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

Investigando o Impacto da Evolução de Casos de Uso em Testes Gerados no Contexto de Teste Baseado em Modelo

Anderson Gustafson Freire da Silva

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

Wilkerson de Lucena Andrade
Everton Leandro Galdino Alves
(Orientadores)

Campina Grande, Paraíba, Brasil

INVESTIGANDO O IMPACTO DA EVOLUÇÃO DE CASOS DE USO EM TESTES GERADOS NO CONTEXTO DE TESTE BASEADO EM MODELO

ANDERSON GUSTAFSON FREIRE DA SILVA

DISSERTAÇÃO APROVADA EM 16/08/2019

WILKERSON DE LUCENA ANDRADE, Dr., UFCG
Orientador(a)

EVERTON LEANDRO GALDINO ALVES, Dr., UFCG
Orientador(a)

PATRICIA DUARTE DE LIMA MACHADO, PhD, UFCG
Examinador(a)

ALEXANDRE CABRAL MOTA, Dr., UFPE
Examinador(a)

CAMPINA GRANDE - PB

S586i Silva, Anderson Gustafson Freire da.
Investigando o impacto da evolução de casos de uso em testes gerados no contexto de teste baseado em modelo / Anderson Gustafson Freire da Silva. - Campina Grande, 2020.
74 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2019.
"Orientação: Prof. Dr. Wilkerson de Lucena Andrade, Prof. Dr. Everton Leandro Galdino Alves.
Referências.

1. Engenharia de Software. 2. Teste. 3. Teste Baseado em Modelo. I. Andrade, Wilkerson de Lucena. II. Alves, Everton Leandro Galdino. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELO BIBLIOTECÁRIO GUSTAVO DINIZ DO NASCIMENTO CRB-15/515

Resumo

Teste de Software é o processo que busca encontrar erros a partir da execução de um determinado sistema. Em outras palavras, intenta detectar divergências entre o comportamento atual e o pretendido para o software em desenvolvimento. No entanto, testar não é uma atividade trivial. Especialmente sistemas complexos, que demandam baterias de teste robustas, podem chegar a dedicar até 50%, do tempo utilizado no desenvolvimento, e dos recursos financeiros, para os testes. Teste Baseado em Modelo (TBM) pode facilitar esse processo, uma vez que viabiliza a geração automática de casos de teste a partir dos modelos que descrevem comportamentos ou funcionalidades do sistema sob teste. No contexto TBM, utilizando como modelos casos de uso descritos em linguagem natural, observamos que cerca de 86% da suíte gerada, para execução manual, se torna obsoleta ao evoluir os modelos do sistema e gerar uma nova suíte. Tais evoluções dos modelos podem ser decorrentes de alterações de requisitos, elicitação incorreta dos requisitos, ou então refatoração para melhorar a qualidade do caso de uso. Além disso, detectamos que parte desses testes, aproximadamente 52%, se tornam obsoletos por correções gramaticais ou melhoria de escrita, não alterando o comportamento do sistema, e, podendo ser reutilizados sem muito esforço. Descartar tais testes implica na perda de seus dados históricos (e.g., histórico de execução/resultados e versão utilizada do sistema). Baseado nisso, o objetivo deste trabalho é buscar identificar automaticamente testes impactados por modificações de aspecto sintático, inseridas durante as evoluções dos modelos do sistema, facilitando o reúso e preservação dos dados históricos.

Abstract

Software testing aiming to find errors from the execution of a particular system. In other words, it attempts to detect divergences between current and intended behavior for a software under development. However, testing is not a trivial activity. Especially complex systems, which require robust test batteries, can devote up to 50% of development time and financial resources to testing. Model-Based Testing (MBT) can facilitate this process by enabling automatic generation of test cases from models that describe behaviors or functionality of the system under test. In the MBT context, using as models use cases described in natural language, we observed that about 86 % of the generated manual execution suite, becomes obsolete when evolving the system models and generating a new suite. Such model evolutions may be due to changing requirements, incorrect requirements elicitation, or refactoring to improve the quality of the use case. In addition, we detected that some of these tests, about 52%, become obsolete due to typos fixing or writing improvements, not modifing system behavior, and can being reused without much effort. Discarding such tests implies the loss of your historical data (e.g., run/results history and associated system version). Based on this, the objective of this paper is to automatically identify tests impacted by syntactic modifications, inserted during the evolution of the system models, facilitating the reuse and preservation of historical data.

Agradecimentos

Agradeço a Deus pelo dom da vida e por seu amor incondicional.

Aos meus queridos pais, Heronides Elias e Maria Luzinete, por todo o cuidado, carinho e incentivo. Ao meu adorado irmão Allan Gustavo e cunhada Carina Dornelas, pelo companheirismo, conselhos e preocupação em sempre querer o meu melhor. A minha família pelo apoio e consideração.

A minha preciosa esposa Otília Martins, por todo amor, compreensão e apoio. Não caberia em palavras expressar a sorte que é ter você ao meu lado. A toda sua família pela torcida e disposição para comigo.

Em especial, agradeço aos meus orientadores Wilkerson Andrade e Everton Galdino, por toda confiança, disposição em me ensinar, motivação e paciência. Carregarei sempre comigo seus ensinamentos.

A Dalton Jorge pela sua prontidão e gentileza em ajudar. Aos amigos da sala 107/9 pela presença durante toda essa jornada e pelos inúmeros momentos de descontração, que tornaram tudo mais leve. Estendo minha gratidão aos membros do laboratório SPLab.

Aos grandes amigos que a UFCG me proporcionou, em especial Gustavo Yamaguchi, pelas várias horas de conversa e solicitude sempre que precisei, e Jonas Zuleido, pelo apoio e parceria.

Aos professores, Patrícia Machado e Alexandre Mota, que compuseram a banca examinadora, pelas considerações feitas para o enriquecimento desse trabalho.

Ao CNPq pelo apoio e suporte financeiro fornecido a este trabalho.

Conteúdo

Lista de Símbolos

Claret - *CentraL Artifact for Requirement Enginnering and model based Testing*

DSL - *Domain-Specific Language*

ICMC - *Instituto de Ciências Matemáticas e de Computação*

LNC - *Linguagem Natural Controlada*

LTS - *Labeled Transition System*

LTS-BT - *Labeled Transition System Based-Testing*

SAST - *Brazilian Symposium on Systematic and Automated Software Testing*

SBES - *Brazilian Symposium on Software Engineering*

SPLab - *Software Practices Laboratory*

SST - Sistema Sob Teste

TBM - *Teste Baseado em Modelo*

TGF - *Trivial Graph Format*

UFBA - Universidade Federal da Bahia

UFCG - Universidade Federal de Campina Grande

UML - *Unified Modeling Language*

USP - Universidade de São Paulo

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Capítulo 1

Introdução

Teste de *Software* é o processo que busca encontrar defeitos a partir da execução de um determinado sistema [2]. Em outras palavras, intenta detectar divergências entre o comportamento atual e o pretendido para o *software* em desenvolvimento. Os testes podem ser de diferentes tipos (e.g., unidade, componente, integração, sistema, funcional), com diferentes características e objetivos [65].

Testar é custoso, pode ocupar 50% do tempo utilizado para desenvolvimento e tomar, em alguns casos, até mais que 50% dos recursos financeiros dedicados ao projeto [45]. Porém, é extremamente importante ter uma boa infraestrutura de teste, uma vez que o custo de solucionar um defeito aumenta à medida que se avançam as fases de desenvolvimento [61].

No contexto de testes de sistema, usualmente não é viável testar todos os cenários de teste possíveis. Um sistema simples pode ter centenas de combinações de entradas e saídas, diversas funcionalidades e comportamentos. Dessa forma, uma execução total demandaria muito tempo e recurso, tornando-se economicamente inviável. Isto é agravado em ambientes de desenvolvimento que utilizam metodologias ágeis (e.g. *Extreme Programming [62]* e *Scrum* [57]), onde as iterações/ciclos ocorrem em uma menor janela de tempo. Estas metodologias têm se disseminado, popularizando-se em diferentes organizações e times de desenvolvimento e possuem como principais características [36]: ciclo incremental de desenvolvimento; entregas frequentes de funcionalidade; colaboração constante com o cliente e capacidade de prover respostas rápidas às mudanças necessárias [23].

A redução do custo e esforço dedicados aos testes têm sido um ponto de atenção. Diferentes pesquisas já foram realizadas trazendo algumas estratégias ou abordagens que con-

tribuem com este tópico, propondo diferentes soluções (e.g., [2, 19, 67, 69]). Uma dessas abordagens, denominada Teste Baseado em Modelo (TBM), viabiliza a geração automática de casos de teste a partir dos modelos que descrevem comportamentos ou funcionalidades do sistema [64]. O modelo utilizado neste tipo de abordagem não é restrito a um tipo específico, podendo se adequar a diferentes contextos. Dentre os utilizados por TBM estão: diagramas *Unified Modeling Language* (UML), como diagrama de máquina de estados [50] e de sequência [55]; sistemas de transição [63]; e documento de requisitos seguindo o padrão de caso de uso com passos descritos em linguagem natural [35].

Podemos citar como vantagens propiciadas do uso de TBM [64]: (i) redução do esforço dedicado à geração dos testes; (ii) redução do custo associado à criação de uma suíte; (iii) criação dos testes de forma sistemática, o que torna o processo replicável; e (iv) capacidade de detecção de faltas equivalente às suítes construídas manualmente. Todavia, limitações também são inerentes ao uso de TBM. Utting e Legeard indicam que [64]: (i) é necessário ter modelos bem construídos, o que demanda um certo grau de experiência de quem escreve; (ii) a quantidade de testes gerada é geralmente grande; (iii) é preciso que exista uma visão abstrata do sistema para melhor construção dos modelos; e (iv) dificuldade de reúso dos testes à medida que o sistema evolui, ocasionado pela sensibilidade da suíte às alterações no modelo, como mostrado por Silva et al. [58].

Estudos estão sendo conduzidos acerca da aplicação de TBM no contexto de desenvolvimento ágil [21, 35, 53], e técnicas, como a de Jorge et al. [35], sendo desenvolvidas para promover o uso de TBM nessas metodologias. Jorge et al. [35] apresenta a ferramenta de nome CLARET, voltada à escrita de casos de uso numa notação simplificada e de fácil aprendizado. Com a CLARET, é possível criar documentos de caso de uso que podem ser utilizados para validação de funcionalidades junto ao cliente, como também elucidar o comportamento de uma determinada funcionalidade do sistema, o que facilita o desenvolvimento e evita retrabalho por parte da equipe de desenvolvimento. Além disso, os artefatos gerados pela CLARET podem ser utilizados para geração automática de testes via ferramentas TBM. Jorge et al. [35], em seu trabalho, une sua ferramenta de especificação de casos de uso a uma outra: LTS-BT [9]. A LTS-BT é responsável por fazer a geração automática de casos de teste. Os testes gerados nesse caso, são funcionais, de alto nível e voltados à execução manual no sistema, uma vez que os casos de uso são descritos utilizando linguagem natural

e não se tem detalhes a nível de código do sistema especificado.

Durante o ciclo de desenvolvimento, é natural que o software passe por modificações, correções ou até alterações nos requisitos. Nesse caso, tais alterações devem ser feitas nos casos de uso CLARET, para que reflitam o comportamento atual do sistema, e gerado novos testes via LTS-BT para que se tenha uma suíte em conformidade com o sistema, evitando executar testes desatualizados. Em meio às modificações feitas nos casos de uso, podemos ter mudança de comportamento do sistema, como também refatorações na especificação, no intuito de melhorar a legibilidade e corrigir erros de escrita. Em virtude dos testes estarem diretamente relacionados aos casos de uso, tais mudanças nos casos de uso impactam diretamente os testes ao se gerar uma nova suíte, contribuindo com a limitação (iv) citada anteriormente: dificuldade de reúso dos testes à medida que o sistema evolui, ocasionado pela sensibilidade da suíte às alterações no modelo. Tal impacto, pode acarretar um maior descarte de testes, visto que, o conjunto de ações e respostas do sistema, que compõem os testes, foi alterado em virtude das atualizações no caso de uso. Categorizamos neste trabalho, as modificações de comportamento inseridas nos casos de uso, como modificações semânticas, por alterarem o comportamento do sistema. Já as modificações voltadas a melhoria e correção da escrita da especificação, denominamos como modificações sintáticas. Um maior detalhe sobre essas modificações e sobre seus impactos nos testes regerados é mostrado na Seção 1.1.

O reúso de testes é importante para que se possa preservar importantes informações históricas acerca dos testes. Nesse caso é interessante identificar quais testes de fato foram modificados em virtude de alterações de comportamento, inseridas durante modificações de um caso de uso, e também identificar quais testes possuem apenas modificações advindas de refatorações realizadas no caso de uso e refletidas nos testes. Não é interessante que se descarte um teste se este possui apenas correções ortográficas ou de pontuação. Se o teste não possui alteração de comportamento, potencialmente pode ser reutilizado, pois testa uma parte não alterada do sistema. Para isso, precisamos de critérios que facilitem identificar se um teste, que passou por modificações, possui alteração de comportamento ou não. É neste sentido que esta dissertação se propõe a acompanhar a evolução de especificações de caso de uso CLARET e seu impacto nos testes TBM, gerados automaticamente utilizando a ferramenta LTS-BT, objetivando evitar o descarte desnecessário de testes. Para alcançar-

mos tal objetivo, foi delineada uma estratégia e desenvolvida uma ferramenta, que utiliza os artefatos gerados através do CLARET, para comparar os casos de teste gerados a partir da versão original de um caso de uso, com os casos de teste gerados após modificações nesse mesmo caso de uso. O resultado obtido dessa comparação provê suítes de teste distintas para os testes identificados como: reutilizáveis; ou obsoletos em razão de modificações de comportamento; ou obsoletos apenas por consequência de pequenas mudanças, potencialmente advindas de refatorações, e por consequência disso capaz de serem reutilizados; ou novos testes, que não existiam quando gerada a suíte para o caso de uso original. Os critérios para se fazer a identificação dos testes se basearam nos estudos trazidos nas Seções 3.1 e 4.1, utilizando funções de similaridade [13] para viabilizar as comparações textuais entre os testes.

1.1 Exemplo Motivante

Em um projeto de desenvolvimento de software, decide-se usufruir dos benefícios propiciados por TBM por meio de casos de uso descritos em linguagem natural. Casos de uso descrevem o uso do sistema por um determinado ator (agente que interage com o sistema), com a intenção de atingir um objetivo específico [32]. Um melhor detalhamento sobre casos de uso pode ser visto na Seção 2.2.

Para o projeto, foi elicitado o seguinte requisito: *"Para se cadastrar no sistema, o usuário deve fornecer seu nome, e-mail e definir uma senha contendo seis ou mais caracteres. Caso o e-mail seja inválido ou senha não esteja no padrão desejado, o sistema deve exibir uma mensagem informando o erro, caso contrário, o sistema deve exibir a mensagem que o usuário foi cadastrado com sucesso."*

A Figura 1.1a apresenta a especificação do requisito elicitado, escrito seguindo a notação de caso de uso. Para este exemplo trazido, o caso de uso descreve como deve ser a funcionalidade de cadastro de usuário para um determinado sistema. No que se refere a estrutura do caso de uso, em termos gerais, podemos ver na Figura 1.1a o ator que fará a interação com o sistema, o objetivo do caso de uso, o fluxo de execução principal para a realização da atividade (fluxo básico), e possíveis fluxos alternativos e de exceção, que fazem desvios na aplicação durante a execução do fluxo básico. Os fluxos básico, alternativo e de exceção

são compostos por passos, que podem ser ações realizadas pelo ator, ou respostas do sistema para tais ações. As pré-condições, são condições que devem ser atendidas previamente à execução dos fluxos para o correto funcionamento. As pós-condições, condições que devem ser satisfeitas ao final da execução do caso de uso e indicam a conclusão esperada para a funcionalidade abordada. A Figura 1.1b exibe o fluxo de execução para a funcionalidade descrita na Figura 1.1a, mas desta vez, no formato de um grafo. Nesse grafo direcionado, os passos descritos no caso de uso passam a ser as arestas, intercalando ações realizadas pelo ator perante o sistema com as respostas fornecidas pelo sistema para tais ações. Os vértices do grafo representam os diferentes estados ou posições no sistema durante o fluxo de execução da funcionalidade representada pelo grafo.

Casos de teste para o sistema poderiam ser gerados por meio de ferramentas TBM, seguindo os fluxos existentes na Figura 1.1b. Uma suíte de teste possível seria: $S1 = \{tc1, tc2, tc3\}$, sendo $tc1 = [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7]$, $tc2 = [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 6 \rightarrow 7]$ e $tc3 = [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7]$.

Suponha que, durante o ciclo de desenvolvimento, decide-se atualizar este requisito nos seguintes pontos: (i) criação de um novo passo no fluxo de exceção 4.2, para que o usuário insira a senha correta e o sistema seja redirecionado para a mensagem de sucesso; (ii) atualização da descrição dos passos descritos no fluxo de exceção 4.1, para melhorar sua leitura. Essas modificações podem ser observadas no caso de uso atualizado, na Figura 1.2a, e no grafo com as ações atualizadas (destacadas com a cor azul), na Figura 1.2b.

Uma vez o modelo atualizado, pode-se facilmente gerar uma nova suíte de testes para que esta esteja em conformidade com a especificação. Seguindo novamente os fluxos do sistema, a nova suíte seria: $S1' = \{tc1', tc2', tc3'\}$, sendo $tc1' = [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7]$, $tc2' = [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \mathbf{5} \rightarrow \mathbf{8} \rightarrow \mathbf{6} \rightarrow 7]$ e $tc3' = [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \mathbf{5} \rightarrow \mathbf{9} \rightarrow \mathbf{6} \rightarrow 7]$.

Comparando *S1* com *S1'*, no intuito de identificar testes que compreendem partes modificadas do sistema, podemos definir *tc1* como reutilizável, por não incluir nenhum passo modificado. Já os testes *tc2* e *tc3*, incluem passos/ações modificados na especificação tornando-os obsoletos ao sistema, de acordo com o que propõe Oliveira et al. [49]. No entanto, os passos modificados em *tc2* não alteram o fluxo do sistema, nem sua semântica (ou comportamento). Nesse caso, *tc2* poderia ser reutilizado com pequenas modificações, evitando seu descarte e

Figura 1.1: Especificação do Requisito

(a) Caso de Uso Textual

Componente do Caso de Uso	Descrição
Nome do Caso de Uso	Cadastrar usuário
Identificador	UC01
Objetivo	Cadastrar usuário
Descrição	Este caso de uso tem por objetivo permitir a inclusão de um novo usuário no sistema.
Atores	Usuário
Pré-condições	Existe conexão ativa com a internet.
Pós-condições	Usuário deve estar cadastrado no sistema.
Fluxo Básico	1. Usuário inicia tela de cadastro 2. Sistema exibe tela com os campos nome, email e senha para cadastro 3. Usuário preenche os campos nome, email e senha 4. Sistema valida campos e exibe mensagem de sucesso
Fluxos Alternativos	*Não há*
Fluxos de Exceção	*4.1 Email inválido* *1. Sistema alerta email inválido* *2. Usuário insere email válido* *3. Retorna ao passo 4 do fluxo básico* *4.2 Senha curta* *1. Sistema alerta senha deve conter mais que 5 caracteres* *2. Retorna ao passo 2 do fluxo básico*

(b) Grafo com o Fluxo do Sistema

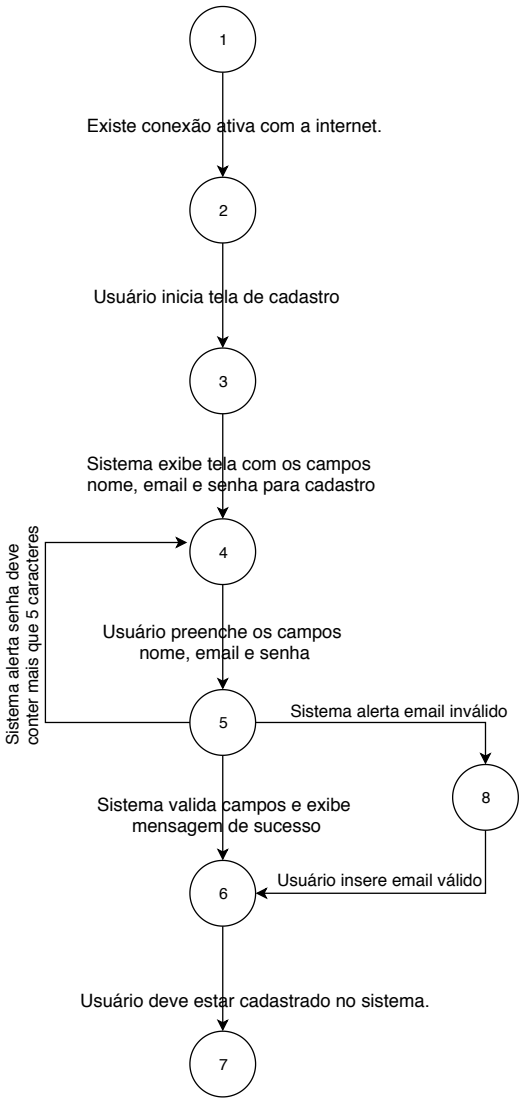
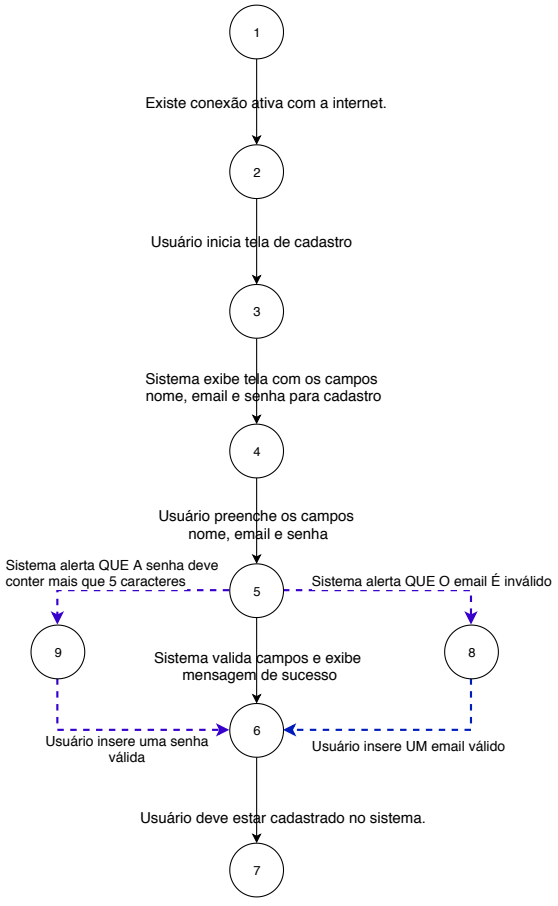


Figura 1.2: Especificação Atualizada do Requisito

(a) Caso de Uso Textual

Componente do Caso de Uso	Descrição
Nome do Caso de Uso	Cadastrar usuário
Identificador	UC01
Objetivo	Cadastrar usuário
Descrição	Este caso de uso tem por objetivo permitir a inclusão de um novo usuário no sistema.
Atores	Usuário
Pré-condições	Existe conexão ativa com a internet.
Pós-condições	Usuário deve estar cadastrado no sistema.
Fluxo Básico	1. Usuário inicia tela de cadastro 2. Sistema exibe tela com os campos nome, email e senha para cadastro 3. Usuário preenche os campos nome, email e senha 4. Sistema valida campos e exibe mensagem de sucesso
Fluxos Alternativos	*Não há*
Fluxos de Exceção	*4.1 Email inválido* *1. Sistema alerta QUE O email É inválido* *2. Usuário insere UM email válido* *3. Retorna ao passo 4 do fluxo básico*
	4.2 Senha curta *1. Sistema alerta QUE A senha deve conter mais que 5 caracteres* *2. Usuário insere uma senha válida* *3. Retorna ao passo 4 do fluxo básico*

(b) Grafo com o Fluxo do Sistema



substituição por um novo teste.

Ao passo que conseguimos reutilizar um maior número de testes, conseguimos focar nos testes que de fato exercitam partes modificadas no sistema, evitando o descarte e perda dos dados históricos (e.g., execuções, resultados obtidos) que foram reutilizados. No exemplo utilizado anteriormente, poderíamos facilmente executar toda a suíte, mas em sistemas maiores isso se torna inviável, pois uma modificação na especificação pode impactar um número maior de testes. Fazer uma análise manual dos testes para saber quais reutilizar não é prático, tornando interessante meios automáticos que os identifiquem.

Baseado nisso, este trabalho busca traçar uma forma automática que auxilie na detecção de casos de testes passíveis de reúso que antes seriam descartados.

1.2 Objetivos

Este trabalho tem como objetivo principal apoiar o uso de TBM em projetos de desenvolvimento de software. Mais especificamente, utilizando funções de similaridade entre sentenças escritas em linguagem natural, buscamos identificar automaticamente casos de teste que, embora exercitem passos modificados do modelo, possam ser reutilizados.

Neste sentido, definimos como questão de pesquisa para guiar nosso trabalho:

- **QP:** É possível minimizar, de forma automática, o descarte de casos de teste após alterações/evolução nos modelos?

No intuito de atingir o objetivo geral e responder nossa questão de pesquisa central, foram traçados os objetivos específicos:

1. Investigar como as edições nos modelos do sistema, ao longo das evoluções realizadas, impactam as suítes de teste;

2. Investigar a viabilidade do uso de funções de distância entre *strings* para classificação de edições inseridas nos modelos;

3. Definir uma estratégia que possibilite a identificação dos casos de testes, ditos obsoletos, que são passíveis de reutilização com pouco esforço;

4. Implementar estratégia para classificação automática de casos de teste em obsoletos sintáticos, semânticos, reutilizáveis e novos.

5. Validar estratégia proposta.

1.3 Escopo

O campo TBM é vasto, com diversas formas de aplicação para diferentes cenários de desenvolvimento. O escopo deste trabalho limita-se aos testes de sistema descritos em linguagem natural, gerados via TBM utilizando modelos de caso de uso descritos em CLARET (modelos CLARET serão explicados na Seção 2.4). Os testes de sistema são manuais por serem de alto nível de abstração e não possuem nenhum tipo de transformação para tornarem-se automáticos.

1.4 Contribuições

Em resumo, as principais contribuições deste trabalho são:

- Um estudo exploratório que avalia as modificações introduzidas nos modelos durante as evoluções do sistema, classificadas nesse trabalho como sintáticas ou semânticas, juntamente com a aferição do impacto produzido por essas modificações nas suítes de teste;

- Uma estratégia para minimizar o descarte de testes TBM, impactados via edições nos modelos;

- Uma ferramenta que aplica a estratégia elaborada, que viabiliza a comparação de duas suítes de teste para detecção de testes impactados por modificações inseridas nos modelos de caso de uso;

- Contribuição no sentido de promover o uso de TBM de forma a se obter um maior reúso de testes.

1.5 Estrutura da Dissertação

Seguindo a organização desta dissertação, o próximo capítulo apresenta a fundamentação teórica (Capítulo 2), que traz conceitos relevantes para facilitar o entendimento deste trabalho. O Capítulo 3 revela a dimensão do problema com um estudo de caso executado em dois projetos industriais. No Capítulo 4 é detalhada a abordagem adotada que busca minimizar o problema alvo. Já o Capítulo 5 relata os estudos realizados, exploratório e empírico, que embasam o problema abordado neste trabalho e validam a estratégia proposta. O Capítulo 6 levanta trabalhos relacionados, ligados aos conceitos por nós abordados, e por fim, no Capítulo 7, apresentamos as conclusões para o conteúdo aqui discutido.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta conceitos teóricos e informações necessárias para o entendimento do trabalho desenvolvido nesta dissertação. Conceitos básicos relacionados ao teste de *software* e teste baseado em modelo estão presentes na Seção 2.1, caso de uso na Seção 2.2, funções de similaridade se encontram na Seção 2.3 e explanações acerca do CLARET e LTS-BT podem ser vistas nas Seções 2.4 e 2.5, respectivamente.

2.1 Teste de Software

Nos dias atuais, estamos envolvidos rotineiramente por uma gama de sistemas e dispositivos. A tecnologia é algo que está presente no nosso cotidiano e nos auxilia nas mais diversas atividades, no trabalho, estudo, lazer, entre outras. Como usuário, sempre esperamos que esses sistemas funcionem de forma satisfatória, executando corretamente os processos a que se propõem. Teste de *software* é o processo que objetiva avaliar sistemas em desenvolvimento e dar a confiança que tais sistemas possuem os comportamentos esperados [2].

A atividade de teste de *software* busca detectar diferenças entre o comportamento atual de um sistema e o pretendido, bem como aumentar a confiança no sentido de que os comportamentos estão em conformidade com o desejado. No desenvolvimento de software, detectar tardiamente um defeito no sistema geralmente está associado a uma maior dificuldade para solucionar tal problema, maior custo associado, ou até mesmo comprometimento da segurança do sistema [61].

Na literatura é comum nos depararmos com três conceitos relacionados com teste de

software: erro, falta e falha. Binder esclarece tais conceitos como [5]:

1. **Erro:** ação humana que acarreta em uma falta no sistema;
2. **Falta:** ausência de código necessário ao programa ou presença de código incorreto, o que pode acarretar em uma falha no sistema;
3. **Falha:** resultado aparente de uma falta presente no sistema. É um comportamento do sistema diferente do esperado ou a falta de um requisito necessário.

Os testes de *software* abrangem diversos níveis do sistema e podem ser classificados em diferentes tipos. Utting e Legeard [64] nos apresentam vários tipos, entre eles:

- **Testes de unidade:** testam uma única unidade do sistema por vez, como um procedimento ou classe;
- **Testes de componente:** testam uma porção maior do sistema, quando comparados aos testes de unidade. Testam os componentes ou subsistemas que formam o sistema como um todo;
- **Testes de integração:** objetivam assegurar que os componentes funcionam corretamente quando em conjunto, uns com os outros;
- **Testes de sistema:** testam o sistema como um todo;
- **Testes funcionais:** exploram as funcionalidades do sistema. Pode-se verificar se são produzidas as saídas corretas para determinadas entradas.

Além dos níveis do sistema, os testes podem ser classificados quanto às características do sistema que se pretende testar (funcional, robustez, performance e usabilidade), e pelo tipo de informação que o teste necessita: i) os testes caixa-preta são testes funcionais e não envolvem detalhes de implementação; e, ii) os testes caixa-branca que são estruturais e fazem uso do código implementado do sistema para testar partes internas do sistema. Testes de unidade voltados à cobertura de caminhos do código testado e responsáveis por testar componentes unitários de código se certificando que tais componentes funcionam como esperado, são exemplo de teste caixa-branca, necessitando de acesso ao código implementado. Já teste baseado em modelo, são testes de maior nível de abstração, que utilizam modelos e

especificações para testar funcionalidades de um sistema, abstraindo os processos internos ao sistema.

2.1.1 Teste Baseado em Modelo

O termo Teste Baseado em Modelo (TBM) é utilizado em diferentes contextos. Para melhor entendermos, listaremos as quatro abordagens mais conhecidas de uso de TBM [64]:

1. Geração de de dados de entrada para testes a partir de um modelo de domínio.
2. Geração de casos de teste a partir do modelo de ambiente.
3. Geração de casos de teste com oráculos utilizando um modelo comportamental.
4. Geração de *scripts* de teste fazendo uso de testes abstratos.

Utting e Legeard [64] explicam TBM como sendo a automação do processo de geração de testes caixa-preta, sendo que cada ponto citado acima tem sua forma particular de geração de testes. De acordo com a abordagem, os testes gerados podem ser automáticos ou não e variar o modelo utilizado para tal. O escopo do nosso trabalho é voltado exclusivamente para a terceira abordagem citada: geração de casos de teste com oráculos utilizando um modelo comportamental. A escolha desta abordagem em particular se deu porque esta envolve tanto os valores de entrada para os testes, a sequência de chamadas ao Sistema Sob Teste (SST), como também os oráculos necessários para avaliar os testes comparando-os com os valores de saída esperados [64]. Tais pontos são necessários ao contexto onde os estudos desta dissertação foram realizados, com uso de ferramentas TBM para testes de projetos em desenvolvimento.

Ferramentas TBM automatizam o processo de construção dos casos de teste, evitando o esforço da criação manual. Para isso, o testador deve descrever os comportamentos do sistema em um modelo abstrato que será lido por uma ferramenta TBM, responsável pela geração dos testes para esse modelo. Para melhor detalhamento, podemos dividir o processo de teste baseado em modelos em cinco etapas [64] ilustradas na Figura 2.1 e elencadas abaixo:

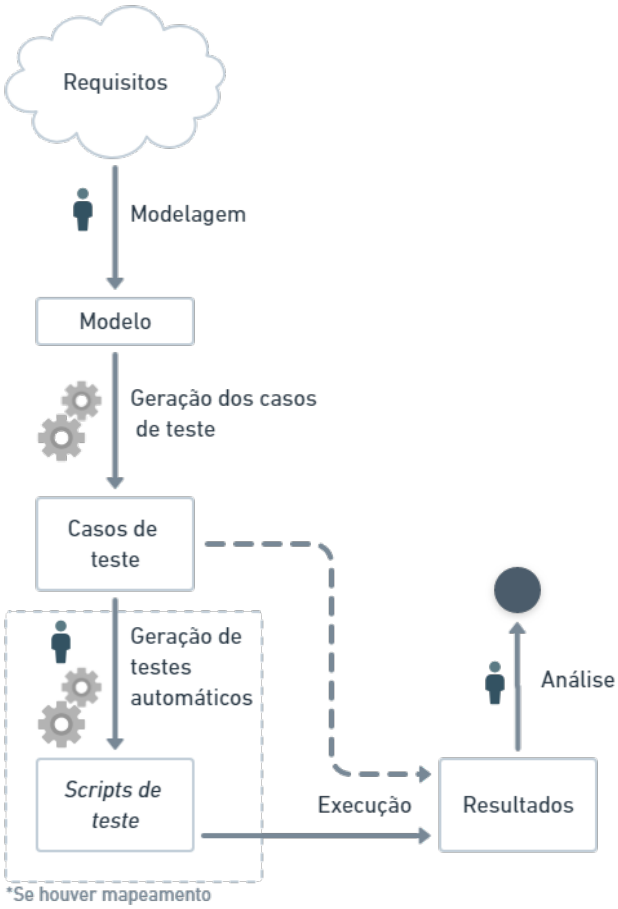


Figura 2.1: Processo TBM

1. Modelagem do Sistema sob Teste (SST).
2. Geração dos casos de teste abstratos.
3. Transformação dos testes abstratos em executáveis.
4. Execução dos testes no SST.
5. Análise dos resultados.

A primeira etapa consiste em escrever o modelo que representa o sistema a ser testado. A modelagem deve focar nos pontos principais do sistema, os quais se quer testar, evitando o detalhamento extremo do SST. Nesta etapa é importante validar o modelo construído, quando necessário pode-se utilizar ferramentas de verificação automática, a fim de garantir que o

modelo está em conformidade com o comportamento desejado. Uma modelagem incorreta refletirá em uma suíte de testes errônea dada sua relação direta com o modelo.

A segunda fase utiliza um critério de seleção, escolhido pelo testador, para geração dos casos de teste abstratos. O critério de seleção define a forma que o algoritmo de geração funcionará. Os testes abstratos gerados são formados por uma sequência de operações indicando o passo-a-passo a ser executado no SST. Usualmente, esses testes não são automáticos por serem oriundos de um modelo que não possui detalhes internos ao sistema.

Já na terceira etapa, os testes abstratos são transformados em testes executáveis, desde que haja uma forma de mapeamento entre os testes em alto nível para baixo nível. Este passo pode ser desconsiderado em cenários onde a execução do teste é manual. No contexto do nosso trabalho, esta etapa não é executada, visto que os modelos são de alto nível (casos de uso descritos em linguagem natural), sem informações estruturais do sistema sob teste.

As etapas quatro e cinco estão presentes em qualquer que seja o processo de teste, não só no TBM. Elas compreendem a execução dos casos de teste no SST junto com o armazenamento dos resultados e na análise dos testes falhos para determinar as causas relacionadas e posteriormente executar as correções. De maneira análoga a outros processos de teste, caso um teste falhe, deve-se verificar se a causa é advinda de uma falta no sistema ou se foi pelo teste estar incorreto, o que indicaria uma inconsistência no modelo.

Adotando TBM como processo de teste, vários benefícios podem ser obtidos. Utting e Legeard [64] mostram que: *(i)* o poder de detecção de faltas no sistema quando se utiliza TBM é igual ou superior a uma suíte de teste construída manualmente; *(ii)* o tempo e custo dedicados ao teste do sistema são reduzidos; *(iii)* a qualidade dos testes aumenta, uma vez que são gerados de forma sistemática cobrindo as várias partes do sistema; *(iv)* o processo de escrita dos modelos ajuda na detecção de falhas nos requisitos, levantando questões importantes que devem ser resolvidas antes do processo de desenvolvimento e teste; *(v)* o relacionamento próximo entre os testes e requisitos contribui na associação dos casos de teste com os requisitos que eles exploram, dando uma visão melhor de quais partes do SST e dos requisitos estão sendo exploradas pelos testes; e *(vi)*, a manutenção dos modelos já implica na manutenção dos testes. Manter os modelos atualizados é suficiente para poder regerar os testes e tê-los atualizados.

Uma vez que foi falado dos benefícios, é válido ressaltar que existem algumas limitações

relacionadas ao tópico. O uso de TBM requer habilidade para abstrair e descrever o sistema em um modelo. Isso pode demandar treinamento e tempo para o aprendizado inicial. O uso de TBM é frequentemente aplicado para testes funcionais, e por este ser um teste de mais alto nível, não é trivial transformar modelos abstratos em testes de baixo nível. O uso de modelos desatualizados acarreta no descobrimento de falsas faltas no SST, por isso é necessário mantê-los sempre atualizados. Além desses, o uso de TBM usualmente gera uma grande quantidade de testes, fazendo necessário um tempo maior para analisar os resultados. E um ponto importante é: pequenas mudanças nos modelos podem afetar um grande número de testes.

Embora a aplicação de TBM tenha se mostrado capaz de gerar bons resultados na prática [25, 47], a criação e manutenção de modelos requer habilidades específicas do testador e demanda tempo, o que pode tornar este um fator restritivo quanto a aplicação de TBM em metodologias de desenvolvimento ágil. Metodologias ágeis priorizam a entrega rápida de artefatos que gerem valor para o cliente e são mais flexíveis à mudanças durante o ciclo de desenvolvimento [60]. Ainda sobre este ponto, soluções como a proposta pelo grupo de Sarmiento et al. [56], Budha et al. [8] e Jorge et al. [35] podem ser utilizadas para unir esses dois campos. Voltaremos nosso foco ao longo desta pesquisa, para o trabalho apresentado por Jorge et al. [35], que propõe uma ferramenta para especificação de casos de uso em linguagem natural visando facilitar a criação e manutenção de requisitos os quais possibilitam a geração automática de testes funcionais. A diminuição da complexidade de escrita dos modelos viabiliza a integração da escrita de requisitos no processos ágeis.

De modo geral, TBM demonstra um avanço na área de Teste de Software e sua aplicação tem se mostrado efetiva na detecção de faltas [64]. As limitações existem mas podem ser minimizadas quando é feito o uso correto, seguindo o processo de forma adequada.

2.2 Casos de Uso

A *Unified Modeling Language* (UML)[1], é uma linguagem voltada para a especificação, visualização e documentação de artefatos relacionados ao desenvolvimento de software. Dentre as modelagens possíveis de se fazer com a UML, temos os casos de uso. Casos de uso são

[1] https://www.uml.org/

modelos comportamentais que descrevem como um ator deve utilizar o sistema para atingir um objetivo desejado e o que esse mesmo sistema deve fazer para que esse objetivo seja atingido [6]. O ator representa o papel de um usuário ou de um sistema terceiro, durante a execução do sistema descrito pelo caso de uso.

Casos de uso facilmente englobam um conjunto de requisitos, descrevendo como deve ser o funcionamento do sistema à medida que interações são feitas por usuários ou outros sistemas. São uma forma de organizar os requisitos do software em desenvolvimento em um formato de fácil gerência, propiciando melhor entendimento das funcionalidades do sistema e facilitando a discussão sobre os comportamentos necessários no sistema.

Uma vez que os casos de uso descrevem um fluxo de ações feitas pelos atores e pelo sistema, podemos utilizá-los na validação dos comportamentos implementados. Além disso, é possível mapeá-los para outros modelos, e.g. *Trivial Graph Format* (TGF), os quais podem ser utilizados para geração automática de casos de teste [3].

Os casos de uso são descritos em formato textual e seguem uma estrutura predefinida com a especificação do fluxo dos eventos, indicando como deve ser a colaboração entre o sistema e atores para que seja obtido o que está definido pelo caso de uso [7].

A estrutura da descrição textual de um caso de uso, é formada principalmente por:

- Pré-condições: critérios que devem ser atendidos para que se possa iniciar a execução do caso de uso. O caso de uso só deve ser executado pelo ator quando atendidas as pré-condições;
- Pós-condições: descreve o estado final que onde caso de uso deve chegar. Representa o objetivo desejado pelo ator e razão da execução do caso de uso;
- Fluxo básico: também conhecido por "caminho feliz", o fluxo básico contém a descrição do fluxo esperado para se atingir o objetivo do caso de uso;
- Fluxos alternativos: desvios do fluxo básico que podem ser tomados por decisões dos atores;
- Fluxos de exceção: tratam os fluxos de exceção no sistema durante a execução do caso de uso.

Componente do Caso de Uso	Descrição
Nome do Caso de Uso	Cadastrar usuário
Identificador	UC01
Objetivo	Cadastrar usuário
Descrição	Este caso de uso tem por objetivo permitir a inclusão de um novo usuário no sistema.
Atores	Usuário
Pré-condições	Existe conexão ativa com a internet.
Pós-condições	Usuário deve estar cadastrado no sistema.
Fluxo Básico	1. Usuário inicia tela de cadastro 2. Sistema exibe tela com os campos nome, email e senha para cadastro 3. Usuário preenche os campos nome, email e senha 4. Sistema valida campos e exibe mensagem de sucesso
Fluxos Alternativos	*Não há*
Fluxos de Exceção	*4.1 Email inválido* *1. Sistema alerta que o email é inválido* *2. Usuário insere umemail válido* *3. Retorna ao passo 4 do fluxo básico* *4.2 Senha curta* *1. Sistema alerta que a senha deve conter mais que 5 caracteres* *2. Usuário insere uma senha válida* *3. Retorna ao passo 4 do fluxo básico*

Figura 2.2: Caso de Uso

Vejamos novamente o requisito encontrado na Seção 1.1: *"Para se cadastrar no sistema, o usuário deve fornecer seu nome, e-mail e definir uma senha contendo seis ou mais caracteres. Caso o e-mail seja inválido ou senha não esteja no padrão desejado, o sistema deve exibir uma mensagem informando o erro, caso contrário, o sistema deve exibir a mensagem que o usuário foi cadastrado com sucesso."*. A Figura 2.2, exibe a descrição textual do caso de uso que modela o requisito em questão, com as informações e fluxo de execução necessários para alcançar o objetivo abordado pelo caso de uso.

O software, uma vez modelado e implementado, não entra em um estado imutável. A sua mudança e evolução se faz necessária para que se mantenha relevante ao contexto onde se está inserido. Da mesma maneira, os casos de uso devem se manter em evolução, para que estes reflitam o comportamento atual do sistema. Edições voltadas à melhoria do caso uso também são bem vindas, objetivando torná-los mais claros e de fácil entendimento [6]. Se tratando de edições realizadas nos casos de uso, elas podem ser classificadas de duas maneiras [58]: i) *semântica:* qual é realizada com o objetivo de promover ou refletir alterações no comportamento do sistema; ii) *sintática:* quando as alterações são de caráter gramatical, visando apenas facilitar a leitura do caso de uso ou corrigir erros de escrita. Esses dois tipos de edições podem ser chamados de semânticos e sintáticos, respectivamente.

2.3 Funções de Similaridade entre *Strings*

Funções de similaridade para *strings* [13] mapeiam um par de *strings* em um número real, onde, quanto maior o valor obtido mais similar é o par analisado. Tais funções são utilizadas em diversas áreas, e.g., detecção de duplicatas durante integração de bancos de dados [18, 70], e implementação de mecanismos de busca e comparação textual [68]. No âmbito de testes também encontramos trabalhos que fazem uso dessas funções para: seleção de casos de teste [10], integração contínua com seleção de testes [48] e priorização de casos de teste [38].

A similaridade entre *strings* pode ser abordada de duas formas: léxica ou semântica. Se duas palavras possuírem a mesma construção de caracteres, elas são consideradas lexicalmente similares. De maneira diferente, se duas palavras possuem o mesmo contexto e significado, mesmo que não sejam construídas da mesma maneira, são tidas como semanticamente similares [24]. Nosso trabalho é voltado apenas ao uso de funções de nível léxico.

Podemos agrupar as funções de similaridade léxica [27] em: i) baseadas em conjunto (*set-based)*: compara dois conjuntos, ignorando ordem ou precedência entre os elementos dos conjuntos, e.g. Jaccard; ou, ii) baseadas em sequência (*sequence-based)*: avalia sequências de *strings* e nesse caso a ordem dos elementos impacta no resultado gerado, e.g. Levenshtein.

Dentre as funções utilizadas no contexto dessa dissertação, a de Levenshtein [40] possui o maior enfoque, conforme justificativa apresentada na Seção 4.3. Sua distância é calculada contando o número mínimo de operações necessárias para transformar uma *string* em outra, utilizando operações de inserção, substituição ou exclusão de um único caractere. Dadas duas *strings* s e t, essa métrica computa quantas operações unitárias são necessárias para transformar s em t. Assim, significa dizer que, quanto menor o resultado obtido para s e t, mais similares elas são. Por exemplo, considere s = sacudir e t = aplaudir, a distância Levenshtein para esse caso seria quatro, indicando que foram necessárias quatro operações para transformar s em t: (i) adição do 'a' no início; (ii) substituição de 's' por 'p'; (iii) substituição de 'a' por 'l'; (iv) substituição de 'c' por 'a'.

As demais funções citadas nesta dissertação são:

- Jaro-Winkler: se baseia na quantidade e na ordem de elementos em comum entre as duas *strings*. Na sua comparação atribui peso maior às diferenças mais espalhadas nas

strings e peso menor para diferenças próximas umas das outras [24].

- Cosine: determina a similaridade entre duas sentenças representando cada uma delas na forma de vetor, levando em consideração a frequência das palavras. Utilizando os vetores, é aplicado a similaridade de cossenos [1].
- N-Gram: compara diferentes subsequências do texto dado buscando verificar as subsequências em comum, dentre todas as existentes, das *strings* analisadas [24].
- Jaccard: computa o número dos termos em comum nas *strings* com relação aos termos únicos de cada *string* avaliada [24].
- Sorensen-Dice: similar a Jaccard, no entanto, ao invés de relacionar o total termos em comum com os termos únicos, relaciona com a soma de todos os termos em ambas as *strings* [24].

2.4 CLARET

Do inglês, *Central Artifact for Requirements Engineering and Model-Based Testing* [35], CLARET é uma linguagem de domínio específico (DSL) que se propõe a ser o elo de ligação entre a engenharia de requisitos e o teste de software. Com CLARET é possível especificar requisitos em forma de casos de uso utilizando uma notação predefinida. Estas especificações são utilizadas para a geração automática de casos de teste abstratos para execução manual e para geração de documentos de requisitos [34].

A ferramenta possibilita a escrita de casos de uso utilizando linguagem natural dentro de uma estrutura semi-controlada utilizando elementos chave já bem difundidos pela UML. O uso da linguagem natural e de elementos usados na UML minimizam a curva de aprendizagem e, principalmente, torna a escrita de casos de uso ágil e simples. Desta maneira, a modelagem se detém a descrição das interações entre o ator, que exercita o sistema, e o próprio sistema, com as respostas para as ações do ator. A descrição das interações possibilita a geração automática de testes de alto nível, independente da implementação do sistema.

A sintaxe da notação proposta por CLARET é formada pelos elementos: *systemName, usecase, actor, preCondition, postCondition, basic, alternative,* e *exception*. Para exemplificar o uso da ferramenta, a Listagem 2.1 ilustra o uso da notação para descrever o seguinte

caso de uso de um sistema de venda de ingresso de um cinema: *"Para realizar a compra de um ingresso, o usuário deve selecionar o filme e a sessão desejada. Feita a seleção pelo cliente, o sistema deve solicitar a inserção do cartão de crédito na máquina e o fornecimento da respectiva senha. Se a senha for válida, deve ser exibido uma confirmação de compra e o ingresso deverá ser impresso pela máquina. Caso contrário, o sistema deve solicitar uma nova senha."*

```
1  system "Cinema", {
2    usecase "Comprar Ingresso", {
3      version "1.0", type:"Criação", user:"Anderson", date:"01/06/2018"
4      actor cliente, "Cliente"
5      preCondition "Existe conexão ativa com a internet"
6      basicFlow {
7        step 1, cliente, "escolhe o filme desejado"
8        step 2, system, "exibe as sessões disponíveis para o filme selecionado"
9        step 3, cliente, "seleciona uma sessão da listagem"
10       step 4, system, "solicita a inserção do cartão de crédito"
11       step 5, cliente, "insere o cartão de crédito na máquina", af:[1]
12       step 6, system, "solicita a senha para o cartão inserido"
13       step 7, cliente, "digita a senha"
14       step 8, system, "confirma a compra e imprime o ingresso", ef:[1]
15     }
16     alternative 1, "Cancela a compra", {
17       step 1, cliente, "cancela a compra em andamento"
18       step 2, system, "solicita confirmação"
19       step 3, cliente, "confirma ação de cancelar", bfs:2
20     }
21     exception 1, "Senha incorreta", {
22       step 1, system, "informa que a senha é incorreta", bfs:7
23     }
24     postCondition "O ingresso deve ter sido impresso."
25   }
26 }
```

Listagem 2.1: Caso de Uso em CLARET

Os primeiros passos (linhas 1 e 2) contém a definição do nome do sistema e do caso de uso. Na linha 3, constam informações de versionamento, seguido pelo ator (linha 4), no nosso caso um cliente do cinema, e pré-condições necessárias na linha 5.

As linhas 6 a 15 compreendem o fluxo principal de execução. Os passos desse fluxo devem ser sempre alternados entre estímulos realizados pelo ator e respostas do sistema.

Cada passo tem um identificador sequencial e sua descrição é feita em linguagem natural podendo conter desvios para fluxos alternativos (linha 11) ou de exceção (linha 14).

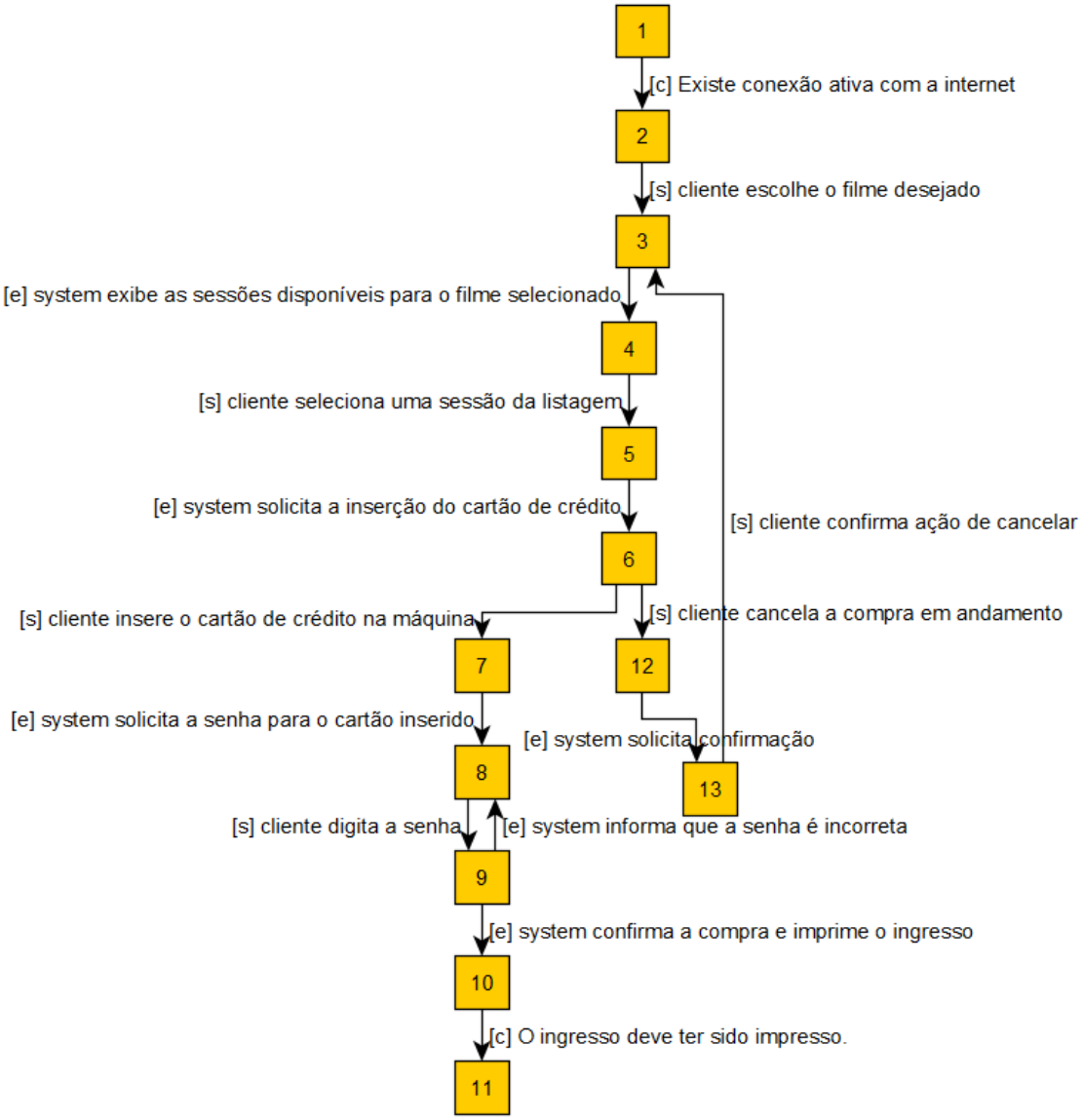


Figura 2.3: Exemplo de TGF Gerado por CLARET

Os fluxos alternativos e de exceção contém um identificador sequencial seguidos por uma descrição. O corpo desses fluxos também deve ter alternância entre estímulo e resposta e podem conter um passo com retorno para o fluxo básico (linhas 19 e 22). Em particular, o fluxo alternativo deve sempre iniciar com um estímulo e o fluxo de exceção com uma resposta, já que estes representam, respectivamente, um caminho alternativo tomado pelo

usuário e um caminho de exceção do sistema. Por fim, é declarada a pós-condição (linha 24), que deve ser atendida quando o caso de uso é realizado.

Terminada a descrição do caso de uso, a ferramenta CLARET nos gera o modelo resultante no formato *Trivial Graph Format* (TGF), que pode ser visto na Figura 2.3. Este modelo pode ser visualizado utilizando ferramentas de visualização, e.g. *yEd*[2], e é o artefato principal para geração dos testes utilizando a ferramenta *Labeled Transition System Based-Testing* (LTS-BT), explanada na Seção 2.5.

2.5 LTS-BT

Como peças chave no processo TBM temos as ferramentas de geração de casos de teste. Em nosso contexto, a ferramenta utilizada é a *Labelled Transition System-Based Testing* (LTS-BT) [9]. Ela possibilita a geração e seleção de casos de teste utilizando estratégias predefinidas.

LTS-BT gera testes funcionais para execução manual no SST. O seu funcionamento se baseia no artefato de entrada, o arquivo no formato TGF gerado por CLARET, e a partir disso são gerados os casos de testes percorrendo os diversos caminhos do sistema de transição TGF, os quais representam os fluxos existentes no SST. Além disso, é possível fazer uma seleção automática dos testes gerados, provendo uma suíte reduzida para quando a execução completa não é viável. A estratégia para esta seleção se baseia na similaridade entre os testes, reduzindo aqueles que possuem certo grau de transições idênticas.

Os testes gerados pela ferramenta são estruturados no formato XML e podem ser importados na ferramenta de gerenciamento de testes TestLink[3]. A Figura 2.4 exemplifica a visualização de um caso de teste importado no Testlink.

[2]https://www.yworks.com/products/yed

[3]http://testlink.org/

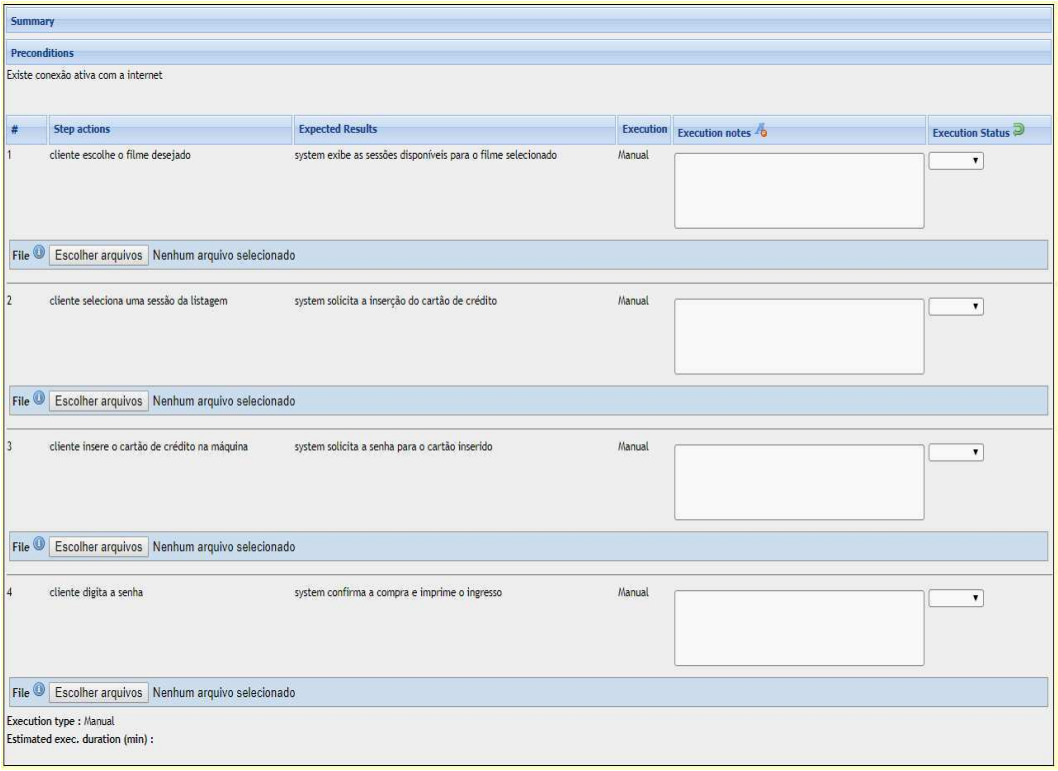


Figura 2.4: Exemplo de Caso de Teste visualizado no *TestLink*

2.6 Considerações Finais

Sobre as ferramentas apresentadas neste capítulo, a Figura 2.5 apresenta onde as ferramentas CLARET e LTS-BT se encaixam no processo TBM. Temos como ponto de partida os requisitos do sistema que precisam ser trabalhados e transformados em modelos CLARET, no nosso contexto, que por sua vez, construirá artefatos para serem utilizados na geração dos testes, utilizando a ferramenta LTS-BT. Esses testes devem ser executados no sistema para que se capture os resultados. O comportamento do sistema implementado e os testes gerados, devem estar em conformidade com a modelagem produzida, para que não se capture falsas faltas.

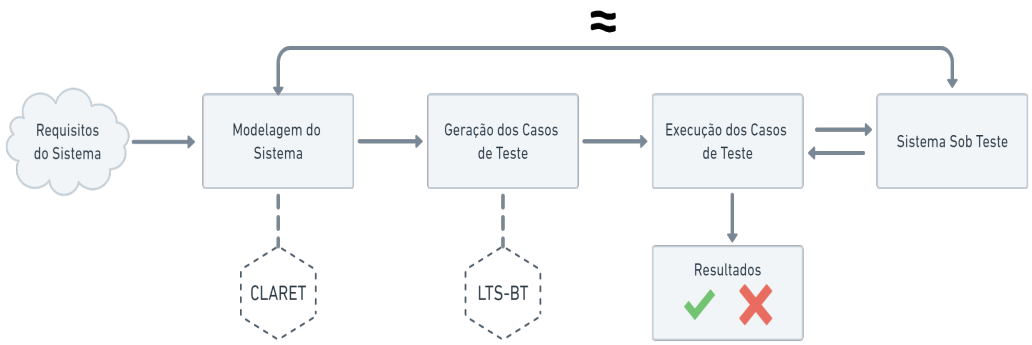


Figura 2.5: Aplicação das Ferramentas Apresentadas, no Fluxo TBM

Por fim, este capítulo apresentou os conceitos teóricos necessários para o entendimento do trabalho desenvolvido. Foram contextualizados conceitos relacionados ao teste de *software*, seus tipos de teste e sua importância. Em seguida, trouxemos informações à cerca do teste baseado em modelo, conceito, aplicação, vantagens e desvantagens. Ressaltamos conceitos sobre casos de uso e funções de similaridade de *strings*, e então, foi explanado sobre ferramentas importantes ao contexto do trabalho, CLARET e LTS-BT.

Capítulo 3

Estudo Exploratório

Como forma de investigar o problema abordado por esta dissertação em um contexto real, foi conduzido um estudo de caso, mostrado na Seção 3.1. Esse estudo serviu como base para a construção de uma estratégia capaz de minimizar o descarte desnecessário de casos de teste, exposto na Seção 4.1.

3.1 Estudo de Caso Exploratório

Este estudo de caso foi realizado em um cenário de cooperação entre o SPLab e as empresas Ingenico do Brasil Ltda[1] e Viceri Solution Ltda[2]. Os projetos SAFF e BZC, resultados dessas parcerias, respectivamente, serviram de base para a análise apresentada em sequência. O SAFF é um projeto focado no desenvolvimento de ferramentas e uso de modelos analíticos para aumento da confiabilidade de produtos da Ingenico e previsão de informações estratégicas. Já o BZC é voltado ao desenvolvimento de um sistema para otimizar as atividades logísticas de comércios eletrônicos.

Os detalhes deste estudo estão dispostos nas seguintes seções: metodologia (Seção 3.1.1), procedimentos (Seção 3.1.2), resultados (Seção 3.1.3), discussão (Seção 3.1.4) e ameaças à validade (Seção 3.1.5).

[1] https://www.ingenico.com.br/

[2] https://www.viceri.com.br/

3.1.1 Metodologia

Ambos os projetos analisados estavam inseridos no mesmo contexto: equipes de desenvolvimento utilizavam Scrum [54] para gerência das atividades, CLARET [34] para especificação dos casos de uso e LTS-BT [9] para geração das suítes baseadas em TBM. Durante o processo de escrita dos casos de uso, todas as edições inseridas ao longo do tempo eram registradas em um sistema de controle de versão, assim como as suítes geradas a partir destes modelos. Estes artefatos eram importantes para as equipes tanto para o processo de elicitação de requisitos (utilizando CLARET) como para a execução de bateria de testes (gerada utilizando LTS-BT) durante o desenvolvimento e antes de entregas do produto de software. Os testes visavam tanto checar novas funcionalidades, como também verificar se recursos já implementados não foram impactados pelos novos.

No que se refere ao processo de desenvolvimento das equipes, havia uma constante interação com o cliente, o que contribuía para alterações frequentes dos modelos de caso de uso. Não havia nas equipes um time específico para elicitação de requisitos, sendo os próprios membros da equipe de desenvolvimento que especificavam os casos de uso e os atualizavam quando necessário. Durante as atualizações dos casos de uso, com adição ou remoção de passos, também se fazia uma refatoração para melhorar a legibilidade e corrigir erros de escrita cometidos anteriormente. Para realizar os testes funcionais no sistema, era regerada a suíte de teste para que incluísse as modificações inseridas nos casos de uso, implicando em diversos testes impactados pelas mudanças. Pela dificuldade em identificar os testes que poderiam ser reutilizados e os modificados, acaba-se optando por executar toda a nova suíte.

A equipe de pesquisa deste trabalho contou a expertise de um dos membros do time de desenvolvimento para realizar a identificação e classificação das edições, inseridas durante o processo de atualização de um caso de uso, e contou com o apoio dos demais integrantes do desenvolvimento para esclarecer quaisquer dúvidas. Outra decisão importante, consideramos como teste obsoleto todo aquele que sofreu alguma modificação após a atualização do modelo de caso de uso e geração da nova suíte. Adotamos essa regra por não se ter disponível os dados dos testes que de fato foram descartados durante as execuções das baterias de teste pela equipe de desenvolvimento e por diversas vezes a equipe desconsiderar completamente a suíte antiga e executar a nova, como relatado anteriormente. Outros trabalhos, a exemplo de Oliveira et al. [49] e Leung e White [39], também assumem como testes obsoletos, aque-

les que foram modificados e possuem as relações de entrada/saída ou o fluxo a ser executado durante o teste, desatualizados.

Objetivo

Partimos do fato de que TBM tende a gerar suítes com grande número de casos de teste [64] e que uma grande quantidade de testes se tornam obsoletos ao evoluir os modelos do sistema. Então, delineamos esse estudo exploratório na intenção de analisar como edições, realizadas ao longo do tempo nos passos das especificações de casos de uso, impactam nas suítes TBM geradas automaticamente.

Questões de Pesquisa

Para alcançar nosso objetivo, definimos o conjunto de questões de pesquisa a seguir:

- **Q1**: Quanto de uma suíte de testes é descartada devido a edições nos casos de uso?
- **Q2**: Qual o impacto de alterações sintáticas e semânticas numa suíte de testes?
- **Q3**: Quanto de um caso de teste obsoleto precisa ser revisado para torná-lo reutilizável?

3.1.2 Procedimentos

Com o propósito de facilitar o entendimento dos procedimentos realizados, dividimos em etapas. As etapas são de finalidades distintas, sendo elas: obtenção do histórico dos casos de uso e suas suítes de teste; extração do conjunto de edições e rastreio de ocorrência nos testes; classificação das edições; e análise. Todas essas quatro etapas são explicadas a seguir.

Etapa 1: Obtenção do Histórico dos Casos de Uso e Suas Suítes de Teste

Como ponto de partida, mineramos os repositórios de código dos projetos para coletar o histórico de evoluções dos casos de uso ao longo do tempo. No contexto do nosso estudo, o histórico de um caso de uso é composto pelas diversas versões deste, contendo conjuntos de edições submetidos ao repositório. Na Tabela 3.1 mostramos o que foi obtido com relação aos casos de uso para cada sistema. Nela se encontra o número de casos de uso; a soma do número de versões envolvendo todos os casos de uso estudados; a média de linhas editadas,

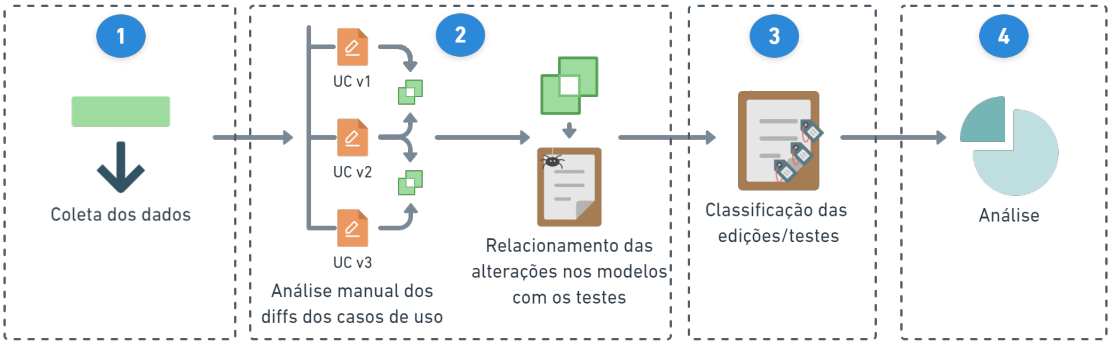


Figura 3.1: Procedimento Realizado.

obtida da comparação de pares de versões consecutivas dos casos de uso; a média de casos de teste nas suítes; e a média de *steps*, ou passos, existentes no casos de teste. Consideramos como sendo uma nova versão de um caso de uso, cada pacote de modificações enviado ao sistema de controle de versão no formato de *commit*, e sobre os *steps*, cada ação do usuário que interage com o sistema e cada resposta dada pelo sistema a ação realizada, foi considerado como um *step*. De forma geral, nosso estudo trabalhou com 28 casos de uso; um total de 79 versões; uma média de 5 linhas editadas entre duas versões destes casos de uso; 53 casos de testes em média, nas suítes; e 10 *steps* compondo os testes.

Tabela 3.1: Resumo dos Dados Analisados

	Casos de Uso	Versões	M. de Edições	M. de Casos de Teste	M. de *Steps*
SAFF	13	42	7	54	12
BZC	15	37	3	52	8

Ainda nesta etapa, coletamos as suítes de teste geradas a partir de cada versão dos modelos CLARET (casos de uso). Todas as suítes foram geradas usando a mesma configuração e critérios para a ferramenta LTS-BT [9].

Etapa 2: Extração do Conjunto de Edições e Rastreio de Ocorrência nos Testes

Nesta fase, extraímos o conjunto de edições entre duas versões adjacentes de um mesmo caso de uso. Dada a versão base de um caso de uso (c) e sua versão subsequente no histórico (c'), foram capturadas todas as edições realizadas nos passos dessas versões. Em nossa análise,

consideramos dois tipos de edições: i) *atualização de step*: qualquer passo apresentado na versão base (*c*) e que sua descrição foi alterada na versão delta do modelo (*c'*); e ii) *remoção de step*: qualquer passo que existisse em *c* e não existisse mais em *c'*. É importante destacar que foi necessário descartamos adições de *steps*.

Para futuramente calcularmos o impacto das alterações com relação a suíte já existente (suíte base), foi preciso descartar este tipo de edição, uma vez que as adições não estão presentes na suíte base e nenhuma adição poderia ser capturada na análise. No Capítulo 4, mais adiante, consideraremos casos de teste com adição de *steps*. O Capítulo 4 apresenta uma classificação para casos de teste, a qual considera os testes que tiveram *steps* adicionados, durante sua evolução, como novos testes, visto que exercitam partes do sistema antes não consideradas pelo teste.

Em seguida, utilizamos o conjunto de edições, construído anteriormente a partir da comparação de versões subsequentes de um mesmo caso de uso (*c* e *c'*), para detectar a presença dessas edições nos casos de teste que compõem a suíte gerada a partir do caso de uso base. Um *script* foi criado para automatizar essa detecção. O *script* percorria os *steps* de cada caso de teste, sempre da suíte gerada a partir do caso de uso base do comparativo de versões (*c*), verificando para cada caso de teste incluso na suíte, se algum de seus *steps* estava contido no conjunto de edições. Consideramos que um caso de teste da suíte de *c* foi afetado por uma modificação, se este incluir em seus passos ao menos um elemento do conjunto de edições. De posse disto e tendo como base as classificações de Leung e White [39] e Oliveira et al. [49] para casos de teste, foi possível agrupar os casos de teste, da suíte respectiva ao caso de uso base, em obsoletos ou reutilizáveis, onde:

- **Casos de teste obsoletos:** casos de teste que tiveram seus *steps* alterados. Estes testes possuem ações ou respostas do sistema diferentes, em comparação com a sua versão anterior.

- **Casos de teste reutilizáveis:** casos de teste que exercitam parte do sistema que não sofreu modificação. Todas as suas ações e respostas do sistema condizem com sua versão anterior.

A Figura 3.1 traz um resumo dos procedimentos realizados no estudo. As etapas 1 e 2 da figura foram explicadas acima e daremos continuidade com a 3 e 4.

Etapa 3: Classificação das Edições

De posse do conjunto de edições realizadas nos casos de uso, conjunto esse construído na etapa anterior, partimos para a classificação de cada uma das edições contidas neste conjunto. A classificação foi um processo manual, executado por um membro da equipe desta pesquisa, que detinha amplo conhecimento sobre os sistemas especificados, e revisado por um segundo membro, para checar possíveis classificações errôneas. Cada edição foi categorizada como sendo do tipo sintático ou semântico. As sintáticas são modificações que não carregam consigo mudanças no comportamento do sistema, mas sim correções ortográficas, troca de palavras por outras mais adequadas ao contexto, etc, de forma que o sentido semântico permanecia intacto. Já as enquadradas no tipo semântico são aquelas edições que diferentemente das sintáticas, alteravam o conteúdo passado, mudando o comportamento do sistema. Edições que possuíam ambos os tipos de edição eram classificados como sintático-semântico.

Etapa 4: Análise

No intuito de observar o impacto geral causado pelas modificações, calculamos a média de casos de teste obsoletos, i.e., quanto de uma suíte de teste seria descartada devido às modificações realizadas na especificação de caso de uso relacionada. Sendo tst_total o número total de casos de teste gerados a partir dos casos de uso; tst_obs o número de casos de teste classificados como obsoletos; e N o número de pares de casos de uso que foram avaliados, utilizados para extrair edições e gerar suítes para comparação seus casos de teste; definimos a Média de Casos de Teste Obsoletos (MCTO) como indicado pela Equação 3.1.

$$MCTO = (\sum \frac{tst\_obs}{tst\_total}) * \frac{1}{N} \tag{3.1}$$

Utilizando o conjunto de modificações com classificação de tipo, foi calculado o impacto individual das diferentes formas de edição. Definimos três derivações da métrica MCTO: MCTO_SIN, que avalia a média de casos de teste obsoletos devido às atualizações sintáticas; MCTO_SEM, avaliando a média de casos de teste obsoletos devido às atualizações semânticas; e MCTO_DUO, a qual indica o impacto das edições com ambos os tipos de alteração. Para essas métricas só foi alterado o parâmetro tst_obs, onde antes representava

todos os testes obsoletos, agora representa em cada uma delas os testes obsoletos em razão do tipo analisado.

Por fim, foi investigada qual a proporção dos passos dos casos de teste que havia necessidade de revisar, a fim de tornar estes testes reutilizáveis. Em outras palavras, foi visto quantos passos com relação ao número total foram atualizados. Para isso, definimos a métrica MPM (Média de Passos Modificados), visto na Equação 3.2, responsável por calcular a proporção interna dos casos de teste que sofrem impacto em decorrência das atualizações nos modelos. stp_total corresponde ao número total de passos de um dado caso de teste; stp_ob o número de *steps* que necessitam de revisão nesse caso de uso; e N o número de casos de teste sob análise.

$$MPM = (\sum \frac{stp\_ob}{stp\_total}) * \frac{1}{N} \tag{3.2}$$

3.1.3 Resultados

Nossos resultados mostraram que suítes de testes TBM são sensíveis aos diferentes tipos de edições efetuadas nos modelos do sistema. Um grande número de casos de teste foram impactados pelas mudanças e consequentemente descartados. Em média, 86% (métrica MCTO) dos testes de uma suíte tornaram-se obsoletos entre duas versões consecutivas de um mesmo caso de uso (Figura 3.2). Esse resultado refletiu uma das principais desvantagens de usar o TBM em projetos ágeis. Metodologias ágeis possuem uma maior flexibilidade quanto a mudanças de requisitos e devem se adequar a isto de forma que gere um produto de maior valor possível para o cliente [4]. E, uma vez que no contexto estudado os casos de teste são derivados dos casos de uso, qualquer tipo de edição efetuada é refletida em impacto no conjunto de testes gerado. Portanto, na prática, as equipes tendem a reutilizar apenas um pequeno número de casos de teste, dificultando o uso dessa estratégia para testes de regressão do sistema. Esse fato poderia inviabilizar o uso de TBM em cenários ágeis. No entanto, as equipes em nosso estudo ainda consideraram as suítes TBM úteis, pois ajudaram a detectar uma série de falhas e reduziram o esforço da criação de testes manualmente.

Q1: Quanto de uma suíte de testes é descartada devido à edições nos casos de uso? Em média, 86% (métrica MCTO) dos testes de uma suíte tornaram-se obsoletos entre duas versões consecutivas de um mesmo caso de uso, enquanto apenas 14% se mantiveram reusáveis.

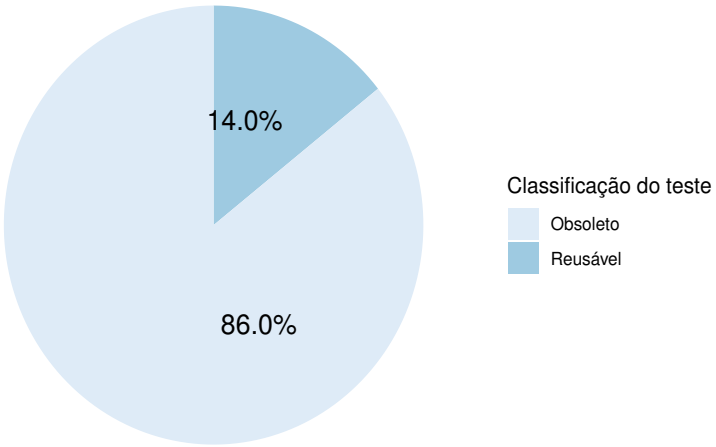


Figura 3.2: Casos de Teste Reusáveis e Obsoletos.

Com relação aos casos de teste que se tornaram obsoletos, fomos investigar a causa desse problema. Analisamos manualmente cada caso e classificamos o tipo de edição do modelo: *edição exclusivamente sintática*, *edição exclusivamente semântica* ou *edição com conteúdo sintático e semântico*. A Figura 3.3 resume esta análise. Como podemos observar, 52% das observações foram causadas por alterações sintáticas nos modelos de casos de uso, enquanto 21% foram causadas por alterações semânticas e 12% por alterações sintáticas e semânticas simultaneamente. Estes resultados mostram que mais da metade dos casos de teste obsoletos é devido apenas a edições sintáticas (*e.g.*, reformulação de um passo, correção gramatical). Tais casos de teste não carregam mudança no comportamento do sistema e poderiam ser revisados, tornado-os reutilizáveis sem muito esforço. Isso contribuiria para a diminuição do descarte de testes e permitiria a preservação do histórico desses testes.

Q2: Qual o impacto de alterações sintáticas e semânticas numa suíte de testes? 52% das observações foram causadas por alterações sintáticas nos modelos de casos de uso, enquanto 21% foram causadas por alterações semânticas e 12% por alterações sintáticas e semânticas simultaneamente.

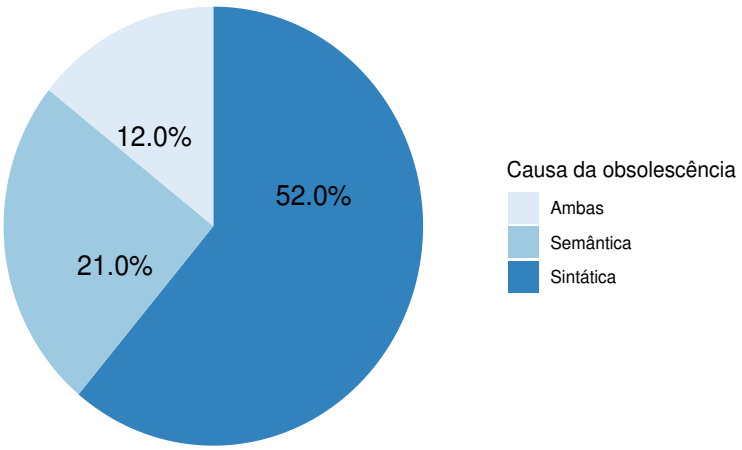


Figura 3.3: Obsolescência dos Testes Devido aos Tipos de Edição.

Sobre a questão do esforço necessário para transformar um caso de teste obsoleto em reutilizável, investigamos o quanto de um caso de teste precisaria ser revisado. É importante ressaltar que essa análise foi baseada apenas no número de etapas que requerem revisão, não medimos a complexidade da revisão. A Figura 3.4 mostra a distribuição dos resultados encontrados. Como podemos ver, metade dos casos de teste obsoletos por alterações sintáticas, denotado pela parte inferior ao corte da mediana, tiveram menos de 25% de seus passos impactados. Isso indica um baixo esforço para revisar esses testes e torná-los reutilizáveis. Tonando-os reutilizáveis, o testador poderia se concentrar nas edições semânticas do modelo, já que elas trazem mudanças na especificação e impacto nas funcionalidades.

Q3: Quanto de um caso de teste obsoleto precisa ser revisado para torná-lo reutilizável? Foi observado com a métrica MPM um valor médio de 4 *steps* alterados para todo o conjunto de casos de teste analisado. Esse valor corresponde proporcionalmente a 38% dos *steps* dos casos de teste em média.

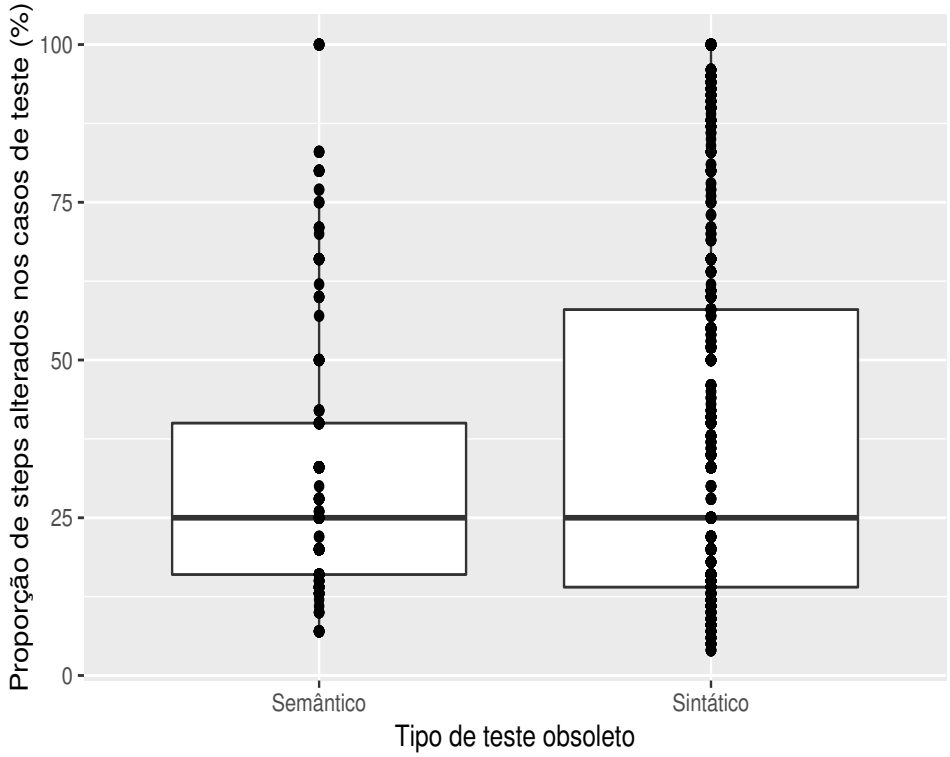


Figura 3.4: Impacto Interno nos Casos de Teste Agrupado por Tipo de Modificação

3.1.4 Discussão

Os resultados obtidos confirmaram a preocupação inicial, sobre o alto índice de descarte de testes ao longo das evoluções dos modelos. Foi detectado um valor além do esperado pelos times de desenvolvimento e pelos pesquisadores, tanto em relação a taxa dos obsoletos em geral (86%), como também no tocante dos obsoletos sintáticos (52% do total de obsoletos).

Este estudo de caso foi determinante para a continuidade das investigações e pela busca de meios que pudessem minimizar o que caracterizamos como problemático. Afinal, parte dos testes poderiam ser reutilizados sem muito esforço, poupando recursos da equipe.

Os resultados descobertos neste estudo certamente estão relacionados ao contexto avaliado. A experiência do time com a escrita de casos de uso, a forma de obtenção das evoluções minerando o repositórios e capturando conjuntos de mudanças e a granularidade dos envios ao repositório, com muitas ou poucas mudanças, podem impactar nos valores obtidos. No entanto, entendemos que a avaliação traz resultados interessantes e atende ao propósito do estudo e pode ser relevante para cenários semelhantes.

De modo geral, elencamos como sendo as principais contribuições trazidas por este es-

tudo:

- As suítes TBM analisadas detectaram um alto impacto devido às evoluções nos casos de uso;
- No contexto do nosso estudo, a maioria das edições realizadas nos casos de uso foram do tipo sintático. Isso pode estar relacionado com a frequência de atualização das informações no cenário de desenvolvimento ágil;
- É necessário avançar no conhecimento a fim de facilitar o uso de TBM. Uma vez que as suítes de teste são sensíveis a pequenas modificações realizadas nos casos de uso, que parte das vezes são voltadas a correções ortográficas ou melhoria da escrita;
- O esforço para revisar e transformar parte dos casos de teste impactados por edições sintáticas em reutilizáveis pode ser baixo. No contexto do nosso estudo, metade do conjunto de testes obsoletos por edições sintáticas, precisariam de ter apenas 25% dos seus *steps* revisados.

O estudo apresentado nesta seção foi publicado na forma de artigo no *III Brazilian Symposium on Systematic and Automated Software Testing* (SAST) - 2018 [58].

3.1.5 Ameaças à Validade

Neste ponto, apresentamos algumas limitações relacionadas ao estudo desenvolvido. Acreditamos que o fator mais impactante está relacionado com o número de casos de uso analisados e de projetos envolvidos, uma vez que utilizamos artefatos industriais, não foi possível obter um conjunto de dados maior, em ambos os sentidos.

A respeito das limitações relacionas à validade externa, entendemos que nosso estudo não pode ser generalizado para todo projeto de software que utilize métodos ágeis e TBM como forma de teste. Porém, acreditamos que nossos resultados são válidos para outros projetos inerentes a um contexto próximo do estudado, utilizando principalmente modelos do sistema descritos em linguagem natural, com uso da notação CLARET para os casos de uso. A equipe e a forma de descrever os casos de uso têm papel importante no cenário de estudo.

Quanto à validade de conclusão, nosso trabalho trata um pequeno conjunto de dados. Embora quiséssemos obter um conjunto maior, não foi possível encontrar outras fontes. Optamos por explorar um contexto próximo ao que estamos inseridos, utilizando o que se tinha disponível. Isso nos permitiu trabalhar com projeto industriais e artefatos que passaram por validação dos engenheiros das equipes e pela equipe que conduziu o estudo.

Sobre as ameaças à validade interna, temos o processo de coleta das edições realizadas nos casos de uso e a classificação das edições em sintático e semântico. Ambas as etapas foram feitas de forma manual, por não ter sido possível realizar de forma automática. Existe o risco inerente de erro nos processos manuais, todavia, esses processos foram conduzidos cuidadosamente, passando por revisão pela equipe que conduziu o estudo e contou com o suporte dos times de desenvolvimento para sanar quaisquer dúvidas a respeito das alterações presentes nos casos de uso.

3.2 Considerações Finais

Este capítulo apresentou um estudo de caso, o qual teve como objetivo investigar o impacto nas suítes de teste, causado pelas edições inseridas durantes as evoluções dos casos de uso. Essa verificação foi importante para validação da obsolescência dos testes em detrimento das evoluções dos modelos. Foi destacada uma diferenciação das edições em sintáticas ou semânticas, quando envolvem ou não alterações no comportamento do sistema, respectivamente. E então, visto que uma parte considerável dos testes são impactados em decorrência das alterações sintáticas, identificamos que estes poderiam ser reutilizados com um baixo esforço.

Capítulo 4

Reduzindo o Descarte de Casos de Teste

Apresentaremos neste capítulo detalhes acerca da construção da estratégia utilizada para resolução do problema do descarte de testes passíveis de reúso. Tal estratégia avalia duas suítes, sendo elas: a gerada a partir da versão base do caso de uso CLARET e outra gerada após modificações no mesmo caso de uso CLARET. O objetivo principal é identificar quais casos de teste potencialmente possuem apenas modificações sintáticas, aquelas que não alteram o comportamento do sistema; quais possuem modificações mais relevantes, que alteram o fluxo ou a semântica da aplicação; os que não passaram por modificações e são reutilizáveis; e os novos testes, presentes apenas na suíte gerada após as modificações.

Na Seção 4.1, apresentaremos a estrutura lógica e funcionamento do algoritmo implementado para classificação dos testes, na Seção 4.2 definimos a arquitetura da ferramenta que implementa os conceitos da estratégia e a forma de utilização, e na Seção 4.3, mostraremos como foi a condução do estudo que embasou a escolha da função distância utilizada.

4.1 Estrutura e Funcionamento da Estratégia

A estratégia definida neste trabalho utiliza a função de distância Levenshtein [40], detalhada na Seção 2.3, e um limiar igual a 15 para esta função, como forma de classificar testes em reutilizáveis, novos, obsoletos sintáticos e obsoletos semânticos. O limiar atua como divisor na classificação dos testes obsoletos, definindo como sendo obsoletos sintáticos, os testes que, quando comparados, obtiveram dissimilaridade máxima igual a 15, e determinando como obsoletos semânticos, testes que obtiveram valor maior que 15 nas comparações uti-

lizando a métrica de Levenshtein. As razões que definiram a escolha da função e do limiar, estão apresentadas no experimento da Seção 4.3, onde foram comparadas seis funções de distância e definido o melhor limiar para cada uma delas, em consequência de apresentarem a melhor *acurácia* para classificação de edições. O algoritmo de classificação desenvolvido, mostrado no Algoritmo 1, confere o resultado retornado pela função Levenshtein, durante a comparação de dois testes, com o limiar definido, para fazer a distinção dos casos de teste. O algoritmo recebe como parâmetros de entrada duas suítes de testes descritas em linguagem natural e, no seu fluxo de execução, cada teste da suíte base é comparado com todos os testes da suíte modificada. A comparação entre um par de casos de testes é feita observando a quantidade de passos que eles possuem e o resultado retornado pela função para cada par de passos.

No processo de comparação, é observado o passo a passo dos testes e realizada a classificação seguindo os critérios:

1. *reutilizáveis*: testes da suíte base e da suíte modificada que coincidem a quantidade de passos e, além disso, comparando passo a passo os testes, todas as comparações devem retornar resultado igual a zero para a métrica Levenshtein, indicando a igualdade entre os testes. Conforme ilustrado na Figura 4.1a;

2. *obsoletos sintáticos*: testes da suíte base e da suíte modificada que coincidem a quantidade de passos e, além disso, comparando passo a passo os testes, nenhuma das comparações deve retornar resultado maior que o limiar (15) para a métrica Levenshtein, indicando uma diferença mínima entre os casos de teste. Seguindo o exemplo ilustrativo, Figura 4.1b;

3. *obsoletos semânticos*: testes da suíte base que não se enquadraram nos critérios dos reutilizáveis ou obsoletos sintáticos. Nesse caso, a estrutura geral pode ter sido alterada (aumento ou diminuição de passos) ou os passos foram modificados, de forma que a métrica Levenshtein tenha retornado valor superior ao limiar adotado (15). O alto valor retornado pela função distância indica baixa similaridade entre os testes. Exemplificado na Figura 4.1c;

4. *novos testes*: testes da suíte modificada que não casaram com nenhum teste da suíte base, tanto na detecção dos reutilizáveis como na dos obsoletos sintáticos.

Result: suítes dos testes reutilizáveis, obsoletos sintáticos, obsoletos semânticos e novos

Data: suite_1, suite_2

inicializa suítes reutilizáveis, obsoletos sintáticos, obsoletos semânticos e novos testes;

for *teste = t1 em suite_1* **do**

  inicializa flag teste_adicionado igual a False;

  for *teste = t2 em suite_2* **do**

    if *if testes são iguais* **then** `/* comparativo Levenshtein = 0 para todos os passos */`

      adiciona t1 na suíte dos reutilizáveis;

      remove t2 da suíte_2;

      altera flag teste_adicionado para True

  end

  if *flag teste_adicionado é falsa* **then**

    for *teste = t2 em suite_2* **do**

      if *if testes são similares* **then** `/* comparativo Levenshtein <= limiar para todos os passos */`

        adiciona t1 na suíte dos obsoletos sintáticos;

        remove t2 da suite_2;

        altera flag teste_adicionado para True

    end

  if *flag teste_adicionado é falsa* **then**

    adiciona t1 na suíte dos obsoletos semânticos

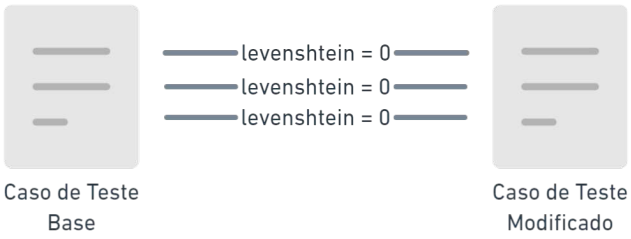
end

a suíte dos novos testes é igual a suíte_2 restante;

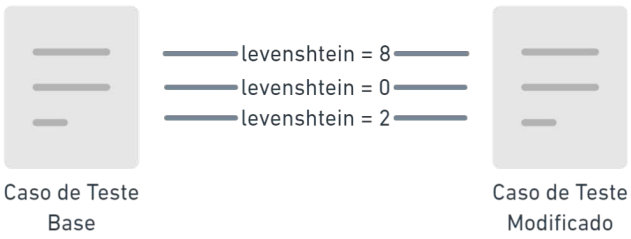
Algorithm 1: Algoritmo de Classificação

Figura 4.1: Representação da Classificação

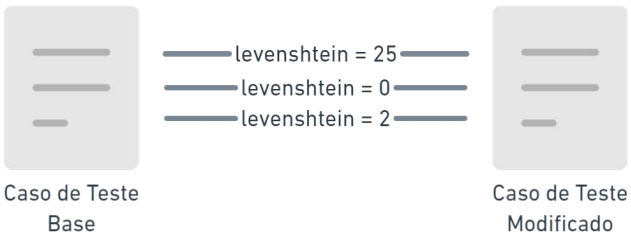
(a) Reutilizável



(b) Obsoleto Sintático



(c) Obsoleto Semântico



4.2 A Ferramenta

Para viabilizar o uso prático da estratégia definida acima, a ferramenta *Test Similarity Tool* foi desenvolvida. Por não necessitar de interações durante sua execução ou extensas configurações para seu uso, a ferramenta está disponível apenas para o uso via linha de comando através do terminal do computador. Foi escrita utilizando a linguagem Java [1] o que permite seu uso em diferentes sistemas operacionais. Nesta seção, encontraremos detalhes acerca da arquitetura definida (Seção 4.2.1), e a forma de utilização (Seção 4.2.2).

[1] https://www.java.com

4.2.1 Arquitetura

O arcabouço arquitetural da estratégia apresentada neste Capítulo 4, compreende 3 módulos: gerador de casos de teste, a partir dos arquivos de entrada; classificador de casos de teste; e o construtor das suítes de teste, utilizando as informações de classificação. A Figura 4.2 ilustra a organização dos componentes na arquitetura, exibe os artefatos de entrada, arquivos tgf gerados a partir do caso de uso sob análise, e as suítes de teste resultado do processo.

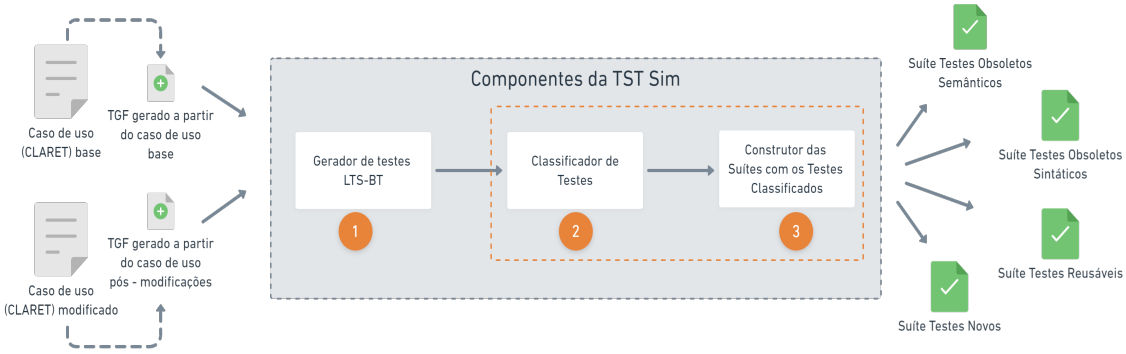


Figura 4.2: Arquitetura da Estratégia Apresentada.

Sobre os artefatos de entrada necessários para o processo de análise, o par de tgf, o primeiro deve ser respectivo à versão base de comparação de um caso de uso e, o outro, respectivo ao mesmo caso de uso após passar por modificações ou refatoração da escrita. Tal dado de entrada é utilizado no módulo de geração de testes.

O módulo responsável pela geração de casos de teste, utiliza a ferramenta LTS-BT. A LTS-BT avalia os possíveis cenários de uso da aplicação, visto no tgf como os diferentes caminhos possíveis de serem feitos partindo do nó raiz, até os nós folha. Nestes caminhos, temos as interações entre o ator, usuário do sistema, e o próprio sistema, com as respostas programadas para as ações tomadas pelo ator.

Já os dois outros módulos da sequência, numerados com 2 e 3, foram desenvolvidos durante o caminhar deste trabalho de dissertação e são de fato nossa contribuição. O módulo 2 é o classificador de testes e segue o fluxo definido no Algoritmo 1. Compara os testes gerados para o caso base com os testes gerados para o caso de uso pós-modificações, seguindo os critérios definidos para identificar os testes obsoletos semânticos, obsoletos sintáticos, reutilizáveis, todos esses referentes ao caso de uso base, e os novos testes, encontrados no conjunto de testes respectivo ao caso de uso atualizado.

O módulo 3 é o construtor das suítes, os testes classificados são organizados em conjuntos distintos de teste, formando as diferentes suítes. As suítes são armazenadas em um destino definido pelo usuário, e segue um formato compatível com a ferramenta TestLink, para futura importação e gerenciamento da execução dos testes.

4.2.2 Utilização

Por não ser um processo complexo, que necessita de muitas configurações por parte do usuário que pretende executar a estratégia elaborada neste trabalho, optamos por desenvolver, a priori, um executável Java [2] que permite a execução completa do processo via linha de comando. Como requisito para utilização do executável Java, basta possuir o Java instalado no computador.

```
1  execute_tstsim <tgf_base> <tgf_pos_modificacoes> <diretorio_para_resultados>
```

Listagem 4.1: Utilização Via Linha de Comando

A Listagem 4.1 apresenta o modelo de execução da ferramenta via linha de comando, passando como parâmetros de execução o tgf base da comparação, o tgf que traz as modificações realizadas no caso de uso e o caminho para o diretório onde deverão ser armazenadas as suítes geradas.

[2] https://www.java.com/

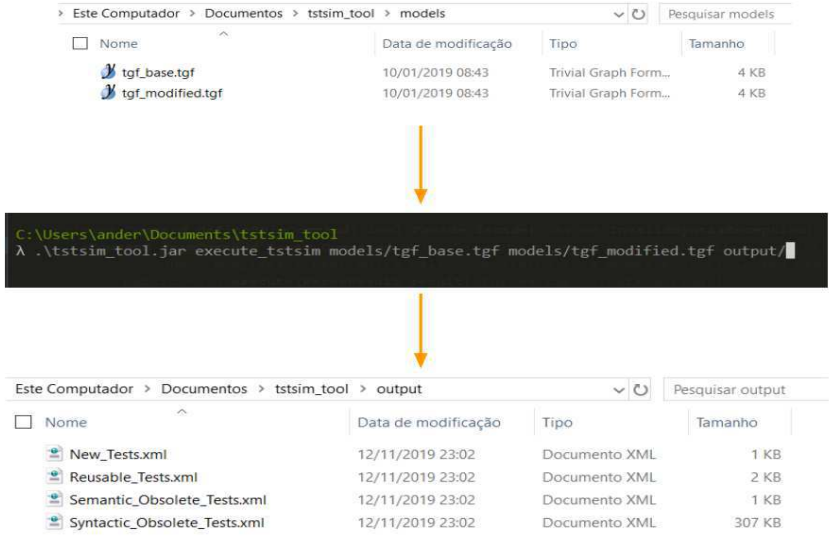


Figura 4.3: Uso Prático da Ferramenta

Na Figura 4.3 é mostrado na prática o meio de uso da ferramenta. São exibidos primeiramente os arquivos do tipo TGF, que são os tgf's anotados gerados automaticamente pela ferramenta CLARET. Os tgf's, base e modificado, são passados para a ferramenta informando o caminho até cada um deles, da mesma forma que se fornece o diretório onde deseja que os resultados sejam salvos. Após executar o comando, serão geradas as suítes de teste, no formato xml, que podem ser importadas na ferramenta Testlink para o gerenciamento dos testes. A ferramenta já se encontra integrada à LTS-BT, realizando internamente o processo de geração de testes, e disponível para *download*[3].

4.3 Estudo Experimental

O experimento trazido nesta seção é do tipo empírico e teve como principal objetivo avaliar diferentes funções de distância, no intuito de avaliar a performance e viabilidade de cada uma quanto ao quesito de classificação de edições em strings.

As próximas seções detalham o experimento, com a metodologia aplicada (Seção 4.3.1), resultados (Seção 4.3.2) e discussão (Seção 4.3.4).

[3] *https://sites.google.com/view/test-similarity/home*

4.3.1 Metodologia

Objetivo

Verificar a viabilidade e capacidade das funções distância aplicadas à classificação de edições realizadas em casos de uso. Além disso, comparar o desempenho de classificação de diferentes funções.

Questões de Pesquisa e Hipóteses

Com o objetivo de avaliar a performance de classificação das funções, definimos nossas questões de pesquisa e levantamos como hipóteses nulas e alternativas:

Q1: É possível classificar edições, inseridas em casos de uso, utilizando funções de distância?

Q2: Qual função apresenta o melhor resultado para classificação de edições?

H_{1-0} : as funções possuem a mesma capacidade de classificação

H_{1-1} : as funções possuem capacidade diferente de classificação

Para respondermos as questões de pesquisa, utilizamos um conjunto de edições extraído de casos de uso de projetos industriais. Cada item do conjunto de edições continha: a *string* base; a *string* modificada; e, a classificação manual dessa mudança, podendo ser obsoleto sintático ou obsoleto semântico. A classificação manual foi utilizada como forma de validação da classificação automática feita pelas funções.

Variáveis Independentes e Dependentes

As variáveis independentes do experimento foram as funções escolhidas para a análise: Levenshtein [40], Jaro-Winkler [15], Cosine [31], Jaccard [41], NGram [37], Sorensen Dice [59]. Utilizamos uma implementação open-source de todas as seis funções[4].

A classificação fornecida pelas funções, sendo obsoleto sintático ou semântico, foi a variável dependente.

Métricas

Para avaliação dos resultados e obtenção de respostas para nossas questões de pesquisa, uti-

[4]https://github.com/tdebatty/java-string-similarity

lizamos três métricas bem difundidas na literatura: *Precisão*, que representa a fração de instâncias recuperadas e relevantes; *Recall*, a fração de instâncias relevantes que são recuperadas dentre todas as instâncias relevantes; e *Acurácia*, que combina a *Precisão* e o *Recall*. Outros estudos empíricos já adotaram essas métricas (e.g., [20, 26, 46]).

As Equações 1, 2 e 3 apresentam as métricas utilizadas, sendo VP o número de classificações feitas pela função como obsoleto sintático que coincidiram com a classificação manual; VN a quantidade de classificações automáticas em obsoleto semântico que foram confirmadas pela classificação manual; FP o número de vezes que a função classificou como obsoleto sintático enquanto que a classificação manual indicava como resultado obsoleto semântico; e FN, ocorrências onde a classificação automática resultou em obsoleto semântico e, a classificação manual, obsoleto sintático.

$$\text{Precisão} = \frac{VP}{VP + FP}$$

$$\text{Recall} = \frac{VP}{VP + FN}$$

$$\text{Acurácia} = \frac{VP + VN}{VP + VN + FP + FN}$$

Execução

Para respondermos nossas questões de pesquisa, inicialmente foi necessário definir limiares para as funções. O limiar é o ponto de corte que indica até qual valor as edições podem ser consideradas sintáticas e a partir de onde passam a ser consideradas semânticas, conforme Figura 4.4. Todas as funções dentre as escolhidas, com exceção de Levenshtein, possuem o resultado normalizado entre zero e um. Por Levenshtein fornecer o número mínimo de operações necessárias para transformar uma *string a* em uma *b*, seu valor pode variar de 0 ao tamanho da maior *string*. A definição dos limiares para as funções foi baseada na melhor acurácia, indicando o ponto de equilíbrio entre a precisão e o *recall* na classificação da nossa base de dados. Os limiares podem ser encontrados na Seção 4.3.2.

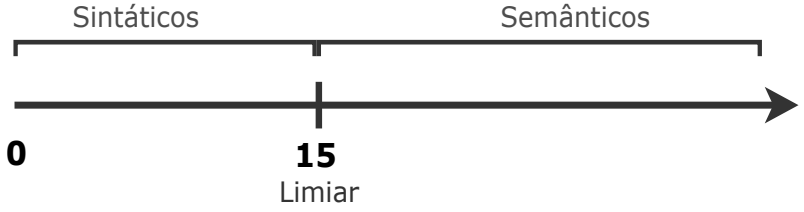


Figura 4.4: Classificação com Base no Limiar

Figura 4.5: Fluxo de Execução do Experimento

O conjunto de dados dessa investigação foi extraído dos projetos de software utilizados no estudo exploratório anterior (SAFF e BZC), descrito na Seção 3. Foram capturados manualmente 507 passos editados, de 28 casos de uso que passaram por evoluções ao longo do desenvolvimento (ao todo, 79 versões de casos de uso), como mostrado na Tabela 4.1. A Figura 4.5 exibe uma visão geral de como foi realizado o experimento, suas três etapas principais (classificação manual, classificação automática e validação da classificação automática) e os processos de cada etapa.

De posse das edições coletadas, classificamos manualmente cada par de edição em obsoleto sintático ou obsoleto semântico para ser o oráculo de validação das classificações feitas

Tabela 4.1: Resumo dos Dados Coletados

	Casos de Uso	Versões	Passos Editados
SAFF	13	42	404
BZC	15	37	103
Total	28	79	507

Tabela 4.2: Exemplo de Classificação

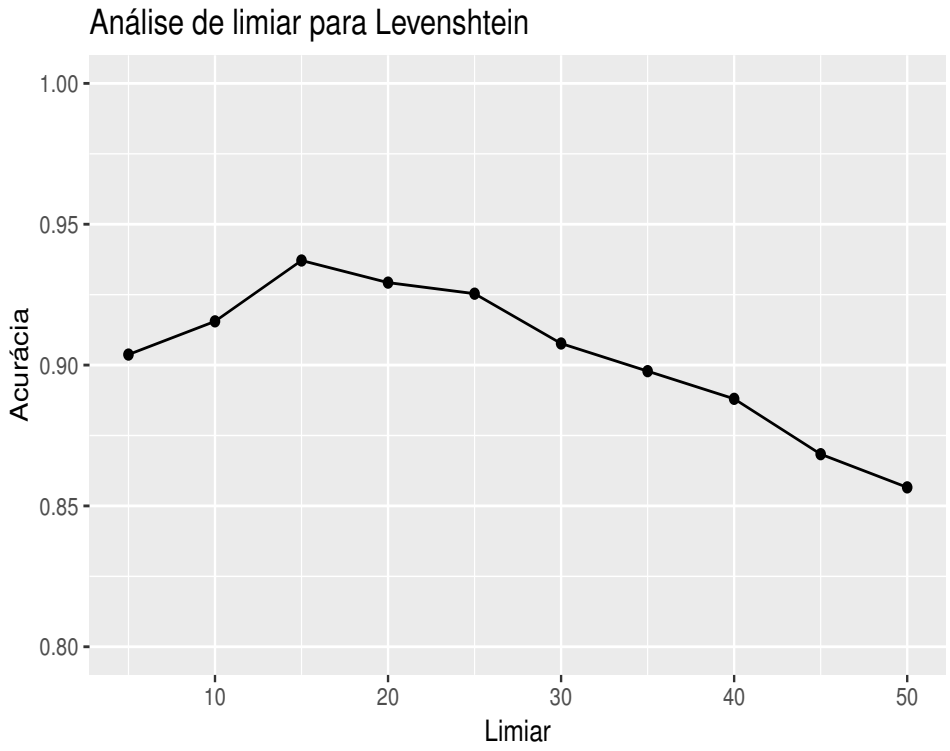
Atualização de Passo		
Versão 1	Versão 2	Classificação
clica no botao voltar	clica no botão 'voltar'	obsoleto sintático
exibe dados da secao	exibe os dados da seção	obsoleto sintático
entra em modo X	exibe pagina que requisita dados do terminal	obsoleto semântico

pelas funções distância. Nossos dados seguiram a estrutura mostrada na Tabela 4.2.

4.3.2 Resultados

No tocante à melhor configuração dos limiares, as Figuras 4.6 e 4.7 sumarizam os valores encontrados para os diferentes limiares testados em cada função. A mesma base de dados foi aplicada para todas as funções, as quais diferem entre si na forma de calcular a distância/similaridade das *strings*.

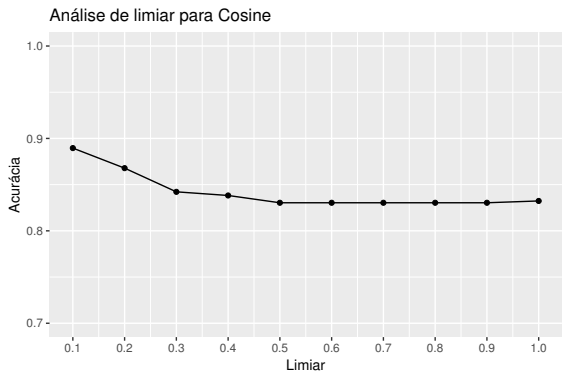
Figura 4.6: Análise do Limiar Levenshtein



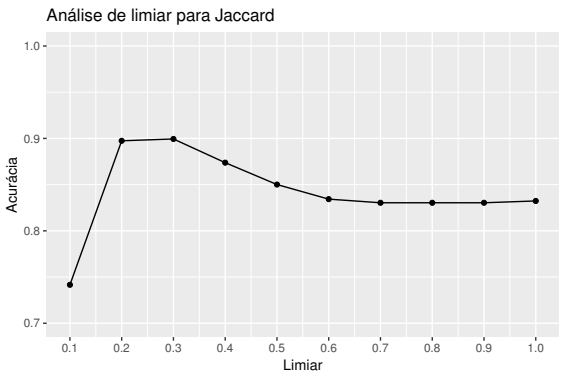
A Tabela 4.3 apresenta a melhor configuração para cada uma das funções e também os valores encontrados para as métricas precisão, *recall* e acurácia. Analisando a mesma tabela, podemos inferir que o uso de funções de distância é capaz de classificar as edições inseridas nos casos de uso, uma vez que os índices de acurácia foram próximos à 90%. Esse resultado responde nossa primeira questão de pesquisa: *"É possível classificar edições, inseridas em casos de uso, utilizando funções de distância?"*.

Figura 4.7: Análise do Limiar

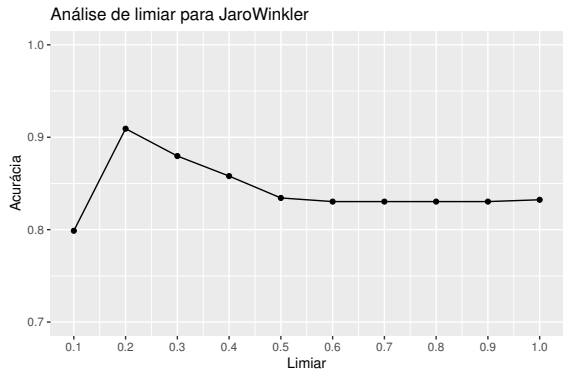
(a) Cosine



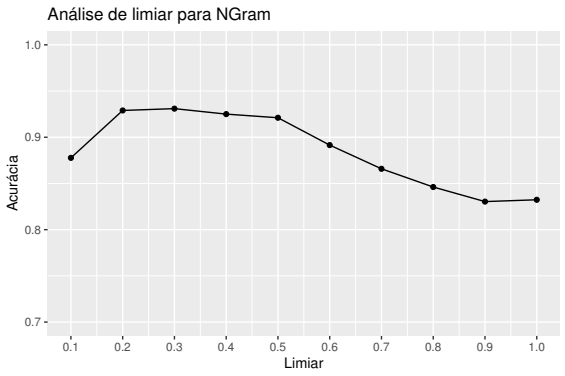
(b) Jaccard



(c) Jaro-Winkler



(d) NGram



(e) Sorensen-Dice

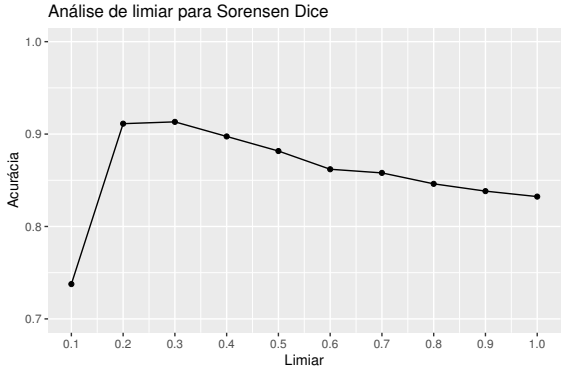


Tabela 4.3: Desempenho dos Limiares Escolhidos

Função	Limiar	Precisão	Recall	Acurácia
Levenshtein	15	94.7%	97.5%	93.7%
Jaro-Winkler	0.2	98.3%	91.4%	90.9%
Cosine	0.1	98.1%	89.6%	88.9%
NGram	0.3	94.3%	97.3%	93.9%
Jaccard	0.3	98.3%	90.4%	89.9%
Sorensen-Dice	0.3	97.1%	92.7%	91.3%

Para respondermos a Q2, testamos a hipótese nula H_{1-0} a fim de identificar se as funções possuem a mesma capacidade de classificação. Para isso, executamos o teste de proporção, *test.prop* do R[5], considerando todas as funções avaliadas. Nossos resultados mostraram, com 95% de confiança, que não há diferença estatística em relação ao poder de classificação das funções, com *p-value* maior que 0.05 (0.179).

4.3.3 Discussão

A estratégia de redução de descarte apresentada neste capítulo compara pares de versões subsequentes de suítes de teste, geradas a partir de um mesmo caso de uso que sofreu atualizações ao longo do tempo. O ponto-chave do processo está na classificação das edições em sintáticas ou semânticas usando como divisor o limiar definido para o resultado da métrica Levenshtein.

No experimento realizado, verificamos que um bom divisor para classificação das edições, utilizando a métrica Levenshtein, é o valor 15. Nesse ponto, encontramos a maior taxa de acerto de classificação dos passos extraídos dos casos de uso. Acreditamos que a utilização desse limiar pode ser estendida para outros ambientes de desenvolvimento que utilizem a ferramenta CLARET para especificação de seus casos de uso, e geração dos testes de alto nível. Nosso objetivo no experimento de utilizar dados de projetos de software diferentes, com equipes de desenvolvimento diferentes, foi elaborado para que nossos resultados não se tornassem específicos a um ambiente, contribuindo para a utilização da estratégia em outros

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

contextos. Ainda sobre essa temática de utilização do limiar e da estratégia em outros cenários, apresentaremos um estudo de validação da estratégia, no Capítulo 5, o qual utilizou um terceiro objeto de estudo para validar o que foi proposto neste capítulo.

Perante o exposto, concluímos que o uso de funções distância é sim capaz de classificar edições inseridas em casos de uso CLARET descritos utilizando linguagem natural. Uma vez que nenhuma função se sobressaiu com relação as outras, a escolha de uma delas pode ser feita baseada em outros aspectos, e.g. disponibilidade do algoritmo, facilidade de implementação, velocidade de execução). No nosso caso, foi dada preferência à Levenshtein, ao longo da dissertação, pela facilidade de se entender seu funcionamento e o que o valor retornado pela função implica.

A tendência das edições sintáticas terem valores mais baixos, do contrário das semânticas, pode ser explicada pelo fato de que correções ortográficas ou melhorias textuais inseridas nos passos são feitas com menos edições. Já quando se altera a funcionalidade, ocorre uma maior troca, inserção ou deleção de palavras.

4.3.4 Ameaças à Validade

No contexto de ameaças, identificamos como ameaças à validade interna: a classificação manual realizada no conjunto de dados para validação das funções; e a extração das edições das evoluções dos casos de uso. A extração foi feita com o suporte da ferramenta para comparação visual de mudanças entre dois arquivos *DiffMerge*[6]. Para mitigar tais riscos, a equipe de pesquisa contou com o apoio das equipes de desenvolvimento que escreveram os casos de uso para sanar quaisquer dúvidas relacionadas ao contexto das mudanças.

Quanto à validade de conclusão, podemos destacar o nosso conjunto de dados utilizado para o experimento. Um conjunto de maior tamanho daria um maior poder estatístico à análise. No entanto, como tratamos de dados reais, não foi possível obter um número maior de casos de uso e edições. Para amenizar o impacto deste ponto, realizamos o experimento utilizando dados de dois projetos de software distintos, evitando viés e vícios existentes em uma mesma equipe de desenvolvimento.

Já em termos de ameaças à validade externa, a generalização dos resultados pode sofrer em ação do contexto de projetos analisados. Não foram utilizados projetos externos à UFCG

[6]https://sourcegear.com/diffmerge/

devido a dificuldade de encontrar projetos com os requisitos necessários.

4.4 Considerações Finais

Apresentamos neste capítulo a estruturação da estratégia implementada neste trabalho. Foi explicado como é feita a classificação dos casos de teste utilizando funções distância e toda a avaliação executada para embasar a escolha da função Levenshtein e sua respectiva configuração utilizada. Na avaliação, foram comparadas diferentes funções no intuito de mensurar a efetividade de classificação de cada uma. Chegamos às melhores configurações de limiares, individuais de cada função, e contatamos a semelhança dos resultados retornados pelas funções por meio de teste estatístico.

Capítulo 5

Validação da Estratégia

Uma vez implementada a estratégia (Capítulo 4), havia a necessidade de avaliar sua capacidade de classificação dos testes. A Seção 5.1 apresenta os detalhes da avaliação das possíveis contribuições deste trabalho.

5.1 Estudo de Caso para Validação

Como forma de avaliar a estratégia proposta neste trabalho, realizamos um estudo de caso para verificação de sua aplicação em um projeto de software diferente dos utilizados nas investigações anteriores. A metodologia (Seção 5.1.1), resultados (Seção 5.1.2) e discussão (Seção 5.1.3) da investigação estão detalhados a seguir.

5.1.1 Metodologia

Objetivo

Mensurar a efetividade da estratégia de classificação de casos de teste em um cenário diferente dos já utilizados. Nesta investigação foi utilizado o projeto TCoM, uma aplicação mobile que possibilita testar periféricos dos terminais de pagamento Ingenico. O desenvolvimento foi conduzido por uma equipe do SPLab e contou com integrantes diferentes com relação aos projetos utilizados nos estudos anteriores (SAFF e BZC).

Questões de Pesquisa

Com o objetivo de averiguar a efetividade de classificação da estratégia definida, definimos como nossa questão de pesquisa central:

Q1: A estratégia implementada é efetiva na classificação de casos de teste?

Métricas

Na avaliação dos resultados, utilizamos as três métricas já utilizadas no estudo anterior (Seção 4.3) para manter a equidade das avaliações. As métricas são: *Precisão*, que representa a fração de instâncias recuperadas e relevantes; *Recall*, a fração de instâncias relevantes que são recuperadas dentre todas as instâncias relevantes; e *Acurácia*, que combina a Precisão e o *Recall*.

Com relação às métricas utilizadas, VP indica número de classificações feitas pela função como obsoleto sintático que coincidiram com a classificação manual; VN a quantidade de classificações automáticas em obsoleto semântico que foram confirmadas pela classificação manual; FP o número de vezes que a função classificou como obsoleto sintático enquanto que a classificação manual indicava como resultado obsoleto semântico; e FN, as ocorrências onde a classificação automática resultou em obsoleto semântico e, a classificação manual, obsoleto sintático.

$$\text{Precisão} = \frac{VP}{VP + FP}$$

$$\text{Recall} = \frac{VP}{VP + FN}$$

$$\text{Acurácia} = \frac{VP + VN}{VP + VN + FP + FN}$$

Execução

O conjunto de dados utilizado na execução do estudo contou com seis casos de uso, envolvendo ao todo 18 evoluções. Foram capturadas as edições inseridas ao longo das evoluções, com o apoio da ferramenta de *diff* DiffMerge[1], e classificadas manualmente entre sintáticas ou semânticas para verificação da presença dessas edições nos casos de teste já

[1] https://sourcegear.com/diffmerge/

classificados. 620 testes foram classificados de forma automática em sintáticos e semânticos. A Tabela 5.1 apresenta a quantidade dos dados coletados.

Tabela 5.1: Dados Coletados

	Casos de Uso	Versões	Total de Testes
TCoM	6	18	620

5.1.2 Resultados

A Tabela 5.2 exibe a quantidade de testes classificados corretamente em sintáticos (verdadeiro-positivo) e semânticos (verdadeiro-negativo), em conjunto com os classificados de forma errônea em sintáticos (falso-positivo) e em semânticos (falso-negativo). Quanto maior o número de verdadeiro-positivos e de verdadeiro-negativos, mostrado na Tabela 5.2, melhor se torna a acurácia. De posse desses dados, obtivemos que nossa classificação teve uma efetividade de 81.4% (acurácia). Quanto à nossa questão de pesquisa, *a estratégia implementada é efetiva na classificação de casos de teste?*, concluímos que sim, é efetiva, uma vez que se obteve uma boa quantidade de acertos e mais de setenta testes poderiam ser reutilizados.

Tabela 5.2: Análise dos Testes Classificados

Classificação	Quantidade
Verdadeiro-Positivo	77
Verdadeiro-Negativo	428
Falso-Positivo	25
Falso-Negativo	90

5.1.3 Discussão

A acurácia menor (81.4%) com relação a obtida ao estudo anterior, apresentado na Seção 4.3 (93.7%), pode ser entendida pelo fato de que neste estudo atual foram classificados testes como um todo, enquanto que no anterior apenas edições de forma isolada.

O valor elevado dos testes semânticos se dá porque os testes podem conter edições de apenas um tipo como também de ambos os tipos, e nesse caso ser classificado como semânticos. Esse número deve variar de acordo com a quantidade e o tipo das edições inseridas nos casos de uso durante as evoluções.

As evoluções dos casos de uso consideradas nesse estudo são respectivas aos conjuntos de mudança enviados para o sistema de controle de versão (*commits*), o que se torna também um fator de variação nos resultados, dependente da cultura da equipe de desenvolvimento. O time de desenvolvimento pode enviar *commits* com conjuntos menores ou maiores de mudanças com mais ou menos edições sintáticas e semânticas.

De modo geral, consideramos a estratégia com uma boa acurácia quando aplicada em outro projeto de desenvolvimento. Outros fatores, como os citados acima, podem ser levados em consideração para o refino da estratégia em estudos futuros.

Um artigo desenvolvido a partir deste estudo e pela investigação mostrada na Seção 4.3, foi aceito para publicação no *XXXIII Brazilian Symposium on Software Engineering* (SBES) - 2019 [17].

5.2 Considerações Finais

Este capítulo trouxe um estudo de caso responsável por validar a estratégia apresentada na Seção 4. Dado o problema elencado no primeiro estudo de caso 3.1, foi implementada a estratégia e validamos a mesma utilizando um novo projeto de software, com especificações, evoluções e edições diferentes das já utilizadas. Observamos a efetividade da classificação automática dos testes, utilizando função distância, e o poder de detecção dos casos de teste obsoletos por razões meramente sintáticas.

Capítulo 6

Trabalhos Relacionados

No desenvolvimento de sistemas, requisitos mudam constantemente, afetando outros documentos (e.g., casos de uso) e implicando na necessidade de modificar o software e os testes associados a este [29]. Se os testes não estão em conformidade com os novos requisitos, defeitos podem ser reportados erroneamente para as novas funcionalidades do sistema. Este é um dos fatores que denotam a importância da detecção dos testes que sofrem impacto devido a alterações nos artefatos do sistema.

Hotomski e Glinz [30] apresentam a ferramenta *GuideGen* capaz de associar mudanças realizadas em requisitos, aos testes de aceitação relacionados a estes. No seu método, um teste é associado a um requisito no momento de sua criação, no entanto, cada requisito só pode ser relacionado a um único teste de aceitação. A *GuideGen* analisa sentenças modificadas no requisito a nível semântico, comparando classe gramatical e dependência entre as palavras que existiam antes e após a modificação. Dessa forma, torna possível a sugestão de formas de adaptação dos casos de teste, que normalmente contém partes do requisito em seu corpo, para que este reflita o novo comportamento. A *GuideGen* e nosso trabalho, se relacionam no quesito de identificar nos testes mudanças efetuadas nos artefatos, que podem ser requisitos cadastrados na ferramenta ou, no nosso contexto, casos de uso. Todavia, observamos todo um conjunto de testes funcionais ao invés de um único teste associado diretamente ao artefato.

A ferramenta TaRGeT, proposta por Ferreira et al. [22], é apresentada como meio de minimizar o gargalo existente entre casos de uso e testes. A TaRGeT, provê um meio de especificação de casos de uso utilizando linguagem natural controlada (LNC) com idioma

inglês, que propicia menos ambiguidade comparando-a a linguagem natural. Dos casos de uso, são extraídas informações acerca das ações do usuário, estados e respostas do sistema, e posteriormente utilizadas na geração dos testes de execução manual. A interseção com nosso trabalho está no seu módulo de gestão de consistência entre suítes. Esse módulo é responsável por, ao modificar um caso de uso e gerar uma nova suíte, comparar a nova suíte com a suíte criada anteriormente para este mesmo caso de uso. A etapa de comparação, utiliza o algoritmo de distância Levenshtein e automaticamente associa testes com similaridade superior a 95% (configurável), retornando uma lista com os novos e antigos casos de teste, e respectivos graus de similaridade. Por outro lado, a estratégia elaborada nesta dissertação, utiliza um contexto independente de idioma e fornece uma classificação dos testes, por meio da comparação deles, em níveis diferentes: reutilizáveis; obsoletos sintáticos; obsoletos semânticos; e novos testes.

Já o conjunto ferramental Conformiq [14], permite a criação de testes automáticos para execução com Selenium[1]. Utiliza artefatos com alto nível de abstração, e.g. testes estruturados em excel, para construção de diagramas de atividades UML que indicam os fluxos a serem executados durante os testes. Para transformar os testes em automáticos, o testador deve montar um *script* de teste fornecendo o conjunto de elementos da página web a ser testada, em associação com as ações a serem executadas durante o teste. Em nossa análise, não observamos estratégias voltadas para detecção de testes passíveis de reúso, além do nosso contexto utilizar casos em uso, diferentemente da ferramenta em questão.

Sobre a evolução de casos de teste, Mirzaaghaei et al. [43, 44], detectaram padrões inerentes à correção de testes durante sua manutenção, para aplicá-los a uma técnica automática de reparo. Alguns desses padrões são: modificação na declaração de métodos, extensão hierárquica de classes; implementação de interfaces; e introdução de sobrescrita de métodos. Já Pinto et al. [51,52], dadas duas versões de um programa Java e suas respectivas suítes, buscaram identificar quais mudanças foram executadas na evolução dos testes. Em seus resultados, identificaram que vários testes são removidos da suíte devido a se tornarem obsoletos. Esses trabalhos citados, tratam da evolução de casos de teste a nível de código. Embora haja, de maneira similar, a necessidade de detectar as mudanças no software que devem ser refletidas nos testes, o nível de abstração desses trabalhos foge do escopo desta dissertação.

[1] https://www.seleniumhq.org/

No tocante as funções de similaridade/distância, Gomaa e Fahmy [24] fazem um apanhado geral sobre uma diversidade de funções. Agrupa os tipos de funções em: i) *string-based* - as que têm sua métrica baseada na sequência e na composição das palavras; *corpus-based* - se baseiam na semântica das palavras utilizando informações de um Corpus linguístico; e *knowledge-based* - também se baseiam na semântica, porém as informações utilizadas são obtidas a partir de redes semânticas, sendo a *WordNet* [42] a mais popular. Vijaymeena e Kavitha [66], trazem um levantamento similar ao trabalho de Gomaa e Fahmy [24], elencando as diferentes abordagens para o cálculo da similaridade. Esses trabalhos nos ajudaram a obter uma maior ciência com relação ao funcionamento e uso das funções de similaridade.

Oliveira et al. [49], propõem uma abordagem (denominada SART) para seleção de testes que exercitam partes modificadas de um sistema, baseando-se na similaridade entre casos de teste gerados em um contexto TBM. Os testes avaliados pela SART são funcionais, estruturados em um modelo LTS anotado [33] com passos, formados por ações do usuário e respostas dadas pelo sistema, descritos em linguagem natural. Na abordagem, é criada uma matriz de similaridade entre os testes de uma suíte (*s*) e os testes desta mesma suíte após passar por modificações (*s'*). Os testes selecionados são todos aqueles que não obtiveram similaridade máxima com algum outro teste, ou seja, testes em *s* que não coincidiram com testes em *s'* (tornaram-se obsoletos) e o oposto, testes em *s'* que não coincidiram com testes em *s* (novos testes). Correlacionado com nosso trabalho, casos de teste são comparados com o auxílio de funções de distância. No entanto, na SART, a comparação entre casos de teste não considera pequenas modificações sintáticas, como foi levantado pelo nosso estudo. Dessa maneira, casos de teste deixam de ser possivelmente reutilizados e são selecionados para serem novamente executados.

Cartaxo et al. [11], denotam uma função de similaridade onde é avaliado um modelo LTS anotado, o qual representa o comportamento de uma determinada aplicação. Comparando dois testes gerados a partir do LTS, a similaridade é calculada com base no número de transições idênticas entre esses testes, em detrimento do número médio de transições entre os dois testes. Além disso, o grupo de Cartaxo et al. [12], apresenta uma maneira sistemática de geração de testes a partir da tradução de diagramas de sequência para modelos LTS. Os estudos conduzidos nesta dissertação nos alertaram sobre o impacto de mudanças (sintáticas ou semânticas) nos casos de uso. Adicionalmente, o LTS pode ser utilizado como modelo

intermediário na geração de testes, como exemplo, casos de uso e estórias de usuário, no processo de geração de testes, se tornando suscetível a incluir modificações sintáticas ou semânticas. Neste caso, a forma de tratativa dos tipos de modificações apresentada nesta dissertação pode aprimorar a aplicação das funções de similaridade.

Ledru et al. [38], aplicaram funções de similaridade entre *strings* na priorização de casos de teste. O processo de priorização feito nesse trabalho, transformava cada caso de teste (a nível de código ou então apenas o conjunto de entradas e saídas) em uma *string*, e então, as comparava com objetivo de marcar com prioridade alta, casos de teste com forte distinção em relação aos já priorizados. O grupo de Ledru comparou a ordenação randômica com quatro priorizações utilizando as funções distância: Hamming, Levenshtein, Manhattan e Cartesiana. Identificou-se que a priorização feita utilizando função distância foi mais eficiente na detecção de mutantes e que Hamming apresentou melhores resultados com um menor custo computacional. A forma de comparação de testes trazida pelo time de Ledru se alinha com a nossa quanto ao uso de funções distância. Em contrapartida, não objetivamos comparar testes de uma única versão da suíte, mas sim, versões distintas com edições inseridas nos modelos utilizados para sua geração.

No trabalho conduzido por Hemmati et al. [28], associa-se função distância com algoritmo genético para seleção de casos de teste pertencentes a suítes geradas automaticamente a partir de máquinas de estado. No seu contexto, não é avaliada a evolução dos modelos utilizados e a geração dos testes é feita a partir de um modelo distinto do utilizado nesta dissertação.

O CLARET apresentado por Jorge et al. [35], provê uma linguagem de especificação de casos de uso, com uma baixa curva de aprendizagem por possibilitar a descrição em linguagem natural desses casos, e é capaz de gerar modelos de maior formalismo que podem ser utilizados para geração automática de casos de teste. A linguagem natural utilizada para descrição dos casos de uso não é controlada, permitindo maior flexibilidade na escrita do passo a passo, que descreve o comportamento do sistema. Essa maior flexibilidade pode abrir espaço para erros de escrita e mal formação das sentenças, dependendo da experiência do usuário, aumentando a chance da presença de edições sintáticas nos modelos. Neste caso, a estratégia apresentada no nosso trabalho pode ser aplicada para tratar essas mudanças e se obter um maior reúso dos testes após modificações nos modelos.

Coutinho et al. [16], apresentam uma extensa avaliação de diferentes funções de similaridade aplicadas no sentido de redução da suíte de testes. De maneira similar, Oliveira et al. [48], examinam as funções distância mas, dessa vez, voltado a seleção de testes para execução no processo de integração contínua. Esses trabalhos demonstram o interesse do uso de funções de similaridade no âmbito de teste de sistema e servem como orientação para o uso dessas funções em diferentes contextos.

Neste capítulo, relacionamos trabalhos importantes e que discutem pontos relevantes para nossa pesquisa. No que concerne a TBM, os estudos indicam uma variedade de modelos a serem utilizados para geração de testes, demonstram também o interesse pelo uso de funções de distância aplicadas para diversos fins, e a preocupação de se manter as suítes atualizadas à medida que os modelos sofrem alterações.

Neste trabalho de dissertação, abordamos o problema pouco explorado na literatura, ao melhor do nosso conhecimento, relacionado a evolução dos modelos e seu impacto em suítes de teste geradas no contexto TBM, ambos descritos em linguagem natural. E, por meio da estratégia elaborada neste trabalho, indicamos uma solução para o alto descarte de testes impactados por mudanças meramente sintáticas e que são passíveis de reúso.

Capítulo 7

Conclusões

Motivado pelo alto descarte de casos de teste, detectado em projetos que adotam o uso de TBM na prática dos testes, este trabalho teve como objetivo principal investigar o uso de algoritmos de similaridade entre *strings* para a classificação automática de casos de teste. Estudos foram executados no intuito de responder a pergunta central: *É possível minimizar, de forma automática, o descarte de casos de teste após alterações/evolução nos modelos?*. Em síntese, verificamos que sim, o descarte pode ser minorado a partir da avaliação da similaridade entre casos de teste. Esse resultado obtido, auxilia a aplicação de TBM, quando utilizado modelos descritos em linguagem natural, no sentido de propiciar um maior reúso dos testes, preservar dados históricos destes e minimizar o esforço dos testadores quando se tem como objetivo testar partes modificadas do sistema.

Para investigar o impacto das edições realizadas nos casos de uso, foi conduzido um estudo exploratório utilizando dois projetos de software industriais. Essa investigação mapeou o impacto das edições nas suítes de teste, ligadas aos casos de uso editados, contabilizando o quanto dessas suítes se tornava obsoleto. Além disso, dividiu a taxa de obsoletos baseado no tipo de modificação envolvida, sintática ou semântica, e detectou a possibilidade de reúso dos testes descartados por modificações sintáticas, uma vez que nesses casos o comportamento do sistema é preservado.

Foi realizado um estudo experimental acerca do uso de funções distância para a classificação dos testes nos dois grupos alvo (sintático e semântico). Seis funções foram analisadas utilizando um conjunto de sentenças que passaram por modificações. Nesse estudo buscou-se utilizar funções com meios distintos de cálculo no comparativo das sentenças, para maior

diferenciação da funções. Demarcamos pontos onde cada função apresentava seu melhore índice de classificação, baseado na acurácia. Esse ponto, o limiar, indica o limite do quão diferentes podem ser as edições para serem consideradas sintáticas ou então passarem a ser semânticas. Como resultado, foi visto o alto potencial do uso destas funções na classificação das edições, e demonstrado que nenhuma se sobressaiu com relação as demais. Todas apresentaram capacidade de classificação equivalente de acordo com o teste estatístico utilizado.

Uma estratégia de classificação foi delineada para classificação dos casos de teste. Tal estratégia se mostrou capaz de avaliar duas suítes de testes, descritas em linguagem natural e geradas no contexto TBM a partir de versões subsequentes de um mesmo caso de uso, e detectar: i) testes passíveis de reúso por possuírem apenas edições sintáticas; ii) testes que de fato devem ser descartados por exercitar partes do sistema que passaram por grandes modificações; iii) testes que se mantiveram intactos e são reutilizáveis; e, iv) os novos testes, que estão presentes na suíte mais nova e não possuem forte semelhança com nenhum teste da suíte mais antiga.

Por fim, o trabalho apresentou a validação da estratégia delineada utilizando um projeto de software industrial. Foi utilizado um projeto diferente dos quais se utilizou para delinear o problema e motivar a implementação da estratégia. Como resultado, foi mostrado a efetividade, com relação à acurácia.

7.1 Contribuições

Diversos artigos podem ser encontrados na literatura acerca de teste de software no contexto de TBM. O uso de TBM auxilia o processo de teste, podendo com menos esforço, gerar suítes tão boas quanto as escrita manualmente. No entanto, sua tendência em gerar suítes com larga quantidade de testes e o alto descarte destes ao evoluir os modelos nos gera uma preocupação. Uma lacuna pode ser vista no cenário de análise de evolução dos modelos TBM e seu impacto nas suítes. Acerca deste ponto, este trabalho trouxe avaliações em projetos industriais que utilizam TBM para apoiar o testes de software. Em específico, voltado aos testes funcionais, descritos em linguagem natural para execução manual.

A estratégia aqui definida, mesclou outro campo da literatura já aplicado em várias áreas da computação, as funções de distância, com o TBM para possibilitar um maior reúso dos

testes gerados ao longo das evoluções. Próximo ao nosso contexto, funções distância já são utilizadas em seleção de casos de teste [10, 49], as quais apresentam bons resultados. No entanto, nesses casos, não são considerados os tipos de modificação, como levantado em nosso trabalho.

O comparativo entre funções distância, em conjunto com a definição das configurações ajustadas para cada uma delas, nos traz um guia para outros trabalhos inseridos no mesmo cenário, uma vez que as funções podem ter desempenhos diferentes quando aplicadas em outros contextos.

O ambiente explorado pela pesquisa, projetos os quais utilizam metodologias ágeis e TBM, também é um campo que deve ser melhor investigado. A aplicabilidade de TBM em metodologias modernas, de forma que se ganhe no reúso dos testes e na preservação dos dados históricos dos testes, contribui com times que já utilizam essas abordagens ou até mesmo na sua adesão.

O estudo conduzido na Seção 3.1 foi publicado no *III Brazilian Symposium on Systematic and Automated Software Testing* (SAST) [58], organizado no ano de 2018 pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC/USP). Tal publicação reforça o interesse da academia pelo tema TBM e pelas evidências encontradas no trabalho, as quais demonstram a aplicação de TBM em um contexto industrial e o impacto, causado pelas evoluções dos casos de uso, nas suítes de teste geradas.

O estudos mostrados nas Seções 4.3 e 5.1, originaram o artigo *"Reducing the Discard of MBT Test Cases using Distance Functions"* que foi aceito para publicação no *XXXIII Brazilian Symposium on Software Engineering* (SBES) [17], conduzido no ano de 2019 pela Universidade Federal da Bahia (UFBA). O trabalho reforça na comunidade científica o uso de funções de similaridade no contexto de testes e evolução de software. A publicação permite que outros pesquisadores discutam sobre o tema, sugiram novos caminhos de investigação e ampliem o assunto com outros trabalhos.

7.2 Trabalhos Futuros

Os estudos realizados nesta dissertação foram de grande importância para se dar o pontapé inicial a respeito do tema abordado. Vislumbramos trabalhos futuros que atacam limitações

enfrentadas no nosso trabalho, ou então, que expandem o trabalho até aqui realizado.

Facilitar o uso da estratégia implementada: Implementar o suporte gráfico para utilização da estratégia facilitaria seu uso. Dessa forma, poderiam ser adicionadas via interface as suítes para análise e ser feita a escolha do caminho destino para as suítes organizadas de acordo com a classificação obtida.

Conduzir estudos de caso com outras fontes de dados: A inclusão de novas fontes de dados aumentaria a confiança nos resultados obtidos ao longo desta dissertação. Embora os dados analisados tenham sido extraídos de diferentes projetos de software, todos fazem parte da parceria firmada entre o SPLab e a Ingenico do Brasil Ltda. Outros contextos de estudo seriam bem vindos e fortaleceriam as análises.

Avaliar outras formas de classificação: Utilizamos como classificação, a função distância Levenshtein, e comparamos diferentes funções. No entanto, em um trabalho futuro pretendemos utilizar uma rede semântica, e.g. WordNet[1], como outra forma de avaliar as edições e os casos de teste.

[1] https://wordnet.princeton.edu/

Bibliografia

[1] S. S. Aljameel, J. D. O'Shea, K. A. Crockett, and A. Latham. Survey of string similarity approaches and the challenging faced by the arabic language. In *2016 11th International Conference on Computer Engineering Systems (ICCES)*, pages 241–247, Dec 2016.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[3] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[4] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[5] Robert Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] Kurt Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[8] Goutam Budha, Namita Panda, and Arup Abhinna Acharya. Test case generation for use case dependency fault detection. In *2011 3rd International Conference on Electronics Computer Technology*, volume 1, pages 178–182. IEEE, 2011.

[9] Emanuela Cartaxo, Wilkerson Andrade, Francisco Oliveira, and Patrícia Machado. Lts-bt: A tool to generate and select functional test cases for embedded systems. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 1540–1544, New York, NY, USA, 2008. ACM.

[10] Emanuela Cartaxo, Francisco Oliveira, and Patrícia Machado. Automated test case selection based on a similarity function. volume 2, pages 399–404, 01 2007.

[11] Emanuela G. Cartaxo, Patrícia D. L. Machado, and Francisco G. Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Softw. Test. Verif. Reliab.*, 21(2):75–100, June 2011.

[12] Emanuela G Cartaxo, Francisco GO Neto, and Patricia DL Machado. Test case generation by means of uml sequence diagrams and labeled transition systems. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 1292–1297. IEEE, 2007.

[13] William Cohen, Pradeep Ravikumar, and Stephen Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the 2003 International Conference on Information Integration on the Web*, IIWEB'03, pages 73–78. AAAI Press, 2003.

[14] Conformiq. Conformiq. https://www.conformiq.com/. [Online; acessado em 20-Março-2019].

[15] Xavier De Coster, Charles De Groote, Arnaud Destiné, Pierre Deville, Laurent Lamouline, Thibault Leruitte, and Vincent Nuttin. Mahalanobis distance , jaro-winkler distance and ndollar in usigesture. 2011.

[16] Ana Emília Victor Barbosa Coutinho, Emanuela Gadelha Cartaxo, and Patrícia Duarte de Lima Machado. Analysis of distance functions for similarity-based test suite reduc-

tion in the context of model-based testing. *Software Quality Journal*, 24(2):407–445, Jun 2016.

[17] Thomaz Diniz, Everton Alves, Anderson Silva, and Wilkerson Andrade. Reducing the discard of mbt test cases using distancefunctions. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES '19, New York, NY, USA, 2019. ACM.

[18] Mohammadreza Ektefa, Marzanah Jabar, Fatimah Sidi, Sara Memar, Hamidah Ibrahim, and Abdullah Ramli. A threshold-based similarity measure for duplicate detection. In *2011 IEEE Conference on Open Systems*, pages 37–41, Sep. 2011.

[19] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[20] Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *J. Syst. Softw.*, 81(5):649–660, May 2008.

[21] David Faragó. Model-based testing in agile software development. *30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing meets Agility*, 2010.

[22] Felype Ferreira, Laıs Neves, Michelle Silva, and Paulo Borba. Target: a model based product line testing tool. *Tools Session of CBSoft*, 2010.

[23] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[24] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, 2013.

[25] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: Tools and methodology. *Softw. Test. Verif. Reliab.*, 21(1):55–71, March 2011.

[26] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Sundaram. Text mining for software engineering: How analyst feedback impacts final results. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[27] Hadi Hemmati and Lionel Briand. An industrial investigation of similarity measures for model-based test case selection. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 141–150, Nov 2010.

[28] Hadi Hemmati, Lionel Briand, Andrea Arcuri, and Shaukat Ali. An enhanced test case selection approach for model-based testing: An industrial case study. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 267–276, New York, NY, USA, 2010. ACM.

[29] Sofija Hotomski, Eya Ben Charrada, and Martin Glinz. Keeping evolving requirements and acceptance tests aligned with automatically generated guidance. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 247–264. Springer, 2018.

[30] Sofija Hotomski and Martin Glinz. Guidegen: a tool for keeping requirements and acceptance tests aligned. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 49–52. ACM, 2018.

[31] Anna Huang. Similarity measures for text document clustering.

[32] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. Object-oriented software engineering - a use case driven approach. In *TOOLS*, 1992.

[33] Claude Jard and Thierry Jéron. A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. 2005.

[34] Dalton Jorge, Wilkerson Andrade, Patrícia Machado, and Everton Alves. Claret - central artifact for requirements engineering and model-based testing. In *Proceedings of the 24th Tools Session / 8th Brazilian Conference on Software: Theory and Practice*, pages 41–48, Fortaleza, CE, BR, 2017.

[35] Dalton Jorge, Patrícia Machado, Everton Alves, and Wilkerson Andrade. Integrating requirements specification and model-based testing in agile development. In *Proceedings of the 2018 26th IEEE International Requirements Engineering Conference (RE)*, pages 1–11, 2018. To appear.

[36] Youry Khmelevsky, Xitong Li, and Stuart Madnick. Software development using agile and scrum in distributed teams. In *2017 Annual IEEE International Systems Conference (SysCon)*. IEEE, apr 2017.

[37] Grzegorz Kondrak. N-gram similarity and distance. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval*, SPIRE'05, pages 115–126, Berlin, Heidelberg, 2005. Springer-Verlag.

[38] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, Mar 2012.

[39] Hareton Leung and Lee White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance - 1989*. IEEE Comput. Soc. Press, 1989.

[40] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[41] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. String similarity measures and joins with synonyms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 373–384, New York, NY, USA, 2013. ACM.

[42] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[43] Mehdi Mirzaaghaei. Automatic test suite evolution. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 396–399, New York, NY, USA, 2011. ACM.

[44] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezze. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, apr 2012.

[45] Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[46] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[47] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects of Computing*, 26(3):441–490, May 2014.

[48] Francisco Oliveira, Azeem Ahmad, Ola Leifler, Kristian Sandahl, and Eduard Enoiu. Improving continuous integration with similarity-based test case selection. In *Proceedings of the 13th International Workshop on Automation of Software Test*, AST '18, pages 39–45, New York, NY, USA, 2018. ACM.

[49] Francisco Oliveira, Richard Torkar, and Patrícia Machado. Full modification coverage through automatic similarity-based test case selection. *Information and Software Technology*, 80:124–137, dec 2016.

[50] Tom Pender. *UML Bible*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[51] Leandro Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 33:1–33:11, New York, NY, USA, 2012. ACM.

[52] Leandro Pinto, Saurabh Sinha, and Alessandro Orso. Testevol: a tool for analyzing test-suite evolution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1303–1306. IEEE Press, 2013.

[53] Olli-Pekka Puolitaival. Adapting model-based testing to agile context. VTT, 2008.

[54] Linda Rising and Norman Janoff. The scrum software development process for small teams. *IEEE software*, 17(4):26–32, 2000.

[55] Monalisa Sarma and Rajib Mall. Automatic test case generation from uml models. In *Proceedings of the 10th International Conference on Information Technology*, ICIT '07, pages 196–201, Washington, DC, USA, 2007. IEEE Computer Society.

[56] Edgar Sarmiento, Julio Cesar Sampaio do Prado Leite, and Eduardo Almentero. C&l: Generating model based test cases from natural language requirements descriptions. In *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, pages 32–38. IEEE, 2014.

[57] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[58] Anderson Silva, Wilkerson Andrade, and Everton Alves. A study on the impact of model evolution in mbt suites. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*, SAST '18, pages 49–56, New York, NY, USA, 2018. ACM.

[59] Thorvald Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Biol. Skr.*, 5:1–34, 1948.

[60] Mario Špundak. Mixed agile/traditional project management methodology–reality or illusion? *Procedia-Social and Behavioral Sciences*, 119:939–948, 2014.

[61] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.

[62] Vinícius Teles. Extreme programming. *São Paulo: Novatec*, 2004.

[63] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing:*

An Outcome of the FORTEST Network, Revised Selected Papers, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

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

[65] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[66] MK Vijaymeena and K Kavitha. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal*, 3(2):19–28, 2016.

[67] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[68] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10:399–417, 2015.

[69] Yanbing Yu, James Jones, and Mary J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 201–210. IEEE, 2008.

[70] Jianfang Zhou. Comparison function for chinese string in entity identification. In *2014 Seventh International Symposium on Computational Intelligence and Design*, volume 1, pages 271–273, Dec 2014.