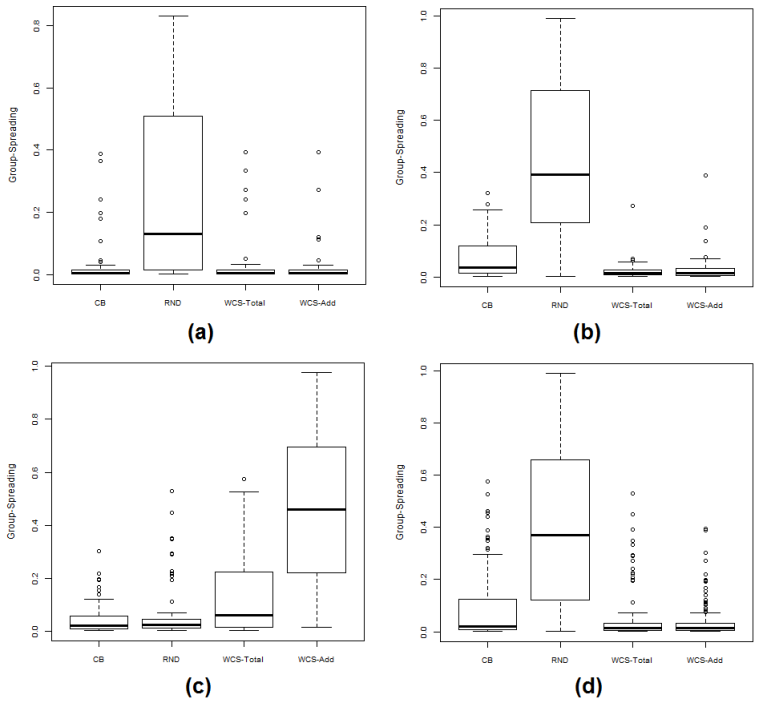

Figura 4.6: Boxplots para a métrica *Group-Spreading*.

confirmada pelos dados da hipótese 20 na Tabela 4.36.

Para inferir a hipótese 21, é possível observar no gráfico *(b)* da Figura que um número maior de mutações são agrupadas no topo da lista ordenada. Visualizando o gráfico da Figura 4.6. Observa-se também um achatamento considerável nas caixas das técnicas propostas, quando comparadas a *RND e CB*. Os *outliers* representam contextos onde CTs cobrem mutações conhecidas (previamente cobertas por CTs de posições superiores na lista priorizada). Esse contexto faz com que o nível de agrupamento diminua, aumentando assim o valor resultante da métrica. A hipótese 21 da Tabela 4.35 justifica a conclusão tirada do gráfico, pois as distribuições dos dados da *WCS-T e WCS-A* diferem da *RND* e da *CB* (*p-values* < 0.05).

O gráfico *(c)* da Figura 4.6 confirma as expectativas de que o nível de agrupamento dos CTs iria diminuir. Esse comportamento ocorre pelo fato do decremento aplicado aos CTs que cobrem mudanças conhecidas. A hipótese 22 da Tabela 4.36 confirma essa diferença, inclusive, a *RND*, nesse contexto específico (Grupo 04 de amostras), conseguiu ser superior as técnicas propostas. O *p-value* para todas as comparações são menores que 0.05, reafirmando a diferença entre os dados.

As distribuições dos dados da hipótese 23 mostra-se divergentes das demais, represen-

Tabela 4.36: Teste Estatístico *PostHoc* Kruskal-Wallis para métrica Group-Spreading

Hipótese	WCS-T x RND	WCS-T x CB	WCS-A x RND	WCS-A x CB
20	4.0e-08	1.0000	7.8e-09	1.0000
21	< 2e-16	0.00068	< 2e-16	0.01021
22	0.00584	0.00089	< 2e-16	< 2e-16
23	< 2e-16	0.0010	< 2e-16	0.0012

tados pelo gráfico *(d)* da Figura 4.6. Nele, observa-se a disparidade entre os resultados das técnicas propostas e das técnicas *CB e RND*. Para confirmar essa superioridade, a Tabela 4.36 confirma, com todos os *p-values* menores que 0.05, para a hipótese 23. Mostra-se que em todas as comparações os dados diferem e, baseando-se no resultado visual, afirma-se que as técnicas propostas são superiores. Logo, reafirma-se que, para todos os dados, as técnicas *WCS-T e WCS-A* agrupam mais CTs que cobrem mudanças inéditas no topo da lista ordenada.

4.3.10 PP06: Existe diferença entre as técnicas para a métrica *Tempo de Execução*?

Para responder a pergunta de pesquisa 06 será necessário analisar os gráficos apresentados nas Figuras 4.1, 4.7, nas Tabelas 4.24 e 4.30. Para este último, foram propostas as hipóteses 24, 25, 26, 27 e 28 apresentadas na Tabela 4.18. O objetivo principal desse questionamento é descobrir se as abordagens *total* e *additional* das técnicas propostas apresentam um custo maior que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados do *Tempo de Execução*.

Com relação ao tempo de execução, afirma-se que a *WCS-A* teve um custo maior com relação a todas as outras, custo esse notório em todos os gráficos apresentados na Figura 4.7. Esse comportamento se dá pela utilização da estratégia *additional*, que agrega maior custo à técnica *WCS-A*, que é comprovado na Tabela 4.37, onde *WCS-T e CB* não apresentam diferença entre os dados, mas a *WCS-A* difere tanto da *RND*, quanto da *CB* para todas as hipóteses (24, 25, 26, 27 e 28).

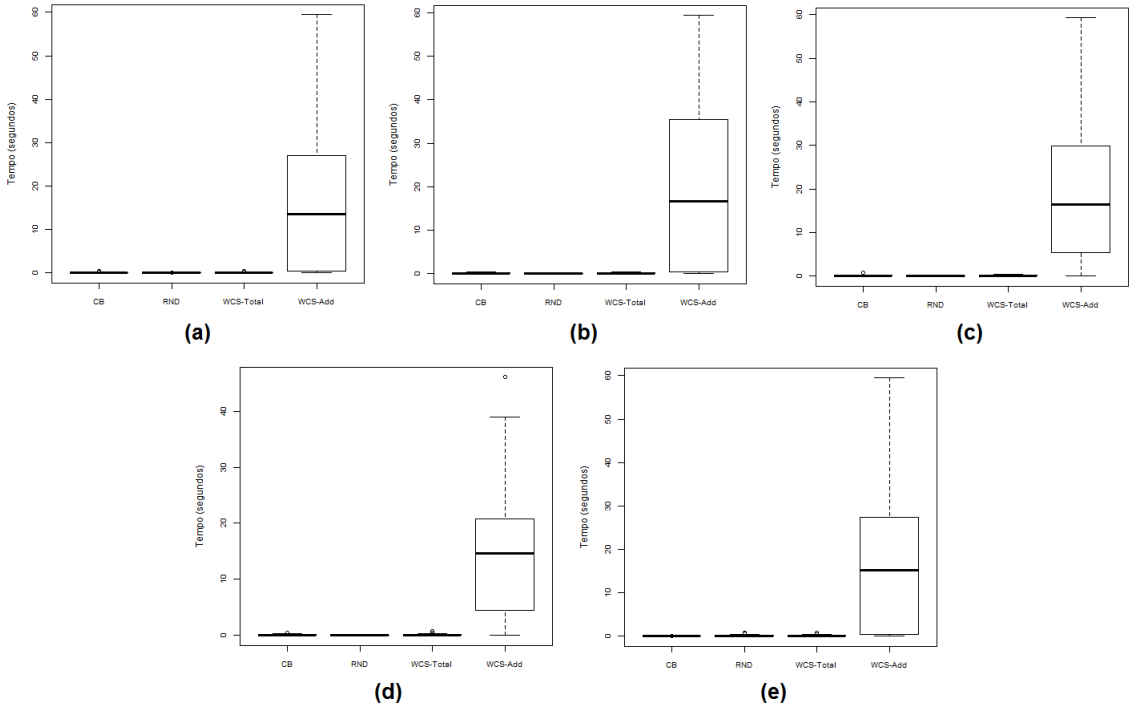

Figura 4.7: Boxplots para a métrica *Tempo de Execução*.

4.4 Ameaças à Validade

As amostras de programas utilizadas, assim como os casos de teste e os defeitos inseridos de forma manual, com certeza são motivos que representam ameaças à validade. Essas ameaças a validade internas e externas são listadas a seguir:

Ameaças à validade externa

1. Nós utilizamos quatro amostras bem específicas de *Java Applications* (pois no momento a arquitetura da ferramenta *PriorJ* está limitada a esse tipo de configurações de

Tabela 4.37: Teste Estatístico *PostHoc* Kruskal-Wallis para métrica Tempo de Execução

Hipótese	WCS-T x RND	WCS-T x CB	WCS-A x RND	WCS-A x CB
24	< 2e-16	1.0000	< 2e-16	< 2e-16
25	< 2e-16	1.0000	< 2e-16	< 2e-16
26	< 2e-16	1.0000	< 2e-16	< 2e-16
27	< 2e-16	1.0000	< 2e-16	< 2e-16
28	< 2e-16	1.0000	< 2e-16	< 2e-16

projetos), isso dificulta a generalização para contextos reais e para outras linguagens de programação além de *Java*.

2. As mutações foram simuladas pelo framework de mutação (MuJava [26]) e inseridas manualmente, isso não pode ser generalizado para amostras com defeitos reais.

3. Os nossos resultados não podem ser generalizados para outros casos de testes, pois cada suite utilizada refere-se ao seu código específico, fica muito difícil generalizar para todas as suites.

Ameaças à validade interna

1. A implementação das técnicas utilizadas no experimento e das variáveis dependentes (*APFD, F-measure, F-Spreading, Group-Measure, Group-Spreading e Weighted Impacted Statements Measure*) podem apresentar defeitos.

A fim de reduzir as ameaças externas, faz-se necessário estudos adicionais das amostras, das suites e das mutações inseridas com o objetivo de tornar cada amostra utilizada o mais próximo do real possível. Com relação as ameaças internas, para reduzir a probabilidade de defeitos na implementação tanto das técnicas quanto das métricas foi utilizada a prática da revisão de código. A ideia dessa prática é que o código escrito por um desenvolvedor, antes de ser enviado para produção, seja revisado por outro membro da equipe. Além disso, para nos certificarmos que as técnicas e métricas funcionavam da forma correta, realizamos alguns testes em cenários em que os resultados eram previamente conhecidos, assim pudemos validar funcionamento de cada uma delas. Todo o código produzido foi revisado antes de iniciarmos o experimento. Para testar a eficiência das técnicas, foi utilizada a métrica APFD que é altamente utilizada para a comparação de técnicas de priorização de casos de teste, porém ela tem suas limitações [35] e para reduzir essa métrica faz-se necessários estudos adicionais a fim de descobrir outras formas de mensurar a eficiência das técnicas.

4.5 Considerações Finais

Este capítulo foi exposto o estudo experimental realizado com a finalidade de aumentar o corpo teórico sobre a priorização geral de casos de teste em um contexto de mudanças. Para

avaliar tal conceito, foi realizado um estudo na literatura, com o objetivo de encontrar as técnicas já desenvolvidas e as métricas utilizadas para medir o seu desempenho.

Com uma experimentação robusta, envolvendo quatro projetos de código aberto, mais de 300 execuções e 5 contextos de mudanças distintos, foi possível deduzir que as técnicas propostas foram superiores à técnica de controle *RND* em todos os cenários e para todas as métricas. Através dos resultados, garantimos também que não ocorreu nenhum impacto negativo para a técnica *CB*. A técnica *WCS Total* se mostrou ainda superior à *CB* em alguns cenários. A técnica *WCS Additional* foi superior à *CB* em alguns cenários, mas isso resultou em um custo no tempo de execução muito acima do esperado.

Capítulo 5

Conclusão

Neste trabalho foi realizada uma investigação acerca da Priorização de Casos de Teste (CT) baseada em mudanças, em vários contextos de simulação de um ambiente real de integração contínua.

Inicialmente, foi conduzido um estudo sobre os trabalhos relacionados, no mesmo contexto, com a finalidade escolher quais melhorias poderiam ser propostas às técnicas existentes na comunidade científica. Além disso, definir quais técnicas e métricas mais utilizadas que pudessem demonstrar as melhorias implementadas e analisadas através do experimento. Concluído este estudo, foi selecionada a técnica *Changed Blocks (CB)* [40] pela observação da negligência da priorização da técnica, para alguns CTs, em contextos de múltiplas mudanças realizadas no código fonte. Tornou-se notória a indiferença de CTs que cobrem percentagens pequenas de código, porém inéditas, com relação aos que cobrem grande parte de código repetidamente privilegiados no resultado final. A melhoria proposta neste trabalho resultou em um algoritmo, chamado de *Weighted Changed Statements (WCS)*, que foi aplicado ao resultado da técnica CB seguindo duas estratégias, *total e additional*. Como técnica de controle, foi utilizada a abordagem aleatória (*random, ou RND*).

Selecionamos quatro métricas para compor o estudo experimental (*APFD, F-measure, F-spreading e Tempo de Execução*) e a propor duas (*Group-measure e Group-spreading*), baseadas nas escolhidas. As quatro técnicas foram comparadas considerando a capacidade de detecção de defeitos, através das métricas *APFD e F-measure*; a capacidade de agrupamento de defeitos, através das métricas *F-spreading e Group-spreading*; a capacidade de privilegiar defeitos inéditos, através da métrica *Group-measure*; e o custo de cada técnica, através da

métrica *Tempo de Execução*.

Além disso, selecionamos projetos que simulariam um ambiente de produção de software real, através criação de versões e da aplicação de mutações nessas amostras. Essas mutações foram geradas pela ferramenta MuJava [26], selecionado um subconjunto de mutações aleatoriamente e gerado uma cópia do projeto com uma das mutações selecionadas inseridas Os projetos foram selecionados através do repositório OpenHub[1], são eles: JBehave[2], XMLUnit[3], JMock[4] e XMLMatchers[5].

Por fim, foi conduzido um estudo experimental a fim de deduzir o impacto, positivo ou negativo, causado pela aplicação do algoritmo *WCS* às técnicas *Changed Blocks (total e additional)*. Para inferir as conclusões a seguir, foi necessário a utilização do Teste de Hipóteses aplicado a cinco grupos de amostras diferentes dos projetos escolhidos, resultando em um total de 303 execuções de priorização no *PriorJ*. Para o teste estatístico foi utilizado o teste de *kruskal-wallis*, seguido do seu teste *post-hoc* com a aplicação da correção de *bonferroni*.

5.1 Resultados e Conclusões

Com base nas técnicas, projetos, grupos de amostras, métricas (selecionadas e propostas), o experimento foi definido e planejado. Como norteadores do experimento foram definidos dois pares de hipóteses (nulas e alternativas), um para cada uma das métricas envolvidas de cada grupo de amostras, gerando um total de 28 hipóteses. As nulas, representadas por H_0, previam igualdade entre as distribuições, enquanto as alternativas, representadas por H_1, previam a diferença entre as distribuições.

Considerando um nível de confiança de 95%, as hipóteses de igualdade testadas para todos os grupos de amostras foram refutadas, através do teste de *kruskal-wallis*. Quando as propostas foram comparadas, o teste rejeitou a hipótese nula. Os testes *post-hoc* mostraram que a técnica *RND* apreasentou resultado diferencial em relação às outras técnicas, indicando

[1] https://www.openhub.net/

[2] http://jbehave.org/

[3] http://www.xmlunit.org/

[4] http://www.jmock.org/

[5] http://xmlmatcher.sourceforge.net/

que há diferenças entre a capacidade de revelar defeitos das técnicas analisadas. Isso é notório na análise dos gráficos apresentados no Capítulo 4, logo, as conclusões estão focadas entre *CB* e às propostas *WCS-T e WCS-A*.

Considerando a Pergunta de Pesquisa 01, que tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *APFD*. A partir do pós-teste realizado em todos os grupos, as hipóteses 01, 02 e 03 confirmam que as técnicas propostas, *WCS-T e WCS-A*, não sofreram nenhum impacto negativo com relação a métrica *APFD*. Os resultados para as hipóteses 04 e 05 ainda apontam que as técnicas propostas foram superiores à *RND*; e, com relação a *CB*, é possível afirmar que a *WCS-A* foi superior, mas a *WCS-T* não, pois os resultados não diferem estatisticamente.

Avaliando a Pergunta de Pesquisa 02, que tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *F-Measure*. A partir do pós-teste realizado em todos os grupos, as hipóteses 06, 07, 08, 09 e 10 confirmam que as técnicas propostas, *WCS-T e WCS-A*, não sofreram nenhum impacto negativo com relação a métrica *F-Measure*. Evidencia-se a superioridade de *WCS-T e WCS-A* em relação a *RND*, para todos os grupos de amostras; e, com relação a *CB*, não podemos afirmar qual técnica é melhor, pois os resultados não diferem. Com exceção da análise dos dados do Grupo 04 de amostras (hipótese 09), que mostrou superioridade nos resultados para a técnica *WCS-A*. Acredita-se que isso ocorreu pelo fato de que nesse contexto, várias mudanças estão inseridas em grande parte do código, simulando um ambiente real de integração contínua. Possivelmente, os dados do Grupo 05 analisados na hipótese 10 não obtiveram os mesmos resultados pelo impacto causado pela união de todas as amostras.

Examinando a Pergunta de Pesquisa 03, que tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas agrupam melhor casos de teste que cobrem defeitos que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *F-Spreading*. A partir do pós-teste realizado em todos os grupos, revela-se que para todas os Grupos de amostras as hipóteses (11, 12, 13, 14 e 15) de que o nível de agrupamento dos casos de teste não seriam impactados após a ordenação se concretizo, quando comparados à técnica *CB*; e ainda foram superiores a técnica *RND*.

Analisando a Pergunta de Pesquisa 04, que tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas antecipam a detecção de defeitos inéditos melhor que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *Group-Measure*. A partir do pós-teste realizado para todas as hipóteses (16, 17, 18 e 19), podemos afirmar que, com exceção do Grupo 02 de amostras, as técnicas propostas (*WCS-T e WCS-A*) foram superiores tanto da *RND*, quanto da *CB*. Isso significa que as técnicas priorizaram de forma mais eficiente casos de teste que cobrem mudanças inéditas no código-fonte.

A fim de analisar a Pergunta de Pesquisa 05, que tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas agrupam melhor casos de teste que cobrem defeitos inéditos que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados da *Group-Spreading*. A partir do pós-teste realizado para todas as hipóteses (20, 21, 22 e 23), podemos afirmar que, com exceção do Grupo 02 de amostras, as técnicas propostas (*WCS-T e WCS-A*) foram superiores tanto da *RND*, quanto da *CB*. Isso significa que as técnicas agruparam de forma mais eficiente os casos de teste que cobrem mudanças inéditas no código-fonte, conquistando assim o objetivo principal deste trabalho de reunir uma maior quantidade de CTs que cobrem mudanças inéditas no topo da lista ordenada.

A fim de analisar a Pergunta de Pesquisa 06, que tem por objetivo descobrir se as abordagens *total* e *additional* das técnicas propostas apresentam um custo maior que a ordem natural (RND) ou a *Changed Blocks* (CB), baseando-se nos resultados do *Tempo de Execução*. A partir do pós-teste realizado para todas as hipóteses (24, 25, 26, 27 e 28), podemos afirmar que, a técnica *WCS-T* não obteve nenhum custo adicional após a implementação da melhoria. Já a técnica *WCS-A* teve impactos negativos no seu custo. Esse comportamento era esperado, por conta da utilização da estratégia *additional*.

Após a análise geral dos resultados, é perceptível que as modificações aplicadas à técnica *Changed Blocks* resultaram em impactos positivos para os contextos em que o algoritmo proposto se propõe a atuar, sem apresentar perdas, de forma geral, no correto funcionamento da técnica. O alto custo da *WCS-A* se dá pela utilização da estratégia de priorização *additional*, que agrega um custo maior à técnica por utilizar uma análise minuciosa. Conclui-se que essa estratégia é responsável por isso, pelo fato de a *WCS-T* ter utilizado o algoritmo *WCS* e ter apresentado custo semelhante a *CB*.

5.2 Trabalhos Relacionados

O objetivo desta seção é reunir informações relevantes das áreas de teste de regressão e priorização de casos de teste de regressão (Subseção 5.2.1), a fim de relacioná-las ao nosso trabalho e nos ajudar no estudo empírico entre as técnicas de priorização existentes atualmente na comunidade científica.

5.2.1 Priorização de Casos de Teste

De acordo com Harrold & Jones [21], o teste é particularmente caro para desenvolvedores de *software* com alta garantia de qualidade, como, por exemplo, *softwares* produzidos para sistemas de bordo comerciais de aviões. Uma das razões para essa despesa é que um dos requisitos da Administração de Aviação Federal é que as suítes de teste sejam adequadas a "condição modificada/decisão de cobertura"(do inglês, *modified condition/decision coverage - MC/DC*). Porém, como o *software* é modificado constantemente e novos casos de teste (CT) são inseridos na suíte, elas crescem e consequentemente o custo da regressão aumenta proporcionalmente. Para enfrentar esse problema do tamanho da suíte de testes, Harrold & Jones [21] investigaram o uso de algoritmos de redução suíte de testes, que identificam um conjunto reduzido de CT que fornece a mesma cobertura do *software*. Os autores conseguiram esse resultado através da aplicação de algoritmos de priorização de testes no conjunto de CTs original. Esse trabalho mostra a importância dos nossos estudos em priorização/redução de testes de regressão e o impacto que esses resultados podem ter em um ambiente de desenvolvimento real.

Gregg Rothermel realiza estudos na área de otimização de testes de regressão há mais de uma década através de várias metodologias, como: redução de grupo de testes, seleção de testes de regressão e priorização de casos de teste. Tomando como foco a priorização de casos de teste, Rothermel et. al. [36] apresenta estudos usando técnicas baseadas em cobertura e apresenta resultados promissores através da utilização da instrumentação de código, abordagem também usada no presente trabalho. Para a realização dos estudos empíricos foram considerados apenas oito programas relativamente pequenos. Os resultados mostram que as técnicas estudadas podem melhorar a taxa de detecção de defeitos mesmo para as técnicas menos sofisticadas (e, consequentemente, menos caras). As abordagens estudadas

por Rothermel et. al. [36] se assemelham com os nossos estudos pela utilização de instrumentação e cobertura de código, com a diferença de que, ao invés de utilizarmos cobertura completa, nos baseamos apenas na cobertura das mudanças entre versões.

Dando continuidade nos estudos dos trabalhos de Rothermel, encontramos Sebastian Elbaum, que conduz estudos na mesma área de priorização de testes. Elbaum et al. [10] realizaram a pesquisa com o objetivo principal é melhorar a taxa de detecção de defeitos em versões específicas do código-fonte. Apesar de reconhecer as limitações da validade dos experimentos, o trabalho mostra que priorização por versão específica pode produzir resultados significativamente melhores, dependendo do contexto. Também foi possível visualizar que as diferenças na análise entre granularidade fina e grossa foram mínimas, em contrapartida, a diferença de custos é grande. Em nosso trabalho foi proposto uma melhoria na técnica *Changed Blocks (CB)*, que é uma técnica de versão específica.

As técnicas examinadas por Rothermel et. al. [36] operam em granularidade fina, o que torna a análise muito cara para *softwares* de larga escala. Uma alternativa visualizada por Elbaum et. al. [10] foi a utilização da análise de código em granularidade grossa, com isso é esperado que ocorra alguma perda na eficácia, mas ganho na eficiência. Elbaum et al. [10] se utilizaram da análise de quatro técnicas apresentadas em [36] e doze novas técnicas que operam no nível de métodos (granularidade grossa). O algoritmo proposto em nosso trabalho, assim como a técnica que aplicamos a melhoria, trabalham com a análise de código em granularidade fina, tornando o algoritmo custoso para *softwares* de larga escala.

Elbaum et al. [10] iniciou o seu trabalho com a utilização dos mesmos programas utilizados em [36], porém, se sentiram seguros o bastante para estender os experimentos com a inclusão da análise em três programas relativamente grandes. As métricas de eficácia utilizadas para a análise das técnicas nos trabalhos citados acima foram realizadas pelo cálculo da *average of the percentage of faults detected (APFD)*. Os programas utilizados por Rothermel et. al. [36] e Elbaum et. al. [10] são provenientes de pesquisas na *Siemens Corporate Research* e *European Space Agency*. Nossos estudos empíricos foram realizados através da utilização de quatro projetos do repositório aberto *OpenHub*[6]. O projeto com maior escala usado na nossa experimentação contém 26.646 linhas de código, aproximadamente quatro vezes maior que o projeto mais robusto utilizado por Elbaum et. al. [10].

[6]https://www.openhub.net/

Baseando-se no conceito de mudanças de *software*, desenvolvedores e *testers* devem ser capazes de visualizar quais CTs serão executados, a fim de capturar possíveis comportamentos inesperados. Essa abordagem foi estudada por Srivastava & Thiagarajan [40] em seu trabalho, e afirmam que "novos defeitos introduzidos recentemente no sistema tem a maior probabilidade de aparecer em mudanças recentes. Assim, uma estratégia efetiva é focar o esforço de testes na partes do programa afetadas pelas mudanças". Srivastava & Thiagarajan [40] propuseram um sistema de priorização de testes de versão específica. Batizado de *Echelon*, o *software* tem por objetivo priorizar um conjunto existente de testes referentes aos cenários que foram modificados no sistema sob testes *(do inglês, System Under Test - SUT)*. É importante ressaltar que ele realiza apenas a ordenação da lista de casos de testes, não tem o papel de selecionar ou retirar casos que por ventura se tornem obsoletos. Para a utilização do Echelon são necessárias a utilização de ferramentas exclusivas da *Microsoft*, o que impede a contribuição de pesquisadores externos. Neste trabalho usamos uma versão do algoritmo, batizada de *Changed Blocks* e vinculada ao plugin do eclipse chamado de *PriorJ* [32], a fim de implementar a melhoria proposta.

Em seus experimentos Srivastava & Thiagarajan [40] utilizaram sistemas relativamente de larga escala, contendo números de linhas de código-fonte em nível de milhão. Neles também foram consideradas medidas utilizadas apenas em seus trabalhos, já que não foram comparadas com nenhuma técnica existente. Entre essas medidas estão: números de casos de teste em cada conjunto; número de blocos impactados em cada conjunto; porcentagem de defeitos detectados, etc. Segundo Srivastava & Thiagarajan [40], a melhor forma de estudar o sistema proposto foi o estudo da detecção de defeitos em arquivos binários no processo de desenvolvimento. Os resultados apresentados mostram taxas entre 71% e 81% dos defeitos existentes detectados logo no primeiro conjunto de casos de teste retornados pela ferramenta. Isso expõe a possibilidade de priorizar casos de teste em softwares de larga escala, se utilizando de priorização de versão específica, e ter bons resultados. Esta técnica trabalha no mesmo contexto que as técnicas propostas, além de tentar solucionar o mesmo problema geral da priorização de casos de teste. Além disso, nós visualizamos contextos em que essa técnica não trabalha muito bem. Por esses motivos, essa técnica foi escolhida como aporte teórico e base para a implementação das melhorias propostas.

Como foi possível observar, no estado da arte na área de priorização de testes existem

técnicas que se utilizam de estratégias como *total* e *additional*. *Total* se baseia na quantidade de elementos cobertos pelo teste e *additional* na quantidade de elementos não cobertos ainda pelo teste. Fundamentado nesse conceito Zhang et. al. [17] propuseram uma abordagem diferente em seu trabalho, se utilizando das duas estratégias em uma só técnica. Segundo ele, *total* e *additional* são estratégias genéricas para diferentes critérios de cobertura. Dado o critério de cobertura, a estratégia total ordena os casos de teste em ordem decrescente de cobertura, enquanto que a additional sempre escolhe o próximo caso de teste baseando-se no número máximo de elementos que ainda não foram cobertos por casos de teste priorizados anteriormente. Zhang et. al. [17] afirmam que a estratégia additional, apesar de ser superior a total em eficiência, apresenta o custo elevado como fraqueza.

Na experimentação realizada por Zhang et al. [17], foram utilizadas 19 versões de quatro programas escritos em Java, que incluem três versões do *jtopas*, três versões do *xml-security*, cinco versões do *jmeter* e oito versões do *ant*. Para simular a presença de defeitos nos programas, já as versões dos objetos de estudo utilizados não contém a quantidade de defeitos necessárias para a realização de uma boa experimentação, Zhang et al. [17] se utilizaram da ferramenta *MuJava* [26] para gerar defeitos nos objetos de estudo. Estratégia também utilizada em nossos estudos experimentais para as melhorias propostas na técnica *Changed Blocks*. As principais contribuições de Zhang et. al. [17] foram: Primeiro, uma nova abordagem que cria melhores técnicas de priorização de casos de teste pelo controle da incerteza da capacidade de detecção de defeitos na priorização; segundo, a apresentação de dois modelos que unificam as estratégias total e additional; terceiro, evidência empírica de que várias estratégias existentes entre as duas gerais são mais efetivas; e quarto, evidência empírica de que essas estratégias podem ter resultados bem melhores que com a abordagem de granularidade fina.

Ripon et. al. [31] apresentou uma nova abordagem batizada de REPiR, com o objetivo de resolver o problema da priorização de testes de regressão através da redução para um problema padrão de recuperação de informação (*Information Retrieval Problem*). Nessa abordagem, as diferenças entre as duas versões do sistema formam a *query* e os testes constituem a coleção de documentos. É importante enfatizar que o REPiR não requer nenhuma análise dinâmica ou estática do programa. A avaliação empírica da ferramenta aponta que a acurácia da ferramenta é maior que as ordens de priorização natural e aleatória (*RND*). Os resultados

mostram que a ferramenta é relativamente independente do tamanho das diferenças entre as versões do *SUT* e do número de métodos de ensaio. Além do mais, a nível de métodos (granularidade grossa), as diferenças altas funcionam melhor que diferenças pequenas. Em contrapartida, a nível de classe as diferenças menores funcionam levemente melhores que diferenças altas, em média. REPiR trabalha melhor que todas as abordagem que utilizam a estratégia *Total* utilizadas na experimentação e funciona igual ou melhor que as abordagens que utilizam a estratégia *Additional*. Mesmo em sua forma mais simples, quando é abstraído da linguagem de programação, não perde acurácia significante.

Ripon et. al. [31] também mostrou que REPiR é mais eficiente e supera as estratégias existentes na maioria das amostras de SUT estudadas. Mas o sucesso da ferramenta depende bastante do uso de termos significativos e similares no código-fonte, casos de teste correspondentes, além de que termos duplicados são ruins para a priorização. Esse trabalho está bem relacionado ao nosso porque pretende resolver problemas na mesma área geral de priorização de casos de teste de regressão, porém, com objetivos específicos e abordagem utilizada diferentes, além de se apresentar como um concorrente com bons resultados.

5.3 Trabalhos Futuros

Os resultados obtidos nesse estudo indicam, na maior parte dos contextos estudados, a eficiência do algoritmo proposto com relação as técnicas comparadas. Porém, sugere-se estudos mais aprofundados como trabalhos futuros, são eles:

- **Evolução da ferramenta PriorJ**: a atualização da ferramenta torna-se necessária para acompanhar as evoluções existentes nas linguagens de programação que trabalha, no caso *Java*. As linguagens de programação se atualizam a cada dia, como exemplo temos o a atualização do *Java 7* para o *8* apresentando novidades como: i. Operadores Diamante; ii. Manipuladores de Métodos; iii. *Lambda Expressions*; etc. Assim, faz-se necessário abrir mão do analisador de cobertura atual e buscar um *framework* que minimize esse trabalho e deixa a ferramenta livre de bibliotecas obsoletas. Transformar o *PriorJ* em um serviço integrável com ferramentas de integração contínua existentes no mercado é um requisito altamente desejável.

- **Execução de mais estudos experimentais**: por ainda conter ameaças à validade, o presente experimento pode ser melhorado de modo a oferecer evidências ainda mais fortes. Aumentar a quantidade de aplicações a serem analisadas em um experimento é uma sugestão imediata de melhoria, incorporando vários tipos de sistemas, com várias suítes diferentes. Isso implicaria em um aumento na capacidade de generalização dos resultados obtidos. Além disso, uma boa melhoria seria analisar a viabilidade de diminuição do custo do algoritmo *WCS* quando utilizado junto a estratégia *additional*, pois ele apresenta resultados muito bons, mas o custo benefício não é viável.

- **Utilização de ambientes reais**: os grupos de amostras utilizados foram bem abrangentes, porém não se compara a um ambiente de desenvolvimento real, enfrentando problemas do dia a dia dos desenvolvedores que se utilizam principalmente das metodologias ágeis e da integração contínua. Podemos pensar em uma ferramenta integrável a algum *framework* de integração contínua, que possibilita a geração uma suíte priorizada para cada versão do repositório de trabalho. Isso pode agilizar a liberação de uma *release* requisitada de última hora pelo cliente, já que a suíte priorizada vai detectar os defeitos mais rapidamente, agilizando assim a estabilização da versão.

Bibliografia

[1] E. L. G. Alves, S. T. C. Santos, P. D. L. Machado, and T. Massoni. Test Case Prioritization Using PriorJ. 2013.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411, May 2005.

[3] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.

[4] K. Beck. *JUnit - pocket guide: quick lookup and advice.* O'Reilly, 2004.

[5] C. E. Bonferroni. Teoria statistica delle classi e calcolo delle probabilità. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze*, 8:3–62, 1936.

[6] R.D. Craig and S.P. Jaskiel. *Systematic Software Testing*. Artech House ITS library. Artech House, 2002.

[7] M. L. Côrtes and T. C. S. Chiossi. *Modelos de Qualidade de Software*. Unicamp, Instituto de Computação, Campinas, 2001.

[8] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Structured programming. 1972.

[9] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, 32(9):733–752, September 2006.

[10] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[11] S. Elbaum, G. Rothermel, S. Kanduri, and A.G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, September 2004.

[12] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.

[13] P. B. Fagundes. Framework para comparação e análise de métodos ágeis. Florianópolis, May 2005. Dissertação de Mestrado, Universidade Federal de Santa Catarina, Centro Tecnológico.

[14] R.A.S. Fisher. *Statistical methods for research workers*. Oliver and Boyd, 1938.

[15] M. Fowler. The new methodology. *Wuhan University Journal of Natural Sciences*, 6(1):12–24, 2001.

[16] S. Freeman, T. Mackinnon, N. Pryce, M. Talevi, and J. Walnes. jmock, 2007.

[17] D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, Piscataway, NJ, USA, 2013. IEEE Press.

[18] J. Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] D. Jeffrey and N. Gupta. Experiments with test case prioritization using relevant slices. *J. Syst. Softw.*, 81(2):196–221, February 2008.

[20] B. Jiang, Z. Zhang, W.K. Chan, and T.H. Tse. Adaptive random test case prioritization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 233–244, 2009.

[21] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 92–, Washington, DC, USA, 2001. IEEE Computer Society.

[22] J.A. Jones and M.J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–209, March 2003.

[23] Y. Kanellopoulos, T. Dimopulos, C. Tjortjis, and C. Makris. Mining source code elements for comprehending object-oriented systems and evaluating their maintainability. *SIGKDD Explor. Newsl.*, 8(1):33–40, June 2006.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, and J. Loingtier. Aspect oriented programming. 1997.

[25] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 442–453, Nov 2003.

[26] Y. Ma, J. Offutt, and Y. R. Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15:97–133, 2005.

[27] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[28] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[29] R.S. Pressman. *Engenharia de software*. Makron Books, 1995.

[30] X. Ren, F. Shah, Frank Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[31] K. S. Ripon, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *37th IEEE/ACM Inter-*

national Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pages 268–279, 2015.

[32] J. H. Rocha, E. L. G. Alves, and P. D. L. Machado. PriorJ: Priorização de Casos de Teste JUnit. 2012.

[33] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. 24(6):401–419, 1998.

[34] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, Aug 1996.

[35] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 179–188, 1999.

[36] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, Oct 2001.

[37] J. Rumbaugh, G. Booch, and I. Jacobson. *Uml - Guia do Usuário: tradução*. Campus, Rio de Janeiro, RJ, Brasil, 2000.

[38] I. Sommerville. *Engenharia de Software*. Addison-Wesley, São Paulo, 2003.

[39] I. Sommerville. *Software Engineering 9*. Pearson Education, 2011.

[40] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106, July 2002.

[41] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[42] W. Weisflog. Michaelis moderno dicionário da língua portuguesa, melhoramentos., 2009.

[43] F. Wilcoxon. *Individual comparisons by ranking methods*. Biometrics Bull, 1945.

[44] Z. Q. Zhou. Using coverage information to guide test case selection in adaptive random testing. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 208–213, July 2010.