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

CLARET: Um Artefato Central para Engenharia de Requisitos e Teste Baseado em Modelos

Dalton Nicodemos Jorge

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 Andrade e Patricia Machado
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

J82c Jorge, Dalton Nicodemos.

Claret : um artefato central para engenharia de requisitos e teste baseado em modelos / Dalton Nicodemos Jorge. Campina Grande, 2018.

73 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.

"Orientação: Prof. Dr. Wilkerson de Lucena Andrade, Profa. Dra. Patrícia Duarte de Lima Machado".

Referências.

1. Teste Baseado em Modelos. 2. Engenharia de Requisitos. 3. Geração Automática. I. Andrade, Wilkerson de Lucena. II. Machado, Patrícia Duarte de Lima. III. Título.

CDU 004.41(043)

"UM ARTEFATO CENTRAL PARA ENGENHARIA DE REQUISITOS E TESTE BASEADO EM MODELO"

DALTON NICODEMOS JORGE

DISSERTAÇÃO APROVADA EM 17/07/2017

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

PATRICIA DUARTE DE LIMA MACHADO, Ph.D, UFCG
Orientador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

JULIANO MANABU IYODA, Ph.D., UFPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

A construção de casos de testes da forma tradicional tem sido, em grande medida, um processo lento e dispendioso em termos financeiros. Nesse sentido, a geração automática de testes baseados em modelos surge como uma alternativa promissora para solucionar tais percalços. A especificação de requisitos do sistema é uma atividade proveniente da engenharia de requisitos que tem como saída diagramas ou documentos detalhados como os casos de uso. Em especial, o documento de caso de uso pode servir como forte candidato para um modelo central no processo de geração automática de testes. Entretanto, o modelo escolhido deve prover informações suficientes para permitir a geração automática de testes e ao mesmo tempo ser sucinto o bastante para não gerar custos e esforços adicionais à engenharia de requisitos. A solução proposta nesta pesquisa tem como objetivo integrar a geração automática de testes baseado em modelos com a engenharia de requisitos, através da proposta de uma notação em linguagem de domínio específico com o apoio de ferramentas visando a geração automática tanto de meta-modelos para auxiliar a criação de casos de testes quanto de documentos formatados de caso de uso.

Abstract

The construction of test cases in the traditional way has been, to a large extent, a slow and costly process in financial terms. In this sense, the automatic generation of model-based tests emerges as a viable alternative to solve such mishaps. The system requirements specification is an activity from the requirements engineering whose output diagrams, or detailed documents as the use cases. In particular, the use case document can serve as a strong candidate for a central model in the process of automatic generation of tests. However, the chosen model should provide sufficient information to allow the automatic generation of testing and at the same time be succinct enough not to generate additional cost and effort to requirements engineering. The solution proposed in this research aims to integrate the automatic generation of model-based testing with requirements engineering, through the proposal of a domain-specific language notation with the support of tools aimed at the automatic generation of both meta-templates to assist the creation of test cases as formatted documents of use case.

Agradecimentos

Aos meus adorados pais, **Lindório Jorge** e **Lourdinha Nicodemos**, por todo o amor, apoio e dedicação ofertados em todos os momentos da minha vida. Aos meus preciosos irmãos, **Italo Jorge**, **Alane Nicodemos** e **Karine Nicodemos**, pelo carinho e consideração.

A minha amada esposa **Maylle Benício**, por estar tão presente na minha vida, ajudando e incentivando cada vez mais. Aos meus queridos sogros, **Bonifácio Benício** e **Noêmia Benício**, pelo acolhimento, apoio e sábios conselhos.

Aos meus orientadores **Wilkerson Andrade** e **Patricia Machado**, pelos votos de confiança que me foram dados antes mesmo do mestrado, pela dedicada orientação e pela paciência e motivação em me ensinar.

A **Adalberto Cajueiro**, pela amizade, respeito e solicitude de longas datas. Muito obrigado por todas as oportunidades profissionais e acadêmicas que tem me proporcionado desde a graduação.

Aos amigos **Achiles Pedro** e **Emmanuel Carvalho**, grandes companheiros e motivadores na saga acadêmica. Aos professores do SPLab, em especial **Franklin Ramalho**, **Tiago Massoni** e **Everton Galdino**, pela atenção e contribuições concedidas.

A todos os membros do SPLab: funcionários, pós-graduandos e integrantes de projetos, que de alguma forma colaboraram com este trabalho. Meu especial agradecimento ao **Alysson Filgueira** e **Arthur Marques**, pelo altruísmo e presteza em compartilharem seus conhecimentos.

Aos professores da banca examinadora, **Tiago Massoni** e **Juliano Iyoda**, pela disposição e gentileza em avaliar este trabalho.

A todos os demais que não foram aqui citados, reconheço e agradeço imensamente a importância de vocês em cada vitória conquistada.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

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

BDD *Behaviour-Driven Development*

CCCU Completude e Consistência de Casos de Uso

CNL *Controled Natural Language*

CR Cobertura de Requisitos

CSP *Communicating Sequential Processes*

DSL *Domain-Specific Language*

EBNF *Extended Backus-Naur Form*

GQM *Goal, Question, Metric*

IEEE *Institute of Electrical and Electronics Engineers*

JSON *JavaScript Object Notation*

LTS *Label Transition Systems*

LTS-BT *Labeled Transition System Based-Testing*

MBT *Model-Based Testing*

ODF *Open Document Format*

SPLab *Software Practices Laboratory*

SST Sistema Sob Teste

T Tempo

TGF *Trivial Graph Format*

UFCG Universidade Federal de Campina Grande

UML *Unified Modeling Language*

Capítulo 1

Introdução

Teste de *Software* é uma atividade crucial para o desenvolvimento de *software*, pois avalia se o sistema está em conformidade com o comportamento esperado para o uso, e também verifica se o sistema atende totalmente os requisitos estabelecidos [1]. Existem diversos tipos de testes, que são classificados quanto ao nível de abstração (unitários, componentes, integração, sistema), quanto às características (funcional, robustez, performance, usabilidade) e quanto ao tipo de informação (caixa-preta ou caixa-branca) [2].

As atividades de teste acarretam custo alto para o desenvolvimento de *software* [3]. O orçamento de um projeto de *software* pode ser comprometido com atividades que compreendem: a elaboração do plano de teste e escrita dos casos de teste; a configuração do ambiente e execução sistemática dos casos de teste segundo um roteiro previamente elaborado; e a localização e correção dos problemas encontrados.

Inúmeras investigações têm sido feitas para amenizar este problema do alto custo da implantação de testes e entre elas podemos citar o Teste Baseado em Modelos — do inglês *Model-Based Testing* (MBT) [4] —, abordagem que se propõe a reduzir custos e tempo. MBT é uma abordagem que possibilita a geração sistemática de casos de testes através de informações fornecidas por um modelo abstrato, tais como os estímulos necessários para o sistema reagir, os valores de saída esperadas do sistema, e a checagem destas saídas para avaliar se estão corretas. Em níveis de abstração, MBT tem sido mais empregrado em testes de sistema. Já em relação às características, MBT é mais utilizado para testes funcionais, em que se quer avaliar o comportamento esperado, e também com testes de robustez, cujo objetivo é medir a resistência às falhas de *software*. Quanto à classificação do tipo de informação, MBT é intrinsecamente caixa-preta, pelo fato de utilizar informações extraídas de modelos de alto nível e não por meio de código para guiar os testes.

Dentre os modelos abstratos que possibilitam a geração de casos de testes, estão artefatos comportamentais da Linguagem de Modelagem Unificada — do inglês *Unified Modeling Language* (UML) — como o diagrama de sequência [5, 6, 7, 8], diagrama de estado [9], diagrama de atividades [9], o documento de requisitos do sistema [10, 11] escrito no formato de casos de uso em linguagem natural, ou alguma notação de modelagem para

satisfazer os objetivos dos testes. Peculiarmente, quando casos de teste são gerados a partir de documentos de requisitos, deve existir necessariamente um relacionamento entre o processo de teste e a engenharia de requisitos.

A engenharia de requisitos é um processo que compreende atividades que auxiliam na criação e manutenção de documentos de requisitos de um sistema. Sendo assim, as práticas da engenharia de requisitos permitem que os envolvidos em um projeto de desenvolvimento conheçam melhor os detalhes do que se espera do *software* antes dele ser construído [12]. Este conhecimento prévio evita um direcionamento errado do projeto e consequentemente um retrabalho, além de custos não previstos no orçamento e no cronograma. Contudo, estudos mais recentes apontam que cinco dos oito principais fatores de fracasso em projetos de *software* se deve a requisitos incompletos, pouco envolvimento com o cliente, perspectivas não realistas, mudanças em requisitos e requisitos inúteis [13]. Tais problemas têm sido combatidos com a adoção das práticas ágeis, como a colaboração frequente dos envolvidos, desenvolvimento iterativo e mudanças de requisitos em todas as fases do projeto. Esta integração dos métodos ágeis com a engenharia de requisitos tem mostrado resultados promissores [14].

Nesse contexto, a escolha do modelo que serve como insumo para a geração de casos de teste deverá seguir alguns critérios: ser abstrato o suficiente em relação ao tamanho do sistema, para que o custo para produzi-lo não seja alto, mas também ser detalhado o bastante, contendo todas as informações sobre o comportamento desejado para o sistema em desenvolvimento.

Para as abordagens de MBT a partir da criação manual do modelo de testes e de requisitos, surgem questões que podem impactar os custos como: *(i)* a manutenção do ciclo de vida de inúmeros modelos, visto que a formalidade da escrita e a quantidade de modelos a serem trabalhados — muitas vezes de forma simultânea — podem demandar um esforço extra; *(ii)* um possível distanciamento semântico entre o modelo e os conceitos tecnológicos do sistema sob teste, que pode ser ocasionado pela inexistência de uma padronização dos termos e conceitos tecnológicos entre a equipe que especifica os casos de testes; e *(iii)* a disponibilidade e custos de ferramentas para criação e manutenção do modelo, que nem sempre estão acessíveis aos participantes do desenvolvimento. Por outro lado, as abordagens de MBT que utilizam o modelo de requisitos como base para o modelo de testes através de formalismos e/ou transformações podem sofrer com a falta de rastreabilidade entre o modelo abstrato e o modelo de testes. Tais questionamentos são originados a partir do problema que é a existência de um cenário no qual se exige documentação suficiente de requisitos para guiar o desenvolvimento e facilitar a comunicação com os envolvidos, como também o emprego de testes manuais de sistema.

Esta dissertação propõe uma abordagem para a especificação de casos de uso por meio de uma nova linguagem de domínio específico, a qual denominamos de Claret. Agregado à linguagem, existe um apoio ferramental com duas finalidades: *(i)* geração automática

de modelos intermediários para testes manuais de *software*; *(ii)* geração automática de documentos de casos de uso formatados para a Engenharia de Requisitos.

É diante deste arcabouço contextual e conceitual exposto que se situa o desenho da pesquisa que alicerçou esta dissertação.

1.1 Objetivos

O objetivo geral desta dissertação é aproximar a engenharia de requisitos com as metodologias ágeis para reduzir os problemas de perda de direcionamento e custos extras em projetos, além de favorecer o processo de teste baseado em modelos. Para atingir este objetivo, propomos uma abordagem para a especificação de casos de uso através de uma linguagem de domínio específico. Esta linguagem tem dois propósitos: servir como artefato central para geração automática de documentos de casos de uso, suprindo as necessidades da engenharia de requisitos; e atuar como modelo de testes em nível de sistema para MBT.

Neste trabalho, buscamos responder a seguinte questão de pesquisa geral:

> É possível estabelecer uma relação entre a especificação e documentação da engenharia de requisitos e minimização de documentos dos métodos ágeis de modo que gere benefícios a ambos e ao teste baseado em modelo?

Para alcançar o objetivo geral da pesquisa e responder as questões de pesquisa, foram delineados e executados os seguintes objetivos específicos:

1. Definição e especificação de uma gramática livre de contexto para a escrita de caso de uso textual, através da notação *Extended Backus-Naur Form* (EBNF) [15];

2. Desenvolvimento de uma ferramenta para: *(i)* avaliação de conformidade de sintaxe nas especificações de requisitos, de acordo com a gramática definida, e *(ii)* produção de uma árvore sintática abstrata para posterior derivação em artefatos de MBT e engenharia de requisitos;

3. Desenvolvimento de dois módulos da ferramenta para, a partir da árvore sintática abstrata gerada, possibilitar a geração automática de: *(i)* modelos de teste para o processo de MBT, e *(ii)* de documentos formatados da engenharia de requisitos a partir da árvore sintática abstrata;

4. Validação da notação e suporte ferramental propostos por meio de dois experimentos e dois estudos de caso.

1.2 Relevância da Pesquisa

A principal contribuição deste trabalho é a elaboração de uma abordagem sistemática para geração automática de artefatos para MBT e para a engenharia de requisitos. Esta dupla função é obtida através de um modelo central escrito em uma linguagem de domínio específico.

Um ponto relevante neste trabalho é que a linguagem proposta não acrescenta custos extras em seu aprendizado, pois a sua sintaxe assemelha-se a estrutura de um caso de uso textual da UML e o formato *JavaScript Object Notation* (JSON). Dessa forma, a curva de aprendizagem é minimizada e o conhecimento prévio na especificação de casos de uso tradicionais é aproveitado.

Outro ponto de relevância deste trabalho é que estamos contribuindo para estender o uso do MBT no contexto das práticas ágeis, que defendem um esforço mínimo em documentações. Ademais, espera-se ainda que as contribuições também sejam relevantes no contexto do fluxo de trabalho, facilitando a criação e manutenção dos modelos de testes e documentos da engenharia de requisitos.

Em resumo, nossa proposta possibilitará uma integração gerenciável e consistente do MBT e da engenharia de requisitos.

1.3 Metodologia

A especificação da gramática para a notação proposta foi alicerçada na definição do modelo de caso de uso textual da UML[1]. Utilizamos a notação EBNF para escrita da gramática, o que colaborou na validação e implementação do respectivo *parser*.

A linguagem escolhida para o desenvolvimento do suporte ferramental foi *Scala*[2], que faz uso do ecossistema da *Java Virtual Machine*[3] (JVM). Empregamos a biblioteca *Parser Combinator*[4] para auxiliar a implementação dos analisadores léxico e sintático, segundo a nossa gramática especificada anteriormente, e obter a árvore sintática abstrata. O módulo responsável pela geração de artefatos para a engenharia de requisitos fez uso de um mecanismo de *templates* baseado no *Mustache*[5] para personalização dos documentos. Já o módulo encarregado pela geração de metamodelos para teste fez uso da biblioteca padrão da linguagem *Scala*.

A pesquisa foi guiada por uma abordagem quali-quantitativa, que abriu espaço não apenas para observações em ambientes controlados, mas também para estudos exploratórios que viabilizaram identificar questões que estavam fora de nosso controle, a exem-

[1] http://omg.org/spec/UML/2.5/
[2] http://www.scala-lang.org
[3] http://docs.oracle.com/javase/specs/jvms/se8/html/index.html
[4] https://github.com/scala/scala-parser-combinators
[5] https://mustache.github.io

plo dos fenômenos individuais e comportamentais do processo. Realizamos um total de dois experimentos e dois estudos de caso, com o intuito de avaliar o trabalho proposto sob diferentes ângulos. A unidade de análise da pesquisa restringiu-se a desenvolvedores experientes pertencentes a projetos de colaboração entre empresas privadas de tecnologia e a Universidade Federal de Campina Grande, por meio do Laboratório de Práticas de Software (SPLab).

O principal objetivo dos experimentos foi avaliar a linguagem proposta. Nos dois experimentos foi realizada uma comparação da notação de CLARET com a escrita livre. No primeiro, utilizamos os critérios de cobertura, consistência, completude e tempo no tocante à especificação de casos de uso. A amostra contou com 8 participantes que realizaram uma tarefa pré-estabelecida. A análise dos dados empregou um *checklist* disponibilizado pelo projeto *Open UP*[6] e o teste de hipótese *Chi-Squared*. Para o segundo experimento, as métricas adotadas foram a legibilidade e a capacidade de escrita, também no contexto de caso de uso. Seis indivíduos colaboraram com a pesquisa, mediante realização de tarefa e respostas a um questionário proposto, baseado na escala *Likert*. A análise dos dados foi feita através de histogramas, gráficos *Q-QPlot*, teste de normalidade *Shapiro-Wilk* e teste de hipótese *Wilcoxon-Mann-Whitney*. Todos os testes estatísticos de normalidade e de hipóteses dos experimentos foram realizados na ferramenta *RStudio*[7].

Quanto aos estudos de caso, o objetivo foi avaliar o processo proposto em CLARET. O primeiro, buscou comprovar a efetividade em capturar defeitos em testes gerados a partir das especificações escritas com a ferramenta. Os participantes foram 3 engenheiros de software e dois projetos de *software* distintos. A análise dos dados foi construída com base nos índices percentuais dos resultados da execução dos testes. O segundo estudo de caso investigou a abordagem proposta por CLARET em sua totalidade. Sete membros de uma equipe de projeto de validação e verificação de *software* participaram do estudo. Os resultados foram analisados em duas etapas: a primeira através dos índices percentuais das respostas objetivas do questionário aplicado (alicerçado na escala *Likert*); a segunda etapa através da análise das respostas dos participantes às questões abertas do questionário, aliado ao nosso registro observacional no decorrer de 3 meses, no período de novembro de 2016 a janeiro de 2017.

1.4 Escopo

Este trabalho abrange dois campos de pesquisa: a engenharia de requisitos e testes baseados em modelos.

Na engenharia de requisitos, este trabalho limita-se à geração automática de documentos formatados de requisitos do sistema, mais especificamente os casos de uso. Em

[6] http://epf.eclipse.org/wikis/openup/
[7] https://www.rstudio.com

relação ao MBT, esta pesquisa restringe-se à transformação automática do modelo intermediário que servirá como entrada para geração de testes manuais, processo este que não é de responsabilidade da abordagem proposta.

1.5 Estrutura da Dissertação

Os capítulos seguintes desta dissertação estão estruturados da seguinte forma:

Capítulo 2 - Fundamentação Teórica Este capítulo fornece o entendimento mínimo necessário para melhor compreensão deste trabalho. São apontados os conceitos fundamentais de teste de *software*, com enfoque no teste baseado em modelo. Também são conceituados a UML, os requisitos, a engenharia de requisitos e casos de uso.

Capítulo 3 - Claret: a linguagem, a ferramenta e o processo Apresenta a abordagem proposta para geração automática de artefatos para MBT e para a engenharia de requisitos. São delineados a linguagem, o suporte ferramental e o processo.

Capítulo 4 - Avaliação Expõe os estudos empíricos e exploratórios para a abordagem de Claret. Ao final, são apresentados e discutidos, por meio de análises, os resultados obtidos.

Capítulo 5 - Trabalhos Relacionados Evidencia os trabalhos relacionados da literatura que fazem uso dos conceitos de MBT e engenharia de requisitos. Além disso, são retratadas as semelhanças e diferenças dos trabalhos relacionados com o proposto nesta dissertação.

Capítulo 6 - Conclusões Traz à tona os resultados obtidos com esta dissertação e as possibilidades para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo explicitar o arcabouço teórico que fundamenta os principais conceitos instrumentalizados neste trabalho. Sendo assim, serão abordados e discutidos os conceitos básicos relacionados a teste de *software* e teste baseado em modelo (Seção 2.1), a UML (Seção 2.2), a requisitos (Seção 2.3), a engenharia de requisitos (Seção 2.4) e casos de uso (Seção 2.5).

2.1 Teste de Software

Cada vez mais, estamos cercados por dispositivos e sistemas com o intuito de nos fornecer algum tipo de serviço. Os comportamentos destes sistemas ou dispositivos são ditados pelos *softwares* que os controlam e se espera que estes funcionem sempre de forma satisfatória.

Mesmo com as melhores práticas de programação, se faz necessário garantir que o *software* esteja de acordo com os seus requisitos esperados. Isso é de suma importância devido a diversos fatores, que vão desde aspectos de segurança em sistemas críticos, até ao fato de que erros detectados tardiamente representam um custo altíssimo para seus desenvolvedores [16]. Dessa forma, teste de *software* é o principal método que a indústria utiliza para avaliar o sistema em desenvolvimento antes de colocá-lo em produção [1].

McGregor and Sykes definem teste de *software* como uma atividade que busca identificar indícios de defeitos que tenham sido inseridos em alguma etapa durante o processo de desenvolvimento ou manutenção de *software*. Estes defeitos são consequência de erros, enganos, omissões ou interpretações equivocadas dos requisitos por parte dos desenvolvedores [17]. No que diz respeito à terminologia empregada em teste de *software*, existe uma certa inconsistência na literatura, gerada pela confusão no emprego de alguns termos, dentre os quais podemos citar:

- **Erro**: é a ação cometida por engano pelo ser humano, em virtude de fatores como: prazos apertados, código complexo, complexidade na infraestrutura, mudanças de

tecnologia e/ou múltiplas interações no sistema [18].

- **Defeito**: é o resultado do erro no código do sistema ou documento. Um defeito pode ou não dar origem a uma falha [18].
- **Falha**: é a manifestação executável do defeito, que impede o sistema de executar o que deveria ser feito, ou o contrário, fazer uma ação que não estava prevista [18].
- **Caso de teste**: é o conjunto de entradas (pré-condições e dados de entrada) e saídas esperadas (pós-condições e dados de saída) para um determinado comportamento de um programa [19].
- **Validação**: avalia se o *software* entregue está em concordância com o uso esperado e depende do conhecimento do domínio [1].
- **Verificação**: determina se os artefatos de *software* resultantes de uma fase do desenvolvimento atendem aos requisitos firmados para esta fase [1].

Em síntese, teste de *software* é a atividade de detectar falhas através da execução de um sistema. É relevante frisar que existem outras técnicas, diferentes e complementares, para melhoria da qualidade do *software*, tais como: análise estática, inspeções, revisões, depuração (*debugging*) e correção de erros. Estes dois últimos processos são realizados após a detecção das falhas [2].

Quanto à classificação do teste de software, Utting and Legeard demonstram os diversos tipos distribuídos em um sistema tridimensional de eixos ortogonais entre si [2] (adaptado de Tretmans [20]) conforme é ilustrado na Figura 2.1. O eixo vertical z informa a escala do sistema sob desenvolvimento, e partindo do ponto mais próximo da origem, encontramos *(i)* o teste de unidade (ou teste unitário), *(ii)* o teste de componente, *(iii)* teste de integração e finalmente *(iv)* o ponto mais distante da origem, que é o teste de sistema.

Já o eixo x compreende as características que se deseja testar e engloba *(i)* o teste funcional, *(ii)* teste de robustez, *(iii)* teste de performance e *(iv)* teste de usabilidade. No eixo y encontramos o tipo de informação necessária para criar o teste, sendo eles: *(i)* o teste de caixa-preta, que significa que não se sabe detalhes da implementação – apenas com informações dos requisitos – e *(ii)* o teste de caixa-branca, que utiliza o código de implementação para guiar os testes.

O tipo de teste mais comum é o teste funcional, também conhecido por teste comportamental. O objetivo do teste funcional é verificar se o programa está de acordo com as suas especificações, sem se deter aos detalhes de implementação. Por consequência, os testes funcionais podem ser adotados bem antes da disponibilização de código executável, e caso as especificações sejam escritas em linguagem formal, os testes funcionais podem

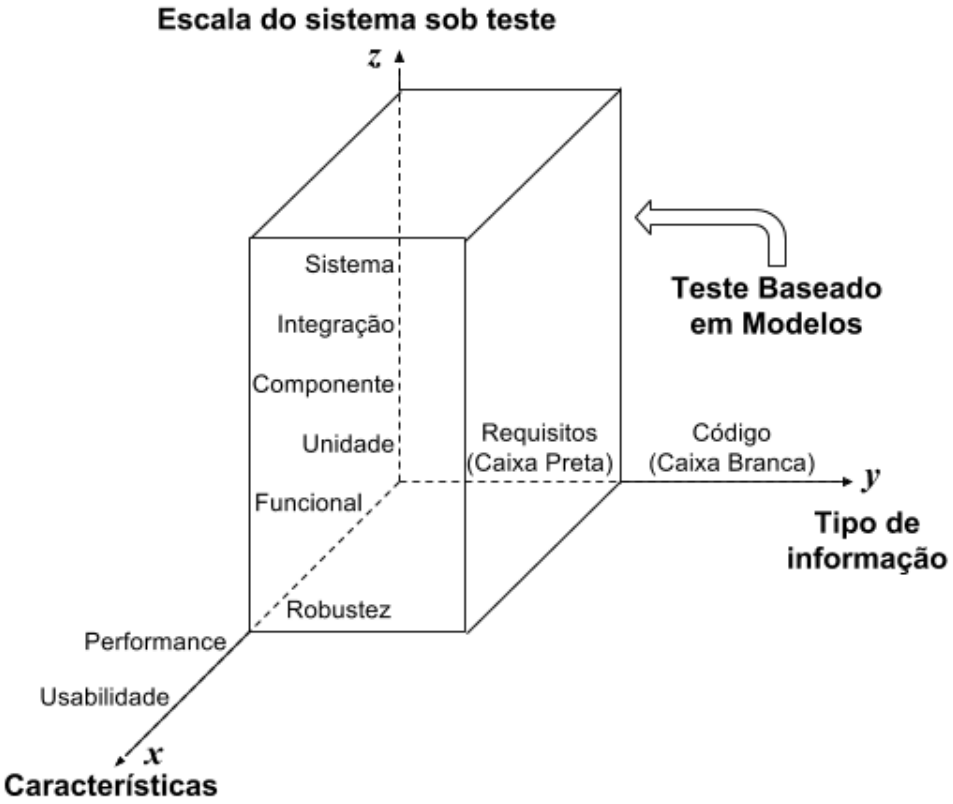


Figura 2.1: Classificação de Teste de Software

ter algum nível de automatização, o que economiza esforços e tempo, além de evitar erros inseridos em um processo de execução manual.

As práticas de desenvolvimento tradicionais, tais como Cascata ou Prototipação, adotam o processo de teste de *software* como uma das etapas finais em um projeto de desenvolvimento. No entanto, com a experiência construída através dos anos, tem se verificado uma maior eficiência quando os testes são aplicados ao longo de todo o processo de desenvolvimento do *software* [17].

2.1.1 *Teste Baseado em Modelo*

Presume-se que em decorrência da popularização da orientação a objetos e do emprego de modelos na engenharia de *software*, tenha surgido um conjunto de técnicas baseado em testes de caixa-preta, que foi intitulado Teste Baseado em Modelo, do inglês *Model Based Testing* (MBT) [4].

Segundo Kaner et al., todo teste é baseado em um modelo do sistema [21]. Sendo assim, o que diferencia o MBT das demais abordagens de testes – apesar de estar explícito no nome – não é o fato de ser baseado em um modelo. Seu diferencial é a geração automática dos casos de teste a partir do modelo que descreve o comportamento esperado pelo sistema. Utting and Legeard definem o MBT como a automação do processo de geração de testes de caixa-preta.

O modelo comportamental que representa o sistema deve possuir duas qualidades: *(i)* ser conciso o suficiente de tal forma que a sua escrita seja rápida e de fácil avaliação dos requisitos e *(ii)* ser preciso o bastante na descrição do comportamento que será avaliado nos casos de teste.

O MBT possui quatro abordagens de geração, listadas a seguir [2]:

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

Apesar destas quatro abordagens de geração, o escopo desta dissertação se limita apenas ao terceiro item: **geração de caso de teste com oráculos a partir de um modelo comportamental**. A razão para a escolha desta abordagem é que ela é a única geração automática que compreende: *(i)* os valores de entrada para os testes, *(ii)* a sequência de chamadas ao Sistema Sob Teste (SST), e *(iii)* os oráculos necessários para avaliar se os testes passaram ou falharam de acordo com os valores esperados na saída [2].

De uma forma geral, o processo de MBT contempla cinco etapas principais, ilustradas na Figura 2.2 e descritas a seguir:

1. Modelar o comportamento do SST;
2. Gerar os casos de teste;
3. Gerar as saídas esperadas;
4. Executar os testes;
5. Analisar os resultados dos testes.

A primeira etapa do processo de MBT consiste na especificação do modelo abstrato que atenda aos requisitos exigidos. Este modelo deve conter as características principais do comportamento do SST e ignorar os demais detalhes. Este modelo deve conter uma identificação para mapear a formalização do comportamento e os respectivos requisitos informais. Uma boa prática nesta etapa é a validação do modelo por alguma ferramenta de suporte. Isso visa identificar anomalias antes de passar para o próximo passo.

A segunda etapa do processo é a geração dos casos de teste a partir do modelo. Para isso, se faz necessária a escolha de um critério de seleção de teste, com o intuito de eleger quais testes farão parte de cada caso de teste. A saída desta etapa é um conjunto de testes, com base na sequência de passos informados no modelo. Outro artefato de saída nesta

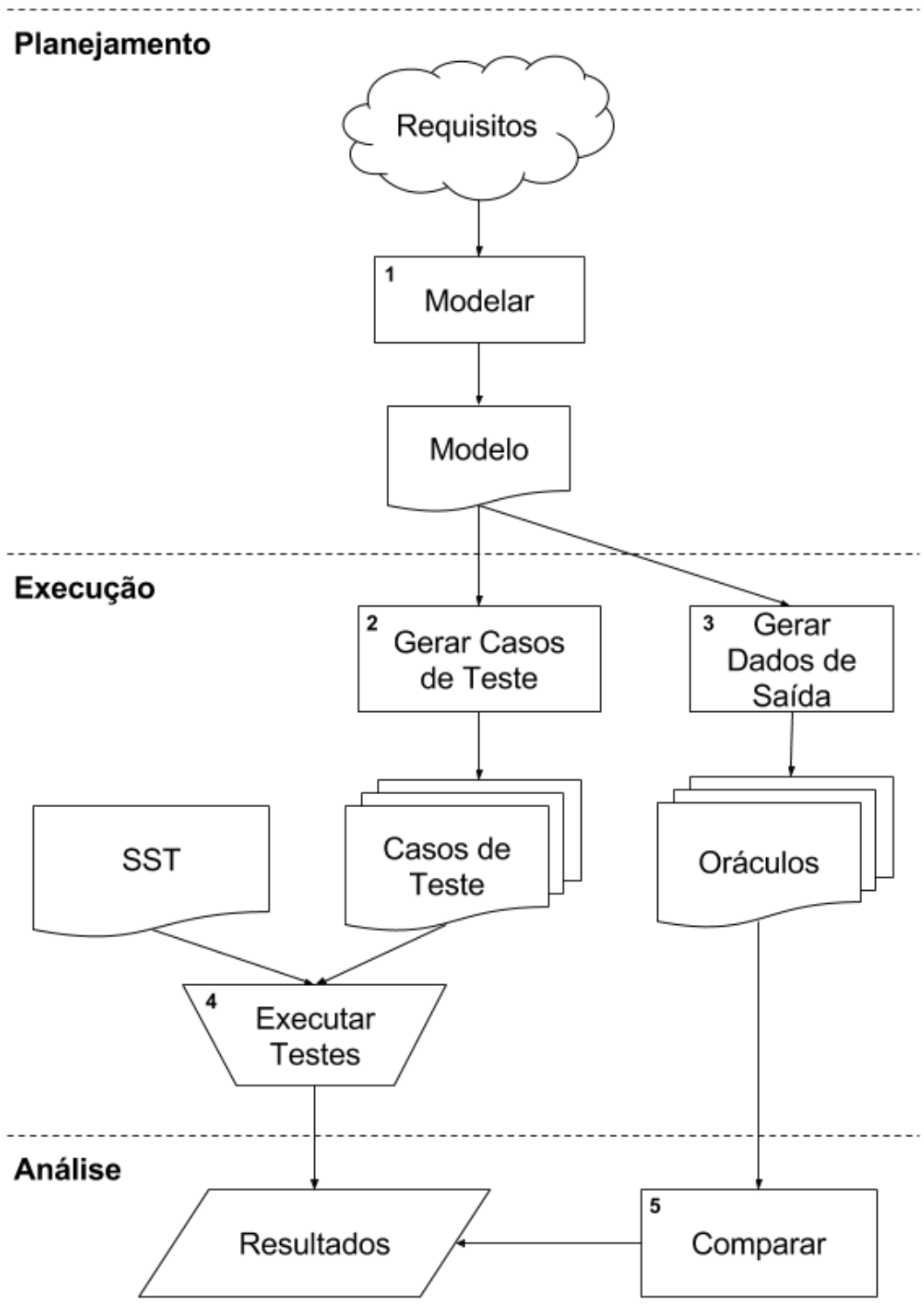


Figura 2.2: Processo do Teste Baseado em Modelo

fase, seria a matriz de rastreabilidade de requisitos, que relaciona o requisito funcional e o caso de teste gerado.

Paralelamente à geração dos casos de teste, é executada a terceira etapa. Aqui são gerados os dados de saída que, na etapa de análise, serão comparados com os resultados dos casos de teste por meio de mecanismos chamados oráculos. O oráculo é um fator

chave na automatização da execução dos casos de teste [4]. Contudo, em uma abordagem de execução manual dos testes, a atividade do oráculo também será igualmente manual.

Já a execução dos testes é a quarta etapa no processo de MBT. Esta fase pode ser executada de forma automática ou manual, de acordo com a abordagem escolhida. Independente da abordagem, esta etapa deve fazer uso de algum suporte ferramental para gerenciar, executar e extrair relatórios dos resultados.

Finalmente, a última etapa se refere à comparação dos resultados dos casos de teste e os valores de saída esperados. Com base na análise dos resultados, serão tomadas as medidas necessárias para corrigir as falhas identificadas pelos testes. Esta etapa também pode servir como um indicador da qualidade dos modelos que geraram os casos de teste.

O emprego de modelos adequados – concisos e precisos – em MBT facilita a geração automática dos casos de teste. Ademais, o modelo também facilita a comunicação entre os envolvidos no desenvolvimento do sistema (*stakeholders*). Um contra-ponto a esta abordagem é que a qualidade dos casos de teste dependerá fortemente da qualidade do modelo especificado. Outro inconveniente observado é a necessidade de conhecimento mínimo da notação exigida para escrever o modelo [4].

2.2 Unified Modeling Language

A UML é uma linguagem padrão para modelar as várias perspectivas de um sistema. Conceitualmente, a UML é uma linguagem para visualizar, especificar, construir e documentar os artefatos de um sistema [22]. Booch et al. afirmam que a modelagem produz a compreensão de um sistema, mas que um só modelo não é suficiente para isso [22]. Sendo assim, a UML provê uma série de modelos que podem ser conectados uns aos outros, para facilitar o entendimento de um sistema nos aspectos estruturais, comportamentais e arquiteturais.

A UML não possui uma semântica bem definida em seus símbolos e por isso não está livre de ambiguidades. No entanto, a linguagem de visualização da UML permite uma comunicação clara. Dessa forma, o emprego de uma padronização na comunicação entre os envolvidos de um projeto evita que cada um utilize um vocabulário próprio para definir e interpretar os modelos.

No que se refere à linguagem de construção, a UML possibilita um mapeamento entre o código e o modelo. Dessa forma, é possível a geração de código a partir desse modelo, e vice-versa em um processo de engenharia reversa. Em relação ao papel de linguagem de documentação, a UML traz diversos benefícios ao desenvolvimento de software, desde as fases iniciais de elicitação de requisitos, até as manutenções após a entrega do produto [23].

Até o presente momento, a última especificação da UML é a versão 2.5[1]. Os modelos

[1] http://omg.org/spec/UML/2.5/

disponibilizados se classificam em estruturais, comportamentais e de interação, conforme a Tabela 2.1.

Modelos (diagramas) da UML 2.5		
Estruturais	Comportamentais	Interação
Classe	Caso de Uso	Sequência
Objeto	Atividade	Comunicação
Componente	Máquina de Estado	Tempo
Estrutura Composta		Interação
Pacote		
Instalação		

Tabela 2.1: Modelos especificados na versão 2.5 da UML

2.3 Requisitos

Os requisitos de *software* descrevem de forma abstrata o comportamento esperado em um sistema durante sua execução [24]. O *Institute of Electrical and Electronics Engineers* (IEEE)[2] define um requisito como uma representação documentada de uma condição ou capacidade que: *(i)* é necessária para um usuário alcançar um objetivo ou resolver um problema, e *(ii)* um sistema deve possuir para satisfazer um acordo, padrão, especificação ou qualquer outro documento rigorosamente firmado [25].

Os requisitos são categorizados em três tipos:

- **Requisitos funcionais**: especificam as funcionalidades que o sistema deve fazer. Aqui deve ser descrito como o sistema deverá responder às entradas do usuário ou como se comportar em determinada situação. Conceitualmente, os requisitos funcionais de um sistema devem possuir duas características importantes: completude e consistência. Um requisito é dito completo, quando descreve todas as funcionalidades que são esperadas pelo usuário. A consistência de um requisito se refere ao quanto a descrição das funcionalidades estão livres de ambiguidade, permitindo dessa forma a correta interpretação e implementação.

- **Requisitos não-funcionais**: descrevem as restrições e conceitos relativos à qualidade dos serviços oferecidos pelo sistema, como por exemplo: performance, facilidade de uso, confiabilidade, robustez, portabilidade, entre outros. Estes requisitos surgem a partir das necessidades dos usuários do sistema, restrições de orçamento, políticas organizacionais, interoperabilidade com outras máquinas ou sistemas e fatores externos como regulamentos de segurança ou legislação de privacidade.

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

- **Requisitos de domínio**: este tipo de requisito contém conceitos específicos do domínio do sistema e como está fora da área de conhecimento da equipe de desenvolvimento, se faz necessário uma orientação de especialistas. Estes requisitos são descritos utilizando uma terminologia específica do domínio e são importantes, pois retratam os elementos fundamentais para que o sistema funcione corretamente.

A especificação de requisitos funcionais através de casos de uso (ver Seção 2.5), tornou-se uma prática amplamente utilizada, não só nas comunidades da orientação a objeto, de onde originou o caso de uso, como também nas demais [26]. Isso se deve ao fato da estrutura do caso de uso permitir a modelagem comportamental do sistema.

Dependendo do nível de detalhamento do requisito, a diferenciação entre estes tipos pode não ser perceptível e isso mostra o quanto há de dependência entre os requisitos [24]. Por exemplo, um requisito funcional que limita o acesso a determinados usuários pode ser interpretado como um requisito não-funcional relativo a segurança. Sommerville ainda atenta para a classificação do nível de abstração dada aos requisitos:

- **Requisitos do usuário**: são descrições mais abstratas dos requisitos com linguagem natural e diagramas para atender as necessidades do usuário do sistema;

- **Requisitos de sistema**: descrevem em detalhes os serviços, funções e restrições que guiarão o desenvolvimento do sistema.

A captura de requisitos envolve uma série de detalhes que devem ser observados minunciosamente para que nenhum detalhe sobre as funcionalidades desejadas em um sistema passem desapercebidas. Um outro cuidado que deve ser tomado é como gerenciar os requisitos e seus relacionamentos quando houver uma mudança durante o ciclo de vida do sistema. Estes cenários exemplificam a complexidade de lidar com os requisitos e esse problema desencadeou esforços para se desenvolver o que conhecemos como engenharia de requisitos.

2.4 Engenharia de Requisitos

A engenharia de requisitos é um processo sistemático que compreende uma série de passos com a finalidade de elicitar, analisar criar e manter a documentação dos requisitos de um projeto de desenvolvimento de *software*. Para isso, a engenharia de requisitos incorpora, após uma análise de viabilidade, as seguintes atividades [27, 28]:

- **Elicitação**: esta atividade compreende a colaboração de clientes e usuário finais do sistema com os engenheiros e analistas de requisitos. Visa compreender o domínio da aplicação a partir de diversas fontes e diferentes técnicas são utilizadas para estabelecer: *(i)* as funcionalidades desejadas como serviço e *(ii)* as restrições que serão impostas no sistema.

- **Análise e negociação**: esta atividade envolve a utilização de técnicas variadas para avaliar e solucionar problemas encontrados nos requisitos levantados na atividade de identificação. Os problemas aqui tratados são relativos a requisitos sem sentidos, contraditórios, inconsistentes, incompletos, vagos ou errados. Também podem ser tratados as interações e dependências entres os requisitos.

- **Especificação e documentação**: o papel desta atividade é transformar os requisitos em estado bruto em um modelo formal. Este modelo pode ser textual (em linguagem natural ou outra notação, como casos de uso ou estórias de usuário), visual (por meio de gráficos ou diagramas da UML) ou matemático (através de métodos formais). O emprego de modelos apropriados para os requisitos facilita a comunicação entre os envolvidos no projeto.

- **Validação**: esta atividade cuida de validar se os requisitos representam corretamente as funcionalidades esperadas pelos usuários. A validação dos requisitos é muito importante, pois evita o retrabalho em problemas encontrados tardiamente, quando o sistema já está em desenvolvimento ou até mesmo em produção. Para isso, esta atividade pode empregar recursos como métodos formais, inspeções, visualizações, entre outros.

- **Manutenção**: esta atividade é ortogonal a todas as demais atividades citadas anteriormente e compreende as ações necessárias para organizar os requisitos de forma consistente e rastreável. Dessa forma, quaisquer mudanças inevitáveis nos requisitos não afetarão a integridade do projeto, garantindo sua implementação.

Sendo assim, o objetivo da engenharia de requisitos é garantir que os requisitos sejam completos, consistentes e relevantes. Ademais, a engenharia de requisitos auxilia a conhecer o que se vai construir antes do início do desenvolvimento do sistema. Com isso, é esperado que custos extras com retrabalho sejam evitados.

A IEEE estabelece uma série de características que uma boa especificação de requisitos deve possuir: corretude, precisão, completude, consistência, ser ordenável por importância ou estabilidade, ser verificável, modificável e rastreável [29]. O documento de especificação de requisitos é o registro formal do que o sistema em desenvolvimento deverá implementar [24].

Este documento deve conter as funcionalidades que os usuários esperam encontrar no sistema e também a descrição detalhada dos requisitos de sistema. A formalização dos requisitos é importante, pois estabelece em detalhes o que será entregue ao cliente em um contrato para desenvolvimento do sistema. No entanto, a metodologia de desenvolvimento ágil[3] alega que esta formalização em documento é desnecessária, em virtude da constante mudança existente nos requisitos.

[3] http://agilemanifesto.org

Na metodologia ágil, os requisitos são apenas coletados e escritos em cartões na forma de estórias de usuário [30, 31]. Esta abordagem faz sentido em projetos de desenvolvimento pequenos e de curta duração, em que o tempo e o orçamento são escassos. Já em projetos de desenvolvimento de sistemas de médio e grande porte, nos quais o número de funcionalidades desejadas e a complexidade são maiores, se faz necessário o emprego de documentação detalhada.

Os requisitos podem sofrer inúmeras mudanças, não só durante a fase de desenvolvimento, mas também depois da entrega do sistema. Por consequência, a engenharia de requisitos tem um processo para tratar o gerenciamento de mudanças de requisitos do sistema. Neste processo, é importante rastrear individualmente os requisitos e gerenciar as dependências existentes entre os requisitos. Dessa forma, é possível avaliar o impacto das mudanças de requisitos para o sistema e decidir se os benefícios das mudanças são justificados pelos custos de desenvolvimento.

2.5 Casos de Uso

O caso de uso faz parte do grupo de modelos comportamentais da UML, e descreve o uso do sistema por um ator para atingir um determinado objetivo [32]. O diagrama de caso de uso fornece uma visão geral dos requisitos do sistema, mas não provê detalhes suficientes para geração de testes [2]. No entanto, a descrição textual de caso de uso contém informações suficientes para permitir a geração dos testes e também guiar os critérios de seleção de testes.

Os casos de uso em formato textual devem ser transformados em algum modelo com nível maior de formalismo, como o *statechart* ou *Trivial Graph Format* (TGF), antes de serem utilizados na geração dos casos de teste [2].

Como modelo para entendimento do comportamento do sistema, o caso de uso descreve os atores, que são os papéis assumidos pelos usuários do sistema em desenvolvimento. Vale ressaltar que os atores podem ser usuários do sistema ou outros sistemas automatizados e que interagem com o sistema em desenvolvimento.

Além dos atores, no caso de uso também são especificados os cenários com as sequências de interações entre atores e sistema [22]. O caso de uso possui um cenário ou fluxo principal de sucesso, na qual o objetivo do caso de uso é alcançado. Também podem existir cenários alternativos, que são sequências de passos diferentes que levam a outras interações dos atores com o sistema. O caso de uso ainda pode conter referências a outros casos de uso, seja incluindo ou estendendo o comportamento básico de outros casos de uso.

Para que o caso de uso seja iniciado, é necessário que critérios sejam atendidos. Estes critérios são descritos no caso de uso em um bloco chamado de **pré-condições**. Os objetivos alcançados pelo caso de uso representam o estado final e são especificados em um outro bloco nomeado de **pós-condições**. O Requisito 1 apresenta a narração de um

requisito que descreve o comportamento de cadastro de um novo usuário para um sistema fictício. A descrição textual do caso de uso, que modela o Requisito 1, é exemplificada na Figura 2.3 e o respectivo diagrama é ilustrado na Figura 2.4

RF01: *Um usuário, com credenciais de acesso e permissão para gerenciar usuários, seleciona a opção cadastrar novo usuário. Deverá ser informado um nome de usuário, contendo apenas letras maiúsculas, minúsculas e sem acentos. O sistema deverá criticar valores inválidos para nome de usuário ou nome de usuário já existente. O sistema exige o cadastro de uma senha contendo letras maiúsculas/minúsculas e sem acentos. Poderá ainda conter números. Qualquer outra combinação deve ser criticada. O sistema ainda solicitará o endereço do usuário com os campos: logradouro, número, bairro, cidade, estado e cep. Caso o usuário informe o cep, o sistema poderá completar automaticamente os campos: logradouro, bairro, cidade e estado.*

Requisito 1: Cadastro de usuários

2.6 Considerações Finais

Apresentamos neste capítulo os conceitos teóricos básicos necessários à compreensão do trabalho proposto. Foram apontados os conceitos fundamentais referentes a teste de *software* e seus tipos de teste, a partir de 3 perspectivas: características, tipo de informação e escala do sistema. Também foi elucidada a abordagem de teste baseado em modelo juntamente com suas vantagens e desvantagens.

No tocante à engenharia de requisitos, foram expostos os conceitos da linguagem de modelagem unificada e os tipos de modelos providos. Em seguida, foi explanado o conceito de requisito de *software*, suas categorias principais e os níveis de abstração. Além disso, foram apresentados a engenharia de requisitos, sua importância para a engenharia de *software* e suas atividades para como lidar com os requisitos. Por fim, foram apresentados os conceitos do modelo de caso de uso e seus blocos essenciais.

Componente do Caso de Uso	Descrição
Identificador	UC01
Nome do Caso de Uso	Cadastrar Novo Usuário
Objetivo	
Escopo	
Ator Primário	Operador
Pré-condições	O operador deve ter credencial válida. O operador deve estar logado. O operador deve ter permissão para cadastrar novo usuário do sistema.
Pós-condições	Novo usuário cadastrado no sistema.
	1) Operador seleciona opção "cadastrar novo usuário"
	2) Sistema apresenta um formulário com os campos de "nome de usuário", "senha", "cep", "logradouro", "número", "bairro", "cidade", "estado".
	3) Operador informa o campo "nome de usuário". (E1, E2)
	4) Operador informa campo "senha". (E3)
	5) Operador informa os campos de "cep", "logradouro", "número", "bairro", "cidade" e "estado". (A1)
	6) Operador seleciona a opção "gravar". (E4)
Fluxos Alternativos	**A1 - Endereço a partir do cep**
	1) Operador informa o campo CEP.
	2) Sistema pesquisa o CEP na base de dados e se encontrar, são preenchidos automaticamente os campos logradouro, bairro, cidade e estado.
	3) Operador preenche o campo número.
	4) Volta ao passo 6 do fluxo básico.
Fluxos de Exceção	**E1 - Nome de usuário inválido**
	1) Operador informa um nome de usuário que contem caracteres diferentes do permitido (letras maiúsculas, minúsculas, sem acento e números).
	2) Sistema informa que o nome de usuário informado já existe no sistema
	3) Volta ao passo 3 do fluxo principal.
	E2 - Nome de usuário existente no sistema
	1) Operador informa um nome de usuário válido, mas já existente no sistema
	2) Sistema informa que o nome de usuário informado já existe no sistema
	3) Volta ao passo 3 do fluxo principal.
	E3 - Senha inválida
	1) Operador informa um nome de usuário que contem caracteres diferentes do permitido (letras maiúsculas, minúsculas, sem acento e números).
	2) Sistema informa que o nome de usuário informado já existe no sistema
	3) Volta ao passo 4 do fluxo principal.
	E3 - Campos não informados
	1) Usuário não informa algum dos campos do formulário.
	2) Sistema informa que todos os campos são obrigatórios.
	3) Volta ao passo 2 do fluxo principal.

Figura 2.3: Descrição textual de um caso de uso

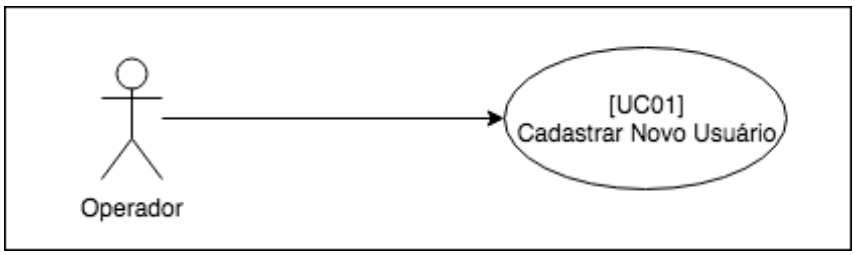


Figura 2.4: Descrição gráfica de um caso de uso

Capítulo 3

Claret: a linguagem, a ferramenta e o processo

Neste capítulo apresentaremos em detalhes a ferramenta CLARET. Inicialmente mostraremos sua linguagem (Seção 3.1), em seguida sua arquitetura (Seção 3.2) e, por fim, o processo (Seção 3.3).

3.1 A Linguagem

CLARET possibilita especificações de requisitos em formato similar ao de casos de uso, por meio de uma notação própria. Estas especificações atuam como artefatos centrais para a geração automática de modelos intermediários para casos de teste manuais. Além disso, a ferramenta viabiliza a geração automática de documentos formatados para a engenharia de requisitos. A finalidade básica da notação de CLARET é, portanto, fornecer uma forma prática e ágil de utilizar linguagem natural, em uma estrutura semi-controlada, para viabilizar atividades dos processos de engenharia de requisitos e MBT.

3.1.1 Gramática

A sintaxe da notação proposta em CLARET se baseia nos principais elementos da escrita tradicional de casos de uso: *usecase*, *actor*, *preCondition*, *postCondition*, *basic*, *alternative*, *exception*. Assim, a curva de aprendizagem é minimizada para quem já conhece a definição de casos de uso da UML.

Devido à similaridade com a definição da notação JSON, se espera que a curva de aprendizagem seja baixa. Dessa forma, é esperado que as dificuldades para compreender ou utilizar CLARET sejam minimizadas, principalmente para os desenvolvedores que já tiveram algum conhecimento prévio com a notação JSON.

[0] http://json.org/

Para a especificação de CLARET, empregamos a EBNF [15], uma família de notações meta-sintaxe que é bastante utilizada para a descrição formal de linguagens de programação. A Listagem 3.1 apresenta a definição da gramática de CLARET.

Listagem 3.1: Regras de produção em EBNF.

```
1  systemName ::= 'systemName' StringLiteral usecase*
2  usecase ::= 'usecase' StringLiteral '{' elem '}'
3  elem ::= version? actor+ pre basic (exception | alternative)* post
4  version ::= 'version' StringLiteral 'type' StringLiteral 'user'
       StringLiteral 'date' StringLiteral
5  actor ::= 'actor' ident StringLiteral
6  pre ::= 'preCondition' StringLiteral (',' StringLiteral)*
7  basic ::= 'basic' steps
8  post ::= 'postCondition' StringLiteral (',' StringLiteral)*
9  alternative ::= 'alternative' [0-9]+ StringLiteral steps
10 exception ::= 'exception' [0-9]+ StringLiteral steps
11 steps ::= '{' (action | response)* '}'
12 action ::= 'step' [0-9]+ ident StringLiteral (af | bs)*
13 response ::= 'step' [0-9]+ 'system' StringLiteral (ef | bs)*
14 af ::= 'af' '[' [0-9]+ (',' [0-9]+)* ']'
15 ef ::= 'ef' '[' [0-9]+ (',' [0-9]+)* ']'
16 bs ::= 'bs' [0-9]+
```

A Linha 1 da Listagem 3.1 mostra a regra de produção **systemName**. Essa produção é o ponto de entrada de um caso de uso especificado em CLARET e inclui todos os casos de usos do sistema (ver Figura 3.1). A regra *StringLiteral* – que aparece em vários pontos da gramática – está implícita para fins de simplificação, e serve para identificar um *token* representado por uma cadeia de caracteres delimitada por aspas. Desse modo, o conteúdo do *token* é livre e pode ser descrito através de linguagem natural.

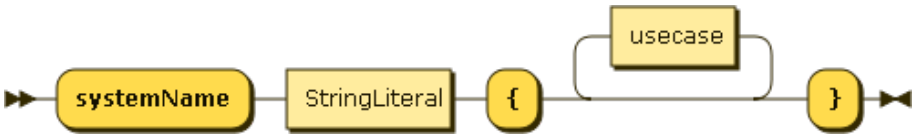


Figura 3.1: Produção **systemName**.

A regra **usecase** (Linha 2) representa um caso de uso (ver Figura 3.2) que é definido por declaração opcional de versão (Linha 4), pelo menos um ator (Linha 5), pre-condições (Linha 6), um fluxo básico (Linha 7), fluxos opcionais alternativos (Linha 9), fluxos opcionais de exceção (Linha 10), e pós-condições (Linha 8).

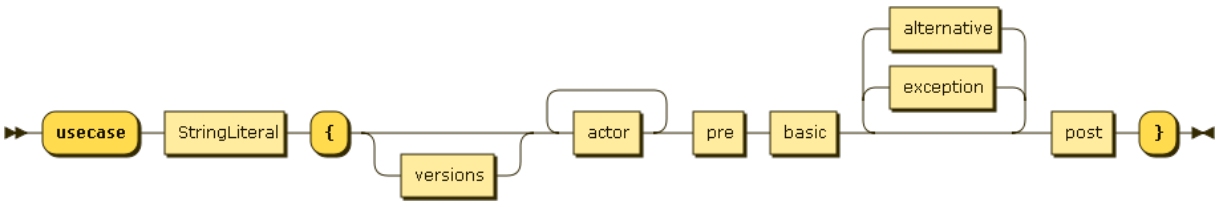


Figura 3.2: Produção **usecase**.

O elemento **version** é uma informação opcional relacionada ao histórico do caso de uso, que contém um número de versão, uma descrição, quem editou o caso de uso e a data da modificação (ver Figura 3.3). Essas informações sobre a versão são cruciais para controlar a evolução do sistema e seus artefatos.

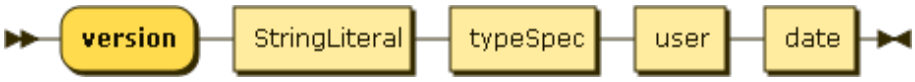


Figura 3.3: Produção **version**.

Todos os atores externos relacionados ao caso de uso são especificados por meio do elemento **actor** (Linha 5). Esta regra (ver Figura 3.4) permite a definição de um identificador ou apelido para cada ator ser referenciado nos passos dos casos de uso, de forma fácil e consistente.

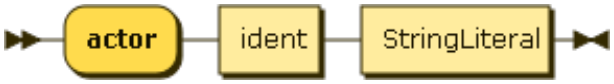


Figura 3.4: Produção **actor**.

A regra **pre** (Linha 6) é usada para definir expressões que devem ser satisfeitas antes da execução do fluxo básico do caso de uso (ver Figura 3.5). Esta palavra chave é obrigatória mesmo quando nenhuma pré-condição exista, que neste caso devem ser utilizadas aspas vazias. As mesmas considerações são aplicadas à regra **post** (Linha 8). A produção pode ser visualizada na Figura 3.6.

Figura 3.5: Produção **preCondition**.

Figura 3.6: Produção **postCondition**.

O fluxo básico é o cenário principal do caso de uso e é especificado usando a regra **basic** (Linha 7) (ver Figura 3.7).

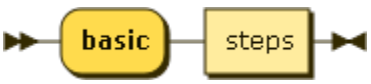


Figura 3.7: Produção **basic**.

Em CLARET, a regra **steps** é composta por uma lista de passos (Linha 11). Cada passo, por sua vez, é formado por: *(i)* um número de sequência, *(ii)* a palavra chave **system** ou identificador do ator, *(iii)* uma cadeia de caracteres entre aspas descrevendo a ação em linguagem natural, e *(iv)* os desvios, caso existam, para outros cenários (*af*–fluxo

alternativo; *ef*–fluxo de exceção; *bs*–passo básico) (Linhas 12 e 13). Cada passo significa uma interação entre um ator e o sistema (Linha 12, ver Figura 3.8), ou a resposta do sistema (Linha 13, ver Figura 3.9).

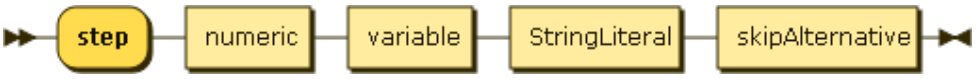


Figura 3.8: Produção **action**.

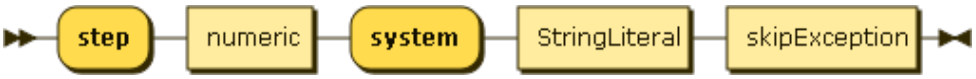


Figura 3.9: Produção **response**.

Depois da execução do fluxo básico, pós-condições podem ser especificadas através da regra **post** (linha 8, ver Figura 3.6). Finalmente, as linhas de 14 a 16 podem ser usadas para especificar todos os desvios que partem do fluxo básico para fluxos alternativos (ver Figura 3.10) ou de exceção (ver Figura 3.11), e de retorno ao fluxo básico (ver Figura 3.12).

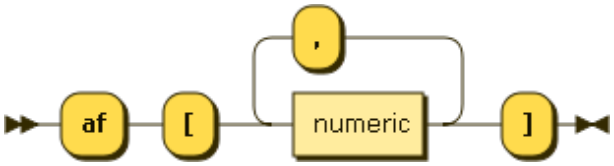


Figura 3.10: Produção **af**.

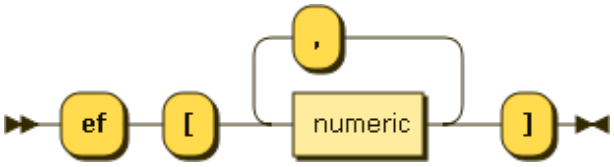


Figura 3.11: Produção **ef**.

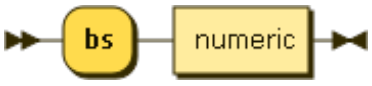


Figura 3.12: Produção **bs**.

Tendo em vista a consistência dos casos de testes, a semântica estática de CLARET exige que cada passo do ator seja seguido por um passo do sistema. A razão disto é que os casos de testes para execução manual são usualmente organizados como sequências de passos de ação (ou estímulo) e seus respectivos resultados esperados (ver Figura 3.13).

Consequentemente, o primeiro passo do fluxo alternativo deve ser um passo do ator (estímulo) que substitui o passo do cenário do qual ele se originou. Já o primeiro passo do fluxo de exceção deve ser um passo do sistema. Além disso, o passo de retorno deve ser de um tipo diferente (sistema ou ator) do passo que ele se origina. Caso o fluxo de exceção não tenha retorno, o último passo deve ser um passo do sistema.

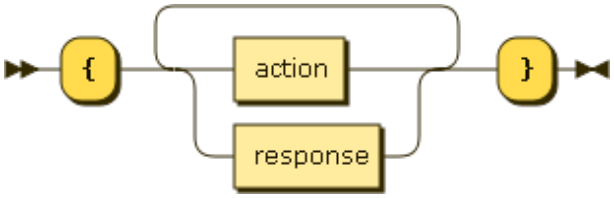


Figura 3.13: Produção **steps**.

3.1.2 Exemplo

A Listagem 3.2 ilustra o uso da notação de CLARET na especificação de um caso de uso para o Requisito 1, apresentado na Seção 2.5 do Capítulo 2. Este requisito funcional de usuário detalha o comportamento esperado para acesso a um sistema de *email*:

> *"Para que um usuário acesse sua caixa de entrada de mensagens, o usuário deve possuir cadastro prévio no sistema e fornecer um nome de usuário e senha válidos. Para facilitar o acesso, o sistema pode sugerir nomes de usuários que foram utilizados anteriormente. No caso de uma combinação incorreta de nome de usuário e senha, o sistema deve apresentar uma mensagem de erro e fornecer uma nova tentativa. Caso o nome de usuário não seja encontrado, o sistema deve informar que o usuário não está cadastrado."*

Listagem 3.2: Caso de uso em CLARET.

```
1  systemName "Mensageiro"
2  usecase "Acesso Sistema" {
3    version "1.0" type "Criação" user "Dalton" date "01/05/2017"
4    actor usuario "Usuário"
5    preCondition "Existir uma conexão de rede.", "Usuário cadastrado no sistema"
6    basic {
7      step 1 usuario "chama a tela de acesso"
8      step 2 system "apresenta um formulário com os campos de nome de usuário, senha,
        e um botão enviar"
9      step 3 usuario "preenche todos os campo e seleciona o botão enviar" af[1]
10     step 4 system "informa uma mensagem de sucesso" ef[1,2]
11   }
12   alternative 1 "Nome de usuário é sugerido" {
13     step 1 usuario "seleciona um nome sugerido, informa uma senha e seleciona a opç
       ão enviar" bs 4
14   }
15   exception 1 "Usuário não existe no sistema" {
16     step 1 system "alerta que usuário não está cadastrado no sistema" bs 3
17   }
18   exception 2 "Combinação de nome de usuário e senha incorretos" {
19     step 1 system "alerta que a combinação de nome de usuário e senha estão
       incorretos" bs 3
20   }
21   postCondition "Acesso ao sistema com sucesso"
22 }
```

Inicialmente, definimos o nome do sistema (Linha 1) e o nome do caso de uso (Linha 2). Depois inserimos algumas informações opcionais para o versionamento do caso de uso (Linha 3). Em seguida, declaramos o ator (Linha 4) e a pré-condição do caso de uso (Linha 5).

Logo após é definido o fluxo básico (Linha 5). Os passos do fluxo básico (Linhas 7 a 10) são alternados entre estímulos do ator e respostas do sistema. Cada passo é declarado com um identificador sequencial, o sujeito (ator ou sistema), a ação descrita em linguagem natural e, opcionalmente, um desvio para fluxos alternativos (Linha 9) ou de exceção (Linha 10).

É relevante frisar que os desvios alternativos partem apenas de passos do tipo estímulo, enquanto que desvios de exceção partem de passos do tipo resposta. Os fluxos alternativos e de exceção são declarados com um identificador sequencial e uma descrição textual (Linhas 12, 15 e 18). Os passos dos fluxos alternativos e de exceção seguem a mesma regra dos passos do fluxo básico. O diferencial é que pode existir um passo de retorno ao fluxo básico (Linhas 13, 16 e 19). Por fim, é declarada a pós-condição (Linha 21) do caso de uso.

3.2 A Ferramenta

Nesta seção apresentamos os detalhes do suporte ferramental de Claret: a organização da arquitetura interna (Subseção 3.2.1), como executar a ferramenta (Subseção 3.2.2) e o *plugin* para auxílio nas especificações no editor Visual Studio Code (Subseção 3.2.3).

3.2.1 Arquitetura

O conjunto arquitetural da ferramenta Claret é composto pelos seguintes módulos: analisador léxico, analisador sintático, analisador semântico, gerador de arquivos *Trivial Graph Format* (TGF) e o gerador de arquivos *OpenDocument Text* (ODT). A Figura 3.14 ilustra a referida arquitetura Claret.

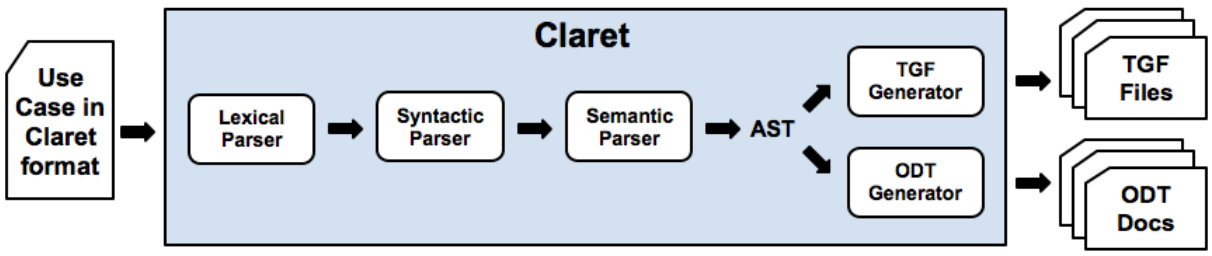


Figura 3.14: Arquitetura da ferramenta.

Analisadores

CLARET foi implementado na linguagem Scala[1] e utiliza a biblioteca *Parser Combinator* para auxiliar a criação dos analisadores léxico e sintático. Desenvolvemos também o analisador semântico que tem como função localizar incoerências na árvore sintática abstrata (AST). O analisador semântico possibilita, por exemplo, a checagem de erros como atores referenciados, mas não declarados para o caso de uso.

Gerador de TGF

O gerador de TGF utiliza o AST para criar um grafo com os todos os caminhos possíveis para cada caso de uso da especificação. Em seguida, é criado um arquivo TGF contendo a representação textual do grafo. O TGF é um formato que contém uma lista com definições de nós com seus respectivos rótulos de identificação (Linhas 1 a 7 da Listagem 3.3) e uma lista de transições. Cada transição contem um par de nós válidos que foram declarados na lista de nós e um rótulo para a respectiva transição (Linhas 9 a 17 da Listagem 3.3).

Listagem 3.3: Arquivo TGF.

```
1  0 0
2  1 1
3  2 2
4  3 3
5  4 4
6  5 5
7  6 6
8  #
9  0 1 [c] Existir uma conexão de rede, Usuário cadastrado no sistema
10 1 2 [s] Usuário chama a tela de acesso
11 2 3 [e] Mensageiro apresenta um formulário com os campos de nome de usuário, senha,
       e um botão enviar
12 3 4 [s] Usuário preenche todos os campo e seleciona o botão enviar
13 3 4 [s] (Nome de usuário é sugerido) Usuário seleciona um nome sugerido, informa
     uma senha e seleciona a opção enviar
14 4 5 [e] Mensageiro informa uma mensagem de sucesso
15 4 3 [e] (Usuário não existe no sistema) Mensageiro alerta que usuário não está
     cadastrado no sistema
16 4 3 [e] (Combinação de nome de usuário e senha incorretos) Mensageiro alerta que a
     combinação de nome de usuário e senha estão incorretos
17 5 6 [c] Acesso ao sistema com sucesso
```

No contexto de nossa ferramenta, os rótulos das transições são as descrições das condições ou os passos de cada cenário (combinação das descrições de "ator" ou "sistema", e a respectiva "ação" de estímulo ou resposta). Este TGF gerado tem a finalidade de validação visual por meio de ferramenta apropriada para grafos. Para fins de geração de testes, CLARET gera uma versão modificada do TGF com anotações que identificam as transições, para auxiliar a posterior geração dos casos de testes [33] (ver Listagem 3.4).

[1] http://www.scala-lang.org

Listagem 3.4: Arquivo TGF anotado.

```
1  0 0
2  1 1
3  2 2
4  3 3
5  4 4
6  5 5
7  6 6
8  7 7
9  8 8
10 9 9
11 10 10
12 11 11
13 12 12
14 13 13
15 14 14
16 15 15
17 #
18 0 7 conditions
19 7 1 Existir uma conexão de rede.,Usuário cadastrado no sistema
20 1 8 steps
21 8 2 Usuário chama a tela de acesso
22 2 9 expected_results
23 9 3 Mensageiro apresenta um formulário com os campos de nome de usuário, senha, e
       um botão enviar
24 3 10 steps
25 10 4 Usuário preenche todos os campo e seleciona o botão enviar
26 3 11 steps
27 11 4 (Nome de usuário é sugerido) Usuário seleciona um nome sugerido, informa uma
       senha e seleciona a opção enviar
28 4 12 expected_results
29 12 5 Mensageiro informa uma mensagem de sucesso
30 4 13 expected_results
31 13 3 (Usuário não existe no sistema) Mensageiro alerta que usuário não está
       cadastrado no sistema
32 4 14 expected_results
33 14 3 (Combinação de nome de usuário e senha incorretos) Mensageiro alerta que a
       combinação de nome de usuário e senha estão incorretos
34 5 15 conditions
35 15 6 Acesso ao sistema com sucesso
```

No TGF anotado são inseridas transições para identificar as transições de condições (Linhas 18 e 34), passos do ator (Linhas 20, 24, 26) e passos do sistema (Linhas 22, 28, 30 e 32).

Gerador de ODT

O Gerador de ODT é o módulo responsável pela geração automática de documentos de casos de uso formatados em arquivos que atende a especificação do Documento de Formato Aberto, do inglês *Open Document Format* (ODF)[2]. Esta especificação é um padrão adotado por diversas organizações como uma versão alternativa, de código aberto, aos

[2] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office

formatos proprietários e sujeitos a licença de uso restrito ou onerosas. O ODF estabelece uma série de tipos de arquivos, cada um com uma finalidade específica: **ods** para planilhas de cálculo, **odp** para apresentações, **odb** para banco de dados, **odg** para desenhos vetoriais, **odf** para equações e **odt** para documentos de texto. Este último tipo é o formato adotado em CLARET para a geração dos documentos.

Por fim, com o intuito de fornecer identidade e flexibilidade aos relatórios gerados, CLARET emprega um sistema de *template* baseado no *Mustache*[3] que possibilita a personalização destes relatórios pelos usuários. O arquivo de *template* é um arquivo chamado "*template.odt*"e está localizado junto ao binário do CLARET. O mecanismo do *Mustache* é livre de lógica e sua função é apenas substituir as propriedades do caso de uso que estão cercados por um par de abre-chaves e outro par de fecha-chaves:

$$\boxed{\{\{< property >\}\}}$$

Apesar da inexistência de lógica no *template* do *Mustache*, é possível iterar os valores de uma propriedade do tipo lista por meio da seguinte construção:

$$\boxed{\{\{< \#list >\}\}\{\{< property_0 >\}\}...\{\{< property_{n-1} >\}\}\{\{< /list >\}\}}$$

A Tabela 3.1 exibe as propriedades e as respectivas descrições de seus conteúdos, que são disponibilizados por CLARET a partir da árvore sintática abstrata de cada caso de uso.

Variável	Descrição
usecase	Nome do caso de uso
system	Nome do sistema
actors	Lista de atores
pre	Lista de pré-condições
basicsteps	Lista de passos do fluxo básico
exceptions	Lista de fluxos de exceção
alternatives	Lista de fluxos alternativos
post	Lista de pós-condições

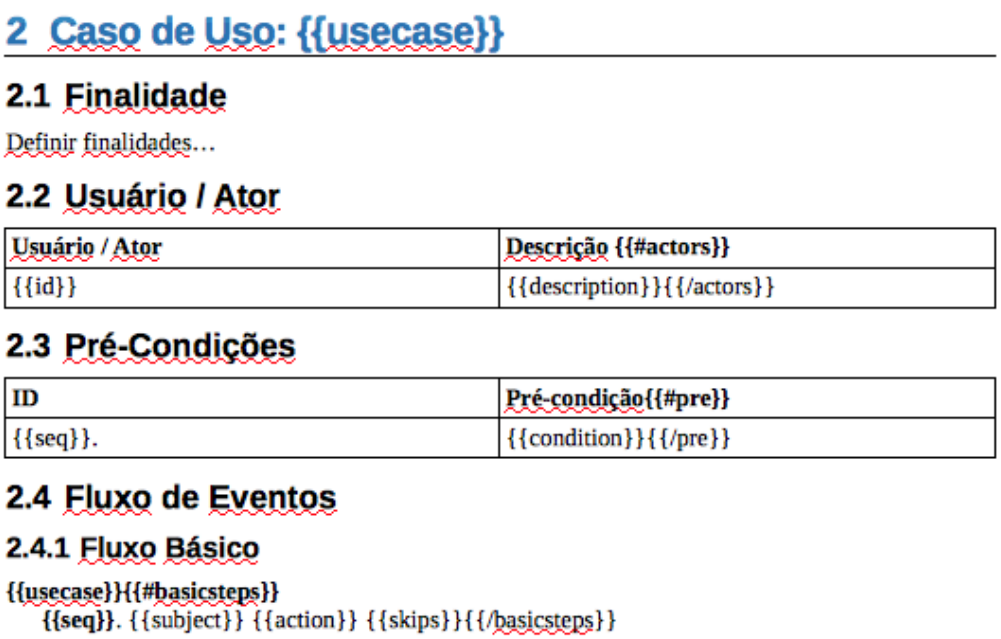
Tabela 3.1: Propriedades para personalização do *template*

A Figura 3.15 exemplifica um trecho do arquivo de *template*.

3.2.2 Execução

CLARET foi desenvolvida para ser uma ferramenta para linha de comando, sem interface gráfica. A razão disso foi obter uma independência visual e permitir a sua execução de três formas: *(i)* diretamente no terminal, *(ii)* através de *scripts* ou *(iii)* por meio de atalhos em editores de texto.

[3] `https://mustache.github.io`



2 Caso de Uso: {{usecase}}

2.1 Finalidade

Definir finalidades...

2.2 Usuário / Ator

Usuário / Ator	Descrição {{#actors}}
{{id}}	{{description}}{{/actors}}

2.3 Pré-Condições

ID	Pré-condição{{#pre}}
{{seq}}.	{{condition}}{{/pre}}

2.4 Fluxo de Eventos

2.4.1 Fluxo Básico

{{usecase}}{{#basicsteps}}
{{seq}}. {{subject}} {{action}} {{skips}}{{/basicsteps}}

Figura 3.15: Arquivo de *template* com o *Mustache*.

A execução via linha de comando possui uma sintaxe simples que compreende apenas um argumento e duas opções (ver Listagem 3.5). A execução com o argumento *filename* serve para compilar o arquivo desejado. Já a execução sem argumento e com a opção "-a" compilará todos os arquivos com extensão *claret* no diretório corrente. A opção "-h" imprime a ajuda do comando no terminal.

Listagem 3.5: Argumento e opções para linha de comando.

```
1 Claret v2.6.1
2 Usage: claret <filename> | -a | -h
3   -a    compiles all claret files in current directory
4   -h    prints this usage text
```

Alguns editores permitem a configuração de teclas de atalhos para comando de *build* ou compilação, como por exemplo, o *Sublime* e o *Visual Studio Code*[4]. Dessa forma, CLARET pode ser executado sem a necessidade de utilizar a linha de comando do terminal, compilando o arquivo que está aberto no editor.

3.2.3 *Plugin* para Visual Studio Code

Para tornar mais produtiva a especificação com CLARET, desenvolvemos um *plugin* para o editor *Visual Studio Code*. A escolha desse editor se deu pela produtividade dos recursos nativos (por exemplo, o suporte ao versionamento com *Git*), pelas constantes atualizações/correções e também pelas facilidades oferecidas para estender suas funcionalidades mediante *plugins*. O *Visual Studio Code* oferece a hospedagem do *plugin* desenvolvido

[4] https://code.visualstudio.com

em uma loja específica, que pode ser acessada dentro do próprio editor.

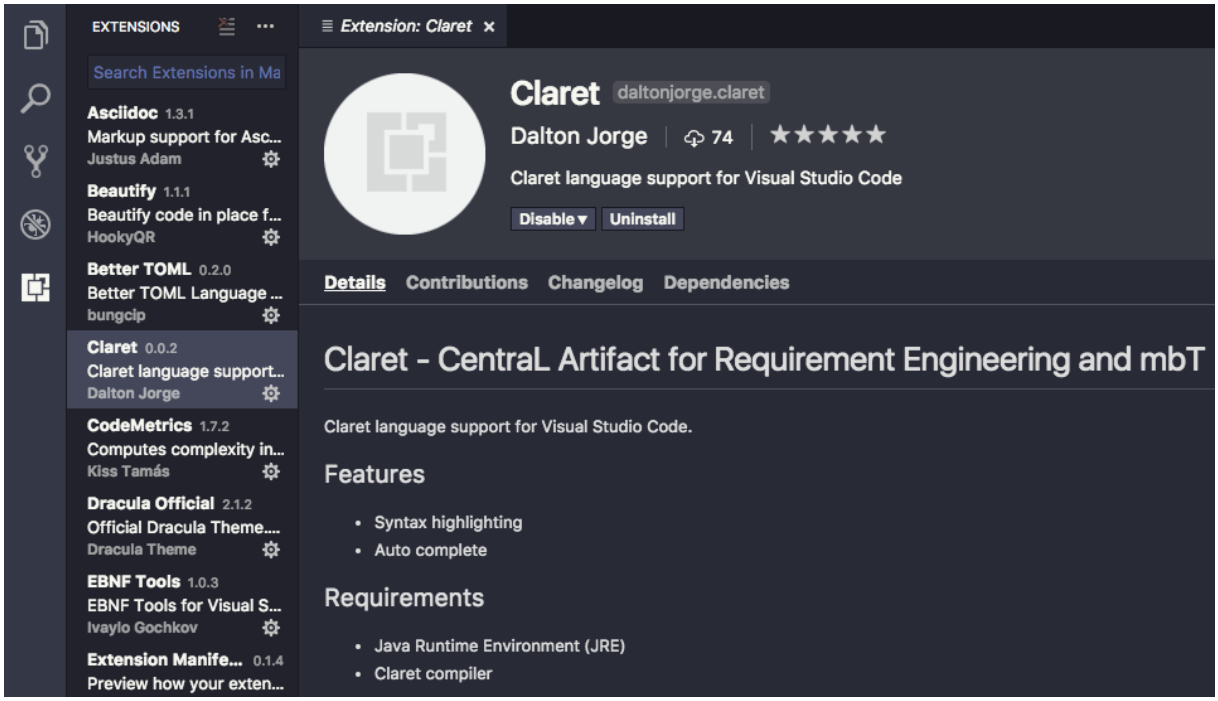


Figura 3.16: Plugin do CLARET para Visual Studio Code.

Além de diferenciar a sintaxe de CLARET mediante cores (*syntax highlight*), o *plugin* permite executar o compilação da especificação através de tecla de atalho e também localizar o erro na especificação CLARET. A Figura 3.17 mostra a captura de um erro no *Visual Studio Code*.

3.3 O Processo

A Figura 3.18 ilustra um panorama geral do processo que usa as especificações com CLARET e seu suporte ferramental. Para fins de simplificação, o diagrama foca nas atividades de MBT (quadro sombreado), enquanto que as atividades de engenharia de requisitos se concentram na criação/manutenção dos casos de uso e geração de documentos. Como pode ser visto, foram incluídas no processo as atividades de Gerenciamento de Mudanças e Gerenciamento de Requisitos, que são pontos chaves para a efetividade tanto da engenharia de requisitos ágil e teste baseado em modelos [34, 35].

O ponto inicial do processo é a lista de requisitos que é o alicerce para a escrita de casos de uso em CLARET. Após a criação e atualização dos casos de uso, cada versão da especificação é compilada pelo ferramental de CLARET para gerar: *(i)* modelos de teste em formato TGF e *(ii)* documentos de engenharia de requisitos para a comunicação entre as partes envolvidas (*stakeholders*). A Figura 3.19 mostra um documento de caso de uso gerado por CLARET.

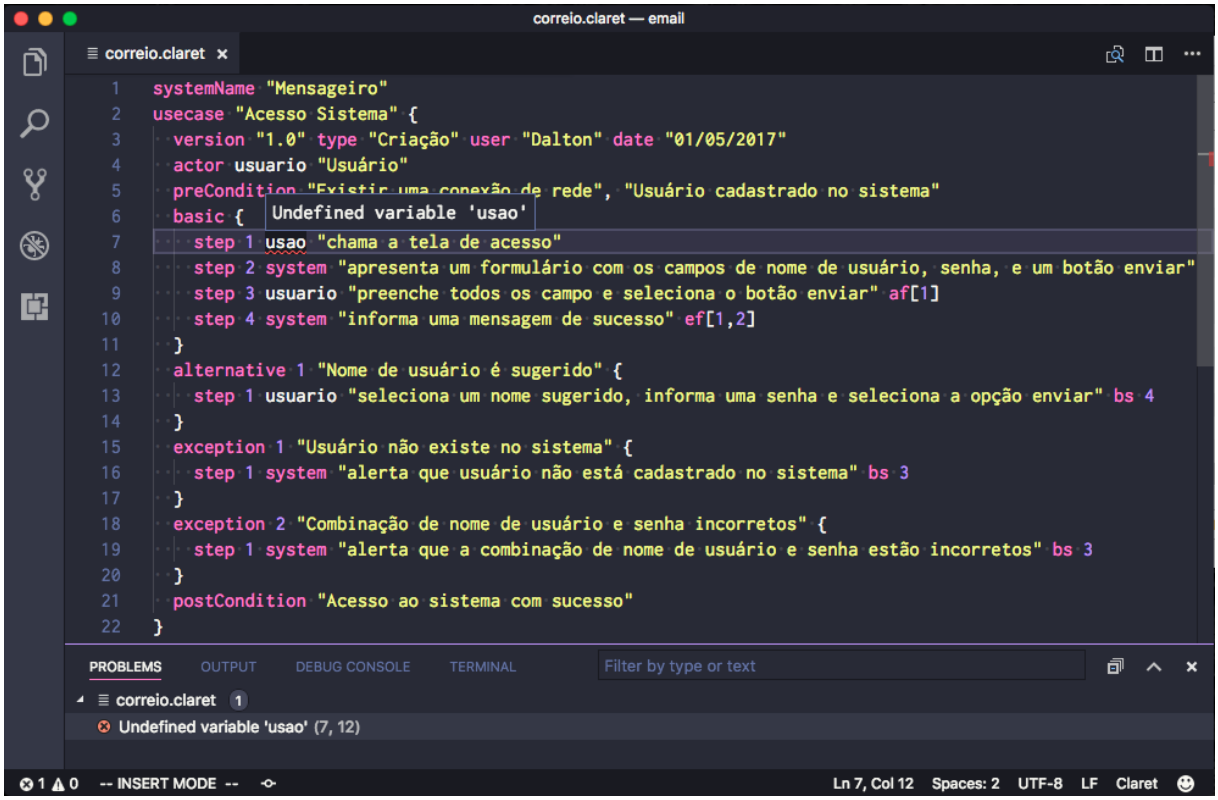


Figura 3.17: Captura de erro no Visual Studio Code.

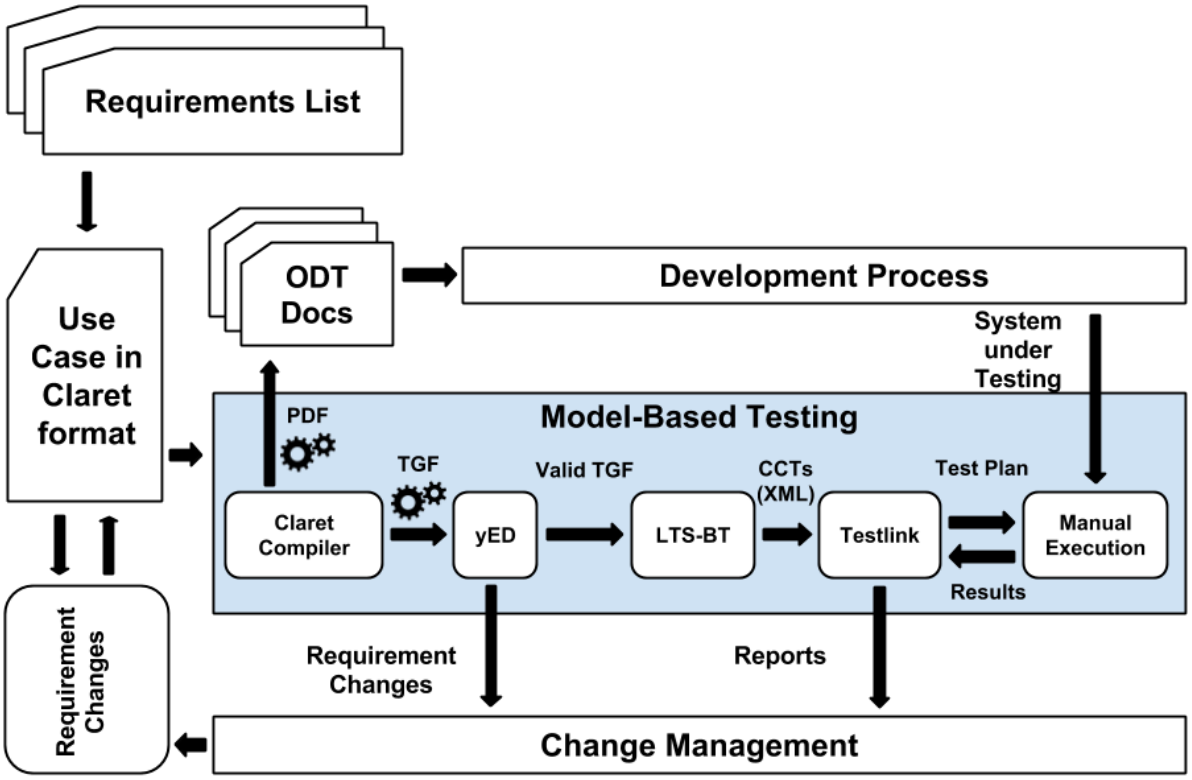
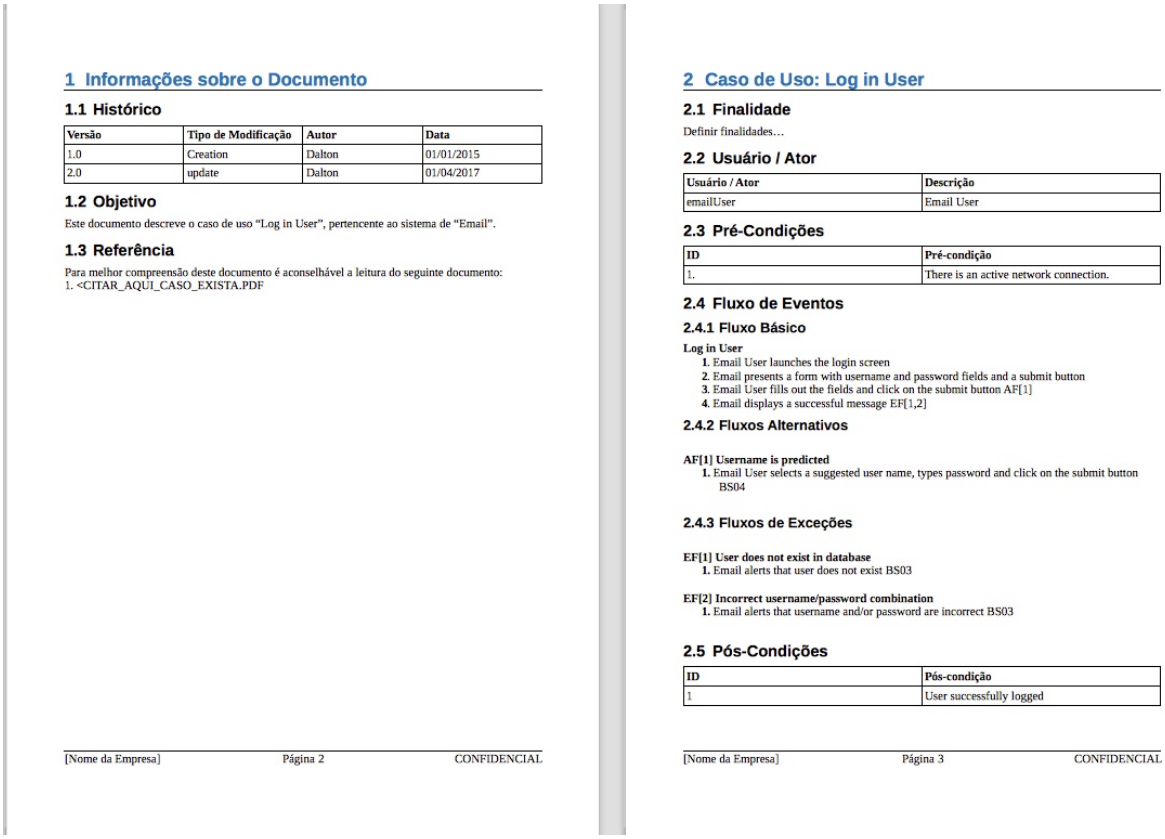


Figura 3.18: Visão geral do processo.

Os modelos TGF podem ainda ser avaliados visualmente através da ferramenta *yEd*[5]

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



1 Informações sobre o Documento

1.1 Histórico

Versão	Tipo de Modificação	Autor	Data
1.0	Creation	Dalton	01/01/2015
2.0	update	Dalton	01/04/2017

1.2 Objetivo

Este documento descreve o caso de uso "Log in User", pertencente ao sistema de "Email".

1.3 Referência

Para melhor compreensão deste documento é aconselhável a leitura do seguinte documento:
1. <CITAR_AQUI_CASO_EXISTA.PDF

[Nome da Empresa] Página 2 CONFIDENCIAL

2 Caso de Uso: Log in User

2.1 Finalidade

Definir finalidades...

2.2 Usuário / Ator

Usuário / Ator	Descrição
emailUser	Email User

2.3 Pré-Condições

ID	Pré-condição
1.	There is an active network connection.

2.4 Fluxo de Eventos

2.4.1 Fluxo Básico

Log in User
1. Email User launches the login screen
2. Email presents a form with username and password fields and a submit button
3. Email User fills out the fields and click on the submit button AF[1]
4. Email displays a successful message EF[1,2]

2.4.2 Fluxos Alternativos

AF[1] Username is predicted
1. Email User selects a suggested user name, types password and click on the submit button BS04

2.4.3 Fluxos de Exceções

EF[1] User does not exist in database
1. Email alerts that user does not exist BS03

EF[2] Incorrect username/password combination
1. Email alerts that username and/or password are incorrect BS03

2.5 Pós-Condições

ID	Pós-condição
1	User successfully logged

[Nome da Empresa] Página 3 CONFIDENCIAL

Figura 3.19: Documentos gerados por CLARET.

(ver Figura 3.20). Esta verificação permite detectar anomalias no grafo, tais como nós órfãos (sem transições), o que pode levar a inconsistências na geração dos casos de teste. Caso existam incoerências no grafo, as devidas correções devem ser efetuadas na especificação em CLARET e novamente compiladas. Após esta fase de checagem visual e consequentemente a conformidade do modelo TGF, o próximo passo é a geração dos casos de teste pela ferramenta *Labeled Transition System Based-Testing* (LTS-BT) de acordo com os critérios de cobertura e estratégias para redução da suíte de testes [36]. Para isso, devem ser utilizados os artefatos TGF anotados – explicados na Seção 3.2.1 – durante a compilação da especificação com CLARET.

Os casos de teste gerados no formato XML são importados no *Testlink*[6] para a etapa de execução manual dos testes (ver Figura 3.21). Por fim, o *Testlink* fornece relatórios com os resultados dos planos de testes que podem levar a novas mudanças nos requisitos ou correções no código-fonte do sistema.

3.4 Considerações Finais

Este capítulo apresentou as particularidades da ferramenta CLARET. Foram explicadas as regras de produção para a gramática da linguagem, tendo como base as construções de

[6] http://www.testlink.org/

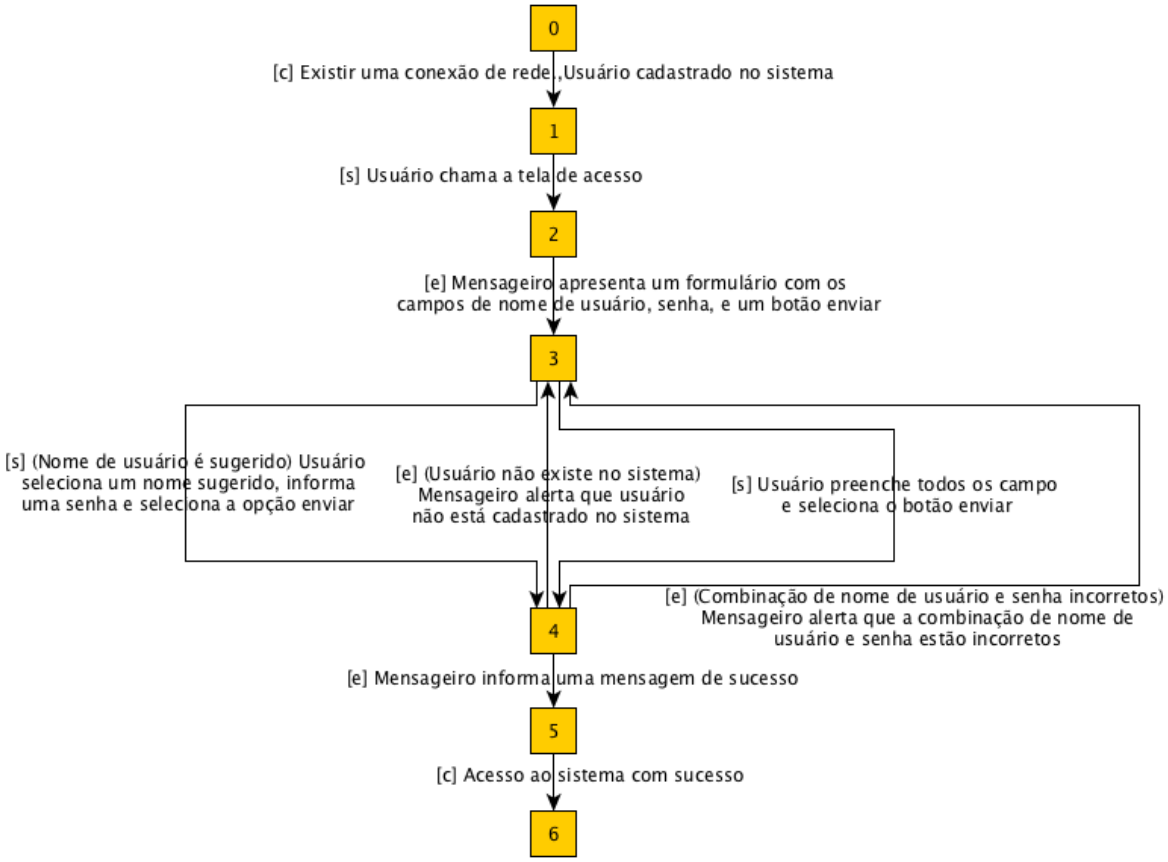


Figura 3.20: Exemplo de TGF gerado por CLARET.

um caso de uso textual. Também foi exemplificada uma especificação com a linguagem proposta a partir de um requisito funcional fictício. Concluindo, detalhamos os módulos internos da ferramenta e o processo empregado para sua utilização em um projeto de desenvolvimento de *software*.

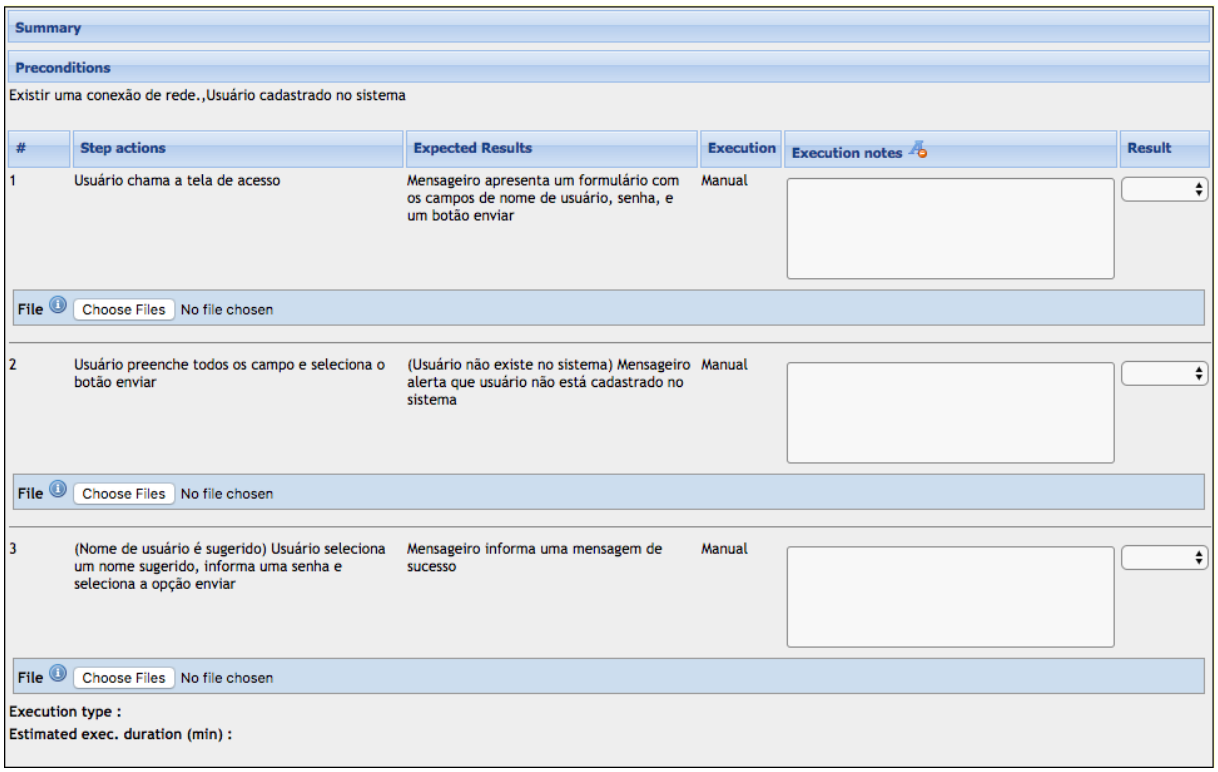


Figura 3.21: Exemplo de caso de teste importado no *Testlink*.

Capítulo 4

Avaliação

Para avaliar as contribuições deste trabalho, foram executados dois experimentos (Seção 4.1 e Seção 4.2) e dois estudos de caso (Seção 4.3 e Seção 4.4) que serão apresentados neste capítulo. Nos experimentos, o objetivo foi avaliar a linguagem proposta, enquanto que nos estudos de caso o foco recaiu sobre o processo proposto.

4.1 Experimento 1

O experimento desta seção é do tipo empírico e foi delineado segundo a abordagem do *Goal, Question, Metric* (GQM) [37]. Sendo assim, objetivamos **analisar** a linguagem proposta para a criação de documentos de requisitos **com a intenção de** avaliá-la em comparação a uma técnica não estruturada **de acordo com** os critérios de cobertura, consistência, completude e tempo, **do ponto de vista de** desenvolvedores e testadores de software **no contexto de** especificação de casos de uso.

Com o intuito de facilitar o entendimento deste experimento, detalharemos a metodologia (Seção 4.1.1), os resultados (Seção 4.1.2) e discussão (Seção 4.1.3) a seguir.

4.1.1 Metodologia

Nas seções seguintes serão expostos: objetivo, participantes, tarefas, variáveis, métricas, hipóteses e desenho do experimento.

Objetivo

O objetivo deste experimento foi investigar a linguagem Claret, comparando-a à escrita livre, em termos de cobertura, consistência, completude e tempo. É importante observar que, apesar da existência na literatura de outras linguagens para especificação de requisitos, foi decidido em nossos estudos focar apenas na escrita livre. As razões disso foram:

(i) Algumas das linguagens disponíveis focam em um nível de abstração mais baixo, pois utilizam recursos de anotação com lógica para a geração de testes de execução automática [38], [33], [39]. Já outras ferramentas fazem uso de diagramas da UML como abordagem para modelar requisitos e necessitam de diretivas lógicas com finalidade de automação dos testes gerados [5], [40];

(ii) A escrita livre satisfaz nosso contexto de investigação, já que é utilizado na engenharia de requisitos em contexto ágil e também em muitos processos de MBT [41];

Existem também anotações com suporte para desenvolvimento orientado ao comportamento — do inglês *Behaviour-Driven Development* (BDD) — [42], na forma de estórias de usuário, como por exemplo a linguagem *Gherkhin* [43]. Entretanto, a abordagem por estórias de usuário visa à descrição de requisitos comportamentais, enquanto que o caso de uso descreve as sequências de interações entre um ator e o sistema em desenvolvimento.

Além disso, a escrita livre foi utilizada como um patamar comparativo para investigar se Claret se encaixa em contexto de escrita para casos de uso em linguagem natural.

Participantes

Tendo em vista que utilizamos pessoas em nossos ensaios, seguimos algumas das recomendações sugeridas por Ko et al. no que diz respeito aos experimentos controlados com participantes humanos para avaliação de ferramentas de software [44].

Sendo assim, o primeiro passo tomado foi o processo de recrutamento de candidatos para o experimento, o qual foi feito através de convites informais dentro do *Software Practices Laboratory* (SPLab). O SPLab é parte da Universidade Federal de Campina Grande (UFCG). Em seguida, aplicamos o seguinte critério de seleção: o candidato deveria possuir um mínimo de experiência acadêmica ou profissional na escrita e leitura de documentos de casos de uso. Com a amostra selecionada, esclarecemos o propósito do experimento, as tarefas a serem executadas e os termos de privacidade das informações cedidas pelos participantes. Posteriormente, seccionamos a amostra em dois grupos numericamente equânimes, de modo aleatório:

(i) Grupo 1, que fez uso de Claret;

(ii) Grupo 2, que utilizou a linguagem natural.

Por fim, realizamos um treinamento da sintaxe de Claret ao Grupo 1, pois se desejou isolar a curva de aprendizado. No total, tivemos um número de 8 participantes (unidades experimentais) para a realização deste experimento. A identificação dos participantes foi preservada por questões éticas, inerentes a um estudo empírico. Assim, as pessoas selecionadas foram referenciadas por meio de código: “C1”, “C2”, “C3” e “C4” para o Grupo 1 e ”E1”, ”E2”, ”E3”, ”E4”para o Grupo 2.

Tarefas

Pedimos para os participantes dos dois grupos especificarem casos de usos a partir de um documento de requisitos, que continha – em alto nível de abstração – os comportamentos desejados em um sistema escolar fictício. A descrição resumida do sistema utilizada no experimento foi: esta aplicação deve ser capaz de registrar as faltas de alunos e caso o número de ausências atinja um determinado limite, os responsáveis pelo aluno devem ser informados por mensagens de texto via celular.

O Grupo 1 utilizou a notação proposta em CLARET e o Grupo 2 fez uso da escrita livre (sem imposição de estruturas ou termos específicos na modelagem). Com o Grupo 1 foi realizado o treinamento prévio, com a finalidade de prover os conhecimentos da sintaxe e semântica de cada comando da linguagem CLARET.

A execução das atividades foi realizada em sessões individuais, em que cada participante teve que especificar um caso de uso na linguagem estabelecida para seu grupo. Ao término de cada sessão, foi coletado o artefato produzido pelo participante para posterior análise.

Variáveis Independentes e Dependentes

O documento de requisitos alocado aos participantes foi elencado como variável independente para este experimento.

As variáveis dependentes deste trabalho experimental foram os critérios de cobertura, consistência, completude e tempo para a escrita de casos de usos em ambas as técnicas. Os valores destas variáveis dependentes foram capturados durante o processo de avaliação da escrita dos artefatos pelos participantes do experimento.

Métricas

Com o intuito de dar suporte às questões de pesquisa (ver a Seção 4.1.1), foi delimitado o seguinte conjunto de métricas:

- Cobertura de Requisitos (CR): representa o número de requisitos cobertos pela especificação de cada participante sobre o total de requisitos esperados. Estes requisitos esperados foram previamente identificados e listados a partir do documento de requisito funcional, conforme apêndice A.

  Sendo:

  - RC: total de requisitos cobertos pelo participante
  - RE: total de requisitos esperados

  A equação de CR é:

$$CR = RC/RE \tag{4.1}$$

- Completude e Consistência de Casos de Uso (CCCU): O projeto *Open UP*[1] disponibiliza uma lista de verificação com 24 questões que avalia se o fluxo principal e os cenários alternativos e de exceção são descritos de maneira consistente e completa. Logo após o estudo, foi solicitado para um engenheiro de requisitos independente inspecionar todos os artefatos produzidos pelos participantes tomando como referência a lista de verificação do *Open UP*. Esta métrica mensura a taxa de respostas positivas encontradas na lista.

- Tempo (T): o tempo em minutos gasto por cada participante para concluir a tarefa designada.

Hipóteses

Visando alcançar os objetivos propostos para este experimento, definimos as seguintes questões de pesquisa:

- Q_1 - Os participantes que utilizaram CLARET para especificação, criaram artefatos com melhor cobertura de requisitos em comparação aos que utilizaram a escrita livre?

- Q_2 - Os artefatos especificados com CLARET foram mais consistentes e completos?

- Q_3 - Os participantes pouparam tempo usando CLARET na especificação de requisitos?

Iniciamos nosso estudo experimental testando as hipóteses nulas, as quais afirmam que não há diferenças entre CLARET e a escrita livre para as métricas CR, CCCU e Tempo:

- $H_{1-0} : CR(\text{CLARET}) = CR(escritalivre)$

- $H_{2-0} : CCCU(\text{CLARET}) = CCCU(escritalivre)$

- $H_{3-0} : T(\text{CLARET}) = T(escritalivre)$

Caso existam evidências que refutem as hipóteses nulas, as hipóteses alternativas direcionais serão investigados mais a fundo.

Desenho do Experimento

A Figura 4.1 ilustra como o experimento foi delineado previamente. Estabelecemos a UFCG como o local para a execução do experimento em um contexto de projeto *offline*. A amostra foi elecanda com desenvolvedores profissionais membros de equipes de projetos de desenvolvimento de software.

[1] http://epf.eclipse.org/wikis/openup/

O *corpus* do experimento foi um documento contendo requisitos de usuário para um sistema fictício, conforme dito anteriormente. Após a aprendizagem do Grupo 1, cada amostra teve seu tempo de execução cronometrado. Com a tarefa realizada, pudemos calcular as métricas adotadas neste experimento. Por fim, efetuamos inferências estatísticas nos dados colhidos e testamos as hipóteses estabelecidas.

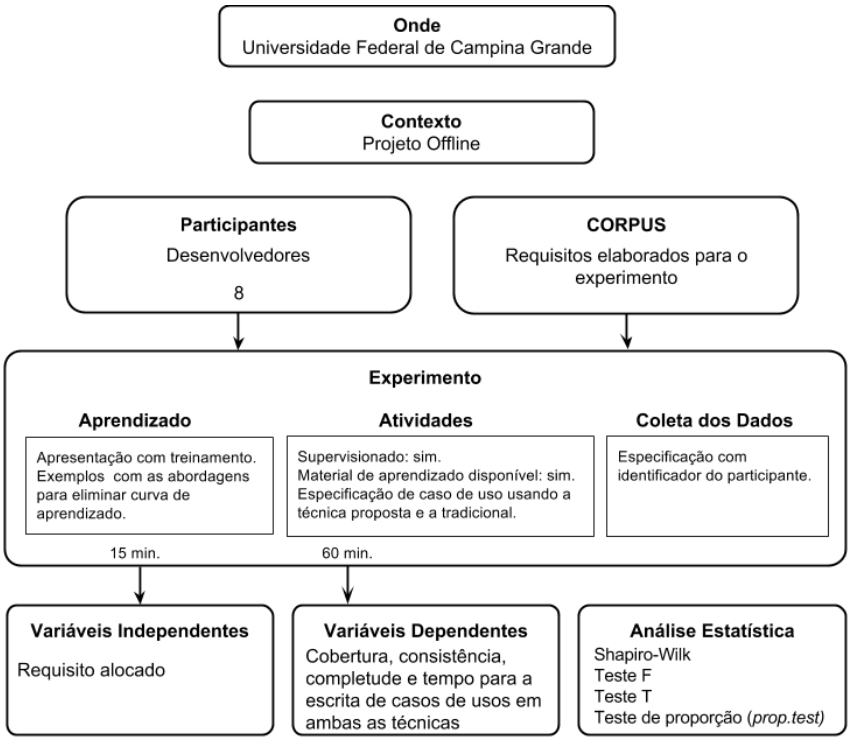


Figura 4.1: Visão geral do experimento

4.1.2 Resultados

Nossa primeira análise foi com os resultados da métrica CR, obtidos por meio da avaliação do caso de uso especificado por cada participante. O grupo que usou Claret cobriu 34 pontos de um total de 36 pontos, enquanto o grupo da escrita livre conseguiu 33 pontos do mesmo total. Para responder a questão de pesquisa Q_1, utilizamos um teste de proporção com intervalo de confiança de 95%. Obtivemos o *p-value* igual a 1, o que indicou que não houve diferença estatística apesar da diferença numérica, e confirma a hipótese nula H_{1-0}. Conforme a Figura 4.2 podemos observar que o grupo que utilizou Claret teve uma percentual de 94% de cobertura de requisitos, enquanto que o grupo que utilizou a escrita livre obteve um percentual de 91%.

A Figura 4.4 mostra os resultados de completude e consistência dos casos de uso para os grupos. Estas qualidades foram atribuídas mediante um *checklist* fornecido pelo projeto *Open UP*. A qualificação da métrica CCCU foi avaliada por um experiente engenheiro de requisitos. Podemos verificar que todos os participantes que utilizaram Claret (C1, C2, C3 e C4) tiveram um percentual individual de CCCU mais elevado do que os participantes

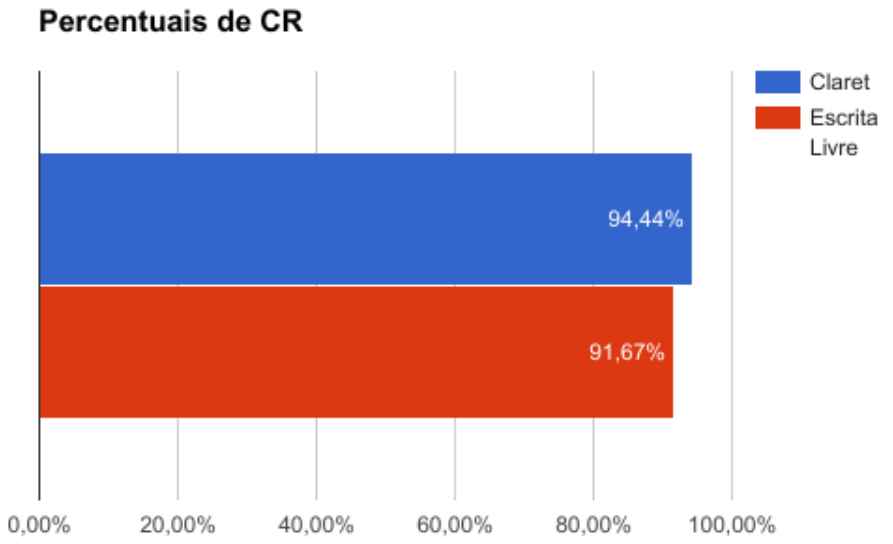


Figura 4.2: Cobertura de Requisitos

do Grupo 2 e atingiram uma média de 88,5%. Já os participantes que fizeram uso da escrita livre (E1, E2, E3 e E4) cobriram uma média de apenas 51%.

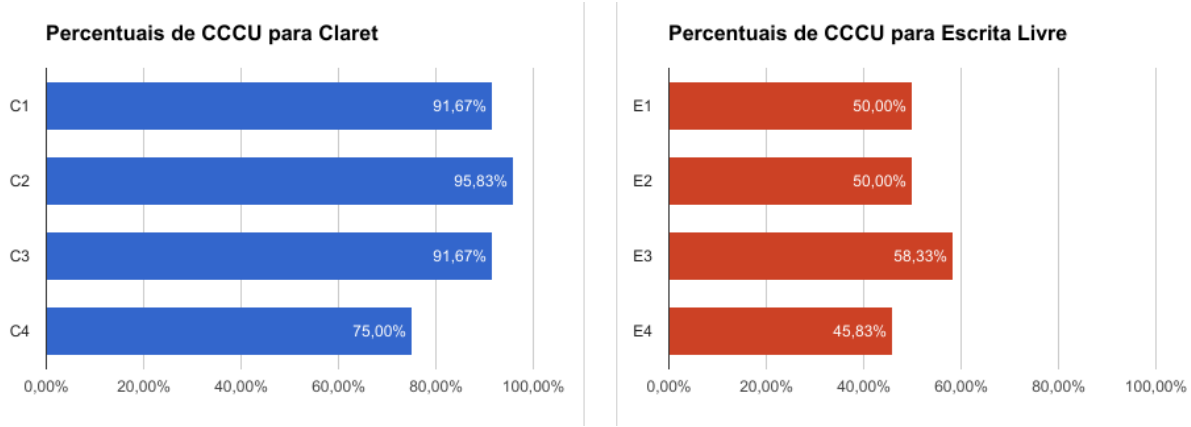


Figura 4.3: Completude e Consistência de Casos de Uso Individuais

Para os testes de hipóteses, escolhemos o teste de proporção *prop.test* do *R* para checar se a diferença encontrada entre as taxas foram significantes estatisticamente. Com 95% de confiança (*p-value* = 3,774e-08), a diferença encontrada foi significativa, o que nos leva a responder nossa segunda questão de pesquisa (Q_2), afirmando que CLARET pode proporcionar a criação de artefatos de especificação mais consistentes e completos.

Por fim, nós medimos o tempo gasto por cada participante na especificação dos casos de uso. O grupo que usou CLARET obteve os seguintes tempos em minutos e segundos: 44:41, 38:14, 15:22, 40:17. Já o grupo que fez uso da escrita livre alcançou os seguintes tempos: 19:35, 24:36, 31:30, 61:01. Pela distribuição normal conferida através de testes de *Shapiro-Wilk* com *p-value* maior que 0.05 (0.1253) para o grupo de CLARET e *p-value* maior que 0.05 (0.2266) para o grupo da escrita livre, foi escolhido o *t-test* para

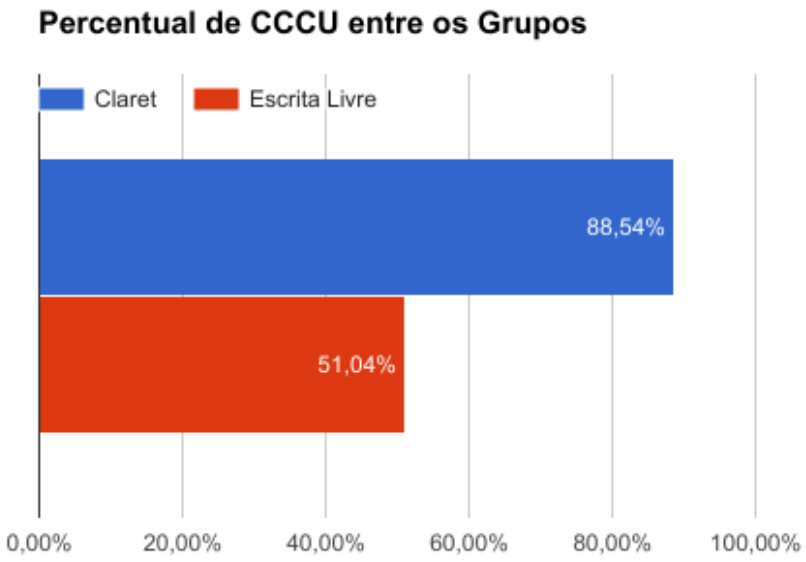


Figura 4.4: Completude e Consistência de Casos de Uso dos Grupos

a comparação das médias dos dois grupos. Antes, porém, se faz necessário avaliar a variância das amostras através de um teste F para examinar a homocedasticidade, o que foi confirmado com o *p-value* maior que 0.05 (0.5846). Em seguida, aplicamos o *t-test* para variâncias homogêneas (*var.equal=TRUE*) e amostras independentes (*paired=FALSE*), e tivemos como resultado um *p-value* maior que 0.05, o que nos indica que as médias dos tempos do grupos são estatisticamente similares.

4.1.3 Discussão

No que se refere à questão de pesquisa Q_1 (artefatos criados com CLARET tiveram melhor cobertura de requisitos em comparação aos que utilizaram a escrita livre?), a diferença entre os grupos foi de 3 pontos percentuais favorável para a especificação com CLARET. Embora este valor não represente uma significância estatística, a escrita com CLARET colaborou para um percentual maior de cobertura de requisitos em relação ao emprego da escrita livre.

Com relação à questão de pesquisa Q_2 (artefatos especificados com CLARET foram mais consistentes e completos?), nós analisamos quais elementos da lista *Open UP* foram mais esquecidos pelos participantes que fizeram uso da escrita livre. Foi observado que, com a escrita livre, os participantes foram inconsistentes na especificação dos atores e na identificação dos passos dentro dos fluxos. Somado a isso, alguns passos não foram claramente definidos ou associados com os atores. Estas falhas tornaram os casos de uso especificados com a escrita livre ambíguos e/ou difíceis de ler, fato que impediria ou tornaria muito difícil uma transformação automática para casos de testes. Muitos desses problemas não foram identificados nos casos de uso escritos com CLARET. A sintaxe

estruturada de CLARET conduziu os participantes a evitarem esse tipo de erro, criando casos de usos mais legíveis.

Por último, foi observado que apesar do tempo gasto pelo Grupo 1 ter sido um pouco maior que o do Grupo 2, a linguagem CLARET se mostrou mais produtiva, tendo em vista que os índices de CR e CCCU foram maiores. É importante mencionar que, mesmo com o treinamento prévio, a maioria dos participantes do Grupo 1 estavam tendo seu primeiro contato com a ferramenta proposta. Nós acreditamos que o tempo tenderia a diminuir na medida em que o desenvolvedor adquirisse mais experiência com a sintaxe de CLARET e sua forma de modelagem. Dessa forma, embora não possamos afirmar que os participantes do grupo experimental economizaram tempo em relação grupo controle, os tempos gastos entre as duas estratégias foram muito próximos.

Análises de Ameaças

No contexto de ameaças à validade de constructo, poderia ser alegado o fato do experimento não ter explorado a totalidade de métricas possíveis para avaliar a qualidade de casos de uso especificados. Todavia, levando em consideração o escopo da nossa investigação, acreditamos que a avaliação por meio das métricas que elencamos (CR, CCCU, Tempo) foi apropriada para o tempo disponível na pesquisa.

Em termos de ameaças à validade interna, há a questão de que CLARET foi apresentada previamente, sendo esclarecidas as suas características principais. Este procedimento, entretanto, teve de ser executado com vistas a nivelar, na medida do possível, o entendimento dos participantes. Outra ameaça refere-se à presença do pesquisador nas sessões de execução das tarefas pelos participantes. Sua presença foi necessária, com intuito de sanar dúvidas exclusivamente sobre a sintaxe de CLARET.

Nos limites das ameaças à validade externa, algumas circunstâncias neste experimento podem dificultar a generalização dos resultados. Por exemplo, o número limitado de requisitos de usuário pode não expressar todas as possibilidades. Contudo, embora simples, os requisitos buscaram englobar os cenários que estão presentes na maioria das especificações de sistemas (fluxo básico, fluxos alternativos e de exceção). Como estávamos lidando com desenvolvedores profissionais, procuramos não saturá-los com extensa série de especificações, acarretando em impacto negativo na motivação para participar do experimento e enviesando os resultados. Mais uma ameaça seria que a amostra não é representativa para a totalidade da população. Para minimizar esta ameaça, nós recrutamos participantes de diferentes projetos (*SAFF*, *Ingetools* e *e-Pol*), que exercem diferentes funções e possuem variados níveis de experiência (graduandos, pós-graduandos e profissionais).

4.2 Experimento 2

O experimento desta Seção é também do tipo empírico e foi delineado segundo a mesma abordagem GQM [37] do experimento da Seção 4.1. Desse modo, objetivamos **analisar** a linguagem proposta para a criação de documentos de requisitos **com a intenção de** avaliá-la em comparação a uma técnica não estruturada **de acordo com** os critérios de legibilidade e capacidade de escrita **do ponto de vista de** desenvolvedores e testadores de software **no contexto de** especificação de casos de uso.

Com o intuito de facilitar o entendimento do experimento, detalharemos a metodologia (Seção 4.2.1), os resultados (Seção 4.2.2) e discussão (Seção 4.2.3) a seguir.

4.2.1 Metodologia

Nas seções seguintes serão expostos: objetivo, hipóteses, participantes, tarefas, questionário, variáveis, métricas e desenho do experimento.

Objetivo

Avaliar a notação CLARET, comparando-a com uma técnica não estruturada (escrita livre) em relação à qualidade dos artefatos criados, de acordo com os seguintes critérios:

- **Legibilidade**: determina a facilidade de leitura e compreensão de uma linguagem;
- **Capacidade de escrita**: determina o quão fácil é a escrita em uma linguagem.

Hipóteses

Para a construção das hipóteses, empregamos a seguinte escala *Likert*: 1 = Muito Fácil, 2 = Um Pouco Fácil, 3 = Nem Fácil e Nem Difícil, 4 = Um Pouco Difícil, 5 = Muito Difícil. Levando-se em consideração a legibilidade e a capacidade de escrita das técnicas estudadas, foram levantadas as seguintes hipóteses nulas e alternativas:

- $H_{1-0} : legibilidade(escrita\_livre) <= 3$
- $H_{1-1} : legibilidade(escrita\_livre) > 3$
- $H_{2-0} : capacidade\_de\_escrita(escrita\_livre) <= 3$
- $H_{2-1} : capacidade\_de\_escrita(escrita\_livre) > 3$
- $H_{3-0} : legibilidade(\text{CLARET}) <= 3$
- $H_{3-1} : legibilidade(\text{CLARET}) > 3$
- $H_{4-0} : capacidade\_de\_escrita(\text{CLARET}) <= 3$
- $H_{4-1} : capacidade\_de\_escrita(\text{CLARET}) > 3$

Participantes

Para a seleção dos participantes, utilizamos os mesmos critérios definidos no experimento da Seção 4.1. No total, tivemos um número de 6 participantes (unidades experimentais) para a realização deste experimento.

Tarefas

Com a finalidade de avaliar os critérios de legibilidade e capacidade de escrita nas duas técnicas em estudo (CLARET e escrita livre), foram atribuídas tarefas de elaboração de casos de uso para os participantes selecionados. Para a execução das tarefas, foram utilizados dois documentos com requisitos funcionais previamente criados durante o desenho do experimento. Tais documentos detalharam um conjunto de comportamentos desejados em sistemas imaginários.

Cada participante teve que modelar um caso de uso a partir de um dos documentos de requisitos (selecionado aleatoriamente), utilizando para isso a escrita livre. Em seguida, os mesmos participantes tiveram um treinamento na técnica de CLARET e depois receberam a tarefa de especificar um segundo caso de uso a partir do outro documento de requisito. Após a finalização das especificações dos casos de uso com CLARET, submetemos os participantes a um questionário de avaliação das técnicas.

Os artefatos especificados pelos participantes foram coletados para fins de histórico. O objetivo da especificação dos casos de uso nas duas técnicas foi dar subsídios suficientes aos participantes para responderem ao questionário com segurança.

Questionário

Os dados foram capturados mediante um questionário realizado com os participantes do experimento. A elaboração do questionário seguiu as melhores práticas para conduzir um experimento controlado com participantes humanos [44]. Além disso, foi levado em consideração o trabalho de Kosar et al. para a comparação de linguagens específicas de domínio [45].

Para quantificar as perguntas do questionário, foi adotada a escala *Likert*, em que os participantes têm que atribuir um grau ao nível de concordância à afirmação em questão (1 = Muito Fácil, 2 = Um Pouco Fácil, 3 = Nem Fácil e Nem Difícil, 4 = Um Pouco Difícil, 5 = Muito Difícil).

Variáveis Independentes e Dependentes

Para a execução do experimento, os documentos de requisitos alocados foram adotados como variável independente.

Em relação às variáveis dependentes, tivemos as características de Legibilidade e a Capacidade de Escrita em ambas as técnicas de descrição de modelos de casos de uso.

Métricas

A métrica utilizada no experimento foi escolhida com a finalidade de examinar o comportamento de cada participante perante as técnicas disponibilizadas. Para isso, adotamos a medida de tendência central, mais especificamente a moda, para as questões que adotaram a escala *Likert*.

Desenho do Experimento

A visão geral do desenho do experimento é esboçada na Figura 4.5. Definimos como local da execução do experimento a UFCG, por meio de um contexto de projeto *offline*. A amostra foi composta por desenvolvedores profissionais membros de equipes de projetos de desenvolvimento de software.

O *corpus* do experimento foi um documento contendo requisitos de um sistema fictício. Todos participantes tiveram treinamento na linguagem de CLARET. Após a aprendizagem, cada unidade experimental teve que especificar um caso de uso com a notação proposta. Realizada a tarefa, aplicamos o questionário para avaliação das técnicas do estudo. Por último, efetuamos inferências estatísticas nos dados colhidos do questionário e testamos as hipóteses estabelecidas.

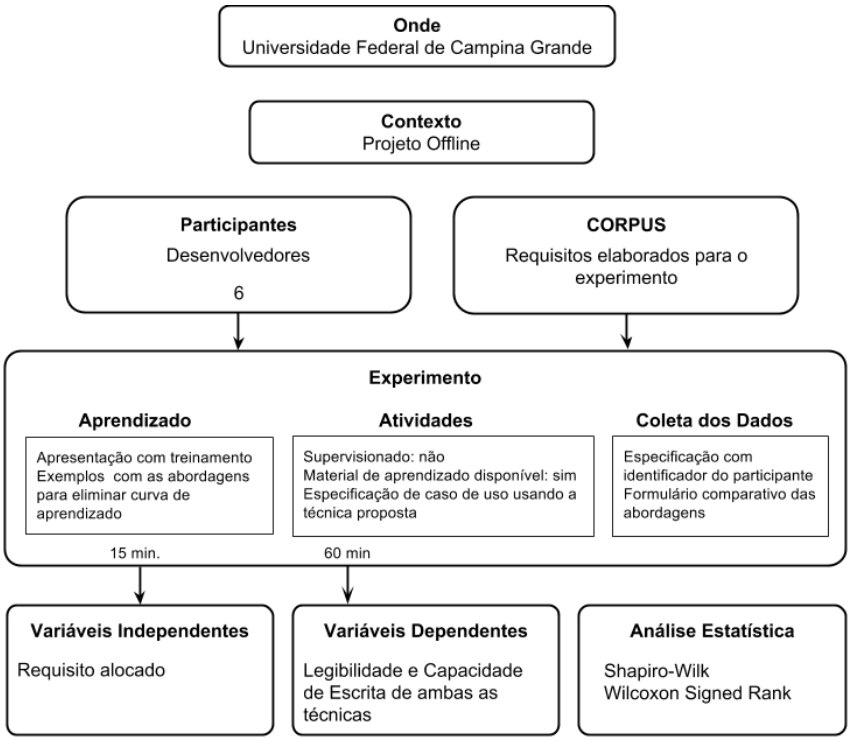


Figura 4.5: Visão geral do experimento

4.2.2 Resultados

Após a execução das tarefas do experimento, os participantes responderam o questionário especificado na Seção 4.2.1. Em seguida, as respostas do questionário foram coletadas e analisadas.

As Figuras 4.6 e 4.7 sintetizam as respostas do questionário em relação à legibilidade e capacidade de escrita das duas linguagens avaliadas.

No que diz respeito à legibilidade da escrita livre, metade dos participantes classificaram como fácil e apenas 16,7% consideraram muito fácil. Os 33,3% restantes se posicionaram de forma neutra. Já em relação à sua capacidade de escrita, a maioria (50%) qualificou como difícil, um terço afirmou não ser nem fácil e nem difícil e somente uma minoria de 16,7% julgou ser fácil.

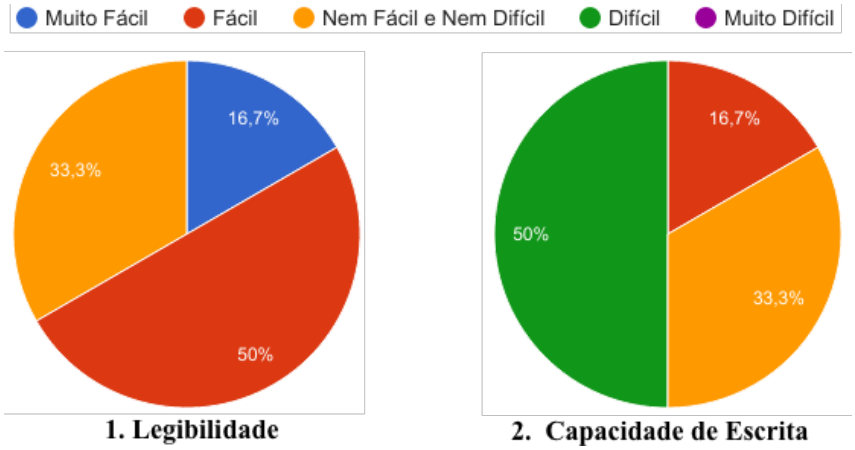


Figura 4.6: Respostas para a Escrita Livre

Quanto à avaliação da legibilidade de Claret, os participantes se dividiram em sua totalidade entre muito fácil e fácil. A capacidade de escrita foi classificada pela maioria (66,7%) como fácil e como muito fácil por 16,7%. Os demais 16,7% se mantiveram neutros.

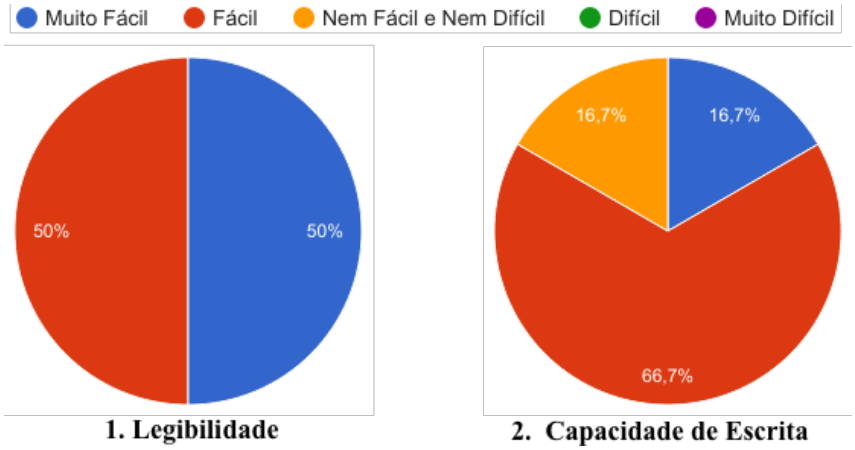


Figura 4.7: Respostas para Claret

Analisamos os resultados do questionário quanto à distribuição da normalidade para cada uma das variáveis estudadas. Primeiramente, fizemos uma análise visual através dos

histogramas e gráficos *Q-QPlot* (Apêndice B), que evidenciaram a não normalidade dos dados. Em seguida, para certificar esta afirmação, aplicamos testes de *Shapiro-Wilk*, que nos retornou baixíssimos valores para *p-value* ($< 0,05$).

Os dados coletados são ordinais, de natureza categórica e não passaram nos testes de normalidade. Estas características justificam o emprego de testes estatísticos não-paramétricos para avaliar as hipóteses propostas. Sendo assim, o teste adotado foi o *Wilcoxon–Mann–Whitney* para uma amostra.

Para a escrita livre, testamos a hipótese nula H_{1-0} e tivemos evidências suficientes para aceitar essa hipótese. Ainda em relação à escrita livre, testamos a hipótese nula H_{2-0} e a mesma foi rejeitada, enquanto que reforça a evidência para a hipótese alternativa.

A hipótese nula H_{3-0} foi refutada com evidências pelos testes e com isso a hipótese alternativa H_{3-1} para a legibilidade de CLARET foi avaliada e aceita com nível de confiança de 95%. Por fim, a hipótese nula H_{4-0} foi rejeitada e o teste com a hipótese alternativa H_{4-1} deu indícios para ser aceita.

4.2.3 Discussão

Em vista do exposto, infere-se que a experiência dos participantes com a notação através de CLARET evidenciou a praticidade e a concisão da ferramenta para o processo de descrição de modelos de casos de uso. De acordo com os resultados obtidos na análise do experimento, verificou-se que houve uma vantagem para a técnica com CLARET, tanto para a legibilidade, quanto para a capacidade de escrita de casos de uso.

A tendência em favor da técnica com CLARET encontrada no experimento pode ser explicada pelas vantagens que a sintaxe oferece, a exemplo da garantia da objetividade, pois a formatação de texto é descartada, ao passo que a escrita é em texto puro. Outra vantagem de suma importância é o agrupamento de múltiplos casos de uso em apenas um artefato, o que evita a utilização de várias instâncias de editor de texto para a navegação dos documentos. Para o profissional que descreve os casos de uso, CLARET demonstrou facilitar substancialmente o fluxo de trabalho, tanto no processo de criação, quanto no de manutenção dos modelos.

Análises de Ameaças

Dentre as ameaças possíveis, encontramos a ameaça à validade de conclusão, relacionada ao baixo poder estatístico, visto que os participantes foram apenas alunos voluntários do curso de Ciência da Computação, tornando a amostra restrita estatisticamente por não incluir outros perfis de especificadores de requisitos.

Outras ameaças identificadas referem-se à Validade de Constructo, especificamente às Ameaças Sociais, sendo elas: Adivinhação de Hipótese e Medo de Avaliação. A primeira decorre do fato de que cada participante avaliou duas técnicas diferentes, e de acordo

com a experiência individual, a adivinhação de hipóteses pode ter sido facilitada. Já a segunda, medo de avaliação, diz respeito à possibilidade de algumas pessoas não se sentirem confortáveis sendo avaliadas e consequentemente comprometer os resultados.

Com o intuito de mitigar parte das ameaças, foi realizada uma seleção criteriosa dos participantes do experimento com suporte de uma pequena entrevista prévia.

4.3 Estudo de Caso 1

Este estudo de caso foi realizado em um contexto de cooperação entre o SPLab e a empresa Ingenico do Brasil Ltda[2]. Através deste estudo exploratório, pudemos comprovar a efetividade da metodologia proposta neste trabalho em capturar defeitos nos testes realizados. Detalharemos esta investigação nas seguintes seções: planejamento (Seção 4.3.1), procedimentos (Seção 4.3.2), resultados (Seção 4.3.3) e discussão (Seção 4.3.4).

4.3.1 Planejamento

Realizamos este estudo no período de setembro de 2015 a dezembro de 2015. Por meio de um treinamento, os participantes tiveram o primeiro contato com a sintaxe de CLARET e o emprego de sua metodologia.

Objetivo

Averiguar a efetividade dos casos de testes, especificados com CLARET, em capturar falhas de dois projetos de software: SAFF e TCoM. O projeto SAFF é um sistema de informação que gerencia relatórios de *status* para sistemas embarcados. Já o projeto TCoM é um sistema que controla a execução e gerencia relatórios de testes em partes de um dispositivo embarcado.

Participantes

Os participantes foram três engenheiros de *software*.

Tarefas

Através de dois projetos distintos, SAFF e TCoM, pedimos aos participantes para modelarem seus requisitos usando CLARET e sua metodologia, para gerar automaticamente as suítes de testes de sistema.

[2] www.ingenico.com

Questões de Pesquisa

Neste estudo, elaboramos as seguintes questões de pesquisa:

- Q_1 - Como a anotação de CLARET e seu suporte ferramental apoiou a especificação dos requisitos?
- Q_2 - As suítes de testes criadas a partir de especificações com CLARET encontraram defeitos de software?

4.3.2 Procedimentos

Este estudo foi realizado nos seguintes moldes: a partir das discussões com o cliente, os engenheiros criaram uma lista de requisitos do usuário. Em seguida, esta lista foi detalhada através de especificações com CLARET. Por meio do suporte ferramental, os engenheiros validaram sintaticamente os arquivos e geraram os modelos em formato TGF que posteriormente foram averiguados visualmente em relação à corretude e completude. Por fim, as suítes de teste foram criadas usando a ferramenta LTS-BT [36] e importadas no ambiente do *Testlink*. Antes das datas de lançamento dos sistemas, as suítes de teste foram executadas e seus resultados analisados.

4.3.3 Resultados

A Tabela 4.1 mostra o número de casos de uso criados para cada um dos sistemas e o tamanho das respectivas suítes de testes. Podemos ver que o TCoM teve apenas uma versão lançada, enquanto o SAFF teve duas versões e consequentemente duas rodadas de requisitos e testes.

Sistema	Total de Casos de Uso	Total de Casos de Teste
SAFF1	17	56
SAFF2	19	63
TCoM	12	32

Tabela 4.1: Totais de Casos de uso e Casos de Teste por sistema e versão

Apesar do esforço inicial para a aprendizagem da sintaxe de CLARET e de como utilizar seu suporte ferramental, os engenheiros não tiveram dificuldades em utilizar a ferramenta para modelar requisitos. Nenhum requisito foi impossibilitado de ser especificado através de CLARET. Além do mais, o suporte oferecido pela ferramenta (checagem de sintaxe, geração de modelos TGF) foi significativamente apreciado, pois forneceu uma grande ajuda na validação dos artefatos de requisitos. Em diversos momentos, o documento de requisitos teve que ser revisado devido a inconsistências encontradas via inspeções no TGF, como por exemplo a inexistência de retornos de fluxos alternativos ou de exceção para

o fluxo principal. Dessa forma, podemos responder nossa primeira questão de pesquisa (Q1), mostrando que de fato CLARET e seu suporte ferramental está apto para especificar requisitos na prática.

A Figura 4.8 informa os resultados percentuais da execução das suítes de testes em duas versões do sistema SAFF e uma versão do sistema TCoM. Os gráficos evidenciam os testes que obtiveram sucesso, os que falharam e os que foram bloqueados. Os casos de testes que foram bloqueados expressam uma pequena porcentagem e ocorreram em virtude de uma defasagem entre a implementação do código e os artefatos de especificação de requisitos. Este é um desafio recorrente no desenvolvimento ágil em que mudanças são continuamente admitidas. Uma chave de resolução para esse problema pode estar em um efetivo processo de gerenciamento de mudanças.

Como pode ser observado em relação aos casos de testes que falharam, no SAFF 1 houve um percentual de 36% em virtude do sistema não estar de acordo com as especificações. Um pouco mais baixas foram as taxas de defeitos encontrados nos dois outros sistemas (SAFF 2 - 24% e TCoM - 25%). Tais falhas dizem respeito a defeitos relacionados aos fluxos de exceção não tratados pelos desenvolvedores e/ou ausência de validações em campos obrigatórios de formulários. Todos os defeitos foram registrados e os trechos reparados foram inclusos na nova versão do sistema. Sendo assim, nós podemos responder a segunda questão de pesquisa (Q2), constatando que as suítes criadas pelos artefatos de CLARET estão aptas a encontrar defeitos reais.

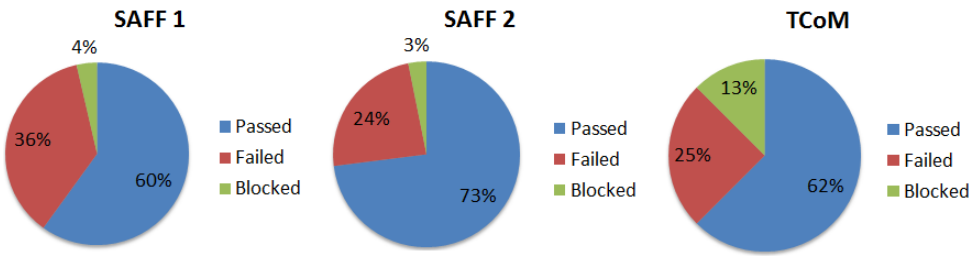


Figura 4.8: Resumo dos resultados da execução de testes nos sistemas

4.3.4 Discussão

Este estudo de caso evidenciou os benefícios práticos de CLARET, seu suporte ferramental e processo proposto. CLARET possibilitou que os engenheiros de software modelassem todos os requisitos das três versões de dois sistemas reais (SAFF e TCoM).

Um importante fato a ser mencionado é que tanto as artefatos escritos em CLARET quanto seus respectivos TGF foram utilizados para guiar o desenvolvimento de código. As representações TGF funcionaram como um diferencial bastante útil que propiciou aos desenvolvedores uma melhor visão geral do sistema e compreensão de seus principais cenários. Apesar da metodologia ágil pregar o mínimo possível de modelos e documentos, o TGF se mostrou bastante pertinente e eficaz na visão dos desenvolvedores.

Ademais, as suítes de teste oriundas dos artefatos de especificação foram capazes de encontrar defeitos em versões de sistemas aparentemente estáveis. Usando CLARET e sua metodologia proposta, os engenheiros criaram artefatos úteis e gerenciáveis e evitaram custos extras com defeitos encontrados tardiamente.

4.4 Estudo de Caso 2

Para realizar uma análise ainda mais completa, efetuamos um segundo estudo de caso de caráter exploratório com foco na utilização de CLARET em um projeto real da indústria. Detalharemos este estudo nas seguintes seções: planejamento (Seção 4.4.1), resultados (Seção 4.4.2) e discussão (Seção 4.4.3).

4.4.1 Planejamento

Este estudo foi realizado no período de novembro de 2016 a janeiro de 2017. Empregamos a observação direta como forma de coletar informações acerca da utilização prática da ferramenta CLARET no processo de MBT e engenharia de requisitos para um sistema legado sem documentação e testes. A observação direta nos auxiliou respondendo questões que estavam fora de nosso controle, como os fenômenos individuais e comportamentais do processo.

Complementando o estudo, também aplicamos um questionário aos participantes. Neste questionário, pedimos para avaliarem CLARET e indicarem pontos positivos e negativos. Os resultados deste questionário serão apresentados na Seção 4.4.2 e discutidos na Seção 4.4.3.

Objetivo

O objetivo deste estudo foi investigar de que forma os participantes deste estudo captaram a abordagem proposta por CLARET em sua totalidade: a sua notação, suporte ferramental e o seu processo em um contexto ágil. A partir disto, identificar as limitações de CLARET em seu uso prático, com vistas a realizar aperfeiçoamentos em trabalhos futuros.

Uma nova investigação da legibilidade e capacidade de escrita foi legitimada pelo fato de ser um estudo de caso prático em um projeto de desenvolvimento real. Em relação ao suporte ferramental, criamos um indicador de facilidade de uso para descobrir o nível de satisfação na utilização de CLARET: *plugin* de editor de texto e mensagens de *feedback*. As respostas desta métrica nos guiaram para estabelecer os trabalhos futuros.

Também inquerimos aos participantes um julgamento acerca da qualidade da documentação gerada por CLARET. Dessa forma, avaliamos se os documentos produzidos por CLARET atenderam as necessidades dos envolvidos no estudo de caso. Por último, questionamos aos participantes o nível de satisfação geral com a ferramenta.

Participantes

Por meio de uma amostragem não-probabilística por conveniência, foram selecionados sete indivíduos, membros de uma equipe de projeto de verificação e validação de software, sendo eles: dois desenvolvedores, quatro testadores e um gerente.

Questionário

O questionário aplicado ao final do estudo exploratório (ver Apêndice C.2) foi composto por oito questões, sendo cinco de caráter objetivo e três de teor subjetivo. As questões objetivas foram quantificadas por meio da escala *Likert*, enquanto que as questões subjetivas foram analisadas mediante uma abordagem qualitativa que uniu o conteúdo apresentado nas falas dos indivíduos ao nosso registro observacional *in loco*.

4.4.2 Resultados

Aqui apresentamos os resultados quantificáveis do questionário. A Figura 4.9 mostra que CLARET teve uma avaliação positiva em um total de 85,7% (muito fácil com 28,6% e fácil com 57,1%) no que concerne a facilidade de escrita. O mesmo percentual foi encontrado nas avaliações em relação à facilidade de leitura (muito fácil com 57,1% e fácil com 28,6%). Essas duas variáveis já haviam sido estudadas no experimento da Seção 4.2 por meio de uma abordagem conduzida em um ambiente controlado. Este estudo de caso por sua vez, representa uma análise dessas questões em um contexto real.

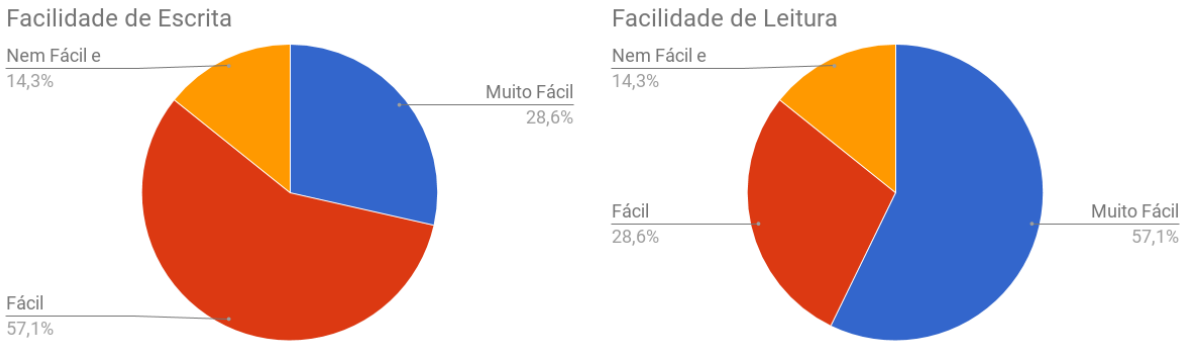


Figura 4.9: Resultados para a facilidade de escrita e leitura

Em relação à qualidade da documentação gerada por CLARET, a Figura 4.10 demonstra um percentual total de 85,7% para os conceitos altíssima qualidade (28,6%) e alta qualidade (57,1%), enquanto que apenas 14,3% indicou uma neutralidade. A mesma figura indica uma unanimidade em relação aos resultados para a facilidade de uso, tendo em vista que todos os participantes classificaram como de altíssima (57,1%) e alta qualidade (42,9%).

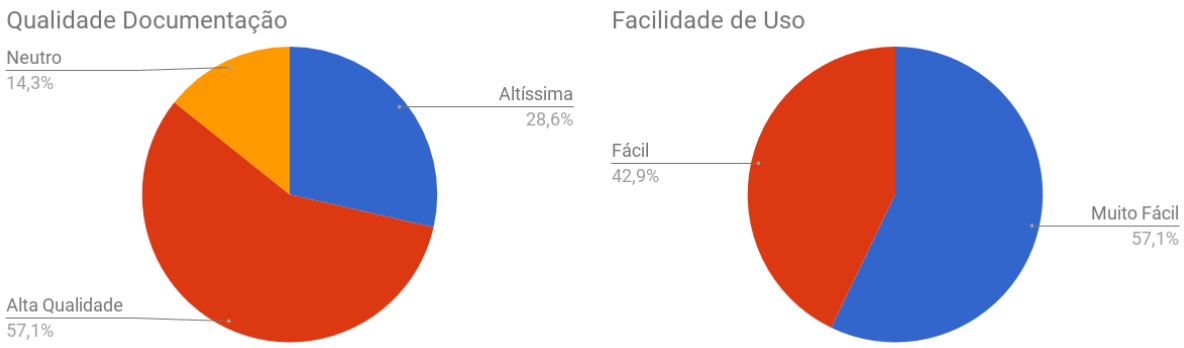


Figura 4.10: Resultados para a qualidade da documentação e facilidade de uso

No tocante ao nível de satisfação, o gráfico da Figura 4.11 demonstra a total satisfação dos envolvidos no estudo, que se conceituaram como muito satisfeitos (28,6%) e satisfeitos (71,4%).

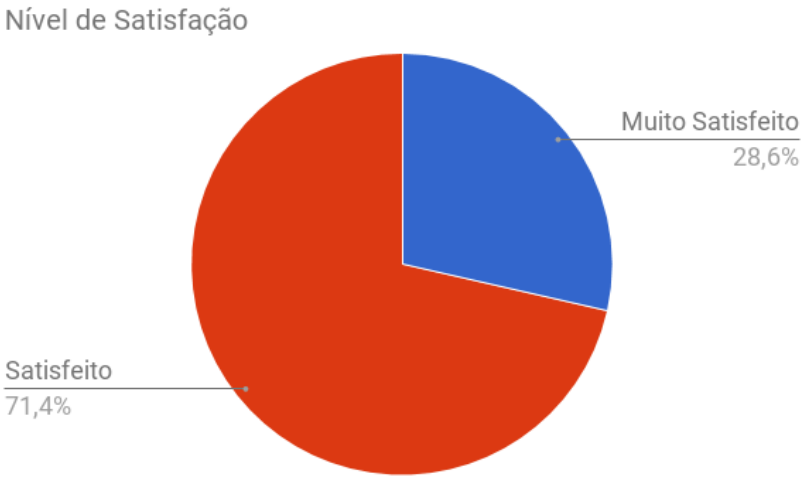


Figura 4.11: Resultados para o nível de satisfação

4.4.3 Discussão

Neste segundo estudo de caso, acompanhamos continuamente a utilização da abordagem da ferramenta e sistematizamos o *feedback* dos participantes sobre CLARET e sua metodologia, com a finalidade de melhorias em trabalhos futuros.

Comprovamos as qualidades propostas por CLARET em seu emprego prático. A legibilidade e a capacidade de escrita foram pontos fortes identificados pelos indivíduos, que consideraram a simplicidade da sintaxe de CLARET como crucial para criar e ler especificações de requisitos de forma mais ágil.

No que se refere a facilidade de uso, CLARET e seu suporte ferramental foram bastante aceitos pelos participantes, visto que as facilidades providas pela ferramenta pouparam tempo e esforços na criação de documentos e casos de testes para os requisitos.

A qualidade da documentação gerada por Claret foi recorrentemente mencionada ao longo de sua utilização, não somente pelos envolvidos no estudo, como também, indiretamente, pelos clientes do projeto. As características positivas apontadas da geração de documentos foram a padronização do *layout* e a formatação do conteúdo, a flexibilidade na personalização do modelo-base para os documentos e a economia de tempo por meio de uma geração facilitada da documentação.

Em relação às melhorias propostas para a ferramenta, nosso registro observacional somado às respostas do questionário permitiu que identificássemos com maior clareza os seguintes pontos: 1) *feedback* de erros mais amigável aos usuários; 2) ambiente com mais recursos e facilidades ao especificador; 3) documentação detalhada e com exemplos de utilização; e 4) identificar más práticas ou anomalias na especificação e sugerir possíveis correções. Isso demonstra que os pontos fracos de Claret estão centrados em recursos que facilitem a utilização da ferramenta, mas que não comprometem a sua funcionalidade.

4.5 Considerações Finais

Este capítulo apresentou os experimentos e estudos de caso realizados com o intuito de validar a abordagem proposta em Claret. O primeiro experimento teve objetivo de comparar Claret e a escrita livre nos quesitos de cobertura, consistência, completude e tempo. Os resultados desse experimento nos deram indicadores sobre a eficiência dos testes gerados a partir dos artefatos criados com nossa ferramenta. Além disso, pudemos tirar conclusões sobre a produtividade durante a escrita. No segundo experimento, confrontamos os critérios de legibilidade e capacidade escrita das técnicas Claret e a escrita livre. Este experimento nos revelou uma vantagem de Claret frente a escrita livre nos dois critérios.

Os experimentos responderam as questões de pesquisas levantadas em um ambiente controlado. Entretanto, sentimos a necessidade de observar nossa proposta em um ambiente real de trabalho para complementar nossa investigação. Desse modo, realizamos dois estudos de caso em dois projetos de desenvolvimento de software. O primeiro estudo de caso objetivou avaliar a capacidade de detectar defeitos a partir dos artefatos de Claret. Os resultados apresentaram indícios positivos, inclusive nos fornecendo relatos de aumento de produtividade com a abordagem. O segundo estudo de caso envolveu um projeto de um sistema legado que não possuia documentação e nem testes. Por meio de análises minuciosas e o emprego da abordagem de Claret para a especificação dos requisitos, foram gerados os artefatos ausentes: testes funcionais e documentos de casos de uso. Com as respostas do estudo, identificamos as percepções dos participantes em relação a escrita e leitura da notação, a facilidade de uso do suporte ferramental, a qualidade da documentação gerada e o nível de satisfação geral da nossa abordagem.

Capítulo 5

Trabalhos Relacionados

Inúmeros estudos na literatura abordam a geração de caso de testes a partir de modelos de requisitos [46, 2], bem como a engenharia de requisitos ágeis. Neste capítulo, faremos uma discussão acerca dos principais trabalhos que em algum momento serviram como alicerce ou como comparativo ao nosso trabalho.

No que concerne ao levantamento bibliográfico realizado sobre MBT, destacamos o trabalho de Machado and Sampaio [33], que discute o problema de como obter casos de testes a partir de modelos de especificação escritos em linguagem natural, contornando problemas como ambiguidade e inconsistências. Também são apresentados quatro modelos de teste e suas estratégias utilizadas para a geração automática de testes. O primeiro modelo é a cadeia de Markov, bastante utilizado para descrever sistemas como processos estocásticos para finalidade de testes estatísticos. O segundo modelo é o *Input-Output labelled transition systems*, que neste estudo emprega as estratégias (modelo e algoritmo) da ferramenta TGV (*Test Generation with Verification technology*) [47]. TGV é uma ferramenta destinada a geração e seleção de automática de testes de conformidade para sistemas reativos e não-determinísticos. O terceiro modelo é o *Annotated labelled transition systems* (ALTS), que é um *Label Transition Systems* (LTS) com anotações para facilitar o processo de geração de testes. O quarto modelo apresentado é a álgebra de processo, mais especificamente o *Communicating Sequential Processes* (CSP) utilizado para descrever o comportamento em sistemas distribuídos e concorrentes.

O artigo propõe a geração automática dos testes baseados em modelos e uma estratégia para traduzir casos de usos escritos no formato CNL (*Controlled Natural Language*) em especificações na notação ALTS. A semelhança com a nossa proposta se encontra na definição do modelo abstrato escolhido, casos de uso, através de uma linguagem controlada que facilita a tradução para os casos de testes.

A ferramenta RTCM, proposta por Yue et al. [38], tem como objetivo remover o distanciamento entre as especificações de casos de teste e as respectivas implementações através de *scripts* para a automatização dos testes. O RTCM toma como base a notação RUCM [48], combinando o uso de linguagem natural com *templates*, um conjunto de

restrições e palavras-chaves. Uma especificação em RTCM possui um fluxo básico e um ou mais fluxos alternativos. É importante ressaltar que a notação herdada de RUCM trata os fluxos de exceção como fluxos alternativos, ou seja, não há sintaxe específica para os cenários de não-sucesso. CLARET, em contrapartida, adota uma abordagem segundo a definição tradicional de casos de uso, diferenciando os cenários de sucesso, alternativos e de exceção. Além disso, CLARET também adota um nível de teste mais abstrato, dispensando a utilização de rótulos de condição, validação e concorrência dentro da especificação de um caso de uso.

Já Wang et al. [11] propôs a ferramenta UMTG, que também se baseia na notação RUCM e emprega o processamento de linguagem natural para a geração automática de casos de teste de sistemas executáveis. A motivação de UMTG é que em alguns cenários mais críticos que exigem uma complexa modelagem comportamental, a especificação de casos de uso e modelos de domínio são considerados difíceis, caros e suscetíveis a erros. A ferramenta objetiva a concepção, sem intervenção humana, de testes executáveis de sistema, além dos dados de saída e oráculos. Para alcançar tais objetivos, UMTG se utiliza das especificações dos casos de uso, através do emprego de técnicas de NLP e do modelo de domínio em questão. A geração de casos de teste de sistemas executáveis foge do escopo da proposta de CLARET. No contexto de MBT, o objetivo de CLARET é a geração automática de meta-modelos no formato TGF, e a transformação destes em casos de teste fica sob responsabilidade de uma outra ferramenta, mais especificamente LTS-BT [36].

Consideramos também o trabalho de Nebut et al. [49], que sugere que a utilização de casos de uso como modelos para a geração automática de casos de teste requer uma atividade de complexa acuidade, por conta das ambiguidades geradas pelos requisitos escritos em linguagem natural. Sendo assim, é proposta uma abordagem na qual os casos de teste são equipados com contratos, escritos na forma de expressões lógicas nas pré-condições e pós-condições. O contra-ponto feito na nossa pesquisa em relação ao trabalho acima citado é que ao invés de utilizarmos contratos nos modelos, oferecemos uma linguagem de domínio específico no contexto de casos de uso para a geração automática de casos de teste para execução manual.

Carvalho et al. [10] mostram uma abordagem de geração de casos de teste, utilizando como insumo, requisitos especificados em linguagem natural. Estas especificações são mapeadas em formato SCR (*Software Cost Reduction*), para posteriormente gerar casos de teste pela ferramenta T-VEC. A proposta faz uso de uma gramática específica para sistemas reativos, o que facilita a especificação do contexto, mas limita seu uso, caso o escopo do projeto mude e os requisitos não sejam atendidos pela gramática. Nosso trabalho se assemelha ao processo proposto, no sentido de utilizar um modelo intermediário para a geração dos casos de teste. Entretanto, CLARET possui uma gramática baseada em casos de uso, que não se prende a um contexto específico, e desse modo se diferencia

do trabalho discutido.

Os casos de teste têm a função de avaliar a precisão de determinadas características de um sistema. Dessa forma, em uma geração automática, a qualidade dos casos de teste é diretamente relacionada com quão bem o modelo representa o sistema. Nogueira et al. propõe melhorias [50] a uma abordagem já existente [51] para geração de casos de teste utilizando álgebra de processo *Communicating Sequential Processes* (CSP), por meio da inserção de dados (entradas, saídas, variáveis e parâmetros) nas especificações e na seleção dos testes. Os requisitos são escritos em *templates* [52] através de *Controled Natural Language* (CNL) e posteriormente traduzidos para CSP, o que garante uma solidez através de relações de conformidade entre entrada e saída definidas formalmente. Nossa proposta se difere por utilizar uma linguagem de domínio específico para casos de uso, em que as descrições das ações dos atores e respostas do sistemas são descritas em linguagem natural, sem ocorrer o processamento das mesmas. O foco de CLARET é permitir a especificação de requisitos de forma ágil, com uma curva mínima de aprendizagem e sem a utilização de informações necessárias para gerar testes de execução automática. Em virtude disso, os modelos TGF gerados por CLARET se destinam exclusivamente a testes com execução manual. Para trabalhos futuros, planejamos estender nossa ferramenta com algum grau de formalidade para os modelos gerados. Com isso, pretendemos garantir modelos robustos para a atual geração de testes manuais, como também possibilitar a geração de testes de execução automática.

A linguagem semi-estruturada *Gherkin* [43] faz parte do *Cucumber*[1], uma ferramenta para criação de testes em abordagens de desenvolvimento baseado em comportamentos (BDD). A sua notação é alicerçada na especificação de estória de usuário, uma técnica bastante difundida em metodologias ágeis e que descreve os valores e cenários de uma funcionalidade esperada pelo cliente. *Gherkin* estabelece as palavras reservadas *given, when e then* para especificar, respectivamente, uma pré-condição, um evento e uma pós-condição a ser avaliada para os cenários. Com essa estrutura, o *Cucumber* gera automaticamente um esqueleto de código em que o desenvolvedor ou testador injeta os testes necessários para validação e verificação das funcionalidades. CLARET por outro lado, emprega casos de uso com especificação mais detalhada dos envolvidos (atores e sistema), dos passos necessários para atingir um objetivo e também dos possíveis passos alternativos e de exceções. Apesar dos casos de uso serem provenientes de metodologias tradicionais e tratarem os requisitos de forma diferente das estórias de usuário, a sintaxe sucinta e direta de CLARET permite agilidade na criação e manutenção das especificações de casos de uso.

O grupo formado por Sarmiento et al. propõem *C & L* [53], um ferramental para geração automática de testes a partir de especificações em linguagem natural. Sua abordagem compreende a transformação das descrições dos requisitos em diagramas de atividades da UML. Tais diagramas são representados na forma de grafos orientados e a

[1] `https://cucumber.io`

partir de estratégias de buscas são gerados os casos de teste. Para isso, *C & L* define que cada sistema em desenvolvimento possua uma linguagem específica em seu contexto, que auxiliará na construção do modelo gráfico de atividades. Assim como *C & L*, CLARET também emprega uma estrutura de grafos orientados para representação interna das especificações. Em vários outros aspectos, este trabalho se difere de nossa ferramenta, entre eles: CLARET utiliza uma linguagem de domínio específico ao invés de linguagem natural e a geração automática compreende apenas documentação e meta-modelos para posterior criação dos testes em uma ferramenta à parte (LTS-BT).

Cartaxo et al. [6] apresentam um procedimento sistemático para a geração de casos de teste, a partir de diagramas de sequência. O foco deste procedimento são os testes em dispositivos móveis e pode ser empregado em processos de desenvolvimento que utilizem diagramas de sequência para especificação dos requisitos. O LTS é utilizado neste contexto como modelo intermediário para a geração de testes, pois possibilita a descrição do conjunto de todos os comportamentos do sistema. De forma análoga, CLARET também emprega um modelo intermediário em seu processo. Este modelo é o TGF anotado e nele está a representação (nós e transições) de todos os caminhos possíveis da especificação. O TGF anotado, como seu nome já revela, possui anotações que auxiliam a geração dos casos de teste na ferramenta LTS-BT. Estas anotações indicam os ações dos atores, os resultados esperados dos sistema e as condições.

O trabalho de Sarma and Mall [5] é outra referência trazida à tona, pois aborda o problema de como obter casos de teste a partir de modelos abstratos, como diagramas de casos de uso e diagramas de sequência. Os autores apresentam uma abordagem para a transformação de diagramas de caso de uso em um grafo chamado UDG (*Use-case Diagram Graph*) e em um diagrama de sequência em grafo SDG (*Sequence Diagram Graph*), para depois combiná-los em uma estrutura denominada STG (*System Testing Graph*), que finalmente será derivada em casos de teste. Nosso trabalho se equipara na utilização de modelos de UML, presente nos artefatos criados a partir de requisitos do sistema.

Papyrus [54] é uma ferramenta de software livre e utiliza a linguagem *SysML* [55], que permite uma modelagem gráfica para especificação, análise e verificação de sistemas complexos. Esta linguagem dá suporte para modelar a estrutura, comportamentos e requisitos de sistema por meio de diagramas da UML (classes, casos de uso, sequência, atividade e outros). Através de *plugins*, é possível gerar automaticamente testes a partir de diagramas especificados no *Papyrus*.

A comunidade de software livre também propôs a ferramenta *Modelio* [56] que utiliza a linguagem *SysML* [55]. Basicamente segue os mesmos princípios de modelagem gráfica da ferramenta *Papyrus* [54] e foi criada com fins de geração automática de código-fonte e documentação a partir dos modelos em formato de diagramas.

CLARET se diferencia dos trabalhos de Cartaxo et al., de Sarma and Mall e das

ferramentas *Papyrus* e *Modelio* – citados anteriormente – por não fazer uso de diagramas da UML e sim de descrições textuais através de uma linguagem de domínio específico para casos de uso.

Budha et al. apresentam um trabalho [57] que se aproxima de CLARET ao implementar requisitos descritos em casos de uso para geração de casos de teste. No entanto, o seu foco é a detecção de falhas de dependências entre os diagramas de casos de uso. Para atingir este objetivo, a metodologia dos autores recorre a uma transformação dos diagramas em uma representação de árvore, que permite checar com maior clareza as faltas de dependência. Apesar de nossa ferramenta empregar casos de uso descritos em formato textual, este trabalho de Budha et al. descortina uma possibilidade de somar esta característica de resolução de falhas de dependências à nossa abordagem em trabalhos futuros.

Os ambientes de desenvolvimento de software são obrigados a lidarem constantemente com mudanças de requisitos em seus projetos. São vários os fatores que desencadeiam mudanças: ameças competitivas, pressões de tempo, tecnologias de desenvolvimento e preferências das partes envolvidas. Muitas vezes os requisitos iniciais se tornam obsoletos antes mesmo do fim do projeto. Em meio a estes desafios, a metodologia ágil surge como um possível norteador e tem conquistado muitos praticantes. Sendo assim, no âmbito da engenharia de requisitos e métodos ágeis, Ramesh et al. apresentam um estudo qualitativo para responder quais são as práticas de ER que os desenvolvedores ágeis empregam e quais os benefícios e desafios de tais práticas [14]. Embora muitos envolvidos na metodologia ágil preguem que é desnecessário fases de planejamento e análise de requisitos formal, estudos recentes [58, 59, 60] relatam que muitos dos problemas em projetos de software poderiam ser ocasionados pela falta de documentos detalhados de requisitos. Em CLARET, adotamos os documentos de requisitos como especificações formais da ER. Contudo a notação simples e o suporte ferramental para geração automática de documentos de casos de uso fazem de CLARET uma ferramenta apta a integrar o espectro das metodologias ágeis.

O trabalho de Inayat et al. propõem reflexões sobre a engenharia de requisitos tradicional e a engenharia de requisitos ágil [12]. Como esse contexto é recente, há poucas informações que indiquem se a filosofia ágil, ao trazer soluções a velhos problemas da engenharia de requisitos, trouxe novos desafios também. Os métodos ágeis incluem a colaboração frequente das partes envolvidas (*stakeholders*), o desenvolvimento iterativo e a aceitação de mudanças de requisitos mesmo em estágios avançados do projeto. Dessa forma, a filosofia ágil oferece uma dinamicidade que satisfaz as necessidades de clientes em menos tempo. O caráter exploratório das metodologias ágeis fazem com que o conhecimento e a ação sejam eventos simultâneos em um projeto [61]. Além disso, os métodos ágeis são guiados pelo valor de negócio e custo (retorno de investimento) e não pela completude dos requisitos implementados [62]. A engenharia de requisitos tradicional compreende as atividades de elicitação, análise, negociação, documentação, validação

e gerenciamento dos requisitos [63]. Todas estas atividades colaboram para o time conhecer o que construirá antes do desenvolvimento iniciar e, consequentemente, evitar custos com o retrabalho. Como resultado da pesquisa qualitativa que os autores realizaram, o artigo em questão expõe as seguintes práticas pelas equipes que adotam a engenharia de requisitos ágil: estórias de usuários, gerenciamento de mudanças, testes de aceitação, refatoração de código, gerenciamento de requisitos, modelagem de requisitos, priorização de requisitos, pareamento para análise de requisitos, dentre outros. Claret se encaixa no contexto das metodologias ágeis, pois emprega um modelo central que se deriva em artefatos para as áreas de MBT e ER simultaneamente. Com isso, poupa-se tempo na criação e manutenção de casos de teste e documentos formais de casos de uso, o que facilita a prática ágil do gerenciamento de mudanças.

Para a discussão do conceito de *Domain-Specific Language* (DSL), focalizamos o trabalho *An approach for the systematic development of domain-specific languages* [64], que apresenta algumas abordagens para a criação de DSL. No entanto, a experiência mostrou que na maioria dos casos, o desenvolvimento é um processo iterativo exploratório. No artigo são explorados os conceitos básicos, assim como os tipos de DSL, e foca mais especificamente em DSL embutidas em linguagens dinâmicas. A nossa pesquisa utiliza algumas das técnicas (modelo de linguagem, comportamento da DSL) citadas na abordagem de desenvolvimento sistemático de DSL do referido artigo.

Neste capítulo, evidenciamos alguns dos trabalhos que compreendem os escopos do teste baseado em modelos e da engenharia de requisitos ágil. No contexto de MBT, os autores apresentam os desafios comuns que são enfrentados e as várias soluções propostas. Problemas como a ambiguidade, a complexidade e a fragilidade dos modelos de requisitos, são contornados com a adoção de CNL, gramáticas de domínio específico e formalismos algébricos.

No que diz respeito à engenharia de requisitos ágil, os estudos ainda são poucos e recentes, no entanto é uma área que tem atraído bastante atenção dos pesquisadores. A filosofia ágil traz novas perspectivas para velhos problemas da ER, ao mesmo tempo que aproveita a maturidade das práticas tradicionais.

Em nosso trabalho, pudemos visualizar uma lacuna existente entre o MBT e a engenharia de requisitos ágil, que é como tornar produtiva a criação e manutenção de casos de teste e de documentos de especificação de casos de uso. É por meio de Claret que propomos nossa solução para geração automática de *(i)* meta-modelos para criação de casos de teste manuais e *(ii)* documentos formais de casos de uso. O ponto de entrada em Claret é um artefato central que contém as especificações dos requisitos, descritos em uma linguagem de caso de uso simples, de fácil legibilidade e capacidade de escrita.

Capítulo 6

Conclusões

O trabalho aqui apresentado propôs a aproximação da engenharia de requisitos com as práticas ágeis, em uma sinergia com o processo de teste baseado em modelo. Neste sentido, nossos esforços visaram responder a questão de pesquisa inicial, evidenciando os benefícios resultantes da associação entre a engenharia de requisitos ágil e o MBT. O caso de uso textual da UML foi o modelo escolhido para a especificação dos requisitos, através de uma notação concisa, com sintaxe voltada ao desenvolvedor e que permitiu capturar as interações entre usuário e sistema de forma consistente.

Os artefatos de saída para a engenharia de requisitos são documentos de textos formatados a partir de um arquivo de *template*, que viabilizou a personalização de *layout* e formatação de texto. Já os artefatos de saída com finalidades de teste, possibilitam a validação visual dos caminhos possíveis dos cenários em ferramentas de suporte a grafos. Estes mesmos artefatos de saídas, quando anotados com informações sobre as transições, permitem a geração de casos de teste na ferramenta LTS-BT.

Dessa forma, o suporte ferramental desenvolvido proporcionou a validação das especificações de requisitos face a sintaxe da notação proposta. Também ofereceu a geração automática de metamodelos para posterior transformação em casos de teste de execução manual, além de documentos formatados para a engenharia de requisitos.

Foram realizados dois experimentos para avaliar os aspectos relacionados à linguagem proposta em Claret e dois estudos de caso que visaram validar o processo proposto em ambientes reais de desenvolvimento de *software*. Os experimentos contemplaram apenas a etapa de especificação de requisitos, nos quais houve uma comparação entre a notação de Claret e a escrita livre para analisar características de legibilidade, capacidade de escrita, cobertura, completude, consistência e tempo. Foi observado que a notação adotada para especificação tem impacto na produtividade e qualidade dos artefatos produzidos. Os resultados obtidos nos deram indícios que a linguagem proposta neste trabalho pode influenciar de forma positiva as especificações de requisitos.

Os estudos de caso objetivaram detectar efeitos não percebidos em um ambiente controlado e avaliar como o processo aqui apresentado se encaixa em um contexto real. Foram

utilizados dois projetos com equipes distintas: o primeiro contemplou uma parte do desenvolvimento e testes de dois sistemas, e o segundo estudo compreendeu a produção de teste e documentação para um sistema legado. Estes estudos nos serviram como fio condutor para validar os objetivos de nosso trabalho e identificar pontos a serem melhorados em trabalhos futuros.

6.1 Contribuições

A literatura possui diversos trabalhos que demonstram os benefícios da aplicabilidade da abordagem de teste baseado em modelos em projetos de *software* e, mais recentemente, estudos que investigam ganhos obtidos com a adição de técnicas ágeis aplicadas na engenharia de requisitos. No entanto, há uma lacuna que é o cenário onde se deseja a adoção do MBT e os modelos da engenharia de requisitos, mas sob a ótica das metodologias ágeis. Nesse contexto, a principal contribuição deste trabalho de dissertação foi a combinação harmoniosa das práticas de engenharia de requisitos, metodologias ágeis e o teste baseado em modelos.

A escrita de requisitos em casos de uso através de uma sintaxe objetiva permite ao especificador focar apenas no conteúdo. Livrar-se de atividades relativas ao *layout* e a formatação de texto, favorece a agilidade no fluxo de trabalho para as especificações. Mais ainda, o emprego de uma gramática que deve ser respeitada e validada, faz com que se minimizem incoerências e consequentemente haja ganhos na qualidade dos artefatos produzidos.

A aplicação da linguagem tem uma curva de aprendizagem minimizada pelo fato de ser embasada nos conceitos já conhecidos dos casos de uso da UML. Desse modo, faz com que se aproveite todo o conhecimento de especificação de requisitos da forma tradicional e não adicione custos extras em termos de treinamento e produtividade. O que observamos em nossos estudos foi justamente o contrário: os ganhos obtidos na qualidade dos artefatos refletiu uma melhor comunicação entre os envolvidos por meio dos documentos gerados automaticamente, além de economia de tempo e custos na criação e manutenção dos casos de teste.

6.2 Trabalhos Futuros

Os estudos de caso foram de grande importância para nos nortear em pontos a serem melhorados nos trabalhos futuros. Dentre os problemas verificados e que podem ser trabalhados, estão:

Suporte ferramental: Melhorias a exemplo de um editor com recursos de geração de estruturas semi-prontas e a sugestão de autocompletar a sintaxe, foi uma das

características mais solicitadas no decorrer dos estudos. Outros pontos-chave que identificamos foram o aperfeiçoamento das mensagens de erros e a localização dessas falhas na especificação. Observamos também que muitos dos erros cometidos pelos especificadores, como por exemplo caminhos ausentes e outros *bad smells*, poderiam ser identificados e sugestões de correções serem oferecidos no decorrer da escrita ou compilação.

Sintaxe da linguagem: Em relação a sintaxe, foram sugeridos que as *strings* que contém as ações nos passos dos cenários, tenham suporte a multilinhas, o que facilitará uma escrita mais extensa e detalhada das interações ator/sistema. Também percebemos que pode haver aperfeiçoamento na identificação do requisito relacionado ao caso de uso, o qual possibilitará a rastreabilidade dos documentos. Outro ponto a ser considerado é o suporte a *includes* e *extends* a outros casos de uso.

Processo de teste: No tocante ao processo de geração de artefatos para teste, tivemos *feedback* no sentido de aperfeiçoar a integração com as demais ferramentas, no caso o LTS-BT para a geração de casos de teste e o *Testlink* para a execução e gerenciamento dos testes manuais. Visamos também possibilidades para a geração automática de artefatos destinados a testes automáticos de sistema, o que proporcionará enormes benefícios em termos de tempo na execução dos casos de teste.

Bibliografia

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008, vol. 54. [Online]. Available: http://www.amazon.com/Introduction-Software-Testing-Paul-Ammann/dp/0521880386

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

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[4] I. K. El-far and J. a. Whittaker, "Model-Based Software Testing," *Encyclopedia of Software Engineering*, pp. 1–22, 2001. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/0471028959.sof207/full

[5] M. Sarma and R. Mall, "Automatic Test Case Generation from UML Models," *10th International Conference on Information Technology (ICIT 2007)*, pp. 196–201, 2007.

[6] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of UML sequence diagrams and labeled transition systems," *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, pp. 1292–1297, 2007.

[7] M. Dhineshkumar and Galeebathullah, "An Approach to Generate Test Cases from Sequence Diagram," *2014 International Conference on Intelligent Computing Applications*, pp. 345–349, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6965069

[8] R. Löffler, M. Meyer, and M. Gottschalk, "Formal scenario-based requirements specification and test case generation in healthcare applications," *Proceedings of the 2010 ICSE Workshop on Software Engineering in Health Care - SEHC '10*, pp. 57–67, 2010. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-77954947794{&}partnerID=tZOtx3y1

[9] M. Shirole and R. Kumar, "UML behavioral model based test case generation," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, p. 1, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2492248.2492274

[10] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "NAT2TESTSCR : Test case generation from natural language requirements based on SCR specifications," *Science of Computer Programming*, vol. 95, pp. 275–297, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2014.06.007

[11] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic Generation of System Test Cases from Use Case Specifications," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771812

[12] I. Inayat, L. Moraes, M. Daneva, and S. S. Salim, "A Reflection on Agile Requirements Engineering : Solutions Brought and Challenges Posed," *Workshop on Requirements Engineering in Agile Development*, 2015.

[13] G. Sillitti, Alberto and Succi, "Requirements engineering for agile methods," *Engineering and Managing Software Requirements*, pp. 309—-326, 2005.

[14] B. Ramesh, L. Cao, and R. Baskerville, "Agile requirements engineering practices and challenges: an empirical study," *Information Systems Journal*, vol. 20, no. 5, pp. 449–480, nov 2007. [Online]. Available: http://doi.wiley.com/10.1111/j.1365-2575.2007.00259.x

[15] R. S. Scowen, "Extended BNF - A generic base standard," *Software Engineering Standards Symposium*, vol. 3, no. 1, pp. 6–2, 1998.

[16] J. Katoen and J. Katoen, "Concepts, algorithms, and tools for model checking," *Arbeitsberichte des*, vol. 2501, no. 32, p. 1, 1999. [Online]. Available: http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Concepts,+Algorithms,+and+Tools+for+Model+Checking{#}0

[17] J. D. McGregor and D. A. Sykes, *A practical guide to testing object-oriented software*. Addison-Wesley, 2001. [Online]. Available: https://books.google.com.br/books?hl=pt-BR{&}lr={&}id=RxwX6yk72YoC{&}oi=fnd{&}pg=PR13{&}dq=A+practical+guide+to+testing+object-+oriented+software.{&}ots=x-CBmTGrzK{&}sig=nf6RAUrU5Oj569WMUBGwJM{_}X0v0{#}v=onepage{&}q=Apracticalguidetotestingobject-orientedsoftware.{&}f=false

[18] F. ISTQB, "Foundation level syllabus version 2011," *International Software Testing Qualifications Board*, 2011.

[19] P. Jorgensen and P. C., *Software testing : a craftsman's approach.* Auerbach Publications, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=2187790{&}CFID=768324456{&}CFTOKEN=38577295

[20] J. Tretmans, "Model-based testing: Property checking for real," in *International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices*, 2004.

[21] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing.* New York, NY, USA: John Wiley & Sons, Inc., 2001.

[22] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2005.

[23] E. Arisholm, L. Briand, S. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: an experimental evaluation," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 365–381, jun 2006. [Online]. Available: http://ieeexplore.ieee.org/document/1650213/

[24] I. Sommerville, *Software engineering*, 2011. [Online]. Available: http://eprints.lancs.ac.uk/12401/

[25] IEEE, "IEEE 24765:2010 - Systems and software engineering – Vocabulary," 2010.

[26] R. Y. Lee, *Software Engineering: A Hands-On Approach.* Springer Science & Business Media, 2013.

[27] P. A. Laplante, *Requirements engineering for software and systems.* CRC Press, 2013.

[28] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - Ireb Compliant*, ser. Rocky Nook computing. Rocky Nook, 2015. [Online]. Available: https://books.google.com.br/books?id=bM1YrgEACAAJ

[29] IEEE, "IEEE 830: Recommended Practice for Software Requirements Specifications," Tech. Rep., 1998.

[30] D. Leffingwell, *Agile software requirements: lean requirements practices for teams, programs, and the enterprise.* Addison-Wesley Professional, 2010.

[31] M. Cohn, *User Stories Applied: For Agile Software Development (Addison Wesley Signature Series)*, 2004, vol. 1. [Online]. Available: http://books.google.com/books?id=SvIwuX4SVigC{&}pgis=1

[32] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, 1992, vol. 2640. [Online]. Available: http://www.mendeley.com/research/objectoriented-software-engineering-a-use-case-driven-approach/

[33] P. Machado and A. Sampaio, "Automatic test-case generation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6153 LNCS, 2010, pp. 59–103.

[34] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects," *Journal of Systems and Software*, vol. 81, no. 6, pp. 961 – 971, 2008, agile Product Line Engineering. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121207002208

[35] I. Inayat, L. Moraes, M. Daneva, and S. S. Salim, "A reflection on agile requirements engineering: Solutions brought and challenges posed," in *Scientific Workshop Proceedings of the XP2015*, ser. XP '15 workshops. New York, NY, USA: ACM, 2015, pp. 6:1–6:7. [Online]. Available: http://doi.acm.org/10.1145/2764979.2764985

[36] E. C. Cartaxo, W. Andrade, F. Neto, and P. Machado, "LTS-BT: a tool to generate and select functional test cases for embedded systems," *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 1540–1544, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1364045

[37] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of Software Engineering*, vol. 2, pp. 528–532, 1994. [Online]. Available: http://maisqual.squoring.com/wiki/index.php/TheGoalQuestionMetricApproach

[38] T. Yue, S. Ali, and M. Zhang, "RTCM: a natural language based, automated, and practical test case generation framework," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*. New York, New York, USA: ACM Press, 2015, pp. 397–408.

[39] G. Cabral and A. Sampaio, "Formal Specification Generation from Requirement Documents," *Electronic Notes in Theoretical Computer Science*, vol. 195, no. C, pp. 171–188, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2007.08.032

[40] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for UML activity diagrams," *Proceedings of the 2006 international workshop on Automation of software test - AST '06*, p. 2, 2006.

[41] R. V. Binder, B. Legeard, and A. Kramer, "Model-based Testing: Where Does It Stand?" *Queue*, vol. 13, no. 1, p. 40, 2014.

[42] M. Hesenius, T. Griebe, and V. Gruhn, "Towards a behavior-oriented specification and testing language for multimodal applications," *EICS 2014 - Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pp. 117–122, 2014.

[43] Gherkhin, https://github.com/cucumber/cucumber/wiki/Gherkin, acessado em: 2017-03-22.

[44] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.

[45] T. Kosar, M. Mernik, and J. C. Carver, "Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments," *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, 2012.

[46] M. J. Escalona, J. J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres, and F. J. Domínguez, "The Journal of Systems and Software An overview on test generation from functional requirements," *The Journal of Systems & Software*, vol. 84, no. 8, pp. 1379–1393, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.03.051

[47] C. Jard and T. Jéron, "TGV: Theory, principles and algorithms. A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 297–315, 2005.

[48] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 1–38, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2430539

[49] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[50] S. Nogueira, A. Sampaio, and A. Mota, "Test generation from state based use case models," *Formal Aspects of Computing*, pp. 1–50, 2012.

[51] ——, *Guided Test Generation from CSP Models*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 258–273. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85762-4_18

[52] G. Cabral, A. Sampaio, and J. S. P. Brazil, "Automated Formal Specification Generation and Refinement from Requirement Documents."

[53] E. Sarmiento, J. C. S. D. P. Leite, and E. Almentero, "C&L: Generating model based test cases from natural language requirements descriptions," *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, pp. 32–38, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6908677

[54] Papyrus, https://eclipse.org/papyrus, acessado em: 2017-04-22.

[55] O. S. Specification, http://www.sysml.org/specs.htm, acessado em: 2017-04-22.

[56] Modelio, http://www.modelio.org, acessado em: 2017-04-22.

[57] G. Budha, N. Panda, and A. A. Acharya, "Test case generation for use case dependency fault detection," in *2011 3rd International Conference on Electronics Computer Technology*. IEEE, apr 2011, pp. 178–182.

[58] F. Paetsch, A. Eberlein, and F. Maurer, "Requirements engineering and agile software development," *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.*, pp. 308–313, 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1231428

[59] B. Boehm, "Requirements that handle IKIWISI, COTS, and rapid change," *Computer*, vol. 33, no. 7, pp. 99–102, 2000.

[60] P. Grünbacher and C. Hofer, "Complementing XP with requirements negotiation," *Proceedings 3rd Int. Conf. Extreme Programming and Agile Processes in Software Engineering*, pp. 1–4, 2002. [Online]. Available: http://cf.agilealliance.org/articles/system/article/file/909/file.pdf

[61] J. Erickson, K. Lyytinen, and K. Siau, "Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research," *Journal of Database Management*, vol. 16, no. 4, pp. 88–100, 2005.

[62] N. A. Ernst, A. Borgida, I. J. Jureta, and J. Mylopoulos, "Agile requirements engineering via paraconsistent reasoning," *Information Systems*, vol. 43, pp. 100–116, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.is.2013.05.008

[63] M. Cohn, *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[64] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Software - Practice and Experience*, vol. 39, no. 15, pp. 1253–1292, 2009.

Apêndice A

Documento de Requisitos

A.1 Requisitos do Usuário

RF01: Um sistema escolar deverá verificar diariamente se a quantidade de faltas dos alunos ultrapassou um limite estabelecido pela coordenação e enviará um SMS aos responsáveis dos alunos informando o número de faltas total. O sistema enviará somente um SMS a cada falta computada além do limite permitido, além disto, os responsáveis dos alunos deverão autorizar o envio e informar um número de telefone. O sistema deverá ainda possuir uma mensagem padrão cadastrada previamente.

A.2 Requisitos Esperados

1. Ator: temporizador, timer, escalonador, etc; (1 ponto)
2. Periodicidade diariamente; (1 ponto)
3. Quantidade de faltas dos alunos ultrapassou o limite; (1 ponto)
4. Um SMS a cada nova falta além do limite estabelecido; (1 ponto)
5. Número de faltas estabelecido; (1 ponto)
6. Mensagem padrão cadastrada para envio do SMS; (1 ponto)
7. Autorização dos responsáveis do aluno para de envio de SMS; (1 ponto)
8. Existir um número de celular cadastrado para envio de SMS; (1 ponto)
9. Pós-condição: Responsáveis dos alunos são avisados das faltas; (1 ponto)

Total de pontos: 9

Apêndice B

Resultados do Experimento 2

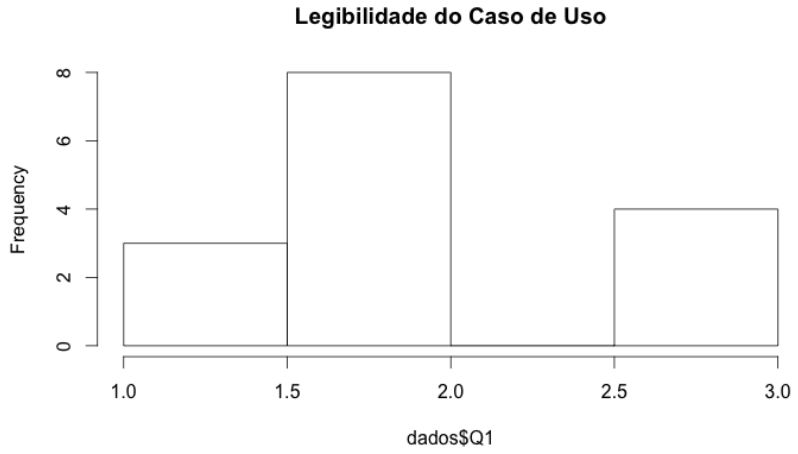


Figura B.1: Histograma para Q1

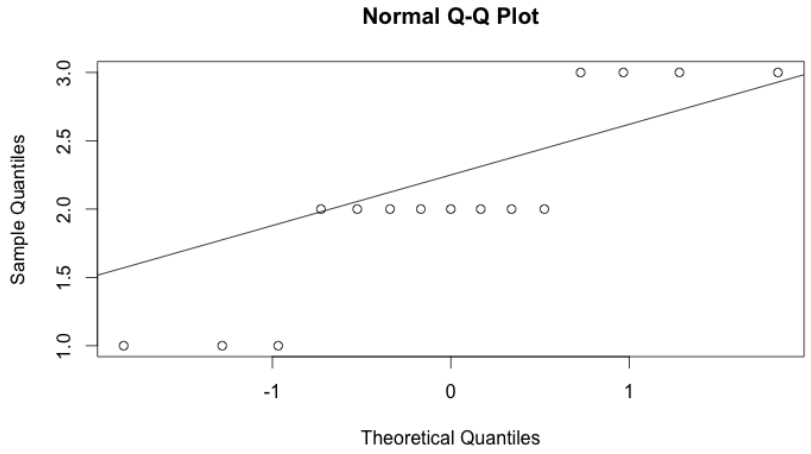


Figura B.2: Q-Q Plot para Q1

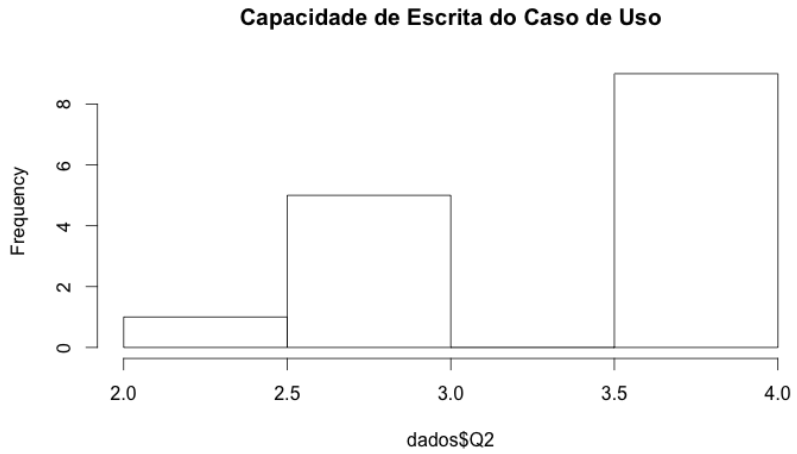


Figura B.3: Histograma para Q2

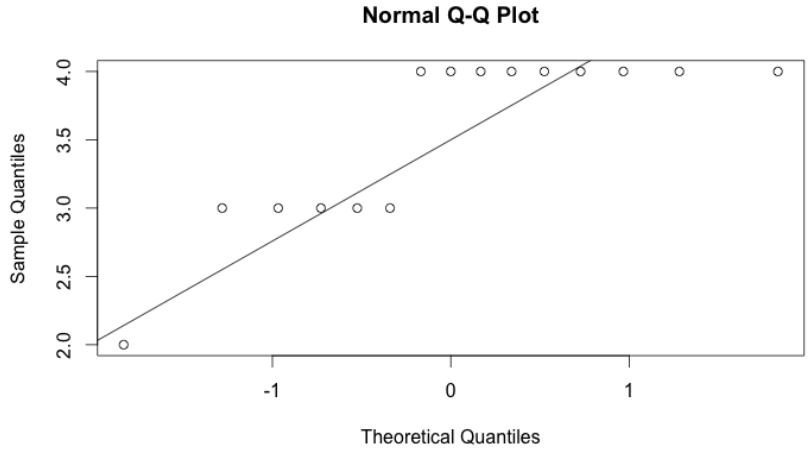


Figura B.4: Q-Q Plot para Q2

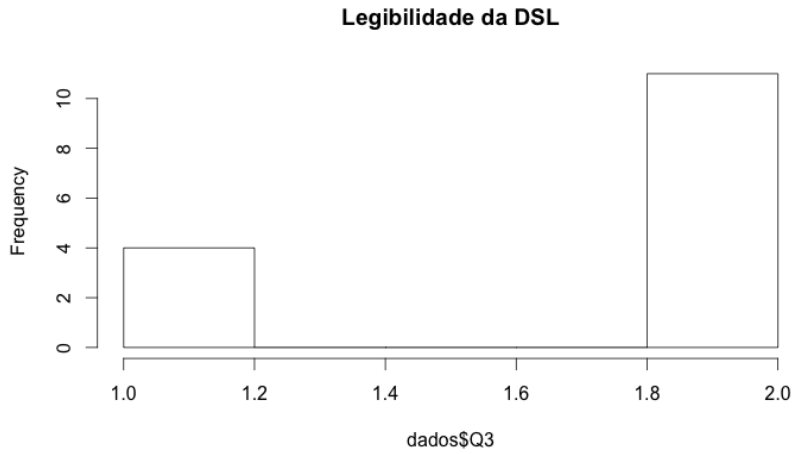


Figura B.5: Histograma para Q3

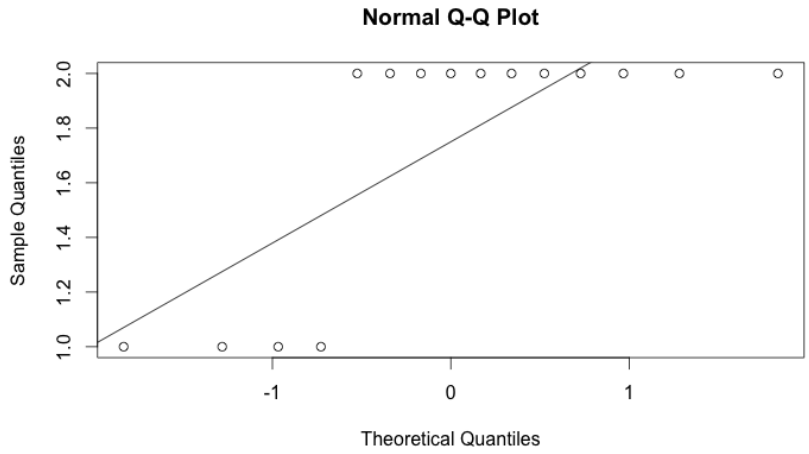


Figura B.6: Q-Q Plot para Q3

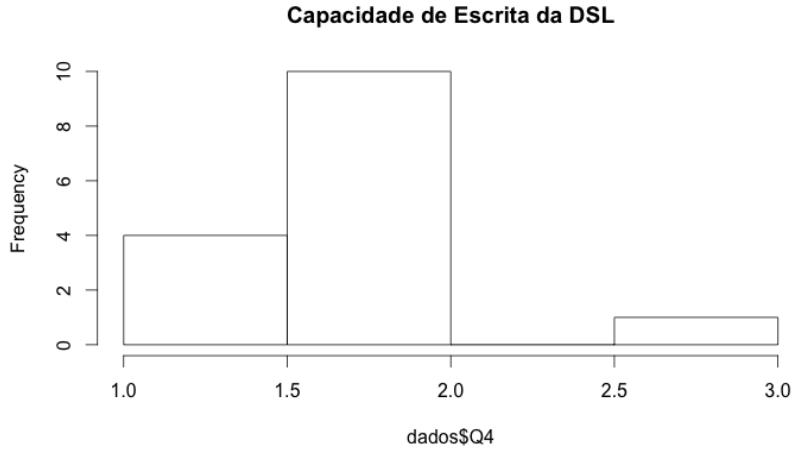


Figura B.7: Histograma para Q4

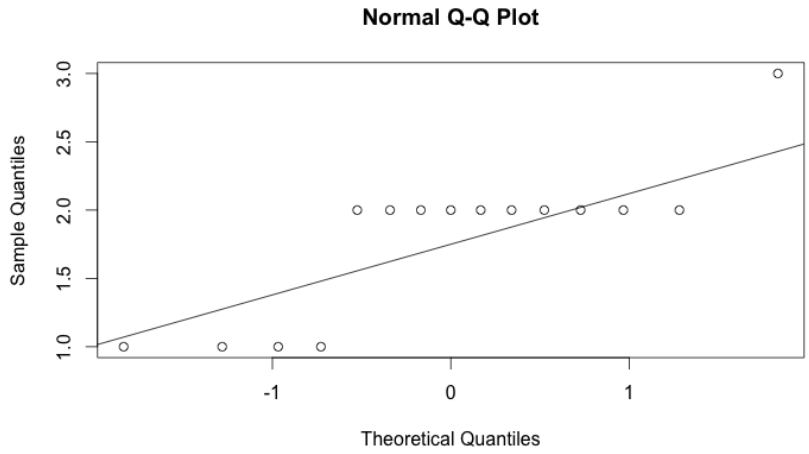


Figura B.8: Q-Q Plot para Q4

Apêndice C

Questionários

Por questões de consciência ambiental, disponibilizamos os questionários de forma eletrônica e os endereços estão informados nas seções seguintes. É pertinente frisar que, apesar de estarem ativos e recebendo respostas, os questionários e respostas são apenas cópias dos originais.

C.1 Experimento 2

- **Questionário**: `https://goo.gl/rLm8B8`
- **Respostas**: `https://goo.gl/FX6mFk`

C.2 Estudo de caso 2

- **Questionário**: `https://goo.gl/bEoKwY`
- **Respostas**: `https://goo.gl/KKZ2C7`