PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

Márcio Assis Miranda

UMA ABORDAGEM PARA GERAÇÃO AUTOMÁTICA DE DIAGRAMAS DE CLASSES E SEQUÊNCIA A PARTIR DA ESPECIFICAÇÃO DE CASOS DE USO

Belo Horizonte

2016

Márcio Assis Miranda

UMA ABORDAGEM PARA GERAÇÃO AUTOMÁTICA DE DIAGRAMAS DE CLASSES E SEQUÊNCIA A PARTIR DA ESPECIFICAÇÃO DE CASOS DE USO

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. Mark Alan Junho Song

Coorientador: Prof. Dr. Humberto Torres Marques Neto

Belo Horizonte

2016

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Miranda, Márcio Assis
M672u Uma abordagem para geração automática de diagramas de classes e sequência a partir da especificação de casos de uso / Márcio Assis Miranda. Belo Horizonte, 2016.
134 f. : il.

Orientador: Mark Alan Junho Song
Coorientador: Humberto Torres Marques Neto
Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática.

1. Engenharia de sistemas. 2. Engenharia de software. 3. Linguagem de programação (Computadores). 4. Estruturas de dados (Computação). I. Song, Mark Alan Junho. II. Marques Neto, Humberto Torres. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. IV. Título.
SIB PUC MINAS
CDU: 681.3.03

Márcio Assis Miranda

UMA ABORDAGEM PARA GERAÇÃO AUTOMÁTICA DE DIAGRAMAS DE CLASSES E SEQUÊNCIA A PARTIR DA ESPECIFICAÇÃO DE CASOS DE USO

Dissertação apresentada ao Programa de Pós-Graduação em Informática como requisito parcial para qualificação ao Grau de Mestre em Informática pela Pontifícia Universidade Católica de Minas Gerais.

Prof. Dr. Mark Alan Junho Song – PUC Minas (Orientador)

Prof. Dr. Humberto Torres Marques Neto – PUC Minas (Coorientador)

Prof. Dr. Fernando Silva Parreiras – FUMEC (Banca Examinadora)

Prof. Dr. Luis Enrique Zárate Gálvez – PUC Minas (Banca Examinadora)

Belo Horizonte, 09 de Novembro de 2016.

Dedico este trabalho a Deus, aos meus pais e minha família, pois sem Eles ao meu lado, não seria possível realizar este grande sonho.

AGRADECIMENTOS

Agradeço primeiramente a DEUS por me conceder saúde, determinação, persistência e sabedoria na realização deste trabalho.

À minha filha Júlia, presente de Deus na minha vida.

À minha esposa Raquel, pela amizade, respeito, amor e companheirismo, principalmente nos momentos mais difíceis.

Aos meus pais, pelo amor incondicional, pelos conselhos, pelo carinho e incentivo na busca dos meus objetivos.

Ao meu irmão Milton e minha irmã Marli, pela força e incentivo.

Aos amigos Ezequiel e Talles, pela parceira, amizade e disposição para os estudos.

Aos amigos André Moreira e Daniel Mendes, que cederam suas casas, abrindo mão do conforto e liberdade, para me acolher.

Aos alunos Marcos Guilherme e Thiago Henrique, pela parceria e comprometimento no desenvolvimento deste trabalho.

Aos amigos Ronaldo Ribeiro, Balbino Soares e André Marlon, por permitirem a realização de testes reais.

Aos meus professores orientadores Mark Alan Junho Song e Humberto Torres Marques Neto, pela confiança, dedicação, disponibilidade e orientação firme durante o mestrado. Por todo o incentivo e direcionamento que me passaram, para que eu pudesse desenvolver este trabalho da melhor maneira possível.

À secretária Giovana, juntamente com toda sua equipe, sempre atenciosos e pacientes.

A todos os professores do Mestrado em Informática da PUC Minas, pelo profissionalismo, dedicação e conhecimento repassado.

A todos que contribuíram no desenvolvimento deste trabalho.

"Determinação, coragem e autoconfiança são fatores decisivos para o sucesso. Não importa quais sejam os obstáculos e as dificuldades. Se estamos possuídos de uma inabalável determinação, conseguiremos superá-los. Independentemente das circunstâncias, devemos ser sempre humildes, recatados e despidos de orgulho."

Dalai Lama

RESUMO

Engenharia de requisitos de software é o processo de levantar, especificar, documentar e manter os requisitos de usuário e de sistema, gerando como artefato o documento de requisitos. Essa é considerada uma fase essencial no desenvolvimento de software. Erros causados durante estas atividades, podem ser custosos para serem consertados em fases posteriores, comprometendo o sucesso do projeto. A utilização de linguagens de domínio específico vem ganhando destaque no processo de análise e levantamento dos requisitos de software, por estabelecer uma comunicação padronizada da equipe, permitir automatizar fases do processo e por possibilitar um ganho na produtividade sem comprometer a qualidade. Neste trabalho foi proposta e implementada a *Language of Use Case to Automate Models* (LUCAM), uma linguagem de domínio específico que possibilita especificar textualmente casos de uso e por meio da ferramenta LUCAMTool, gerar automaticamente os diagramas de casos de uso, classes e sequência. Para verificar a viabilidade da solução proposta, foram realizados testes em ambiente simulado e real, buscando contemplar os variados cenários do desenvolvimento de sistemas. A abordagem auxilia na análise e modelagem de requisitos e busca minimizar os problemas apresentados nas especificações em linguagem natural, tais como: incerteza, ambiguidade, complexidade e dependência de conhecimento do domínio por parte dos especialistas.

Palavras-chave: Automatically Generating. Class Diagram. Domain-Specific Language. Sequence Diagram. Use Case Specification.

ABSTRACT

Requirements Software Engineering is the process of analyzing, specifying, documenting and managing user and system requirements, generating the requirement document as its main artifact. It is considered to be essential and complex stage of software development. Errors generated during this stage may be very costly to fix in further stages, compromising the success of the project. The use of domain-specific languages has been gaining traction in the process of analyzing and identifying software requirements since it establishes standardized team communication, enables the automation of some stages of the process and results in productivity gains with no impact in quality. In this paper, we propose and implement LUCAM (*Language of Use Case to Automate Models*), a domain-specific language which allows users to textually specify requirements and, using LUCAMTool, automatically generate use case, class and sequence diagrams. To assess the viability of the proposed solution, we conducted several tests in both simulated and real environments, seeking to encompass a variety of software development scenarios. Our solution helps in requirement analysis and modeling, and strives to mitigate problems that occur in natural language specifications, such as: uncertainty, ambiguity, complexity and reliance on specialists' knowledge of the domain.

Keywords: Automatic Generation. Class Diagram. Domain-Specific Language. Sequence Diagram. Use Case Specification.

LISTA DE FIGURAS

FIGURA 1 – Transição da LUCAM para os diagramas UML 36
FIGURA 2 – Fluxo geral da LUCAM 37
FIGURA 3 – Diagrama de sequência do fluxo AddAccount - Parte 1 51
FIGURA 4 – Diagrama de sequência do fluxo AddAccount - Parte 2 52
FIGURA 5 – Diagrama de sequência AddAccount - UC SB_BankAccount 53
FIGURA 6 – Diagrama de sequência CloseAccount - UC SB_BankAccount 55
FIGURA 7 – Diagrama de classes - SB_BankAccount 56
FIGURA 8 – Ferramenta LUCAMTool 58
FIGURA 9 – Fluxo de testes com a LUCAM 61
FIGURA 10 – Funcionalidades - Testes reais 63
FIGURA 11 – Tempo de especificação e modelagem - Entidade Simples 64
FIGURA 12 – Tempo de especificação e modelagem - Entidade Composta 65
FIGURA 13 – Diagrama de classes - SB_BankAccount 89
FIGURA 14 – Diagrama de sequência AddAccount - UC SB_BankAccount 90
FIGURA 15 – Diagrama de sequência CloseAccount - UC SB_BankAccount 91
FIGURA 16 – Diagrama de sequência Withdraw - UC SB_BankAccount 92
FIGURA 17 – Diagrama de sequência MakeTransfer - UC SB_BankAccount 93
FIGURA 18 – Class Diagram UC CRUDPatient 96
FIGURA 19 – Sequence Diagram AddPatient 98
FIGURA 20 – Sequence Diagram ModifyPatient 99
FIGURA 21 – Sequence Diagram DisablePatient 100
FIGURA 22 – Sequence Diagram ReadPatient 101
FIGURA 23 – Diagrama de classes - SA_StudentEnrollment 103

FIGURA 24 – Diagrama de classes - SA_StudentEnrollment 104
FIGURA 25 – Diagrama de classes do caso de uso PIV_SGC_MaintainCivilWorks . . 108
FIGURA 26 – Diagrama de sequência do fluxo *AddCivilWorks* 109
FIGURA 27 – Diagrama de sequência do fluxo *ModifyCivilWorks* 110
FIGURA 28 – Diagrama de sequência do fluxo *DisabledCivilWorks* 111
FIGURA 29 – Diagrama de sequência do fluxo *SearchCivilWorks* 112
FIGURA 30 – Diagrama de classe - UC *PIV_SGC_WeighingLoadConcrete* 115
FIGURA 31 – Diagrama de sequência do fluxo *MaintainWeighingLoadingConcrete* . 116
FIGURA 32 – Diagrama de classes - UC *CNB_SCOF_FormRequest* 120
FIGURA 33 – Diagrama de sequência do fluxo *AddFormRequest* 121
FIGURA 34 – Diagrama de sequência do fluxo *AttachStandardizedFile* 122
FIGURA 35 – Diagrama de sequência do fluxo *ApprovedStandardizedDocument* 123
FIGURA 36 – Diagrama de sequência do fluxo *FinalizeRequest* 124
FIGURA 37 – Diagrama de sequência do fluxo *CancelFormRequest* 125
FIGURA 38 – Diagrama de sequência do fluxo *SearchFormRequest* 126
FIGURA 39 – Diagrama de classes do UC CNB_SCOF_RelSupplier 128
FIGURA 40 – Diagrama de sequência do fluxo CNB_SCOF_RelSupplier 128

LISTA DE TABELAS

TABELA 1 – Relação de funcionalidades dos sistemas selecionados 62

LISTA DE QUADROS

QUADRO 1 – Artigos selecionados 31

QUADRO 2 – Retorno do método *identificaOracao()* 41

QUADRO 3 – Padrões de sentenças da LUCAM 41

QUADRO 4 – Convenções léxicas da LUCAM 44

QUADRO 5 – Nomeclaturas dos identificadores da LUCAM 44

QUADRO 6 – Símbolos especiais da LUCAM 44

LISTA DE CÓDIGOS

LISTAGEM 4.1 – Template LUCAM 38
LISTAGEM 4.2 – Cabeçalho - UC SB_BankAccount 47
LISTAGEM 4.3 – Fluxo AddAccount - UC SB_BankAccount 49
LISTAGEM 4.4 – Fluxo alternativo CloseAccount - UC SB_BankAccount 54
LISTAGEM 4.5 – Rodapé - UC SB_BankAccount 57
LISTAGEM C.1 – Especificação do caso de uso - SB_BankAccount 86
LISTAGEM D.1 – Cabeçalho - UC SCM_CRUD_Patient 94
LISTAGEM D.2 – Fluxo Principal - UC SCM_CRUD_Patient 94
LISTAGEM D.3 – Fluxos Alternativos - UC SCM_CRUD_Patient 95
LISTAGEM E.1 – Especificação do caso de uso SA_StudentEnrollment 102
LISTAGEM F.1 – Especificação do caso de uso PIV_SGC_MaintainCivilWorks 105
LISTAGEM G.1 – Especificação do caso de uso PIV_SGC_WeighingLoadingConcrete 113
LISTAGEM H.1 – Especificação do caso de uso CNB_SCOF_FormRequest 117
LISTAGEM I.1 – Especificação do caso de uso CNB_SCOF_RelSupplier 127

LISTA DE ABREVIATURAS E SIGLAS

DSL – *Linguagem de Domínio Específico*

DSLs – *Linguagens de Domínio Específico*

EBNF – *Extended Backus-Naur Form*

EJB – *Enterprise Java Beans*

ERP – *Enterprise Resource Planning*

GPL – *General Purpose Language*

IDE – *Integrated Development Environment*

LATUC – *LAnguage of Textual Use Cases*

LRC – *Last Receiver Class*

LUCAM – *Language of Use Case to Automate Models*

MDD – *Model Driven Development*

MVC – *Model-View-Controller*

RUP – *Rational Unified Process*

SAGSA – *Semi-Automatic Generator of Software Artifacts*

UML – *Unified Modeling Language*

XMI – *Extensible Markup Language Metadata Interchange*

XML – *EXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO **17**
1.1 Visão geral do problema 17
1.2 Objetivos 18
1.3 Visão sucinta da solução proposta 18
1.4 Estrutura do texto 20

2 CONCEITOS FUNDAMENTAIS **21**
2.1 Casos de uso 21
2.2 Diagrama de classes 22
2.3 Diagrama de sequência 23
2.4 Desenvolvimento Dirigido a Modelos (MDD) 24
2.5 DSLs - Linguagens de Domínio Específico 26
2.5.1 Conceitos 26
2.5.2 Motivação para usar Linguagens de Domínio Específico (DSLs) 27
2.5.3 Problemas com DSLs 28
2.5.4 Como projetar DSLs 29
2.6 Considerações finais 29

3 TRABALHOS RELACIONADOS **31**

4 ABORDAGEM PROPOSTA **36**
4.1 Template da especificação na LUCAM 38
4.2 Elementos preliminares da especificação 39
4.3 Módulo Mapear Operações 40
4.4 A linguagem LUCAM 43
4.4.1 Convenções léxicas da LUCAM 43
4.4.2 Regras gramaticais da LUCAM 44
4.5 A ferramenta LUCAMTool 57
4.6 Considerações finais 59

5 PROVA DE CONCEITO E ESTUDO DE VIABILIDADE 60
5.1 Prova de conceito e testes 60
5.2 Análise dos resultados 63
6 CONCLUSÃO 68
6.1 Contribuições 69
6.2 Trabalhos Futuros 70
REFERÊNCIAS 72
APÊNDICE A – QUESTIONÁRIO 77
APÊNDICE B – GRAMÁTICA LUCAM 80
APÊNDICE C – CASO DE USO “CONTA BANCÁRIA” 86
APÊNDICE D – CASO DE USO “MANTER PACIENTE” 94
APÊNDICE E – CASO DE USO “MATRICULAR ESTUDANTE” 102
APÊNDICE F – CASO DE USO “MANTER OBRAS CIVIS” 105
APÊNDICE G – CASO DE USO “CARREGAMENTO DE CONCRETO” 113
APÊNDICE H – CASO DE USO “REQUISIÇÃO DE FORMULÁRIO” . 117
APÊNDICE I – CASO DE USO “RELATÓRIO DE FORNECEDOR” ... 127
APÊNDICE J – GRAMÁTICA IMPLEMENTADA NO XTEXT 129
ANEXO A – TEMPLATE OPENUP 134

1 INTRODUÇÃO

Neste capítulo introdutório será apresentado de forma sucinta o problema abordado nesta pesquisa, o objetivo geral e os específicos, resumidamente descreve-se a solução proposta e os resultados alcançados, e por fim, a estrutura do texto desta dissertação.

1.1 Visão geral do problema

Ter um processo de engenharia de requisitos definido é fundamental para desenvolver softwares com qualidade. Entender as propriedades de um problema e especificá-las, são tarefas difíceis enfrentadas pelos engenheiros de software, e às vezes são negligenciadas por estes. Os requisitos de software são com frequência, artefatos esquecidos, pois os desenvolvedores preferem avançar com a codificação sem compreender o problema a ser resolvido (BROWNE; RAMESH, 2002; TUN et al., 2009; SULTANOV; HAYES, 2010; SAVIC et al., 2011; GUPTA, 2015).

No decorrer do desenvolvimento, engenheiros de requisitos em parceira com clientes e usuários, realizam a negociação do escopo e do comportamento do software, gerando a especificação de requisitos, que é uma descrição detalhada do comportamento do sistema e dos usuários, considerando as interações entre eles.

A captura e mapeamento dos requisitos na maioria das vezes é especificada em casos de uso textuais, visando possibilitar também o entendimento de leigos. No entanto, a sua utilização inerente da linguagem natural pode apresentar problemas decorrentes de ambiguidade, redundância, inconsistência e incompletude, afetando a qualidade dos artefatos gerados (ACHOUR et al., 1999; COCKBURN, 2000; JAYARAMAN; WHITTLE, 2007; TIWARI; GUPTA, 2015).

Existe um distanciamento entre a especificação de casos de uso e a implementação, o que reproduz resultados não satisfatórios e tardios. Procedimentos fundamentais para geração dos modelos a partir da especificação de requisitos em nível de sistema continuam com atividades manuais e dependentes de conhecimento dos especialistas (EL-ATTAR, 2011). Outro problema existente em projeto de software é a falha na comunicação entre analistas, clientes e usuários. Porém, espera-se que ambos tenham um vocabulário comum, possibilitando expressar conceitos do domínio da aplicação de forma que todos sejam interpretados corretamente.

Uma das formas de resolver o problema é representar requisitos de maneira com-

preensível para todos os participantes do projeto, possibilitando automatizar parte do processo (SAVIC et al., 2011). A utilização de metodologias e ferramentas na execução de um processo de software disciplinado, tende a melhorar a produtividade da equipe, reduzir os custos e os riscos associados com o desenvolvimento e melhorar a qualidade dos produtos (GHOSH, 2010). A aplicabilidade de *Linguagem de Domínio Específico* (DSL) vem ganhando destaque, pois, apesar de ser considerado um método com um foco limitado, permite aprimorar a produtividade, a comunicação entre engenheiros e especialistas de domínio, eliminando inclusive a ambiguidade da linguagem natural. A legibilidade e compreensibilidade são características fundamentais para justificar a utilização desse recurso na descrição de casos de uso (FOWLER; PARSONS, 2011).

1.2 Objetivos

Nesta seção será apresentado o objetivo geral e os objetivos específicos que norteiam este trabalho de pesquisa.

O objetivo geral é apresentar uma abordagem que define um padrão para especificar casos de uso e que automatize parcialmente o processo de desenvolvimento de softwares orientados a requisitos textuais, possibilitando a geração de diagramas (*Unified Modeling Language* (UML)) estruturais e comportamentais.

Especificamente propõe-se:

- definir uma DSL para representar a especificação de casos de uso textuais;
- implementar uma ferramenta de apoio para gerar automaticamente diagramas de casos de uso, classes e sequência a partir da especificação de casos de uso;
- demonstrar a expressividade da DSL proposta com a realização de testes em ambiente simulado e real; e
- realizar um estudo de viabilidade da abordagem proposta para especificações de requisitos funcionais.

1.3 Visão sucinta da solução proposta

A modelagem de casos de uso é uma estratégia essencial para documentação de software e permite ao analista especificar as funcionalidades pretendidas pelos usuários. Essa técnica foi introduzida por Jacobson (1987) e posteriormente adotada na UML (OMG, 2015b)*.

*OMG: *Object Management Group* ®

O propósito dos casos de uso é delinear os requisitos de tal forma que as pessoas com menos conhecimento técnico (por exemplo, clientes e usuários) possam compreendê-los e analisá-los. Geralmente, casos de uso consistem em duas partes: o diagrama e as descrições textuais. A primeira representa a estrutura dos casos de uso (atores, funções e as relações entre si). A segunda o comportamento e o fluxo do processo.

Portanto, nota-se que especificar casos de uso em linguagem natural, pode gerar falhas no processo, conforme descrito na primeira seção desse capítulo. Fowler e Parsons (2011) destacam a utilização de DSL como uma alternativa para reduzir as falhas na modelagem e definição dos artefatos, possibilitando a equiparação dos artefatos desejados com artefatos gerados. DSL permite também obter ganhos significativos na produtividade e qualidade do produto final, proporcionando uma maior aceitação dos clientes ao adquirirem produtos que atendam suas necessidades e expectativas (GHOSH, 2010).

Neste trabalho é proposta e implementada uma DSL externa denominada *Language of Use Case to Automate Models* (LUCAM), a qual define padrões para uma linguagem que permite analistas de requisitos especificar casos de uso textuais. Foi proposta também uma ferramenta denominada LUCAMTool que contempla a geração automática dos diagramas de casos de uso, classes e sequência a partir da especificação de casos de uso.

A ferramenta faz o mapeamento da especificação para o padrão de entrada *Extensible Markup Language Metadata Interchange* (XMI), reconhecido por softwares de modelagem UML, como *Astah*[†] e outros, levando em consideração as regras definidas para a linguagem proposta. Outro recurso existente é a geração automática e direta de diagramas para arquivos de extensão *.astah*, possibilitando ao usuário já visualizá-los, caso tenha o software instalado em seu computador.

Procurou-se durante o projeto da DSL e na implementação da LUCAMTool, guiar-se por um conjunto de requisitos relevantes, para atender características como simplicidade e objetividade, de modo a oferecer uma sintaxe autoexplicativa e intuitiva para o usuário.

O cenário escolhido para apresentar os recursos e a expressividade da linguagem LUCAM é de um sistema bancário, com o caso de uso "*SB_BankAccount*". O gerente como ator primário é o responsável por manter os dados dos clientes e de suas contas. Já o cliente como ator secundário tem o papel de repassar ao atendente seus dados para que sejam cadastrados no sistema e poderá também realizar movimentações como saques e transferências.

Os testes foram realizados em ambiente simulado e real, sendo o segundo conjunto executado na forma de prova de conceito em projetos escolhidos pelas empresas colaboradoras. A abordagem proposta apresenta recursos que contribuem para a padronização das

[†]Astah: <*http://astah.net/*>

etapas do desenvolvimento de sistemas, como a fase de especificação e modelagem de casos de uso, possibilita uma melhora na produtividade, permite uma comunicação padronizada da equipe e minimiza os problemas derivados da linguagem natural. Ao contrário do que acontece com a maioria das especificações executadas manualmente, os diagramas gerados pela LUCAMTool refletem fielmente ao que foi especificado e são consistentes entre si.

1.4 Estrutura do texto

Além deste capítulo introdutório, o restante do trabalho está divido da seguinte forma:

- No capítulo 2 são apresentados os conceitos fundamentais para o entendimento do trabalho, como casos de uso, diagramas de classes e sequência, *Model Driven Development* (MDD) e linguagens de domínio específico.
- São descritos no capítulo 3, trabalhos relevantes que abordam a automatização do processo de desenvolvimento de software e que destacam a aplicabilidade de notações formais, como DSL.
- No capítulo 4, é apresentada a abordagem proposta, os padrões de orações suportados pela linguagem, as regras gramaticais da LUCAM e os passos da geração dos diagramas, a ferramenta LUCAMTool e as considerações finais deste capítulo.
- No capítulo 5 são apresentados os testes realizados em ambiente simulado e em ambiente real. Apresenta-se também os resultados gerados com a LUCAMTool e a análise dos dados coletados com a aplicação de um questionário nas empresas colaboradoras com este projeto de pesquisa.
- Por fim, no capítulo 6 é apresentada uma análise conclusiva, as contribuições deste trabalho de pesquisa, as limitações e orientações de trabalhos futuros.

2 CONCEITOS FUNDAMENTAIS

Neste capítulo serão abordados conceitos relacionados ao tema da dissertação, apresentando uma visão resumida sobre modelagem de casos de uso, diagramas de classes e sequência, desenvolvimento baseado em modelos e linguagens específicas de domínio.

2.1 Casos de uso

Modelagem em casos de uso é um recurso que foi introduzido por Jacobson (1987) e posteriormente adotado de forma significativa pela UML. Atualmente é amplamente aceito e reconhecido como uma estratégia de especificação, que representa os requisitos funcionais de um software e a interação que esses possuem com os atores e com outros sistemas (TIWARI; GUPTA, 2015).

Cockburn (2000) define casos de uso como a especificação de uma sequência de ações (fluxos e eventos) de um requisito funcional de software e de seus atores. Esses fluxos podem possuir variações, ou seja, além do fluxo principal, podem existir fluxos alternativos ou de exceção (BOOCH; RUMBAUGH; JACOBSON, 2010). Para Cockburn (2000), um caso de uso descreve o comportamento do sistema sob diversas condições, conforme o software responde a uma requisição de um dos *stakeholders**, denominados como atores. Os atores podem ser pessoas ou quaisquer elementos externos que possam interagir com o software de alguma forma. Considera-se como ator primário aquele que inicia uma interação com o sistema para atingir um objetivo, estimulando alguma ação por parte da aplicação, que reponde, protegendo os interesses dos envolvidos no processo.

A modelagem de casos de uso é uma técnica de descoberta de requisitos funcionais e já se tornou uma característica fundamental da UML. O caso de uso identifica os atores envolvidos e sua interação com o sistema, sendo complementado por uma especificação textual e outros diagramas. É um mecanismo de captura dos contratos entre os envolvidos (*stakeholders*) com o sistema e seus comportamentos (ADOLPH; COCKBURN; BRAMBLE, 2002; GUEDES, 2011).

De acordo com Guedes (2011), os casos de uso possibilitam a compreensão do comportamento externo do sistema por qualquer pessoa, pois, apresenta os requisitos do ponto de vista do usuário, indo além de representações de funcionalidades, pois eles podem ser utilizados na análise e elicitação dos requisitos para se estabelecer os cenários

*Stakeholders: parte interessada - com interesses - envolvida no desenvolvimento do sistema.

operacionais necessários do software (CHRISSIS; KONRAD; SHRUM, 2011). Conforme descrito no modelo *Rational Unified Process* (RUP), os casos de uso são especificações dos comportamentos do software em termos de sequências de ações, ou seja, são representações dos cenários do sistema. Adolph, Cockburn e Bramble (2002) conceituam um cenário como uma sequência de interações (metas), com a intenção de atingir os objetivos do ator primário e atores secundários do sistema. Com os casos de uso, é possível gerar automaticamente boa parte dos artefatos do software, inclusive os códigos fontes.

Para Guedes (2011), os casos de uso são importantes na etapa de análise e levantamento dos requisitos, pois ajuda a especificar, visualizar e documentar as características, funções e serviços desejados pelos usuários. Também é possível identificar os tipos de usuários que irão interagir com o software, quais papéis desempenhados por eles e quais funções serão requisitadas por cada ator específico.

2.2 Diagrama de classes

Segundo Guedes (2011), o diagrama de classes tem seu enfoque em permitir a visualização das classes do sistema com seus atributos e métodos, bem como em demonstrar como as classes se relacionam, complementam e transmitem informações entre si. Já Fowler (2005) afirma que o diagrama de classe descreve os tipos de objetos do sistema e os tipos de relacionamentos estáticos entre eles. Este diagrama também apresenta as propriedades, as operações (métodos) de uma classe e as restrições que se aplicam à maneira como os objetos estão conectados. São úteis para ilustrar as relações entre as classes e interfaces. Elementos como generalizações, agregações e associações refletem herança, composição e conexões, respectivamente (SAWPRAKHON; LIMPIYAKORN, 2014).

Esse diagrama apresenta uma visão estática de como as classes estão organizadas, preocupando-se em como definir a estrutura lógica do sistema. Destaca-se que o diagrama de classes serve como base para a construção da maioria dos outros diagramas da UML (BOOCH; RUMBAUGH; JACOBSON, 2006).

As classes geralmente possuem atributos e métodos, sendo que o primeiro item mencionado armazena os dados dos objetos da classe. Já o segundo são as funções que uma instância da classe pode executar. Os valores dos atributos podem ser variáveis de instância para instância, e graças a essa variação, é possível identificar individualmente cada objeto do sistema. Ao contrário dos atributos, os métodos são idênticos para todas as instâncias da classe. Os métodos são rotinas que podem receber valores como parâmetros e retornar valores, que podem ser o resultado produzido pela execução do método ou um valor representando se o método foi realizado com sucesso ou não (FOWLER, 2005; GUEDES, 2011).

Embora os métodos sejam declarados no diagrama de classes, identificando os possíveis parâmetros que são por eles recebidos e os possíveis valores por eles retornados, no diagrama de classes, não se preocupa em definir as etapas que tais métodos deverão percorrer ao serem invocados e não considera o sequenciamento das trocas de mensagens entre os objetos, sendo esta uma função atribuída aos diagramas de atividades e sequência. O diagrama de sequência está sendo contemplado neste trabalho e será explicado na próxima seção.

2.3 Diagrama de sequência

Os diagramas de sequência são usados para modelar o comportamento combinado de um grupo de objetos, que se resume em mensagens e eventos (SOMMERVILLE, 2011; SELLAMI et al., 2015). Para Sellami et al. (2015), esse diagrama é uma forma de representar as interações existentes entre os objetos de um sistema (FOWLER, 2005). Segundo Guedes (2011), é um diagrama comportamental que tem como objetivo determinar a sequência de eventos que ocorrem e um processo, identificando quais mensagens devem ser disparadas entre os elementos envolvidos e em que ordem. O diagrama de sequência também permite que o analista especifique a sequência de mensagens enviadas entre os objetos em colaboração. Ao contrário do diagrama de colaboração, diagramas de sequência enfatizam a sequência das mensagens, em vez das relações entre os objetos (SAWPRAKHON; LIMPIYAKORN, 2014). Determinar a ordem em que os eventos ocorrem, as mensagens que são enviadas, os métodos que são invocados e como os objetos interagem entre si num processo, é o objetivo deste diagrama.

Para Sawprakhon e Limpiyakorn (2014), o diagrama de sequência se baseia nos casos de uso, portanto, deve-se ter em mente que o fato de haver normalmente um diagrama de casos de uso, não significa que haverá apenas um diagrama de sequência. Normalmente existem vários diagramas de sequência em um projeto, sendo um para cada processo específico do sistema. Sellami et al. (2015) destacam que os diagramas de sequência são destinados a mostrar um fluxo detalhado para um caso de uso específico ou uma parte dele, e entre os diagramas da UML, é classificado como um diagrama de interação, pois mostra o fluxo da troca de mensagens entre os objetos ou agentes, de uma forma sequencial (BASIT-UR-RAHIM; ARIF; AHMAD, 2014).

De acordo com Sharma, Gulia e Biswas (2014), um diagrama de sequência tem duas dimensões, sendo vertical e horizontal. A primeira representa a sequência de mensagens em tempo e ordem em que ocorrem; e, a dimensão horizontal mostra as instâncias de objetos ou de agentes para que as mensagens sejam enviadas. Esse diagrama é importante do ponto de vista da obtenção e da compreensão de um caso de uso, que envolve interações entre os vários objetos.

O diagrama de sequência depende também do diagrama de classes, já que as classes dos objetos utilizados no diagrama de sequência estão descritas nele. No entanto, o diagrama de sequência é uma forma de validar o diagrama de classes, pois é ao modelar o diagrama de sequência que percebe quais métodos necessários a declarar e em quais classes eles deverão estar. Já a descoberta dos métodos se dá através do detalhamento dos processos com a especificação de casos de uso, com os diagramas de interação, como os de sequência (GUEDES, 2011).

Nos diagramas de sequência, podem ser encontrados os elementos:

- *Lifelines*: são linhas verticais que representam na maioria das vezes a instância de uma classe que participa de uma interação, ou seja, o tempo de vida de um objeto;
- Foco no controle ou ativação: indica os períodos em que um objeto está participando do processo, ou seja, identifica momentos em que um objeto está executando os métodos utilizados em um processo específico.
- Mensagens ou estímulos: são representadas por linhas horizontais ou diagonais. São utilizadas para demonstrar a ocorrência de eventos, que normalmente forçam a execução de um método nos objetos envolvidos no processo. As mensagens podem ser disparadas entre:
  - um ator e outro ator;
  - um ator e um objeto;
  - um objeto e outro objeto; e
  - um objeto e um ator.

Outros elementos são os fragmentos combinados, representados por um retângulo que determina a área de abrangência do diagrama e permite expressar questões de testes *se...então*, *laços* ou *processamentos paralelos* representados pelos operadores de interação incluídos a ele. A linguagem LUCAM contempla os fragmentos combinados *Alt e Loop.*

2.4 Desenvolvimento Dirigido a Modelos (MDD)

Diretamente relacionado com a abordagem proposta neste trabalho, o MDD possibilita aumentar o nível de abstração no processo de desenvolvimento do software e permite que analistas organizam os requisitos do sistema de forma lógica, separando modelo conceitual do modelo de implementação, independente da plataforma (SELIC, 2003; BLOBEL; GOOSSEN; BROCHHAUSEN, 2014). Para Atkinson e Kuhne (2003), há uma tendência no crescimento da elevação dos níveis de abstração em projetos de desenvolvimento de software, o que permite que os desenvolvedores resolvam de forma eficaz problemas de maior

complexidade (MUKERJI; MILLER, 2001; ATKINSON; KUHNE, 2003; PETRASCH; MEIMBERG, 2006).

De acordo com Panach et al. (2015), o paradigma MDD estabelece que o esforço dos analistas devem ser concentrados a princípio no modelo conceitual e posteriormente o sistema é implementado baseado nas regras de negócio estabelecidas nele, sendo que parte da geração do código-fonte pode ser automatizada. Em outras palavras, o paradigma MDD distingue os modelos de domínio do problema do modelo de domínio da solução.

Entretanto, MDD é uma abordagem de desenvolvimento de software emergente que visa preencher a lacuna semântica entre o domínio do problema e domínio da solução. Especificamente, os modelos não são só utilizados para construir as especificações em alto nível, mas também para gerar artefatos essenciais do processo de desenvolvimento de todo o sistema. Fu et al. (2011) destacam que essa é uma melhoria sobre os processos tradicionais de desenvolvimento de software, nos quais o desenho e implementação da aplicação são praticamente desconectados. Por exemplo, durante a fase de análise e projeto, casos de uso, diagramas de interação, de classes, e outros diagramas UML são utilizados na modelagem dos problemas a serem resolvidos. No entanto, esses artefatos são compreensíveis pelos analistas, não pelas máquinas (computadores).

Segundo Atkinson e Kuhne (2003), o paradigma do MDD possui como objetivo aumentar a produtividade da equipe de desenvolvimento através da automatização de persistência de dados e a interoperabilidade. Ao usar os modelos como artefatos primários do processo de desenvolvimento, esse método de desenvolvimento dá um enfoque maior na modelagem do software em comparado com sua codificação (implementação). O objetivo com isso é a automatização de parte do desenvolvimento do software, o que inclui a criação dos diagramas e a implementação, aumentando a produtividade no desenvolvimento do produto (OMG, 2015a). Em MDD, normalmente a implementação é gerada de forma semiautomática a partir dos modelos estabelecidos em fases anteriores.

De acordo com o portal do OMG (2015a), um modelo é uma representação abstrata que descreve a parte estrutural e comportamental de um software, sendo normalmente apresentado pela combinação de desenho e texto. Destaca-se também que todo modelo possui seu próprio grau de abstração, ou seja, o nível de detalhe existente nele.

A abordagem MDD é focada na diferença entre o projeto e a implementação de um sistema, de modo que o projeto possa ser realizado sem levar em conta detalhes de implementação, como técnicas de programação ou plataformas específicas. Isso é necessário para a geração automática de código para múltiplas plataformas, a partir de transformações de modelos abstratos (PETRASCH; MEIMBERG, 2006; CLEMENTE et al., 2009).

2.5 DSLs - Linguagens de Domínio Específico

2.5.1 Conceitos

De acordo com Kosar, Bohra e Mernik (2016), as DSLs são linguagens adaptadas a um domínio de aplicação específica, que oferecem ganhos significativos e facilidade de uso em comparação com linguagens de programação de uso geral (MERNIK; HEERING; SLOANE, 2005; SOBERNIG; HOISL; STREMBECK, 2016). Fowler e Parsons (2011) definem DSL como uma linguagem de programação de computadores com expressividade limitada, focada em um domínio específico. Os autores destacam que o desenvolvimento de DSL não é uma tarefa trivial, pois necessita de conhecimento de domínio e expertise no desenvolvimento da linguagens, e portanto, poucas pessoas têm ambas habilidades (LIU; ZHANG; KRAFT, 2014).

Fowler e Parsons (2011) destacam duas categorias de DSLs: externas e internas.

- Uma DSL externa é uma linguagem separada da linguagem da aplicação com a qual ela trabalha. Normalmente, uma DSL externa possui uma sintaxe customizada, mas é habitual usar a sintaxe de outra linguagem, como por exemplo, a linguagem *EXtensible Markup Language* (XML). Um *script* em uma DSL externa costuma ser analisado sintaticamente por um código na aplicação hospedeira usando técnicas de análise sintática textual. Exemplos de DSL externa são: SQL, Awk, e aquivos de configuração para sistemas como *Struts* e *Hibernate*.

- Uma DSL interna é uma maneira específica de usar uma *General Purpose Language* (GPL), ou seja, uma linguagem de propósito geral. Um *script* em uma DSL interna é um código válido em sua linguagem de programação, mas somente um subconjunto dos recursos da linguagem em um estilo para tratar de um aspecto do sistema. O resultado deve assemelhar com o de uma linguagem customizada e não da linguagem hospedeira. As linguagens LISP e Ruby são exemplos de DSLs internas.

Segundo Liu, Zhang e Kraft (2014), as linguagens de programação atuais oferecem recursos para o desenvolvimento de soluções de software com alto grau de complexidade. Diante disso são necessários mecanismos de abstração e modelagem, como classes, tipos abstratos de dados, objetos, interfaces, etc. Já uma DSL suporta o mínimo de recursos necessários para atender ao seu domínio, o que facilita seu aprendizado da equipe (MERNIK; HEERING; SLOANE, 2005; FOWLER; PARSONS, 2011).

No desenvolvimento de software, normalmente é de se esperar que analistas e usuários possuam um vocabulário em comum, servindo como suporte para o entendimento, possibilitando aos mesmos expressarem conceitos dentro do domínio da aplicação, de

forma que todos sejam interpretados corretamente no contexto em que estão inseridos. Um vocabulário padrão, sendo compartilhado por todos os envolvidos no projeto, serve como uma forma de unificação de todos os artefatos que são partes da implementação (FOWLER; PARSONS, 2011). Para Visic et al. (2015), o reuso dos artefatos é outra característica relevante e serve como base para definição de casos de testes e para os fontes do software. Os analistas de sistemas e os especialistas do domínio entenderão melhor os requisitos desenvolvidos, caso utilizem uma mesma linguagem de domínio. Então, um vocabulário padrão aproxima o domínio do problema e o domínio da solução (FOWLER; PARSONS, 2011; GHOSH, 2010).

A partir do momento que a complexidade do software aumenta, cresce com a necessidade de uma linguagem próxima do domínio do problema (MATTHEE; LEVITT, 2011). Dentre as lições e melhores práticas para a utilização de DSL em projetos, estão o fato de ser uma forma de comunicação entre analistas e especialistas de negócio. Portanto, é recomendável envolver o especialista de domínio durante o projeto da linguagem. Antes de se projetar uma DSL, é imprescindível pensar nos prós e nos contras, já que elas possuem expressividade limitada. Ghosh (2010) destaca que outro ponto a ser levado em conta é o fato de que a sintaxe da DSL deve ser expressiva o suficiente para os usuários finais, sem se tornar difícil de projetar e utilizar.

Kosar, Bohra e Mernik (2016) destacam que as ferramentas que auxiliem no desenvolvimento de DSLs estão aumentando continuamente. Isso reduz o esforço de implementação para DSLs e permite o desenvolvimento de linguagens em tempo satisfatório. No entanto, a falta de processos adaptativos e de fácil aplicação, muitas vezes transforma o desenvolvimento de DSLs num conjunto de atividades criativas, cujos resultados dependem da experiência dos desenvolvedores envolvidos. Devido a isso, muitos trabalhos estão sendo desenvolvidos e aprimorados, com o propósito de formalizar e padronizar o desenvolvimento das DSLs, aplicando esse recurso na prática (MERNIK; HEERING; SLOANE, 2005; GHOSH, 2010; FOWLER; PARSONS, 2011; KOSAR; BOHRA; MERNIK, 2016).

2.5.2 Motivação para usar DSLs

Diferentemente de paradigmas como orientação a objetos ou como processo ágeis, as DSLs são ferramentas com um foco limitado, ou seja, elas não introduzem uma mudança fundamental na maneira de pensar sobre o desenvolvimento de software. Em vez disso, são ferramentas específicas para condições particulares (VISIC et al., 2015; SOBERNIG; HOISL; STREMBECK, 2016). Portanto, além dos fatores já citados, Fowler e Parsons (2011) destacam outros pontos que incentivam o uso das DSLs, como: o aprimoramento na produtividade de desenvolvimento, a padronização da comunicação com especialistas de domínio (usuários e clientes), a mudança no contexto de execução e a oferta de um

modelo computacional alternativo (MERNIK; HEERING; SLOANE, 2005; GHOSH, 2010; KOSAR; BOHRA; MERNIK, 2016).

Svenningsson e Axelsson (2015) afirmam que a expressividade limitada das DSLs facilita na prevenção de erros por parte dos desenvolvedores. O modelo, por si só, fornece uma melhoria significativa na produtividade, pois, ele evita duplicação ao encapsular código e fornece uma abstração para que se possa pensar sobre o problema, facilitando especificar o que está acontecendo de maneira que pode ser entendida. Uma DSL melhora isso ao fornecer uma forma expressiva de ler e de manipular essa abstração (MERNIK; HEERING; SLOANE, 2005; GHOSH, 2010; FOWLER; PARSONS, 2011).

Fowler e Parsons (2011) destacam que se tratando da comunicação com especialistas do domínio, pode-se afirmar que é uma parte difícil dos projetos de software e uma fonte de falhas. Ao fornecer uma linguagem eficaz e intuitiva para lidar com os domínios, uma DSL pode ajudar a melhorar essa comunicação.

Para Ghosh (2010), ao abordar o tema mudança no contexto da execução, ressalta-se que ao usar uma DSL, essa pode frequentemente cobrir limitações em uma linguagem hospedeira, permitindo-se expressar as coisas em uma DSL confortavelmente e, então gerar código para o ambiente de execução a ser usado. Um modelo pode facilitar tal tipo de deslocamento. Uma vez que se tem o modelo, é fácil executá-lo diretamente e gerar código a partir dele. Modelos também podem ser preenchidos a partir de uma interface como formulários, como a partir de uma DSL, sendo que a segunda opção possuem vantagens em relação a primeira, pois as DSLs são melhores que formulários na representação de lógicas mais complexas. Além disso, podem-se usar as mesmas ferramentas de gerenciamento de código, tal como sistemas de controle de versão para gerenciar essas regras. Ao informar regras por um formulário e armazená-las em bancos de dados, muitas vezes o controle de versão é ignorado (MERNIK; HEERING; SLOANE, 2005; FOWLER; PARSONS, 2011).

2.5.3 Problemas com DSLs

Para Fowler e Parsons (2011), mesmo as DSLs aplicáveis podem vir com problemas, que são superestimados, normalmente porque pessoas não conhecem o suficiente a construção de DSLs e seu ajuste no panorama amplo do desenvolvimento de software.

Fowler e Parsons (2011), Visic et al. (2015) destacam três problemas:

- *cocofonia de linguagem*: a preocupação de que linguagens são difíceis de serem aprendidas;
- *construção da linguagem*: mesmo uma DSL representando um custo incremental menor em relação a sua biblioteca subjacente, ela demanda um custo considerável

em sua construção, pois, há a necessidade de escrever código e de mantê-los de acordo que as mudanças no contexto surgem; e

- *abstração restrita*: uma DSL fornece uma abstração que pode ser usada para pensar acerca de uma área de conhecimento qualquer. Tal abstração é valiosa, pois lhe permite expressar o comportamento de um domínio de um modo facilitado em comparado com o pensamento em termos de construções em baixo nível. Entretanto, qualquer abstração, seja ela uma DSL ou um modelo, carrega um perigo, o de colocar restrições em seu pensamento. Com uma abstração restrita, depende-se de esforços para fazer o mundo se encaixar em sua abstração, do que aconteceria ao contrário. Como qualquer outra abstração, deve-se olhar para a DSL como algo que está sempre em evolução e inacabável.

Um amplo entendimento de como as DSLs são implementadas é fundamental para entender essas questões e evitar problemas com a implementação e utilização das mesmas.

2.5.4 Como projetar DSLs

De acordo com Fowler e Parsons (2011), o objetivo de uma DSL é a clareza para o leitor. Pretende-se que o leitor típico, sendo ele programador ou especialista de domínio, entenda rapidamente o que as sentenças escritas na DSL significam. Para isso, não há uma receita de bolo, o que deve fazer é manter-se o esforço em atender essas características.

Segundo Membarth et al. (2016), ao projetar uma DSL é indispensável realizar uma análise cuidadosa do domínio da aplicação. Isso inclui a análise de operações e operadores, e da análise dos componentes básicos característicos de um domínio.

Explorar características de projetos iterativos, colocando suas ideias à prova em um público-alvo. Esteja preparado para fornecer múltiplas alternativas e veja como as pessoas reagem. Obter uma boa linguagem envolverá testar e rejeitar muitos passos equivocados, pois, ao corrigi-los, é provável que encontre o caminho (GHOSH, 2010).

Outro ponto é não tentar fazer uma DSL ser lida como uma linguagem natural, pois isso pode complicar o entendimento semântico da linguagem e acaba-se perdendo o benefício que as DSLs proporcionam, que é o padrão de comunicação entre os envolvidos no projeto. Uma DSL é uma linguagem de programação e seus usuários devem sentir isso.

2.6 Considerações finais

Neste capítulo apresentou-se os conceitos relacionados a DSLs, como recurso chave no processo de desenvolvimento de software, possibilitando um aumento na qualidade e

na produtividade de artefatos de software em processos orientados a requisitos. Isso inclui modelagem de casos de uso e os modelos gerados a partir daí.

No contexto deste trabalho, o objetivo de se utilizar uma DSL externa é automatizar a criação de artefatos de um processo de software orientado a requisitos. Com isso, espera-se evitar problemas em todas as fases do desenvolvimento, padronizar a comunicação entre os envolvidos, aumentar a produtividade e a qualidade dos sistemas desenvolvidos.

3 TRABALHOS RELACIONADOS

Serão apresentados nesta seção os trabalhos selecionados para fundamentar este projeto de pesquisa, conforme Quadro 1.

Quadro 1 – Artigos selecionados

ID	Título	Origem	Autores
1	The draco approach to constructing software from reusable components	IEEE Xplore	Neighbors (1984)
2	Draco-puc: a technology assembly for domain oriented software development	IEEE Xplore	Leite, Sant'Anna e Freitas (1994)
3	Translating use cases to sequence diagrams	ACM	Leite, Sant'Anna e Freitas (1994)
4	Toward engineered, useful use cases	JOT	Williams et al. (2005)
5	Supporting use case based requirements engineering	SD	Somé (2006)
6	The impact of model driven development on the software architecture process	ACM	Heijstek e Chaudron (2010)
7	Language for use case specification	IEEE Xplore	Savic et al. (2011)
8	Dsl for the uninitiated	ACM	Ghosh (2011)
9	A Textual Domain Specific Language for User Interface Modelling	SPRINGER	Karu (2013)
10	Language-based software engineering	ACM	Thakur e Gupta (2014)
11	atoucan: An automated framework to derive uml analysis models from use case models	ACM	Yue, Briand e Labiche (2015)
12	Quantifying usability of domain-specific languages: An empirical study on software maintenance	SD	Albuquerque et al. (2015)
13	Domain-Specific Languages: A Systematic Mapping Study	SD	Kosar, Bohra e Mernik (2016)

Fonte: Elaborado pelo autor

A seguir é apresentada a descrição de cada trabalho e as diferenças existentes em relação à abordagem apresentada nesta dissertação.

Na década de 80, Neighbors (1984) propôs uma abordagem denominada DRACO. Trata-se de um paradigma para construção de software que se fundamenta na reutilização de componentes através de um esquema baseado em conhecimento. Sua característica é a distinção e identificação, não só dos variados tipos de conhecimento, mas também dos diferentes níveis de abstração presentes no processo de desenvolvimento de software (PRADO; BARRÉRE; BONAFE, 1999).

A classificação dos domínios é baseada no grau de abstração e dividida em Domínios de Aplicação, Domínios de Modelagem e Domínios Executáveis. O primeiro grupo expressa a semântica dos domínios, ou seja, descreve o mundo real no qual os sistemas de software serão inseridos. O segundo se trata do grupo de domínios que encapsulam conceitos de forma a fornecer o recurso de reúso e facilita a implementação dos domínios de

aplicação. Já o terceiro, são aqueles domínios que podem ser diretamente implementados em uma plataforma computacional (NEIGHBORS, 1992; LEITE et al., 1995).

Com base no paradigma DRACO, Leite, Sant'Anna e Freitas (1994) apresentaram o DRACO-PUC, um sistema de transformação que tem por objetivo desenvolver, colocar em prática e testar o paradigma transformacional para o desenvolvimento de software orientado a domínios, fundamentado em reúso e em múltiplas visões. Uma especificação pode ser obtida através da aplicação de regras de transformação, descritas formalmente, sobre uma especificação de entrada. As regras de transformação são responsáveis pela automatização do processo de construção de software. Os sistemas de transformação manipulam as entradas (programas ou especificações), preservando a sua semântica. O objetivo é transformar um programa X em um programa Y, aplicando um grupo de transformações que mantem a semântica original de X em Y.

Foi apresentada por Leite et al. (1995) uma abordagem que usa a máquina transformacional DRACO-PUC para converter programas na linguagem COBOL para C/C++. A estratégia não tem por objetivo gerar código segundo os preceitos da programação orientada a objetos. Posteriormente, Freitas, Leite e SANT'ANNA (1996) incluíram na máquina DRACO-PUC recursos que atendem além das linguagens Cobol, C/C++, também as linguagens Visual Basic e Java.

Em Prado, Barrére e Bonafe (1999) foi proposta a MVCASE. Trata-se de uma ferramenta CASE orientada a objetos, que foi desenvolvida com o propósito de investigar a automação do processo de desenvolvimento de sistemas de software, desde a análise e especificação de requisitos, utilizando técnicas de diversos métodos, até a implementação automática em linguagem executável. Ela é composta por uma ferramenta gráfica, denominada *JavaRC*, que suporta a especificação de requisitos do sistema usando diferentes métodos da orientação a objetos, inclusive o DRACO, que é responsável pela implementação automática do sistema (PRADO; LUCRÉDIO, 2000).

Prado e Lucrédio (2001) incrementaram recursos na ferramenta MVCASE, permitindo suportar o desenvolvimento de softwares orientados a objetos, baseados em componentes. Na ferramenta, o engenheiro de software modela o sistema seguindo o método *Catalysis* (D'SOUZA; WILLS, 1999), e gera seu código para uma plataforma distribuída, usando componentes distribuídos com a tecnologia *Enterprise Java Beans* (EJB). A ferramenta disponibiliza os componentes em um *browser*, que facilita seu reúso através da herança ou instanciação de suas classes. Lucrédio (2005) estendeu a MVCASE e incluiu serviços remotos de armazenamento e busca de artefatos de software, visando aumentar a produtividade com o reúso de componentes de software.

No início da década passada, foi proposto por Li (2000) um conjunto de regras para normalizar a especificação de casos de uso textuais. Adotando o padrão definido, é

possível um analista deduzir quais classes, objetos, associações, atributos e operações um caso de uso possui e gerar os diagramas de sequência. Porém, trata-se de uma atividade mecanizada e não automatizada, já que não foi implementada uma ferramenta para testar na prática o padrão proposto, ficando a solução dependente do conhecimento de um especialista.

Foi proposto por Williams et al. (2005) um projeto que permitiu explorar os problemas comuns existentes na modelagem de casos de uso, mostrando que há inconsistência e desalinhamento entre o modelo de casos de uso e suas especificações textuais. Posteriormente foi apresentado por Hoffmann et al. (2009), um metamodelo para descrições de casos de uso textuais. Este define uma representação textual do comportamento do caso de uso, sendo facilmente compreensível por leitores que não dominam o assunto. Para a modelagem de casos de uso narrativos, foi desenvolvida a ferramenta *Narrative Use Case Description Toolkit for Evaluation and Simulation* (NaUTiluS).

Em Somé (2006), foram gerados cenários com sequenciamento baseado nas pré-condições e pós-condições de casos de uso textuais, descritos em linguagem controlada. Os cenários gerados estão em alto nível de abstração, diferentemente do que se propõe neste trabalho, em que se trabalha com nível detalhado. Os mesmos autores já tinham desenvolvido um trabalho em que verificam inconsistências dos dados, como entidades descritas com mais de um nome, operações no caso de uso que não se refiram a uma operação de conceito no modelo, entre outras. São gerados cenários com a composição dos casos de uso, e simulada sua execução em uma máquina de estados, numa ferramenta denominada UCEd. Propõe-se um modelo de domínio com informações de valores possíveis, porém com formas variadas em sua declaração e pretende-se gerar os cenários com o sequenciamento dos casos de uso, porém, para execução de testes.

Em (SAVIC et al., 2011), é proposta uma linguagem para especificação de casos de uso, denominada *SilabReq*. A partir dos casos de uso são gerados os modelos de domínio, a lista de operações do sistema, o modelo de caso de uso e os diagramas de estado e de atividades. No ano seguinte, foi proposta pelo mesmo autor a divisão da especificação em níveis de diferentes graus de abstração, uma vez que casos de uso são utilizados por pessoas de diferentes papéis, com diferentes necessidades, durante o desenvolvimento do software, desde usuários finais, engenheiros de requisitos e analistas de negócio, até projetistas, desenvolvedores e *testers*. Diferente da abordagem aqui citada, a LUCAMTool gera além dos diagramas de casos de uso e sequência, também o diagrama de classes, sendo esse considerado um dos mais importantes da UML. Outro diferencial da LUCAM em relação à *SilabReq*, são os testes realizados em ambiente real de desenvolvimento de software.

Em Karu (2013) foi proposta uma DSL que permite especificar casos de uso e gerar automaticamente protótipos de telas para interface web. A linguagem de descrição

de interface de usuário e o metamodelo correspondente consistem em duas partes distintas. A primeira descreve os aspectos e a granularidade dos elementos de interação fornecidos ao usuário do sistema durante cada etapa de interação. A segunda parte da linguagem descreve as possíveis interações fornecidas pela interface. Neste trabalho não é gerado diagramas UML e na abordagem proposta nesta dissertação, não se gera protótipos de telas.

Foi proposta e implementada por Tavares (2014) uma DSL denominada *LAnguage of Textual Use Cases* (LATUC) que permite gerar na forma semiautomática, diagramas de classes a partir de especificação de casos de uso. Também foi implementada uma ferramenta de apoio denominada *Semi-Automatic Generator of Software Artifacts* (SAGSA), a qual faz o *parser* da especificação e gera os diagramas de classes. Portanto, a ferramenta não identifica classes de entidade que estão implícitas na especificação, o que gera inconsistência entre o diagrama de classe e a especificação de casos de uso. Além da limitação de gerar somente o diagrama de classes, a *Integrated Development Environment* (IDE) proposta deixa a desejar em relação a usabilidade e interatividade com o usuário.

Thakur e Gupta (2014) apresentaram uma ferramenta que permite a geração automática de diagramas de sequência a partir da especificação de casos de uso escritas em linguagem natural e no idioma Inglês. Na abordagem, usou-se o analisador de expressões em linguagem natual desenvolvido pelo *The Stanford Natural Language Processing Group* (*Stanford NLP Group*[*]) para identificar os objetos e as interações entre eles a partir de especificação de casos de uso. O *Stanford Parser* analisa as sentenças e mapeia cada palavra do discurso, como adjetivo, advérbio, artigo, pronome, substantivo, verbo, etc. Portanto, a abordagem não trata situações no diagrama de sequência como fragmentos combinados, auto-chamadas e não gera outros diagramas UML.

Em Yue, Briand e Labiche (2015), foi proposta uma abordagem denominada *aToucan*, que teve como base trabalhos já existentes, para gerar automaticamente um modelo de análise UML contemplando os diagramas de classes, sequência e atividade a partir de um diagrama de casos de uso. Os pesquisadores também adotaram especificação em linguagem natural (Inglês) e similar ao trabalho anteriormente citado, utilizaram o *Stanford Parser* para fazer o mapeamento dos termos que compõem as sentenças da especificação de casos de uso.

Apesar de os resultados na geração automática dos modelos terem evoluído, ressalta-se que ao adotar a linguagem natural, assuma-se o risco de os problemas inerentes da mesma permanecerem, pois é livre, ambígua e não estabelece um padrão de comunicação entre a equipe. O que dificulta a automatização da geração de artefatos e a rastreabilidade entre especificação, diagramas e código-fonte.

[*]Stanford NLP Group: <*http://nlp.stanford.edu/*>

Já as linguagens de domínio específico, se projetadas adequadamente, ao contrário da linguagem natural, estabelece uma linguagem padrão a ser utilizada por todos os membros da equipe e sua formalidade padroniza a comunicação entre os envolvidos no projeto, possibilitando uma maior produtividade, confiabilidade e qualidade dos artefatos gerados (HEIJSTEK; CHAUDRON, 2010; GHOSH, 2011).

Além disso, as DSLs permitem que as soluções sejam expressas no nível de domínio da aplicação, possibilitando que os analistas de negócio entendam, validem, modifiquem e desenvolvam funcionalidades utilizando a linguagem definida. Destaca-se que especialistas de domínio e engenheiros de software já estão acostumados com a formalidade das linguagens de programação e isso reduz o tempo de aprendizado da linguagem e otimiza o aproveitamento dos recursos da mesma (GHOSH, 2011; FOWLER; PARSONS, 2011; GUPTA, 2015; KOSAR; BOHRA; MERNIK, 2016). DSLs visam facilitar a construção de artefatos de software através de abstrações e notações formais especializadas. São cada vez mais utilizadas em atividades da engenharia de software, incluindo a concepção, especificação e verificação de requisitos, auxiliando na geração automática de modelos (ALBUQUERQUE et al., 2015).

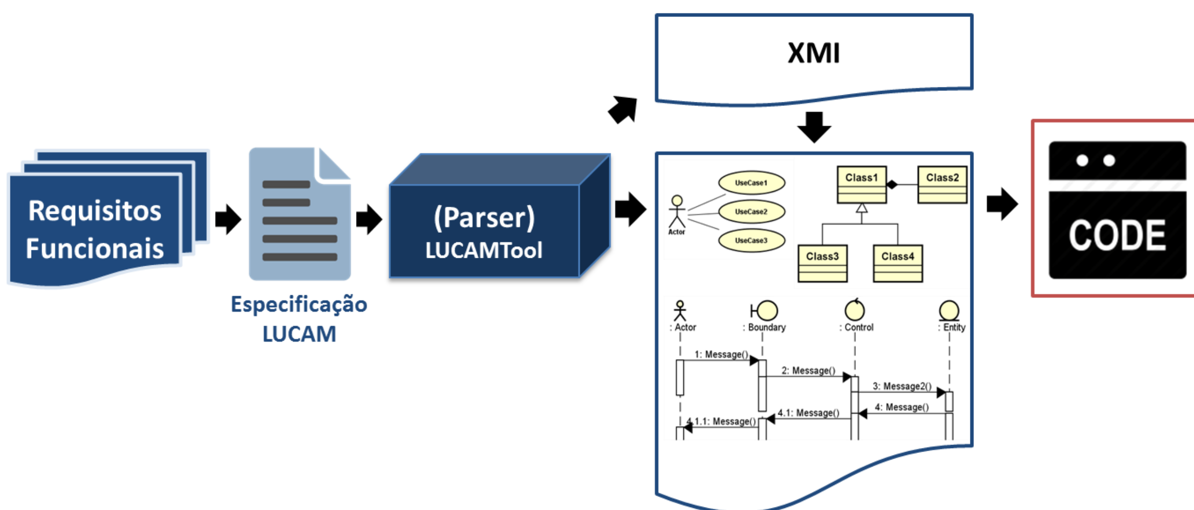
4 ABORDAGEM PROPOSTA

Este trabalho propõe uma DSL denominada *Language of Use Case to Automate Models* (LUCAM), a qual contempla o detalhamento de casos de uso textuais e juntamente com a ferramenta LUCAMTool permite a geração automática de diagramas UML em processos de software orientados a requisitos, especificamente os diagramas de casos de uso, classes e sequências. Esses diagramas são gerados seguindo premissas do paradigma da orientação a objetos e do modelo arquitetural *Model-View-Controller* (MVC).

A LUCAMTool é uma ferramenta de apoio que foi implementada para possibilitar ao usuário especificar os requisitos funcionais do software na linguagem LUCAM, que com um *parser* realiza o mapeamento das informações relevantes do detalhamento de casos de uso e posteriormente gera artefatos para um padrão de entrada XMI e no formato `.astah`. Os detalhes sobre a ferramenta serão apresentados na Seção 4.5.

A transição da especificação de casos de uso para os diagramas UML se dá pelo processo ilustrado, conforme Figura 1. O processo se inicia com a análise e levantamento dos requisitos por parte dos analistas de negócio. Posteriormente os casos de uso (requisitos funcionais) são especificados em forma textual na linguagem LUCAM e em seguida são mapeados e transformados em modelos orientados a objetos.

Figura 1 – Transição da LUCAM para os diagramas UML



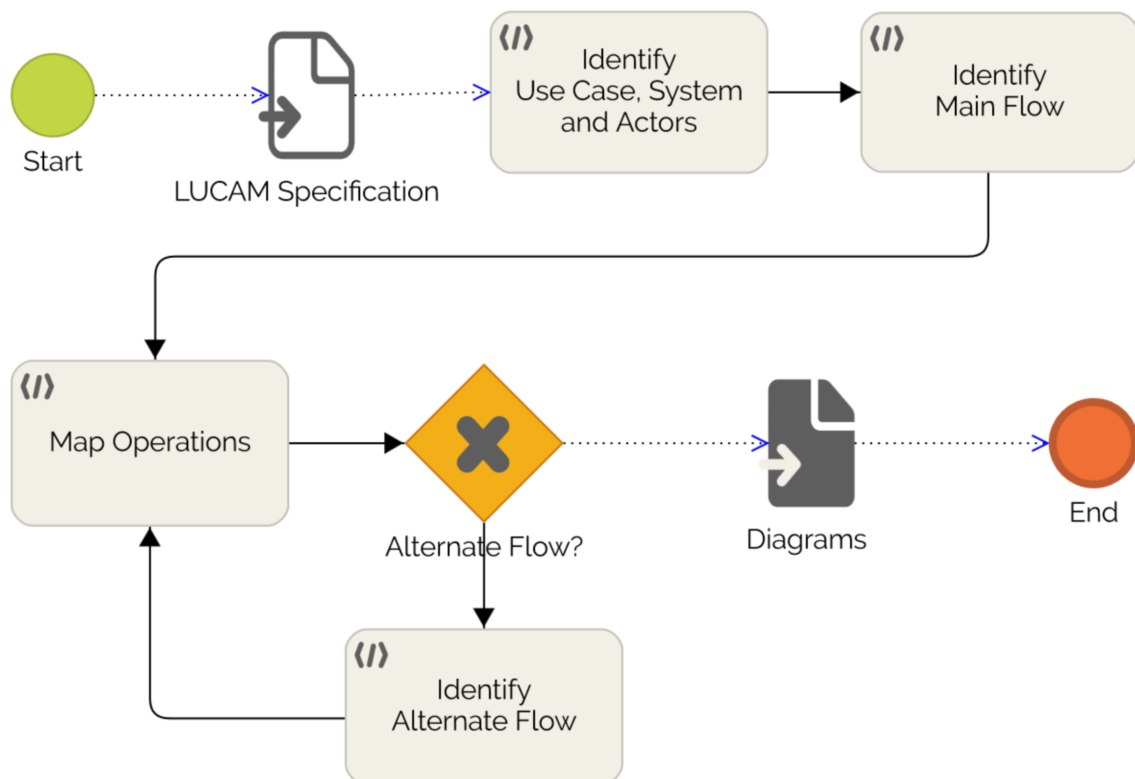
Fonte: Elaborado pelo autor

Repare que uma etapa posterior possível a ser realizada é a geração automática

de código-fonte referente aos diagramas, já que a maioria das ferramentas de modelagem possuem o recurso de geração de código-fonte a partir dos diagramas UML.

A geração automática de artefatos simplifica e agiliza o processo de desenvolvimento de software e permite contemplar diversos casos práticos, os quais em sua maioria são trabalhosos se feitos manualmente. E uma vez que os artefatos refletem fielmente o que está na especificação do sistema, eles se tornam confiáveis, reduzindo as possibilidades de falhas no projeto.

Figura 2 – Fluxo geral da LUCAM



Fonte: Elaborado pelo autor

Na Figura 2 é apresentado o fluxo que representa o processo de mapeamento da especificação de casos de uso e posteriormente a geração dos diagramas. Conforme será mostrado nas regras da gramática LUCAM, o fluxo alternativo deriva as mesmas regras do fluxo principal, sendo que suas sentenças obedecem o mesmo padrão. Além disso, num caso de uso pode ter vários fluxos alternativos, caracterizando assim o ***loop*** presente no fluxograma.

4.1 Template da especificação na LUCAM

A princípio será apresentado o esqueleto da especificação na DSL LUCAM (Listagem 4.1), o qual foi baseado no *template* proposto pelo processo unificado de desenvolvimento de software *OpenUP* (Processo Unificado Aberto) (ECLIPSE, 2015), conforme Anexo A.

Listagem 4.1 – Template LUCAM

```
Use Case: Use_Case_Name.
Brief Description
"Brief Description of the use case.".
System: System_Name.
Primary and Secondary Actors
Primary Actors: Actor_name_01, ... , actor_name_N.
Secondary Actors: Actor_name_01, ... , actor_name_N.

Main Flow: Main_Flow_Name.
Actor starts Use Case.
.....................................
Actor finishes Use Case.

Alternate Flows
Alternate Flow 01: Alternate_Flow_Name.
.....................................
Alternate Flow N: Alternate_Flow_Name.
.....................................

Key Scenario
Key Scenario 01: Key_Scenario_Name.
.....................................
Key Scenario N: Key_Scenario_Name.

Preconditions
"Preconditions associated with this use case".
Postconditions
"Postconditions associated with this use case".
Special Requirements
"Special Requirements associated with this use case".
Extension Points
"Extension Points associated with this use case".
```

A especificação é composta pelos atributos: nome do caso de uso, breve descrição, nome do sistema, atores, fluxos básicos e alternativos, cenários principais, pré-condições, pós-condições, requisitos especiais e pontos de extensão, conforme *template* da Listagem 4.1. Os termos destacados são as palavras reservadas da DSL proposta.

O *template* original (Anexo A) tem como foco especificações em linguagem natural, portanto, esse tipo de linguagem não exige uma padronização e dispensa qualquer tipo de restrição. Também apresentam fenômenos sintáticos que dariam origem à imprecisão semântica, o que dificulta a automatização do processamento dos casos de uso. Para o correto mapeamento de um detalhamento de casos de uso para os diagramas de classes e sequências, foi necessário adaptar o *template* base para que a DSL contivesse as informações necessárias e suficientes para uma posterior automatização da geração dos artefatos de software suportados pela LUCAM.

Com base no propósito da DSL, identificou-se a necessidade de incluir elementos na especificação de casos de uso que não constam no *template* original, como: explicitação dos atributos e métodos de cada classe; além dos atores primários e secundários, é necessário especificar o nome do sistema que irá interagir com os mesmos; para o diagrama de sequência, são necessários especificar também os fragmentos combinados de decisão, repetição e simultaneidade (*if, loop* e *concurrency* respectivamente), atendendo as regras estabelecidas para a sintaxe da DSL proposta.

Não é usual incluir esses elementos no detalhamento de casos de uso, portanto, ao incluí-los, aumenta-se a expressividade da linguagem, tornando seu vocabulário rico e com opções de escrita que permitem especificar a maioria dos cenários existentes no desenvolvimento de software. Esse recurso também é essencial para a geração automática dos diagramas, e inclusive, possibilita manter a consistência entre a especificação e os diagramas gerados.

A LUCAM possui mecanismos para descrever situações em que um ator interage com um sistema afim de executar uma ação. A padronização proposta pela linguagem facilita o mapeamento das informações do detalhamento de casos de uso para as classes, como os relacionamentos, atributos e métodos.

4.2 Elementos preliminares da especificação

A especificação de casos de uso na linguagem LUCAM possuem em seu cabeçalho os elementos a seguir: nome do caso de uso, descrição, nome do sistema, atores primários e secundários.

A partir do nome do caso de uso é identificado o nome da classe de controle. Essa classe tem a finalidade de controlar o sequenciamento do comportamento do caso de uso, ou seja, representa a dinâmica do requisito funcional. Para cada fluxo (principal e alternativos), gera-se um método para a classe de controle e para cada método dessa classe, são gerados automaticamente um diagrama de sequência. Inicialmente, já são identificados também os nomes dos atores primários e secundários e o nome do sistema.

4.3 Módulo Mapear Operações

O módulo ***Mapear Operações*** é o responsável por reconhecer as sentenças (orações) da especificação de casos de uso, validando-as com base nos padrões definidos e nas regras da gramática. É nele que atua o núcleo do *parser* da LUCAMTool, o qual contém dois métodos fundamentais, denominados *identificaOracao()* e *identificaMensagem(Oracao)*.

Primeiramente serão apresentados exemplos de sentenças escritas na linguagem LUCAM e esses serão usados para demonstrar como é realizado o mapeamento e a geração dos diagramas. Cada sentença segue os padrões que serão apresentados no Quadro 3, inclusive estão apresentados na mesma ordem.

1. ***Customer*** *informs the attributes to the* ***Manager****.*

2. ***Manager*** *selects "InsertAccount" on* ***MainForm****.*

3. ***Manager*** *enters attributes (ID, Name, DateBirth, ...) of* ***Customer****.*

4. ***System*** *returns "Input mode screen" to* ***Manager****..*

5. ***System*** *sends the notification by* ***e-mail****.*

6.1 ***System searches*** *for the* ***Customer****.*

6.2 ***System returns*** *the attributes of* ***Customer****.*

7. ***System saves*** *the attributes of* ***Customer****.*

8. ***System validates*** *attributes of* ***Customer****.*

9. ***System verifies*** *the attribute (condition) of the* ***Customer****.*

10. ***System displays*** *the attributes of* ***Customer*** *on* ***MainForm****.*

11. ***System cancels*** *the operation.*

O método *identificaOracao()* se encarrega de analisar cada termo das orações que compõem a especificação textual, identificando sua classe gramatical (sujeito, verbo, método, substantivo, atributos, preposição, classes e atores) e retorna um objeto com esses termos. O Quadro 2 exemplifica como é composto esse objeto para as sentenças que foram apresentadas anteriormente.

Quadro 2 – Retorno do método *identificaOracao()*

Nº	Subject	Verb	Method	Noun/ Attributes	Prep1	Prep2	Entity Class	Boundary Class	Communication Class	Actor
1	Customer	informs		attributes		to				Manager
2	Manager	selects	Insert Account			on		MainForm		
3	Manager	enters		attributes (...)	of		Customer			
4	System	returns	Input ... screen			to				Manager
5	System	sends		notification		by			E-mail	
6.1	System	searches				for	Customer			
6.2	System	returns		attributes	of		Customer			
7	System	saves		attributes	of		Customer			
8	System	validates		attributes	of		Customer			
9	System	verifies		attributes	of		Customer			
10	System	displays		attributes	of	on	Customer	MainForm		
11	System	cancels		operation						

Fonte: Elaborado pelo autor

Observe que cada sentença é fragmentada de acordo com seus termos e são preenchidos os atributos do objeto. Após o mapeamento realizado com o método *identificaOracao()*, o objeto retornado é passado via parâmetro para o método *identificaMensagem(Oracao)*. Com base nas regras apresentadas no Quadro 3, esse método mapeia os elementos existentes no diagrama de sequência e retorna um objeto com o rótulo da mensagem, as classes de origem e destino.

Quadro 3 – Padrões de sentenças da LUCAM

Nº	Syntactic Structure	Sender	Receiver	Operation
1	**ActorS** Verb Noun Preposition **ActorR**	-	-	-
2	**Actor** Verb Noun/Method Preposition **Boundary**	Actor	Boundary	Verb+Noun/Method
3	**Actor** Verb Noun Preposition **Entity**	Actor	Boundary	Verb+Noun+Preposition+Entity
4	**System** Verb Noun/Message Preposition **Actor**	Controller	Boundary	Verb+Noun/Message
5	**System** Verb Noun Preposition **Communication**	Controller	Controller	Verb+Noun+Preposition+Commun.
6	**System VerbEntityWithReturn** Noun Preposition Entity	Last Receiver Class (LRC)	Entity	Verb+Noun+Preposition+Entity
		Entity	Controller	Return+Noun+Preposition+Entity
7	**System VerbEntityWithoutReturn** Noun Preposition Entity	Controller	Entity	Return+Noun+Preposition+Entity
8	**System VerbValidation** Noun Preposition Entity	Controller	Controller	VerbV.+Noun+Preposition+Entity
9	**System VerbProcessing** Noun Preposition Entity	Controller	Controller	VerbP.+Noun+Preposition+Entity
10	**System VerbReturnInBoundary** Noun Preposition Boundary	LRC	Boundary	VerbReturn+Noun
11	**Actor/System Verb** Noun	Controller	Controller	Verb+Noun

Fonte: Elaborado pelo autor

A regra 1 apresentada no Quadro 3 reconhece um padrão de mensagem que repre-

senta uma interação entre dois atores, portanto, optou-se por não mapear essa operação para o diagrama de sequência, sendo a sentença utilizada para identificação de classes.

A regra 6 é um tipo de padrão que pode gerar duas operações. Por exemplo: se o usuário fazer uma pesquisa (verbo *searches*), a mensagem deverá ser repassada da classe do tipo *Boundary* até a classe do tipo *Entity*. O mesmo acontece quando o sistema recuperar os dados pesquisados, uma mensagem deverá sair da classe do tipo *Entity* e ser replicada até a classe do tipo *Controller*. Tem situações que a mensagem tem origem na classe do tipo *Controller*, por isso definiu-se o termo *Last Receiver Class* (LRC), pois, no fluxo é necessário ter controle e identificar qual a classe em que o método foi invocado. Essa situação poderá ser observada e será evidenciada nos diagramas de sequência que serão apresentados na Seção 4.4.

Além da classe do tipo controle, é necessário identificar a tipologia de outras classes. Houve então a necessidade de definir regras baseadas nas preposições que precedem o nome de cada classe na sentença. A preposição ***Of*** precede o nome da classe do tipo *Entity*, ***On*** precede a classe *Boundary*, ***To/For*** precedem os nomes dos *Atores* e ***By*** precede a classe do tipo *Communication*, conforme pode ser observado nos exemplos apresentados no Quadro 2.

Visando aumentar a expressividade da linguagem e atender requisitos como clareza, simplicidade e interatividade, criou-se três tabelas que referenciam palavras previamente cadastradas na LUCAMTool. O termo *TABTRANSVERB* se refere ao conjunto de verbos transitivos e *TABINTRANSVERB* aos verbos intransitivos. Tem verbo que só faz sentido no contexto de uma frase se estiverem relacionados a um ator. Já outros só fazem sentido se tiverem relacionados ao sistema. Existem também os verbos de exceção que necessitaram ser identificados por uma classe de equivalência de verbos. Verbos como *validates* e *verifies* validam restrições de entrada de dados ou regras de negócio e apresentam comportamento de *auto chamada* no diagrama de sequência. Já verbos como *searches* e *retrieves*, reivindicam uma mensagem de retorno à classe de controle.

A terceira tabela é a *TABNOUN*, que representa o conjunto de substantivos pré-cadastrados na LUCAMTool e que compõem o vocabulário da linguagem proposta. As sentenças contidas nas especificações escritas na linguagem LUCAM devem seguir as regras de acordo com os padrões definidos, pois o *parser* fará o mapeamento correto das interações para geração automática do diagramas de classes e sequência, sem comprometer a qualidade dos mesmos.

O Algoritmo 1 resume os passos do processo de mapeamento dos elementos para os diagramas de casos de uso, classes e sequência.

A entrada no processo se dá com a especificação de caso de uso na linguagem LUCAM e a saída é composta pelos elementos mapeados para os diagramas. Os méto-

Algoritmo 1: Passos executados pelo *parser*

```
Entrada: Especificação do caso de uso na linguagem LUCAM
Saída: Elementos mapeados para os diagramas
1  inicio
2      Nome da classe de controle <- Nome do caso de uso;
3      Atores [ ] <- Nomes dos atores primários e secundários;
4      Sistema <- Nome do sistema;
5      for CADA FLUXO DO CASO DE USO do
6          Métodos da classe de controle [ ] <- Nome do fluxo;
7          repeat
8              Lê a oração;
9              ...
10             Aciona o método identificaOracao();
11             Aciona o método identificaMensagem(Oracao);
12             ...
13             Elementos dos diagramas [ ] <- elementos mapeados na especificação;
14         until FIM DO FLUXO;
15     return ELEMENTOS DOS DIAGRAMAS [ ];
```

dos *IdentificaOracao()* e *IdentificaMensagem(Oracao)* são os responsáveis por realizar o mapeamento, conforme processo explicado nesta seção. O mapeamento dos elementos é baseado nos termos e no comportamento de cada sentença. Já os elementos são armazenados numa estrutura de dados e repassados para uma classe responsável pela geração dos diagramas.

4.4 A linguagem LUCAM

Esta seção apresenta a linguagem proposta neste trabalho, detalhando suas convenções léxicas, as regras sintáticas e semânticas da LUCAM, exemplos de especificações e diagramas gerados automaticamente pela LUCAMTool.

4.4.1 Convenções léxicas da LUCAM

As convenções léxicas da linguagem LUCAM estão apresentadas no Quadro 4.

Foram definidas as nomeclaturas para os identificadores, conforme Quadro 5.

No Quadro 6, são apresentados os símbolos especiais reconhecidos pela LUCAM.

O espaço em branco é ignorado, exceto como separador de identificadores (ID) e palavras-chave. A linguagem permite ao usuário inserir comentários (*// ou /* ... */ ou #*) nas especificações, os quais serão ignorados pelo *parser* ao realizar o mapeamento do detalhamento dos casos de uso.

Quadro 4 – Convenções léxicas da LUCAM

Símbolo	Regras
ID	letra (letra | dígito | '_')*
NUM	dígito (dígito)*
letra	'a'..'z' | 'A'..'Z'
dígito	'0'..'9'

Fonte: Elaborado pelo autor

Quadro 5 – Nomeclaturas dos identificadores da LUCAM

ID	Descrição
ActorID	Os nomes dos atores primários e secundários
AttributeID	Os atributos de uma classe
AttributeTypeID	Tipos dos atributos de uma classe
BoundaryClassID	Nome da classe de fronteira
CommunicationID	Forma de envio de uma mensagem
ControlClassID	Nome de uma classe de controle
EntityClassID	Nome de uma classe entidade
MethodBoundaryID	Nome do método da classe de fronteira
MethodControlClassID	Nome do método da classe de controle
UseCaseID	Nome do caso de uso
SystemID	Nome do sistema

Fonte: Elaborado pelo autor

Quadro 6 – Símbolos especiais da LUCAM

Descrição	Símbolo
Aspas	" "
Barra-barra	//
Barra-asterisco	/* ... */
Cerquilha	#
Colchetes	[ ]
Dois pontos	:
Hifen	-
Parênteses	( )
Ponto	.
Vírgula	,

Fonte: Elaborado pelo autor

4.4.2 Regras gramaticais da LUCAM

A gramática da linguagem de especificação LUCAM na notação *Extended Backus-Naur Form* (EBNF) é apresentada e explicada a seguir. Na versão de EBNF utilizada, símbolos terminais são grafados em negrito e símbolos não terminais iniciam-se com maiúsculas. Além disso, {A} indica zero ou mais repetições de A, A seguido de {A} indica uma

ou mais repetições de A e [A] indica que A é opcional.

A gramática EBNF e a explicação detalhada de cada regra, está no Apêndice B. Nesta seção serão apresentadas as regras mais relevantes. Juntamente com a apresentação das regras gramaticais, serão apresentados também fragmentos dos diagramas gerados automaticamente pela LUCAMTool, mostrando com detalhes o processo de mapeamento dos elementos nos diagramas de classes e sequências.

Também é apresentada no Apêndice J a gramática implementada no XText *, sendo esse um *plugin* do Eclipse destinado para projetos e implementações de linguagens de domínio específico. Esse *framework* permite desenvolver DSL e obter um editor de texto correspondente à linguagem, baseado no IDE Eclipse7, com o suporte característico, incluindo a validação, *IntelliSense* , formatação, coloração de sintaxe e reconhecimento de palavras chave (*highlighting*), visão estrutural dos elementos e referência cruzada (SAVIC et al., 2011).

O caso de uso denominado "*Conta Bancária*" será utilizado para apresentar os recursos e a expressividade da linguagem LUCAM. Neste cenário, o gerente como ator primário é o responsável por manter os dados dos clientes e de suas contas. Já o cliente como ator secundário, tem o papel de repassar ao atendente seus dados para que sejam cadastros no sistema e poderá também realizar movimentações como saques e transferências.

Na primeira parte da gramática, estão quatro regras que compõem a estrutura (esqueleto) do detalhamento de casos de uso, conforme apresentado a seguir.

1. `LUCAM ::= UseCaseHeader UseCaseFlows UseCaseFooter`
2. `UseCaseHeader ::= UseCaseName UseCaseBriefDescription SystemName PrimarySecondaryActors`
3. `UseCaseFlows ::= MainFlowName MainFlowScope [AlternativeFlows]`
4. `UseCaseFooter ::= Key Scenario {KeyScenario} PreConditions PosConditions SpecialRequirements ExtensionPoints`

Associada a cada regra, segue a explicação semântica das mesmas:

- **Regra 1 (*LUCAM*)**: Essa regra representa o esqueleto do detalhamento de casos de uso, sendo divida em três grupos:
  - Cabeçalho do detalhamento do caso de uso: representado pela regra ***UseCaseHeader***.

*Xtext: <https://eclipse.org/Xtext/>

- Fluxos do detalhamento do caso de uso: representado pela regra ***UseCaseFlows***.
- Rodapé do detalhamento do caso de uso: representado pela regra ***UseCaseFooter***.

- **Regra 2 (*UseCaseHeader*)**: essa deriva um conjunto de regras que define as partes preliminares da especificação de casos de uso.
- **Regra 3 (*FlowControl*)**: por meio dessa regra, é possível a geração dos fluxos básicos e alternativos do caso de uso.
- **Regra 4 (*UseCaseFooter*)**: são derivadas dessa regra, as regras que representam as informações existentes no rodapé do detalhamento de casos de uso.

A seguir as regras que geram o cabeçalho da especificação de casos de uso na linguagem LUCAM.

5. `UseCaseName ::= "Use Case: "ControlClassID POINT`
6. `UseCaseBriefDescription ::= "Brief Description"[Text] POINT`
7. `SystemName ::= "System: "SystemID POINT`
8. `PrimarySecondaryActors ::= "Primary and Secondary Actors "`
   `PrimaryActorName [SecondaryActorName]`
9. `PrimaryActorName ::= "Primary Actors: "`
   `ActorID {","ActorID} POINT`
10. `SecondaryActorName ::= "Secondary Actors: "`
    `ActorID { ","ActorID} POINT`

Associada a cada regra, segue a explicação semântica das mesmas:

- **Regra 5 (*UseCaseName*)**: define o nome do caso de uso e mapeia o nome da classe de controle.
- **Regra 6 (*UseCaseBriefDescription*)**: representa a descrição resumida do caso de uso.
- **Regra 7 (*SystemName*)**: define o nome do sistema.
- **Regra 8 (*PrimarySecondaryActors*)**: define a estrutura para especificar os nomes dos atores primários e secundários.

- **Regra 9 (*PrimaryActorName*)**: regra define os nomes dos atores primários.
- **Regra 10 (*SecondaryActorName*)**: se existir atores secundários, essa regra define seus nomes.

Será utilizado para exemplo de especificação e de geração dos diagramas o caso de uso *SB_BankAccount*, referente ao cenário do sistema bancário. Para formação do cabeçalho da especificação na Listagem 4.2, utilizam as regras 5-10, apresentadas anteriormente. Nessa parte, a LUCAMTool já realiza um mapeamento e identifica o nome da classe de controle, os atores e o nome do sistema. O nome da classe de controle tem origem do nome do caso de uso, mapeado com a regra ***UseCaseName***, a qual deriva para o identificador ***ControlClassID***.

Listagem 4.2 – Cabeçalho - UC SB_BankAccount

```
1 Use Case: SB_BankAccount.
2 Brief Description
3 "Allows the user perform operations such, as insertion and closing of
     account data. Withdraw cash and carry out bank transfers.".
4 System: System.
5 Primary and Secondary Actors
6 Primary Actors: Manager.
7 Secondary Actors: Customer.
```

As regras que compõem o fluxo principal dos casos de uso são apresentadas a seguir.

11. `MainFlowName ::= "Main Flow:  "MethodControlClassID POINT`
12. `MainFlowScope ::= [ActorID "starts Use Case"POINT] MainFlow [ActorID "finishes Use Case"POINT]`
13. `MainFlow ::= MainFlowElements {MainFlowElements}`
14. `MainFlowElements ::= MainFlowCore | FlowIf | FlowLoop | FlowConcurrency`
15. `MainFlowCore ::= ((ActorID | SystemID) (MainFlowTabTransVerb | TABINTRANSVERB) POINT) | ReturnMessage | InteractionUseCase POINT`
16. `MainFlowTabTransVerb ::= TABTRANSVERB ["MethodBoundaryID"] ( MainFlowAttibutes | "on"MainFlowBoundaryClass)`
17. `MainFlowAttibutes ::= [["the"] TABNOUN ["("[AttributeTypeID] AttributeID {, [AttributeTypeID] AttributeID}")"]] (("on"MainFlowBoundaryClass | ("of") MainFlowEntityClass)) | MainFlowsActorClass)`

18. `MainFlowBoundaryClass ::= BoundaryClassID`

19. `MainFlowActorClass ::= ("for"| "to") ["the"] ActorID`

20. `MainFlowEntityClass ::= EntityClassID ["on"MainFlowBoundaryClass | "by"MainFlowCommunication | MainFlowsActorClass) ]`

21. `MainFlowCommunication ::= CommunicationID`

22. `ReturnMessage ::= (SystemID TABTRANSVERB SimpleReturnMessage ("to" | "for") ["the"] ActorID | SystemID )`

23. `FlowIf ::= "If "Condition MainFlow ["Else " MainFlow ]"EndIf"`

24. `FlowLoop ::= "Loop "Condition MainFlow "EndLoop"`

25. `FlowConcurrency ::= "StartConcurrency "MainFlow "concurrent" MainFlow "EndConcurrency"`

26. `InteractionUseCase ::= ActorID "executes"( "usecase"| "alternate flow") "useCaseID"(","| "and"| "or") "useCaseID"`

Associadas às regras que contemplam o fluxo principal, seguem as explicações semânticas.

- **Regra 11 (*MainFlowName*)**: define o cabeçalho do fluxo principal da especificação do caso de uso.

- **Regra 12 (*MainFlowScope*)**: estabelece o escopo do fluxo principal da especificação do caso de uso.

- **Regra 13 (*MainFlow*)**: deriva para a regra ***MainFlowElements***, a qual definirá todos os elementos existentes nas sentenças que compõem o fluxo principal e os fluxos alternativos da especificação.

- **Regra 14 (*MainFlowElements*)**: define quais elementos fazem parte das orações que descrevem as ações dos atores e do sistema.

- **Regra 15 (*MainFlowCore*)**: define um conjunto de padrões de escrita das orações, as quais descrevem as interações existentes entre o sistema e os atores.

- **Regra 16 (*MainFlowTabTransVerb*)**: complementa a regra ***MainFlowCore*** e possibilita a utilização de verbos transitivos na especificação do caso de uso.

- **Regra 17 (*MainFlowAttibutes*)**: complementa a regra ***MainFlowCore*** e possibilita a definição dos atributos existentes nas classes.

- **Regra 18 (*MainFlowBoundaryClass*)**: define o nome da classe do tipo fronteira.

- **Regra 19 (*MainFlowActorClass*)**: complementa a regra ***MainFlowCore***.

- **Regra 20 (*MainFlowEntityClass*)**: complementa a regra ***MainFlowCore*** e define os nomes das classes do tipo entidade.

- **Regra 21 (*MainFlowCommunication*)**: complementa a regra ***MainFlowCore*** e define o tipo de comunicação que o sistema terá no envio de mensagens para o usuário, como por exemplo, uma mensagem enviada por *e-mail*.

- **Regra 22 (*ReturnMessage*)**: complementa a regra ***MainFlowCore*** e possibilita a identificação de uma mensagem de retorno.

- **Regra 23 (*FlowIf*)**: define uma estrutura condicional para descrever ações condicionadas, existentes no fluxo do processo. Representará fragmentos combinados condicionais no diagrama de sequência.

- **Regra 24 (*FlowLoop*)**: define uma estrutura que permite executar ações enquanto uma condição for verdadeira. Representará o fragmento combinado ***loop*** no diagrama de sequência.

- **Regra 25 (*FlowConcurrency*)**: permite descrever ações concorrentes uma com a outra.

- **Regra 26 (*InteracionUseCaseID*)**: permite descrever interações entre casos de uso.

Na Listagem 4.3, é apresentado o fluxo *AddAccount* do caso de uso *SB_BankAccount*.

Listagem 4.3 – Fluxo AddAccount - UC SB_BankAccount

```
8  Main Flow: AddAccount.
9  Manager starts Use Case.
10     Customer informs the attributes to the Manager.
11     Manager enters attributes (Int ID, String name, String
          socialSecurity, Date  birthDate) of Customer on MainForm.
12     Manager enters attributes (String agency, String numAccount) of
          Account on MainForm.
13     Manager selects "InsertAccount" on MainForm.
14     System validates attributes of Customer.
15     System validates attributes of Account.
```

```
16    If ["Inconsistency"]
17       System returns "Incorrect data" to Manager.
18    Else
19       System saves attributes of Customer.
20       System saves attributes of Account.
21       System returns "Account successfully inserted" to Manager.
22    EndIf
23 Manager finishes Use Case.
```

Com base nos padrões e nas regras (11-26) apresentadas, é realizado o mapeamento das classes do tipo fronteira e de entidade, seus atributos e métodos. Para que os atributos das classes e seus tipos sejam mapeados, o analista deve explicitá-los na especificação, caso contrário eles não aparecerão nos diagramas. As regras que representam os atributos das classes e seus tipos são ***AttributeID*** e ***AttributeTypeID*** respectivamente. O nome da classe de controle é gerado a partir do nome do caso de uso, pela regra ***ControlClassID***. Já os nomes das classes de fronteira e entidades são gerados pelas regras ***BoundaryClassID*** e ***EntityClassID*** respectivamente, tendo base o padrão apresentado na Seção 4.3. Os métodos da classe do tipo controle são gerados a partir dos nomes de cada fluxo (principal e alternativos), pela regra ***MethodControlClassID***. Já os métodos das classes de fronteira são identificados pela regra ***MethodBoundaryID***.

Para a especificação apresentada na Listagem 4.3, mapeia-se as informações relevantes para geração dos diagramas de classes e sequência.

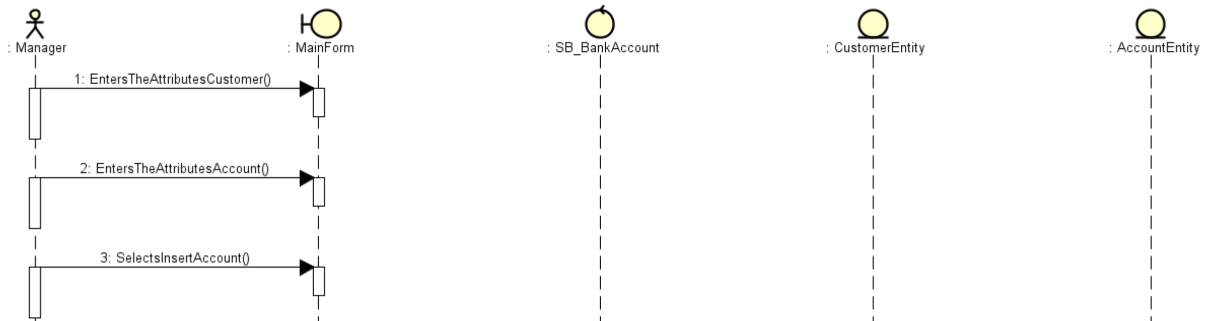
- Linha 8: identifica o método *AddAccount()* para a classe de controle *SB_BankAccount*. A regra da linguagem LUCAM para representação dessa estrutura é ***MainFlowName***.

- Linha 11: identifica a classe de entidade *Customer* e seus atributos. Também define a classe de fronteira *MainForm*. Para o diagrama de sequência, é mapeada uma mensagem entre o ator *Manager* e a classe de fronteira *MainForm*. As regras da gramática que representam essa estrutura são ***MainFlowElements, MainFlowCore, MainFlowTabTransVerb, MainFlowAttributes, MainFlowEntityClass*** e ***MainFlowBoundaryClass***.

- Linha 12: identifica a classe de entidade *Account* e seus atributos. Também é mapeado uma mensagem entre o ator *Manager* e a classe de fronteira *MainForm* para o diagrama de sequência. O padrão dessa oração é idêntico ao anterior.

- Linha 13: identifica o método *InsertAccount()* da classe de fronteira *MainForm* e a interação no diagrama de sequência, conforme Figura 3. As regras da gramática que representam essa estrutura são ***MainFlowElements, MainFlowCore, MainFlowTabTransVerb*** e ***MainFlowBoundaryClass***.

Figura 3 – Diagrama de sequência do fluxo AddAccount - Parte 1



Fonte: Elaborado pelo autor

Observe que no diagrama de sequência parcial ilustrado pela Figura 3, já aparecem as classes identificadas até o momento, mesmo que não haja interações entre elas.

Interessante observar que as mensagens dos diagramas de sequência são mapeadas e obedecem cada padrão de oração definido na Seção 4.3, como por exemplo, a primeira mensagem do diagrama apresentado tem como origem o ator (*Manager*), como destino a classe de fronteira (*MainForm*) e o rótulo é formado pela junção do *Verbo + Substantivo + Preposição + Entidade* (*EntersTheAttributesCustomer()*), conforme a terceira regra apresentada no Quadro 3.

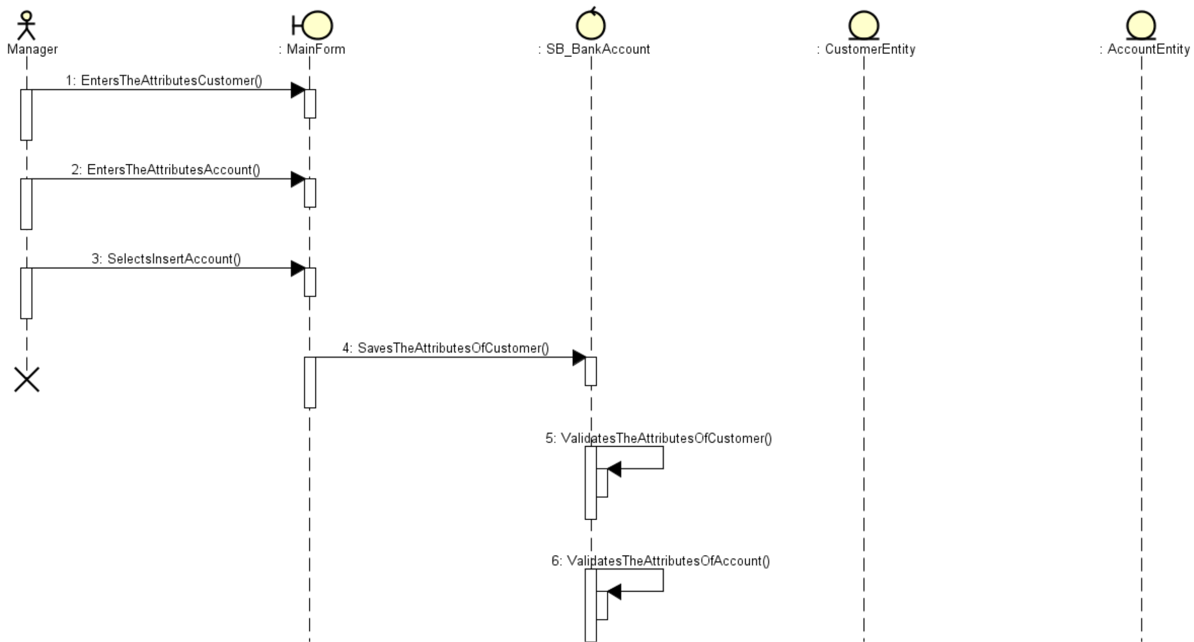
- Linha 14: o verbo *validates* é previamente cadastrado na tabela de verbos e é usado para especificar e identificar métodos/mensagens de validação, o que acontece nas linhas 14 e 15 da especificação apresentada anteriormente. No diagrama de sequência, essa situação resulta numa *auto chamada* na linha de tempo da classe de controle, conforme pode ser observado na Figura 4.

- Linha 16: identifica-se um comando ***If...Else...EndIf***, que se caracteriza numa tomada de decisão no fluxo. Esse fragmento é mapeado pela regra 23 (***FlowIf***) da gramática da linguagem LUCAM. No diagrama de sequência, tal regra é mapeada pelo *parser* para o fragmento combinado *Alt*, conforme pode ser observado na Figura 5.

- Linha 17: identifica-se uma mensagem de retorno para o usuário, sendo que a mesma é disparada caso haja alguma inconsistência. Essa mensagem pode ser observada no diagrama a seguir.

- Linha 19: caso não haja inconsistência, conforme descrito nas linhas 19, 20 e 21 da especificação, o sistema dispara o estímulo para salvar os dados do cliente e da conta e posteriormente retorna uma mensagem para o usuário, conforme pode ser observado na Figura 5. O verbo *returns* é utilizado quando houver a necessidade

Figura 4 – Diagrama de sequência do fluxo AddAccount - Parte 2



Fonte: Elaborado pelo autor

de emitir uma mensagem de retorno ao usuário ou às classes. Esse verbo e todos os outros reconhecidos pela linguagem LUCAM são previamente cadastrados na tabela de verbos.

Está ilustrado pela Figura 5, o diagrama referente ao fluxo *AddAccount* apresentado na Listagem 4.3, no qual o usuário poderá inserir uma conta para o cliente. Foi possível explorar os recursos presentes num diagrama de sequência, como auto chamada, mensagens de retorno, validações e fragmentos combinados.

A seguir são apresentadas as regras gramaticais que representam os fluxos alternativos.

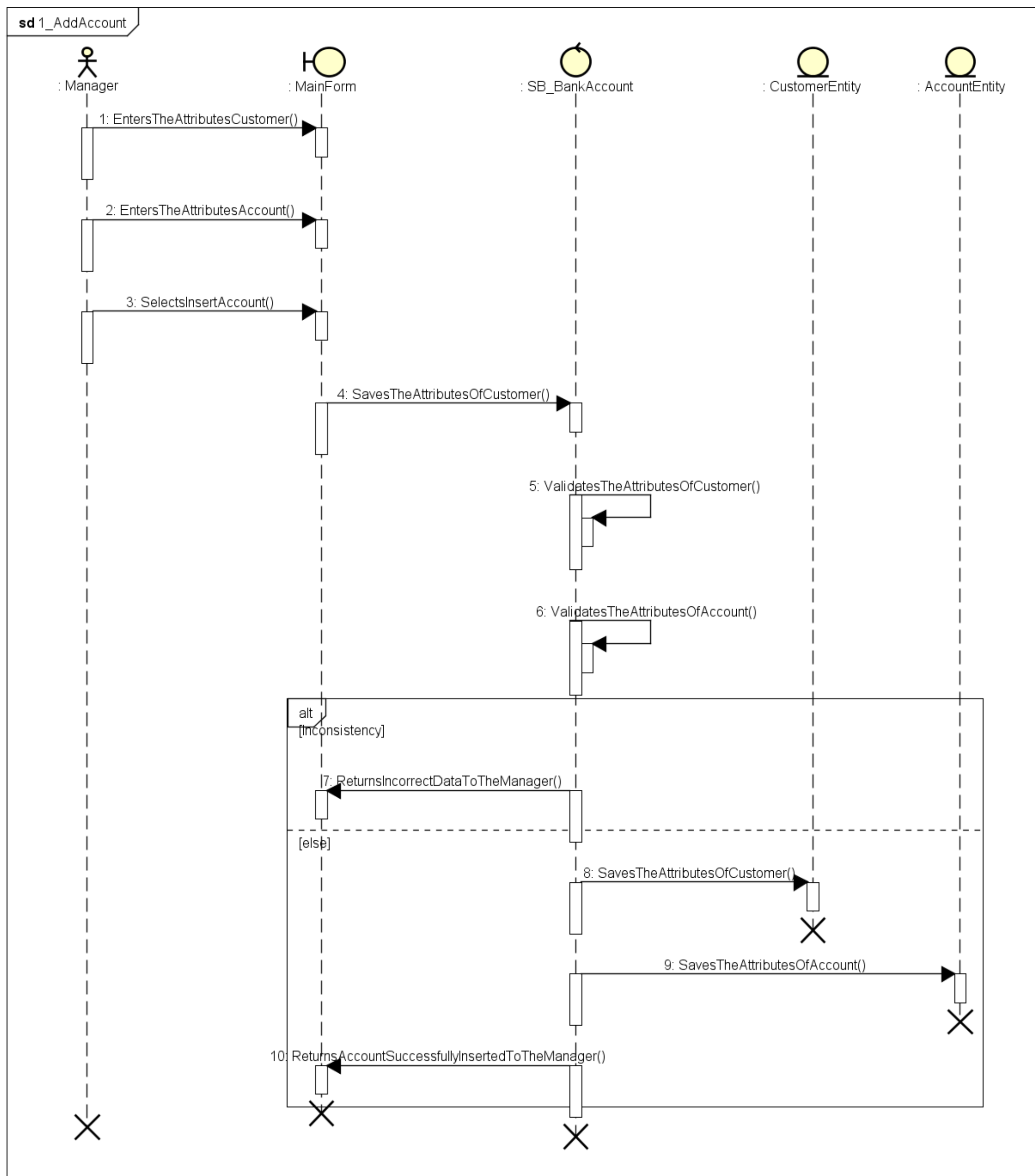
27. `AlternativeFlows ::= "Alternative Flows "AlternativeFlowScope { AlternativeFlowScope}`

28. `AlternativeFlowScope ::= "Alternative Flow "NUM ":  " MethodControlClassID POINT AlternativeFlowCore`

29. `AlternativeFlowCore ::= MainFlow`

A parte da gramática que contempla o fluxo alternativo é trivial, pois a regra (***AlternateFlowCore***) deriva para a regra do fluxo principal (***MainFlow***). Isso implica

Figura 5 – Diagrama de sequência AddAccount - UC SB_BankAccount



Fonte: Elaborado pelo autor

que os mesmos padrões para especificação e mapeamento do fluxo básico são aceitos também para fluxo alternativo.

Associadas às regras que contemplam os fluxos alternativos, segue as explicações semânticas.

- **Regra 27 (*AlternativeFlows*)**: define o cabeçalho dos fluxos alternativos.
- **Regra 28 (*AlternativeFlowScope*)**: define o escopo dos fluxos alternativos.

- **Regra 29 (*AlternativeFlowCore*)**: deriva para a regra ***MainFlow***. Todo padrão de oração aceito no fluxo principal, também será aceito nos fluxos alternativos.

A seguir é apresentada a especificação do fluxo alternativo *CloseAccount* do caso de uso *SB_BankAccount*, conforme a especificação da Listagem 4.4.

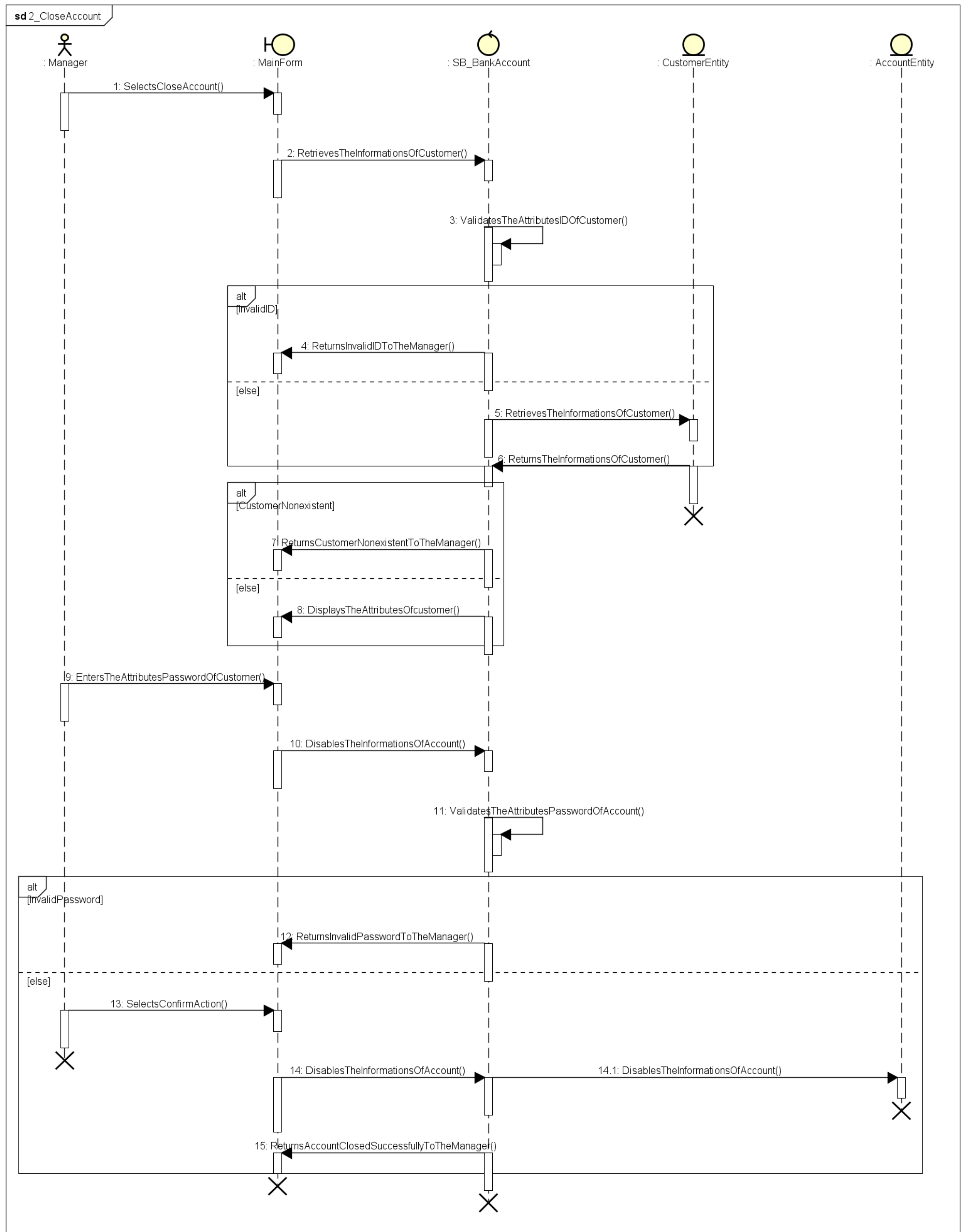
Listagem 4.4 – Fluxo alternativo CloseAccount - UC SB_BankAccount

```
25 Alternate Flow 01: CloseAccount.
26    Customer informs the attributes (ID) for the Manager.
27    Manager selects "CloseAccount" on MainForm.
28    System validates the attributes (ID) of Customer.
29
30    If ["Invalid ID"]
31       System returns "Invalid ID" to Manager.
32    Else
33       System retrieves the informations of Customer.
34    EndIf
35
36    If ["Customer  nonexistent"]
37       System returns "Customer nonexistent" to Manager.
38    Else
39       System displays the attributes of customer on MainForm.
40    EndIf
41
42    Customer informs the attributes (Password) for Manager.
43    Manager enters the attributes (Password) of Customer on MainForm.
44    System validates the attributes (Password) of Account.
45
46    If ["Invalid Password"]
47       System returns "Invalid password" to Manager.
48    Else
49       Manager selects "ConfirmAction" on MainForm.
50       System disables the informations of Account.
51       System returns "Account Closed Successfully" to Manager.
52    EndIf
```

O diagrama de sequência ilustrado pela Figura 6 é referente ao fluxo alternativo *CloseAccount* conforme especificado na Listagem 4.4, no qual o usuário poderá encerrar uma conta bancária caso não exista inconsistência, ou seja, após validação de todos os dados necessários. Para checar essas condições, utilizou-se o comando ***If*** existente na linguagem LUCAM.

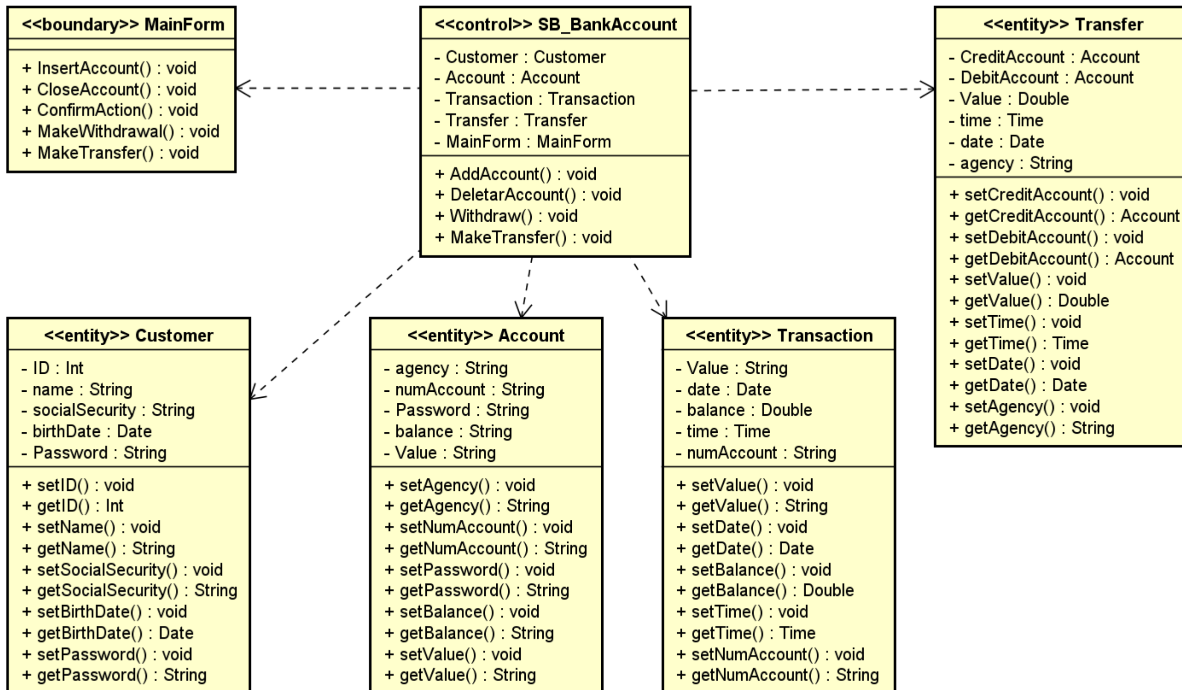
Também foi gerado automaticamente o diagrama de classes, contendo os relacionamentos entre as classes, os atributos e métodos das mesmas, conforme diagrama da Figura 7. Para que os atributos e seus tipos sejam mapeados, o analista deve explicitá-los

Figura 6 – Diagrama de sequência CloseAccount - UC SB_BankAccount



Fonte: Elaborado pelo autor

Figura 7 – Diagrama de classes - SB_BankAccount



Fonte: Elaborado pelo autor

na especificação, caso contrário, eles não aparecerão no diagrama. O usuário tem a opção de especificar os atributos da classe sem seus tipos, portanto, o *parser* definirá o tipo *String* como padrão para esses atributos.

A seguir são apresentadas as regras que compõem o rodapé da especificação de casos de uso.

29. `KeyScenario ::= "Key Scenarios" "Key Scenario" NUM ":" MethodControlClassID POINT`

30. `PreConditions ::= "Preconcitions" [Text] POINT`

31. `PostConditions ::= "Postconditions" [Text] POINT`

32. `SpecialRequirements ::= "Special Requirements" [Text] POINT`

33. `ExtensionPoints ::= "Extension Points" [Text] POINT`

Associadas às regras que contemplam o rodapé da especificação de casos de uso, seguem as explicações semânticas de cada regra.

- **Regra 29 (*KeyScenario*)**: define quais os cenários chaves do caso de uso.

- **Regra 30 (*PreConditions*)**: define as pré-condições de execução do caso de uso.
- **Regra 31 (*PostConditions*)**: define as pós-condições de execução do caso de uso.
- **Regra 32 (*SpecialRequirements*)**: define os requisitos especiais do caso de uso.
- **Regra 33 (*ExtensionPoints*)**: define os pontos de extensão do caso de uso.

O rodapé da especificação do caso de uso *SB_BankAccount* é apresentado a seguir.

Listagem 4.5 – Rodapé - UC SB_BankAccount

```
101 Key Scenarios
102    Key Scenario 01: AddAccount.
103    Key Scenario 02: DisabledAccount.
104 Preconditions
105    "Before this use case begins the actor has logged onto the system".
106 Postconditions
107    "There are no post conditions associated with this use case".
108 Special Requirements
109    "There are no special requirements associated with this use case".
110 Extension Points
111      "There are no extension points associated with this use case".
```

A especificação deste caso de uso e todos os diagramas gerados, encontram-se no Apêndice C.

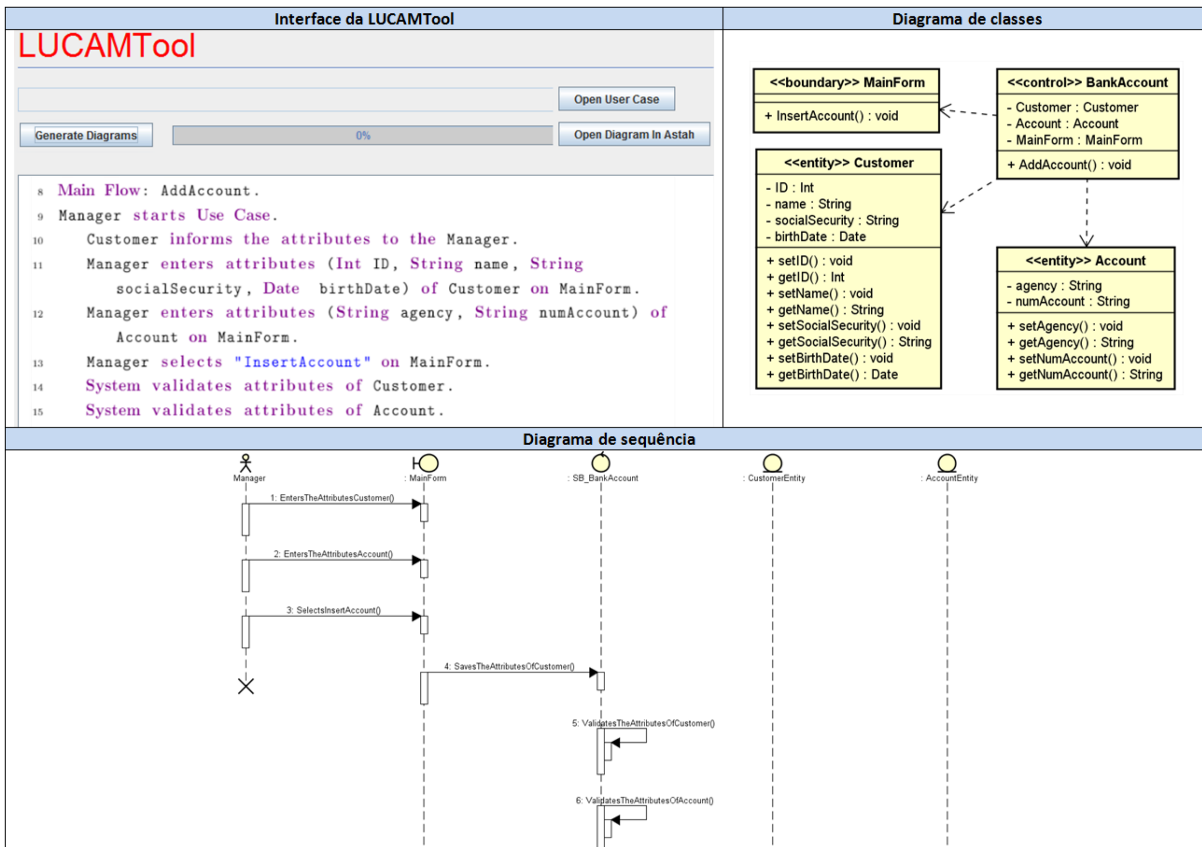
4.5 A ferramenta LUCAMTool

A ferramenta LUCAMTool foi projetada e implementada para apoiar o processo de geração automática dos artefatos e sustentar a linguagem LUCAM. Trata-se de um *parser* que tem o papel de interpretar as especificações elaboradas na linguagem proposta, realizar o mapeamento das mesmas de acordo com as regras e padrões descritos anteriormente, e por fim, gerar os diagramas. A ferramenta LUCAMTool, a gramática da linguagem proposta e os testes realizados estão disponíveis em `https://github.com/assismiranda/LUCAMTool`.

A Figura 8 apresenta a interface da aplicação, juntamente com um exemplo de geração dos diagramas de classes e sequência.

Observe que os diagramas de classes e sequência são fiéis ao que foi especificado textualmente. Outro ponto relevante é a consistência existente entre os diagramas, pois ambos foram gerados seguindo os mesmos critérios e padrões, sendo compostos pelos mesmos elementos, sem divergências.

Figura 8 – Ferramenta LUCAMTool



Fonte: Elaborado pelo autor

A LUCAMTool suporta dois formatos de artefatos, sendo XMI e *.astah*. A geração no formato XMI é interessante, pois é reconhecido por vários softwares de modelagem. Caso não tenha o aplicativo *Astah* instalado em seu computador, o usuário consegue importar o arquivo na extensão *.xmi* e visualizar os diagramas normalmente. O segundo formato é específico do aplicativo *Astah*, o qual foi usado para visualização dos diagramas gerados neste trabalho. Uma das funções da ferramenta LUCAMTool é a abertura dos artefatos diretamente no segundo formato, inicializando o aplicativo *Astah* com os diagramas já carregados. Com essa opção, além da qualidade, ganham-se usabilidade, produtividade, praticidade e agilidade.

Para implementação, utilizou-se a IDE de desenvolvimento *NetBeans*[†] e a linguagem de programação Java.

Um próximo passo do projeto será trabalhar na implementação de uma IDE robusta, tendo como base as melhorias que foram sinalizadas pelas empresas colaboradoras no projeto e identificadas pelos pesquisadores com os testes.

[†]NetBeans: <https://netbeans.org/>

4.6 Considerações finais

Neste capítulo foram apresentados os padrões de orações reconhecidos pela linguagem, sendo os mesmos definidos para que seja possível identificar os elementos necessários para a geração dos diagramas.

Foi apresentada também a linguagem de domínio específico externa denominada LUCAM, proposta neste trabalho de pesquisa. É uma abordagem que permite a automatização e a geração de diagramas de casos de uso, classes e sequência em processos de software orientados a requisitos. Explicou-se o escopo e as regras gramaticais da linguagem, sendo que a mesma se restringe a reconhecer as estruturas de frases definidas em sua gramática, conforme foi descrito na Seção 4.4.

Apresentou-se a ferramenta LUCAMTool desenvolvida para efetuar o mapeamento das especificações de casos de uso textuais na linguagem LUCAM e gerar os artefatos em formato XMI e .astah. Destaca-se que o formato XMI é reconhecido pela maioria dos aplicativos de modelagem de software.

Conclui-se que a abordagem proposta é uma forma de padronizar as especificações de casos de uso textuais, possibilitando a geração automática de diagramas de casos de uso, classes e sequência.

Na próxima seção (Seção 5) serão descritos os métodos utilizados para realização dos testes, os resultados obtidos com os testes realizados em ambiente simulado e real, e uma análise de viabilidade da solução proposta.

5 PROVA DE CONCEITO E ESTUDO DE VIABILIDADE

Nesta seção serão apresentados os resultados gerados com os testes realizados com a linguagem LUCAM e a ferramenta LUCAMTool. Os testes foram divididos em duas etapas, sendo a primeira realizada em ambiente simulado e a segunda em ambiente real, conforme serão descritos a seguir.

5.1 Prova de conceito e testes

Os testes com a abordagem proposta foram realizados em duas fases, sendo a primeira em ambiente simulado, utilizando cenários clássicos retirados na literatura de Engenharia de Software. Posteriormente testou-se a ferramenta com sistemas em empresas de diferentes áreas de atuação, no formato de prova de conceito, incluindo cenários variados.

Prova de conceito ou PoC (*Proof of Concept*) é a definição utilizada para denominar um modelo prático que possa provar o conceito (teórico) estabelecido por uma pesquisa ou trabalho técnico. Pode ser considerado também como uma implementação, resumida ou incompleta, de uma ideia ou método, realizada com o propósito de verificar que o conceito ou teoria em questão é suscetível de ser explorado de uma maneira proveitosa (HENDERSON-SELLERS et al., 2014; GONZALEZ-PEREZ et al., 2016).

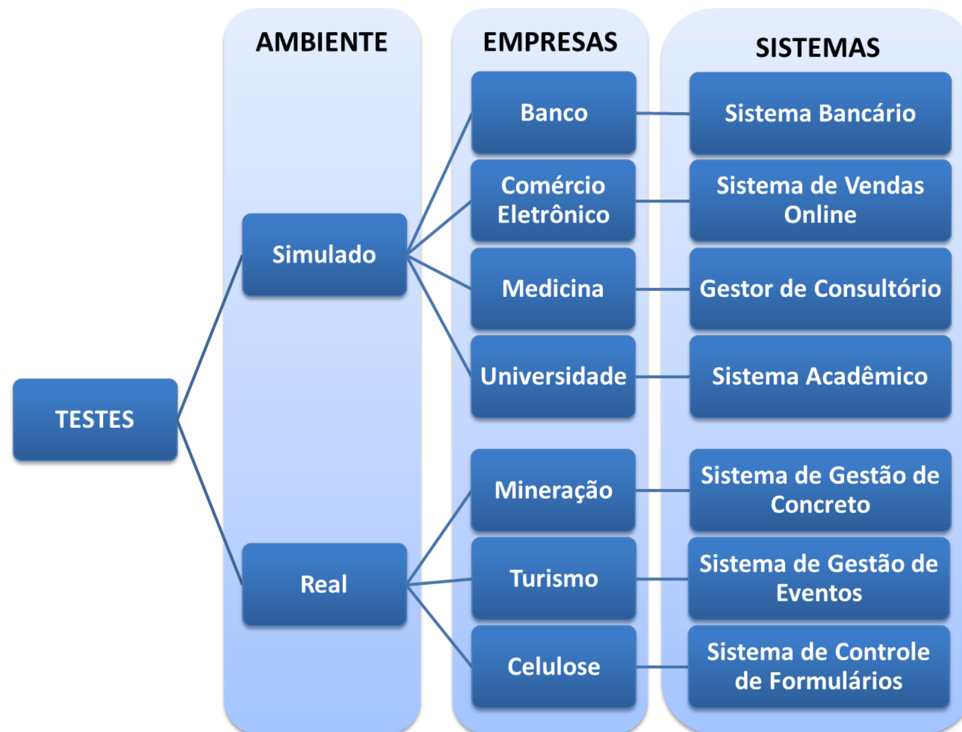
Com a prova de conceito foi possível demonstrar na prática a metodologia proposta, os conceitos e as tecnologias envolvidas na elaboração do projeto. De acordo com Gonzalez-Perez et al. (2016), PoC tem natureza colaborativa, envolvendo a expertise dos projetistas, fornecedores e das competências do cliente. No caso deste projeto, estiveram envolvidos os pesquisadores e a equipe de colaboradores das organizações que contribuíram com os testes, compostas por gerentes, analistas e desenvolvedores.

O fluxo dos testes e os sistemas de softwares selecionados para cada etapa, serão apresentados por meio da Figura 9.

Os cenários dos testes preliminares (ambiente simulado) envolveram especificações de funcionalidades para sistemas acadêmico, bancário, comércio eletrônico e medicina.

No Apêndice C é apresentada a documentação (especificação e diagramas) gerada para o caso de uso *SB_BankAccount*. Esse caso de uso foi utilizado para explicar a DSL proposta e para demonstrar o processo de geração dos artefatos na Seção 4.4.2.

Figura 9 – Fluxo de testes com a LUCAM



Fonte: Elaborado pelo autor

O caso de uso apresentado no Apêndice D é referente ao cenário de uma clínica médica. O atendente é o ator primário e o cliente o ator secundário. O cliente tem o papel de repassar ao atendente seus dados para que sejam cadastrados no sistema. Já o atendente pode inserir, editar, desativar e pesquisar registros de clientes da clínica. O fluxo (*AddPatient*) descreve o processo da inserção de um novo paciente no sistema, no qual o usuário seleciona a opção desejada e o sistema exibe a tela em modo de inserção. Ao receber os dados do paciente como entrada, o sistema valida os campos obrigatórios até que todos sejam preenchidos. Posteriormente o sistema verifica se há registro duplicado no banco de dados e exibe uma notificação com base no resultado de retorno. Por fim, o registro é gravado na base de dados e exibe um *feedback* positivo ao usuário. As especificações dos fluxos alternativos *ModifyPatient*, *DisablePatient*, *ReadPatient* e seus respectivos diagramas são apresentados no Apêndice D.

Nos testes em ambiente simulado foram explorados cenários de um sistema acadêmico e no Apêndice E é apresentada a especificação e os diagramas gerados para o caso de uso “Matricular Estudante”.

Para os testes no ambiente real, foram selecionados três sistemas em três empresas distintas. Todos os sistemas tiveram seus requisitos especificados por meio de casos de usos textuais em linguagem natural. Os diagramas e os códigos-fonte originais foram gerados por especialistas.

O primeiro software escolhido é utilizado para gerenciar o processo de produção e entrega de concreto numa empresa do ramo de mineração, a qual possui atuação em 8 cidades no estado de Minas Gerais. O software possui 115 funcionalidades, sendo 53 do tipo "Gerir Entidade Simples" (que agrupam sub-funcionalidades do tipo ações CRUD-Create, Read, Update e Delete), 25 do tipo de "Gerir Entidade Composta" (que agrupam, cada uma, de quatro a nove sub-funcionalidades "Gerir Entidade", não computadas no grupo anterior), 7 do tipo "Processo de Workflow" e 30 do tipo "Execução de Relatórios", conforme Tabela 1. Esse sistema tem integração com um *Enterprise Resource Planning* (ERP) que já é utilizado na empresa, o qual boa parte dos dados que o alimenta são vindos da base de dados do ERP. A equipe envolvida foi composta por 3 analistas desenvolvedores e 1 gerente.

O segundo software selecionado é utilizado para gestão de eventos de uma empresa nacional, do ramo de turismo. O software selecionado para testes possui 57 funcionalidades, sendo 21 do tipo "Gerir Entidade", 16 do tipo de "Gerir Entidade Composta", 5 do tipo "Processo de Workflow" e 15 do tipo "Execução de Relatórios", conforme Tabela 1. A equipe foi composta por 1 analista desenvolvedor e 1 gerente.

Por fim, selecionou-se um software para o controle e padronização de formulários gerados numa empresa nacional do ramo de celulose. O software possui 109 funcionalidades, sendo 42 do tipo "Gerir Entidade", 33 do tipo de "Gerir Entidade Composta", 6 do tipo "Processo de *Workflow*" e 28 do tipo "Execução de Relatórios", conforme Tabela 1. Esse software também tem integração com um ERP de grande porte já utilizado pela empresa. Estiveram envolvidos 4 analistas desenvolvedores e 1 analista de negócio e 1 gerente.

Tabela 1 – Relação de funcionalidades dos sistemas selecionados

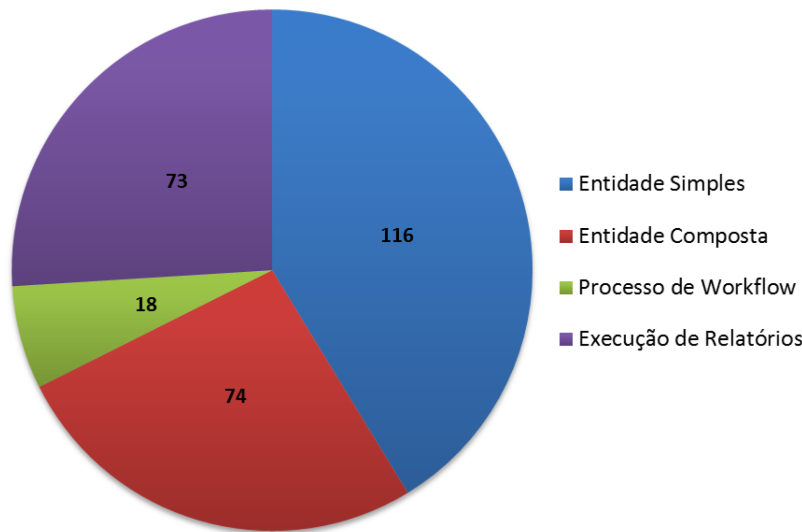
Sistemas de software	**Gerir Entidade**	**Gerir Entidade Composta**	**Processo de Workflow**	**Execução de relatórios**	**Total**
Sistema de Gestão de Concreto	53	25	7	30	115
Sistema de Gestão de Eventos	21	16	5	15	57
Sistema de Controle de Formulários	42	33	6	28	109

Fonte: Elaborado pelo autor

O gráfico apresentado na Figura 10, representa um mapeamento das funcionalidades que fizeram parte dos testes em ambiente real, agrupadas por tipo.

Com base na documentação disponibilizada pelas empresas, os requisitos funcionais foram classificados por tipo, conforme Tabela 1, visando selecionar um conjunto de funções que contemplasse cenários deversificados, inclusive com características existentes nas maiorias dos softwares. Com os testes, teve-se a condição de realizar uma avaliação

Figura 10 – Funcionalidades - Testes reais



Fonte: Elaborado pelo autor

de qualidade com a abordagem proposta neste trabalho.

São apresentadas as especificações e os diagramas obtidos com os testes realizados em dois sistemas, sendo um na empresa do ramo de mineração e o outro na empresa do ramo de celulose. Para fins de demonstração, foram selecionados 2 casos de uso da primeira empresa e 2 casos de uso da segunda, conforme Apêndices F, G, H e I.

Os resultados obtidos (especificações e diagramas (XMI e *.astah*)) e a ferramenta LUCAMTool estão disponíveis em `https://github.com/assismiranda/LUCAMTool/`.

Para manter as regras de *compliance* entre os pesquisadores e as empresas, por pedido das organizações, serão mantidas em sigilo as identidades das mesmas e os documentos originais fornecidos por elas para a realização dos testes. Só serão apresentados os artefatos já escritos na linguagem LUCAM e gerados pela ferramenta LUCAMTool, sendo que todos os documentos foram validados pelas organizações, e após a autorização, foram expostos neste trabalho. Um questionário foi aplicado para as equipes, para que fosse realizada uma análise de viabilidade da abordagem proposta neste trabalho.

Na próxima seção serão apresentadas as considerações sobre os resultados gerados e a análise dos dados coletados com o questionário de pesquisa aplicado para as equipes das empresas colaboradoras com os testes.

5.2 Análise dos resultados

Optou-se por apresentar nesta dissertação, testes com especificações completas, sendo possível explorar cenários que envolvem os elementos dos diagramas de casos de

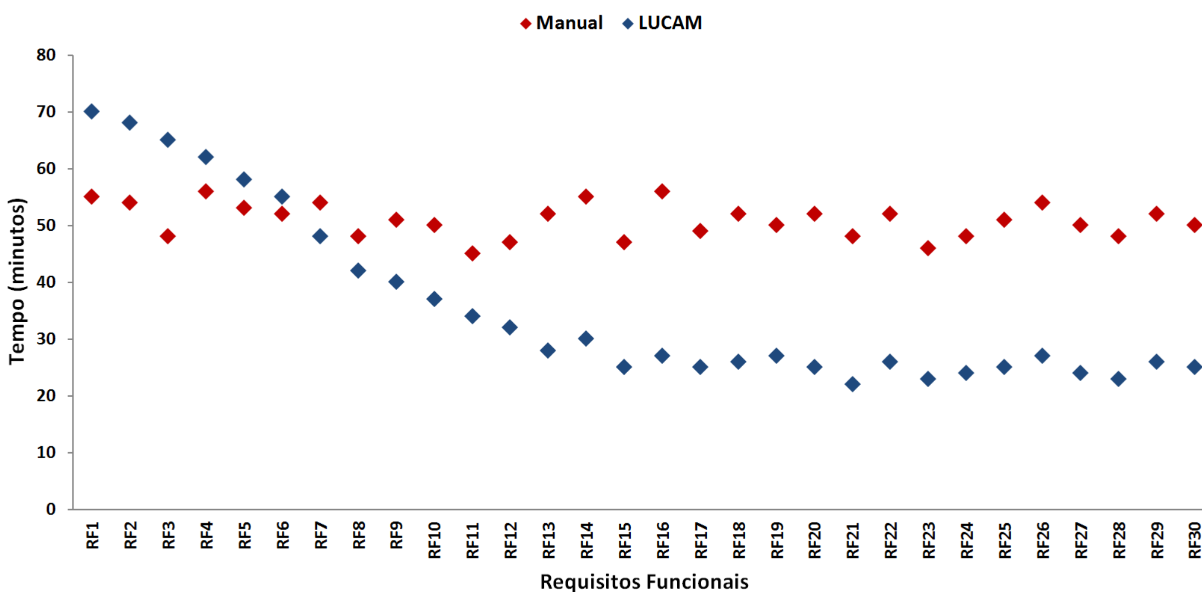
uso, classes e sequência, comprovando uma boa expressividade da linguagem. Além dos elementos básicos, fragmentos combinados (*Alt* e *Loop*), auto chamada e mensagens de retorno também foram contemplados nos diagramas de sequência gerados pela LUCAM-Tool.

Com os testes realizados, observa-se que os diagramas gerados são fiéis ao que foi especificado na linguagem LUCAM. Isso é um fator que incentiva o uso de métodos formais, pois na prática, nota-se que a maioria dos diagramas elaborados manualmente por especialistas, divergem em parte das especificações textuais, não refletindo fielmente o que foi especificado.

A empresa do ramo de celulose disponibilizou seus registros de *TimeSheets*. Foi selecionado para análise um conjunto de 60 Requisitos Funcionais (RF), sendo 30 do tipo "Entidade Simples" e 30 do tipo "Entidade Composta". Nessa análise foi levada em consideração o tempo que cada analista precisou para especificar (textualmente e graficamente) cada caso de uso, na forma manual e pela LUCAMTool.

O gráfico apresentado pela Figura 11, representa o tempo (minutos) necessário para especificação de 30 funcionalidades classificadas como "Entidade Simples".

Figura 11 – Tempo de especificação e modelagem - Entidade Simples



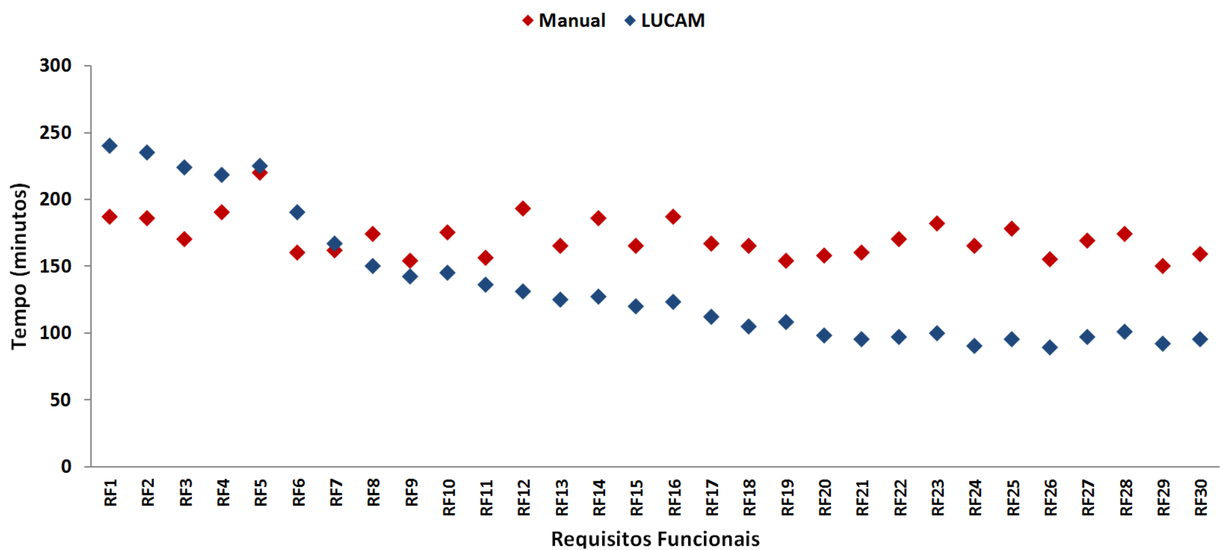
Fonte: Elaborado pelo autor

Analisando o resultado levando em consideração todos os casos de uso, houve aproximadamente 30% de ganho na produtividade. Já a partir do RF15, observa-se um comportamento estável e ao analisar os resultados a partir desse ponto, observa-se que o ganho na produtividade foi de 50% aproximadamente, em comparado com a especificação

realizada manualmente.

No segundo gráfico, representado pela Figura 12, demonstra o tempo necessário para especificação de cada uma das 30 funcionalidades classificadas como "Entidade Composta".

Figura 12 – Tempo de especificação e modelagem - Entidade Composta



Fonte: Elaborado pelo autor

Para os casos de uso classificados como "Entidade Composta", analisando todos os resultados, inclusive no período de aprendizado da linguagem, houve aproximadamente 20% de ganho na produtividade. O comportamento estabilizou a partir do RF20 e a partir desse ponto, o ganho na produtividade foi de aproximadamente 40%.

Nota-se que no início do processo de aprendizagem da linguagem proposta, o analista consome um tempo significado para especificar, pois está conhecendo os elementos e recursos da LUCAM. Por se tratar de uma DSL, seu foco limitado acelera o aprendizado, sobretudo comparando com o aprendizado de uma linguagem de propósito geral. A partir do momento que o usuário conhece a linguagem, ganha-se produtividade, conforme resultados apresentados nos dois gráficos.

Além dos testes realizados em ambiente simulado e real, realizou-se também uma pesquisa com as equipes das empresas colaboradoras com os testes. O modelo do questionário da pesquisa está no Apêndice A e em seguida serão destacados os pontos relevantes que foram identificados ao analisar as respostas sinalizadas pelos entrevistados.

Foi questionado se é usual elaborar documentação de software na empresa. Observou-se que a empresa de menor porte não tem o hábito de documentar os softwares desenvolvidos. Alegaram que não compensa, pois os softwares implementados são pequenos.

Já nas empresas de maior porte, os entrevistados responderam que os softwares desenvolvidos são documentados. Eles também alegam que esse processo de documentação é essencial para contribuir com a qualidade do processo de desenvolvimento e manutenção dos sistemas, inclusive considerando a demanda de manutenção e mudanças constantes. Relataram que com o passar do tempo, é comum a documentação ficar desatualizada ou não refletir fielmente ao que foi implementado.

Ao serem questionados sobre os motivos pelos quais levam a não utilização efetiva da documentação de software, as respostas que ganharam destaque serão apresentadas a seguir.

[...] a documentação demanda muito tempo da equipe e não compensa desenvolvê-la.

[...] os diagramas UML são complexos e demanda muito esforço por parte da equipe de analistas.

Para a questão relacionada ao conhecimento de linguagens de programação, a maioria dos entrevistados tem domínio uma ou mais linguagens, e estariam determinados a aprender outra linguagem para especificar software, sem maiores dificuldades ou resistência, desde que os ganhos sejam concretos, que a linguagem seja de fácil aprendizado e que os problemas inerentes da linguagem natural sejam resolvidos, como por exemplo, a falta de padrão dos documentos e a ambiguidade nas especificações.

A maior parte dos entrevistados consideram vantajosa uma abordagem que automatize parte do processo de desenvolvimento de software e ao serem questionados sobre os resultados apresentados pela ferramenta LUCAMTool, eles consideram viável aplicar a abordagem proposta na prática.

Relacionado a viabilidade da abordagem proposta ou suas características, serão apresentados os depoimentos coletados como respostas ao questionário aplicado.

[...] a linguagem utilizada pela ferramenta é direcional e de fácil aprendizado, tornando rápida a especificação dos casos de uso.

[...] a utilização da linguagem proposta proporciona uma maior precisão na construção dos diagramas gerados e retira em parte a dependência da utilização de softwares específicos para geração dos diagramas...

Ao serem questionados se os diagramas gerados pela LUCAMTool estão de acordo com o que foi especificado e se as especificações que foram transcritas para a linguagem LUCAM estão fiéis aos modelos fornecidos em linguagem natural, a maioria dos entrevistados responderam positivamente, validando os testes realizados.

Após o processo de testes realizados nas empresas, as equipes descreveram os pon-

tos positivos e negativos da abordagem proposta neste trabalho, sendo os depoimentos apresentados a seguir.

- **Positivos:**

  [...] linguagem objetiva, clara e de fácil aprendizado.

  [...] os modelos são definidos de forma fiel à especificação.

  [...] geração de artefatos no formato XMI.

  [...] ganho na produtividade e na qualidade.

- **Negativos:**

  [...] não suportar especificações de processos específicos, como ambiente SAP.

  [...] a IDE necessita de melhorias no editor de código e de inclusão de recursos de visualização dos diagramas gerados.

Os depoimentos a seguir destacam as sugestões de melhorias para a linguagem LUCAM e para a ferramenta LUCAMTool.

[...] integrar com a geração de cenário de testes automatizados.

[...] melhoria no editor e interface da ferramenta LUCAMTool.

[...] refletir alterações no modelo gráfico à especificação de casos de uso.

[...] incluir na linguagem recursos que possibilite especificar no idioma Português.

Destaca-se que os testes realizados nas empresas com cenários variados, foram fundamentais para o sucesso do projeto, mostrando que a abordagem proposta é viável.

As organizações demonstraram boa vontade em ajudar e estão dispostas a contribuírem com a evolução do projeto, acreditando ser vantajoso investir em métodos de documentação e desenvolvimento de sistemas, os quais auxiliem as equipes de analistas e desenvolvedores, possibilitando um ganho de qualidade e produtividade.

Na próxima seção será apresentada a análise conclusiva do projeto, as contribuições e as sugestões de trabalhos futuros.

6 CONCLUSÃO

A DSL LUCAM, juntamente com a LUCAMTool possui recursos que contribuem para uma padronização das etapas do desenvolvimento de sistemas, como a fase de especificação e modelagem de casos de uso e possibilita uma comunicação eficaz entre a equipe, minimizando os problemas derivados da linguagem natural, como ambiguidade, incerteza e complexidade. A ferramenta permite também a geração automática de artefatos de software como diagramas de casos de uso, classes e sequência e consegue-se gerar um esqueleto do código-fonte. A solução oferece inúmeros benefícios, como: melhora na reutilização de artefatos anteriormente desenvolvidos, aumento na qualidade e na produtividade, maior controle sobre cada fase do processo, redução de falhas que podem surgir pela falta de padrão e de uma linguagem trivial entre analista e usuários.

DSLs são compostas por regras sintáticas e semânticas, tornando o treinamento dos usuários uma atividade essencial (FOWLER; PARSONS, 2011; GUPTA, 2015). Percebe-se com os testes realizados que o esforço empenhado no início é compensado à medida que a equipe tem o domínio da linguagem e da ferramenta utilizada. Conclui-se que ao projetar uma DSL, é necessário pensar no seu propósito e nos seus usuários, sendo fundamental definir construções gramaticais simplificadas e buscar atender premissas como simplicidade e objetividade, de modo a oferecer uma sintaxe autoexplicativa, que possibilite reduzir o tempo no aprendizado da linguagem e que a mesma não possua recursos ociosos.

Para aumentar a consistência automática entre um modelo de caso de uso e seu correspondente conjunto de especificações textuais, as representações textuais dos relacionamentos dos casos de uso existentes no diagrama UML devem ser feitas através de identificações eficientes, o que não é trivial. Isso porque a coerência entre um diagrama UML e as descrições textuais exigem formalidade nas especificações.

Conclui-se que o processamento de DSL permite a criação automática de artefatos. Destacam-se vantagens, como:

- Melhora na reutilização de artefatos de software anteriormente desenvolvidos;
- Aumento de produtividade e qualidade do processo de desenvolvimento;
- Melhora na documentação e padronização dos artefatos desenvolvidos;
- Maior controle sobre cada fase do processo de desenvolvimento do software;
- Melhora a comunicação entre analistas e usuários.

Aplicar a abordagem proposta neste trabalho em ambientes diversificados foi fundamental para identificar falhas e para direcionar o projeto, pois, mesmo que em ambiente simulado consegue-se testar a ferramenta com cenários variados, tem situações que só acontecem na prática e à medida que aplicadas em projetos maiores. Os *feedbacks* repassados pelas equipes das organizações contribuíram para o sucesso do projeto e além dos recursos já implementados, direcionaram vários trabalhos futuros.

Percebe-se também que as organizações buscam a cada dia aperfeiçoar seus processos de desenvolvimento de software e estão determinadas a experimentar outras metodologias, desde que auxiliem e propiciam resultados satisfatórios.

6.1 Contribuições

Nesta seção serão descritas as contribuições deste trabalho.

- Definição de uma abordagem que formaliza o processo de especificação textual e modelagem de casos de uso, estabelecendo um padrão para especificar requisitos funcionais de software, o qual possibilite a formalização e, por conseguinte, a identificação de seus elementos para derivações posteriores;
- Uma metodologia que possibilite gerar diagramas UML fiéis ao que foi especificado no detalhamento dos casos de uso;
- Disponibilização de uma ferramenta de apoio (LUCAMTool), que gera automaticamente diagramas de casos de uso, classes e sequência a partir da especificação de casos de uso, eliminando boa parte do trabalho manual realizado pela equipe de analistas e desenvolvedores;
- Disponibilização de um aplicativo que oferece ao usuário uma interface simplificada, com boa usabilidade que auxilie de forma eficiente na tarefa de especificação e modelagem, favorecendo o aumento da produtividade e da qualidade das especificações de caso de uso;
- Incentivar a documentação de software, uma vez que se propõe a geração automática de modelos a partir da especificação de casos de uso;
- Auxiliar na redução dos custos no processo de documentação de software;
- Realização de um estudo de viabilidade robusto, demonstrando a viabilidade técnica do uso da abordagem proposta para três sistemas de softwares reais;
- Publicação de artigo: MIRANDA, Márcio A.; RIBEIRO, Marcos G.; TAVARES, Renan; DIAS, Thiago; MARQUES, Humberto Torres; SONG, Mark. A. J. Song.

An Approach for Generating Class and Sequence Models. In: International Conference on Software Engineering Research & Practice (SERP'16), 2016, Las Vegas. International Conference on Software Engineering Research & Practice. Las Vegas: CSREA Press, 2016. p. 253-259.

- Submissão de artigo: MIRANDA, Márcio A.; RIBEIRO, Marcos G.; MARQUES, Humberto Torres; SONG, Mark. A. J. Song. A Domain-Specific Language for Automatic Generation of UML Models. In: IET Software, 2016.

6.2 Trabalhos Futuros

A seguir são apresentadas propostas de possíveis trabalhos futuros, considerando também as sugestões sinalizadas pelas equipes das organizações que contribuíram com os testes.

- visando ampliar os recursos de reúso, sugere-se implementar um repositório dos artefatos gerados, inclusive com controle de versões;
- ampliar a DSL proposta para suportar também especificações no formato de estórias de usuários, permitindo mapear além dos requisitos funcionais, também os não funcionais;
- propor e implementar uma abordagem que verifique se os modelos gerados estão em conformidade com padrões de projetos preestabelecidos;
- ampliar os testes da ferramenta em ambiente corporativo, expandindo os recursos da LUCAMTool para contemplar a geração de documentação de sistemas mais específicos, como por exemplo, especificações que contemplem processos de software ERP;
- permitir gerar automaticamente outros diagramas UML que não foram contemplados;
- permitir gerar cenários de testes automatizados;
- propor uma abordagem que avalie e otimize o desempenho da ferramenta LUCAMTool na geração dos modelos; e
- implementar uma ferramenta que seja independente, possibilitando aferir a especificação caso os diagramas ou fontes sejam alterados, refletindo as alterações feitas no modelo gráfico à especificação.

Acredita-se que com a continuidade do projeto e com as implementações dos recursos apontados, a abordagem poderá ser amplamente aplicada e contribuirá de forma significativa com a comunidade desenvolvedora de software.

REFERÊNCIAS

ACHOUR, C. B. et al. Guiding use case authoring: Results of an empirical study. In: PROCEEDINGS OF THE 4TH IEEE INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING. Washington, DC, USA: IEEE Computer Society, 1999. (RE '99), p. 36–43. ISBN 0-7695-0188-5.

ADOLPH, S.; COCKBURN, A.; BRAMBLE, P. PATTERNS FOR EFFECTIVE USE CASES. United States of America: Addison-Wesley Longman Publishing Co., Inc., 2002.

ALBUQUERQUE, D. et al. Quantifying usability of domain-specific languages: An empirical study on software maintenance. JOURNAL OF SYSTEMS AND SOFTWARE, v. 101, p. 245 – 259, 2015. ISSN 0164-1212.

ATKINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. SOFTWARE, IEEE, v. 20, n. 5, p. 36–41, Sept 2003. ISSN 0740-7459.

BASIT-UR-RAHIM, M. A.; ARIF, F.; AHMAD, J. Formal verification of sequence diagram using divine. In: COMPUTER APPLICATIONS AND INFORMATION SYSTEMS (WCCAIS), 2014 WORLD CONGRESS ON. Tunisia: [s.n.], 2014. p. 1–6.

BLOBEL, B.; GOOSSEN, W.; BROCHHAUSEN, M. Clinical modeling—a critical analysis. INTERNATIONAL JOURNAL OF MEDICAL INFORMATICS, v. 83, n. 1, p. 57 – 69, 2014. ISSN 1386-5056.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. UML: GUIA DO USUÁRIO. Rio de Janeiro, RJ: Elsevier Brasil, 2006.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. The unified modeling language user guide, 1999. ADDISON-WELSLEY LONGMAN INC, 2010.

BROWNE, G. J.; RAMESH, V. Improving information requirements determination: a cognitive perspective. INFORMATION & MANAGEMENT, v. 39, n. 8, p. 625 – 645, 2002. ISSN 0378-7206.

CHRISSIS, M. B.; KONRAD, M.; SHRUM, S. CMMI FOR DEVELOPMENT: GUIDELINES FOR PROCESS INTEGRATION AND PRODUCT IMPROVEMENT. United States of America: Pearson Education, 2011.

CLEMENTE, P. J. et al. Haais-dsl: Dsl to develop home automation and ambient intelligence systems. In: PROCEEDINGS OF THE SECOND WORKSHOP ON ISOLATION AND INTEGRATION IN EMBEDDED SYSTEMS. New York, NY, USA: ACM, 2009. (IIES '09), p. 13–18. ISBN 978-1-60558-464-5.

COCKBURN, A. WRITING EFFECTIVE USE CASES. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201702258.

D'SOUZA, D. F.; WILLS, A. C. Objects, Components, and Frameworks with UML: The Catalysis Approach. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-31012-0.

ECLIPSE. OpenUP. 2015. Acesso em: 02 ago. 2015. Disponível em: <http://epf.eclipse.org/wikis/openup/index.htm>.

EL-ATTAR, M. A systematic approach to assemble sequence diagrams from use case scenarios. In: Computer Research and Development (ICCRD), 2011 3rd International Conference on. Shanghai, China: [s.n.], 2011. v. 4, p. 171–175.

FOWLER, M. UML essencial: um breve guia para linguagem-padrão de modelagem de objetos. 3.ed. [S.l.]: Bookman, 2005. ISBN 85-363-0454-5.

FOWLER, M.; PARSONS, R. Domain-specific languages. Upper Saddle River (N.J.), Boston, Paris: Addison-Wesley, 2011. (The Addison-Wesley signature series). ISBN 978-0-321-71294-3.

FREITAS, F.; LEITE, J.; SANT'ANNA, M. Aspectos implementacionais de um gerador de analisadores sintáticos para o suporte a sistemas transformacionais. I Simpósio Brasileiro de Linguagens de programação, Belo Horizonte, p. 115–127, 1996.

FU, J. et al. Model-driven development: Where does the code come from? In: Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on. Palo Alto, CA, USA: [s.n.], 2011. p. 255–262.

GHOSH, D. DSLs in Action. 1st. ed. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN 9781935182450.

GHOSH, D. Dsl for the uninitiated. Commun. ACM, ACM, New York, NY, USA, v. 54, n. 7, p. 44–50, jul. 2011. ISSN 0001-0782.

GONZALEZ-PEREZ, C. et al. An ontology for {ISO} software engineering standards: 2) proof of concept and application. Computer Standards & Interfaces, v. 48, p. 112 – 123, 2016. ISSN 0920-5489. Special Issue on Information System in Distributed Environment.

GUEDES, G. T. UML 2 : Uma Abordagem Prática 2A Edição. São Paulo, SP, Brasil: Novatec Editora, 2011.

GUPTA, G. Language-based software engineering. Science of Computer Programming, v. 97, Part 1, p. 37 – 40, 2015. ISSN 0167-6423. Special Issue on New Ideas and Emerging Results in Understanding Software.

HEIJSTEK, W.; CHAUDRON, M. The impact of model driven development on the software architecture process. In: Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on. Lille, France: [s.n.], 2010. p. 333–341. ISSN 1089-6503.

HENDERSON-SELLERS, B. et al. An ontology for iso software engineering standards: 1) creating the infrastructure. Computer Standards and Interfaces, v. 36, n. 3, p. 563–576, 2014. Cited By 11.

HOFFMANN, V. et al. Towards the integration of uml-and textual use case modeling. Journal of Object Technology, v. 8, n. 3, p. 85–100, 2009.

JACOBSON, I. Object-oriented development in an industrial environment. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. New York, NY, USA: ACM, 1987. (OOPSLA '87), p. 183–191. ISBN 0-89791-247-0.

JAYARAMAN, P.; WHITTLE, J. Ucsim: A tool for simulating use case scenarios. In: Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on. [S.l.: s.n.], 2007. p. 43–44.

KARU, M. A textual domain specific language for user interface modelling. In: _____. Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering. New York, NY: Springer New York, 2013. p. 985–996. ISBN 978-1-4614-3558-7.

KOSAR, T.; BOHRA, S.; MERNIK, M. Domain-specific languages: A systematic mapping study. Information and Software Technology, v. 71, p. 77 – 91, 2016. ISSN 0950-5849.

LEITE, J. C. S. et al. O uso do paradigma transformacional no porte de programas cobol. IX Simpósio Brasileiro de Engenharia de Software, pg, p. 397–415, 1995.

LEITE, J. C. S. do P.; SANT'ANNA, M.; FREITAS, F. G. D. Draco-puc: a technology assembly for domain oriented software development. In: IEEE. Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on. [S.l.], 1994. p. 94–100.

LI, L. Translating use cases to sequence diagrams. 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 293, 2000. ISSN 1527-1366.

LIU, L.; ZHANG, J.; KRAFT, N. Using domain specific language for large screen game interaction. In: Games Media Entertainment (GEM), 2014 IEEE. [S.l.: s.n.], 2014. p. 1–6.

LUCRÉDIO, D. Extensão da Ferramenta MVCASE com serviços Remotos de Armazenamento e Busca de Artefatos de Software. Dissertação (Mestrado) — Universidade Federal de São Carlos, 2005.

MATTHEE, M. H.; LEVITT, S. P. Domain specific languages contextualized. In: Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment. New York, NY, USA: ACM, 2011. (SAICSIT '11), p. 310–313. ISBN 978-1-4503-0878-6.

MEMBARTH, R. et al. Hipacc: A domain-specific language and compiler for image processing. IEEE Transactions on Parallel and Distributed Systems, v. 27, n. 1, p. 210–224, Jan 2016. ISSN 1045-9219.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. ACM COMPUT. SURV., ACM, New York, NY, USA, v. 37, n. 4, p. 316–344, dez. 2005. ISSN 0360-0300.

MUKERJI, J.; MILLER, J. Model driven architecture. OMG DOCUMENT, JULY, 2001.

NEIGHBORS, J. M. The draco approach to constructing software from reusable components. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE, n. 5, p. 564–574, 1984.

NEIGHBORS, J. M. The evolution from software components to domain analysis. INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, World Scientific, v. 2, n. 03, p. 325–354, 1992.

OMG. OMG - MODEL DRIVEN ARCHITECTURE. Abril 2015. Acesso em: 25 set. 2015. Disponível em: <http://www.omg.org/mda/>.

OMG. OMG UNIFIED MODELING LANGUAGE TM (OMG UML). 2.5. ed. [S.l.], mar 2015. Acesso em: 22 nov. 2015. Disponível em: <http://www.omg.org/spec/UML/2.5>.

PANACH, J. I. et al. A framework to identify primitives that represent usability within model-driven development methods. INFORMATION AND SOFTWARE TECHNOLOGY, v. 58, p. 338 – 354, 2015. ISSN 0950-5849.

PETRASCH, R.; MEIMBERG, O. Model driven architecture. HEIDELBERG: DPUNKT. VERLAG, 2006.

PRADO, A.; BARRÉRE, T.; BONAFE, V. Case orientada a objetos com múltiplas visões e implementação automática de sistemas-mvcase. In: PROCEEDINGS OF THE XIII BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING. [S.l.: s.n.], 1999. p. 113–128.

PRADO, A. F.; LUCRÉDIO, D. Mvcase: Ferramenta case orientada a objetos. In: XIV SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SESSÃO DE FERRAMENTAS. [S.l.: s.n.], 2000.

PRADO, A. F. do; LUCRÉDIO, D. Ferramenta mvcase-estágio atual: Especificação, projeto e construção de componentes. 2001.

SAVIC, D. et al. Language for use case specification. In: SOFTWARE ENGINEERING WORKSHOP (SEW), 2011 34TH IEEE. [S.l.: s.n.], 2011. p. 19–26. ISSN 1550-6215.

SAWPRAKHON, P.; LIMPIYAKORN, Y. Model-driven approach to constructing uml sequence diagram. In: INFORMATION SCIENCE AND APPLICATIONS (ICISA), 2014 INTERNATIONAL CONFERENCE ON. [S.l.: s.n.], 2014. p. 1–4.

SELIC, B. The pragmatics of model-driven development. IEEE SOFTWARE, v. 20, n. 5, p. 19–25, Sept 2003. ISSN 0740-7459.

SELLAMI, A. et al. A measurement method for sizing the structure of {UML} sequence diagrams. INFORMATION AND SOFTWARE TECHNOLOGY, v. 59, p. 222 – 232, 2015. ISSN 0950-5849.

SHARMA, R.; GULIA, S.; BISWAS, K. Automated generation of activity and sequence diagrams from natural language requirements. In: EVALUATION OF NOVEL APPROACHES TO SOFTWARE ENGINEERING (ENASE), 2014 INTERNATIONAL CONFERENCE ON. [S.l.: s.n.], 2014. p. 1–9.

SOBERNIG, S.; HOISL, B.; STREMBECK, M. Extracting reusable design decisions for uml-based domain-specific languages: A multi-method study. JOURNAL OF SYSTEMS AND SOFTWARE, v. 113, p. 140 – 172, 2016. ISSN 0164-1212.

SOMMERVILLE, I. ENGENHARIA DE SOFTWARE. São Paulo, SP, Brasil: PEARSON BRASIL, 2011.

SOMé, S. S. Supporting use case based requirements engineering. INFORMATION AND SOFTWARE TECHNOLOGY, v. 48, n. 1, p. 43 – 58, 2006. ISSN 0950-5849.

SULTANOV, H.; HAYES, J. Application of swarm techniques to requirements engineering: Requirements tracing. In: REQUIREMENTS ENGINEERING CONFERENCE (RE), 2010 18TH IEEE INTERNATIONAL. [S.l.: s.n.], 2010. p. 211–220. ISSN 1090-705X.

SVENNINGSSON, J.; AXELSSON, E. Combining deep and shallow embedding of domain-specific languages. COMPUTER LANGUAGES, SYSTEMS & STRUCTURES, v. 44, Part B, p. 143 – 165, 2015. ISSN 1477-8424. SI: {TFP} 2011/12.

TAVARES, R. D. UMA LINGUAGEM DE DOMÍNIO ESPECÍFICO PARA GERAÇÃO SEMINAUTOMÁTICA DE DIAGRAMA DE CLASSES A PARTIR DE DETALHAMENTO DE CASOS DE USO. Dissertação (Mestrado) — Pontifícia Universidade Católica de Minas Gerais, PUC-MG, 2014.

THAKUR, J. S.; GUPTA, A. Automatic generation of sequence diagram from use case specification. In: PROCEEDINGS OF THE 7TH INDIA SOFTWARE ENGINEERING CONFERENCE. New York, NY, USA: ACM, 2014. (ISEC '14), p. 20:1–20:6. ISBN 978-1-4503-2776-3.

TIWARI, S.; GUPTA, A. A systematic literature review of use case specifications research. INFORMATION AND SOFTWARE TECHNOLOGY, v. 67, p. 128 – 158, 2015. ISSN 0950-5849.

TUN, T. T. et al. Are your lights off? using problem frames to diagnose system failures. In: REQUIREMENTS ENGINEERING CONFERENCE, 2009. RE '09. 17TH IEEE INTERNATIONAL. [S.l.: s.n.], 2009. p. 343–348. ISSN 1090-705X.

VISIC, N. et al. A domain-specific language for modeling method definition: From requirements to grammar. In: 2015 IEEE 9TH INTERNATIONAL CONFERENCE ON RESEARCH CHALLENGES IN INFORMATION SCIENCE (RCIS). [S.l.: s.n.], 2015. p. 286–297. ISSN 2151-1349.

WILLIAMS, C. et al. Toward engineered, useful use cases. JOURNAL OF OBJECT TECHNOLOGY, v. 4, n. 6, p. 45–57, 2005.

YUE, T.; BRIAND, L. C.; LABICHE, Y. atoucan: An automated framework to derive uml analysis models from use case models. ACM TRANS. SOFTW. ENG. METHODOL., ACM, New York, NY, USA, v. 24, n. 3, p. 13:1–13:52, maio 2015. ISSN 1049-331X.

APÊNDICE A – QUESTIONÁRIO

Questionário

1. **É comum documentar os softwares que são desenvolvidos na empresa?**
Mark only one oval.

- Sim
- Não

2. **A equipe acredita ser importante documentar os softwares, considerando que essa atividade é essencial no processo de desenvolvimento de software?**
Mark only one oval.

- Sim
- Não

3. **Caso não utilize efetivamente a documentação de software, por quais motivos isso ocorre?**
Check all that apply.

- [ ] A documentação demanda muito tempo da equipe e não compensa desenvolvê-la.
- [ ] A documentação não é útil, pois no futuro não é consultada.
- [ ] Falta de profissionais capacitados para a realização da atividade de documentação de software.
- [ ] Falta de ferramentas para documentar software.
- [ ] Os diagramas UML são complexos e demanda muito esforço por parte da equipe de analistas.

4. **Quando documentados, os casos de uso são especificados utilizando linguagem natural?**
Mark only one oval.

- Sim
- Não

5. **Ocorre algum problema com a documentação em linguagem natural? Se sim, sinalize-os de acordo com as opções abaixo.**
Check all that apply.

- [ ] Ambiguidade.
- [ ] Falta de padrão na documentação.
- [ ] Falta de padrão na comunicação da equipe.
- [ ] Produtividade baixa na modelagem e no desenvolvimento do software.
- [ ] Os desenvolvedores tem dificuldades em entender o que foi especificado.
- [ ] O software desenvolvido não reflete fielmente o que foi especificado.

6. **Todos os analistas da empresa conhecem pelo menos uma linguagem de programação?**
Mark only one oval.

- Sim
- Não

7. **A equipe estaria disposta a aprender uma nova linguagem para especificar software, que não seja a linguagem natural?**
Mark only one oval.

- Sim
- Não

8. **Descreva aqui os motivos que te levaram a escolher a resposta da questão anterior (Sim ou Não)**

9. **A equipe acha interessante uma abordagem que automatize parte do processo de desenvolvimento de software?**
Mark only one oval.

- Sim
- Não

10. **Após resultados apresentados pela ferramenta LUCAMTool, a equipe acha viável a aplicação da abordagem proposta em ambiente real de desenvolvimento de software?**
Mark only one oval.

- Sim
- Não

11. **Descreva aqui os motivos que te levaram a escolher a resposta da questão anterior (Sim ou Não)**

12. **Os diagramas gerados pela LUCAMTool estão de acordo com o que foi especificado?**
Mark only one oval.

- Sim
- Não

13. **As especificações que foram transcritas para a linguagem LUCAM estão fiéis aos modelos fornecidos em linguagem natural?**
Mark only one oval.

- Sim
- Não

14. **Descreva os pontos positivos e negativos da LUCAMTool.**

15. **Descreva aqui as sugestões de melhoria para a LUCAMTool.**

Powered by Google Forms

APÊNDICE B – GRAMÁTICA LUCAM

A gramática completa da linguagem de especificação LUCAM na notação EBNF é apresentada a seguir:

1.LUCAM ::= UseCaseHeader UseCaseFlows UseCaseFooter

2.UseCaseHeader ::= UseCaseName UseCaseBriefDescription SystemName PrimarySecondaryActors

3.UseCaseFlows ::= MainFlowName MainFlowScope [AlternativeFlows]

4.UseCaseFooter ::= Key Scenario {KeyScenario} PreConditions PosConditions SpecialRequirements ExtensionPoints

5.UseCaseName ::= "Use Case: "ControlClassID POINT

6.UseCaseBriefDescription ::= "Brief Description"[Text] POINT

7.SystemName ::= "System: "SystemID POINT

8.PrimarySecondaryActors ::= "Primary and Secondary Actors " PrimaryActorName [SecondaryActorName]

9.PrimaryActorName ::= "Primary Actors: " ActorID {","ActorID} POINT

10.SecondaryActorName ::= "Secondary Actors: " ActorID { ","ActorID} POINT

11.MainFlowName ::= "Main Flow: "MethodControlClassID POINT

12.MainFlowScope ::= [ActorID "starts Use Case"POINT] MainFlow [ActorID "finishes Use Case"POINT]

13.MainFlow ::= MainFlowElements {MainFlowElements}

14.MainFlowElements ::= MainFlowCore | FlowIf | FlowLoop | FlowConcurrency

15.MainFlowCore ::= ((ActorID | SystemID) (MainFlowTabTransVerb | TABINTRANSVERB) POINT) | ReturnMessage

16.MainFlowTabTransVerb ::= TABTRANSVERB ["MethodBoundaryID"] (MainFlowAttibutes | "on"MainFlowBoundaryClass)

17.MainFlowAttibutes ::= [["the"] TABNOUN ["("[AttributeTypeID] AttributeID {, [AttributeTypeID] AttributeID}")"]] (("on"MainFlowBoundaryClass | ("of") MainFlowEntityClass)) | MainFlowsActorClass)

18.MainFlowBoundaryClass ::= BoundaryClassID

19.MainFlowActorClass ::= ("for"| "to") ["the"] ActorID

20.MainFlowEntityClass ::= EntityClassID ["on"MainFlowBoundaryClass | "by"MainFlowCommunication | MainFlowsActorClass)]

21.MainFlowCommunication ::= CommunicationID

22.ReturnMessage ::= (SystemID TABTRANSVERB SimpleReturnMessage ("to" | "for") ["the"] ActorID | SystemID)

23.FlowIf ::= "If "Condition MainFlow ["Else " MainFlow]"EndIf"

24.FlowLoop ::= "Loop "Condition MainFlow "EndLoop"

25.FlowConcurrency ::= "StartConcurrency "MainFlow "concurrent" MainFlow "EndConcurrency"

26.InteractionUseCase ::= ActorID "executes"("usecase"| "alternate flow") "useCaseID"(","| "and"| "or") "useCaseID"

27.AlternativeFlows ::= "Alternative Flows "AlternativeFlowScope { AlternativeFlowScope}

28.AlternativeFlowScope ::= "Alternative Flow "NUM ": " MethodControlClassID POINT AlternativeFlowCore

29.AlternativeFlowCore ::= MainFlow

30.KeyScenario ::= "Key Scenarios" "Key Scenario" NUM ":" MethodControlClassID POINT

31.PreConditions ::= "Preconcitions" [Text] POINT

32.PostConditions ::= "Postconditions" [Text] POINT

33.SpecialRequirements ::= "Special Requirements" [Text] POINT

```
34.ExtensionPoints ::= "Extension Points" [Text] POINT

35.POINT := .
```

A seguir é apresentada a explicação semântica associada a cada regra:

- **Regra 1 (*LUCAM*)**: Essa regra representa o esqueleto geral do detalhamento de casos de uso, sendo divida em três grupos:
  - Cabeçalho do detalhamento de caso de uso: representado pela regra ***UseCaseHeader***;
  - Fluxos presentes no detalhamento de caso de uso: representado pela regra ***UseCaseFlows***;
  - Rodapé do detalhamento de caso de uso: representado pela regra ***UseCaseFooter***.
- **Regra 2 (*UseCaseHeader*)**: essa deriva um conjunto de regras que definem as partes preliminares da especificação do caso de uso, como o nome do caso de uso, uma breve descrição do caso de uso, o nome do sistema, e os nomes dos atores primários e secundários.
- **Regra 3 (*FlowControl*)**: por meio dessa regra, é possível a geração dos fluxos básicos e alternativos do caso de uso. Mais detalhes dessa regra serão explicados posteriormente.
- **Regra 4 (*UseCaseFooter*)**: são derivadas dessa regra, as representações de informações presentes no rodapé do detalhamento de casos de uso, consideradas informações complementares da especificação. Estão presentes nessa parta da especificação, informações como cenários principais, pré e pós condições, requisitos especiais e pontos de extensão.
- **Regra 5 (*UseCaseName*)**: essa regra define o nome do caso de uso e por meio dele também é mapeado o nome da classe de controle, presentes no diagrama de classes e no diagrama de sequência.
- **Regra 6 (*UseCaseBriefDescription*)**: essa regra representa uma breve descrição presente no detalhamento do caso de uso.
- **Regra 7 (*SystemName*)**: essa regra define o nome do sistema. Essa regra é importante, principalmente para a geração do diagrama de sequência, pois, a partir dessa informação, é possível identificar as ações realizadas pelo sistema e as ações

realizadas pelos atores. Além disso é possível verificar inconsistências do nome do sistema sinalizado ao longo da especificação dos casos de uso.

- **Regra 8 (*PrimarySecondaryActors*)**: essa regra define a estrutura para especificar os nomes dos atores primários e secundários.

- **Regra 9 (*PrimaryActorName*)**: essa regra define os nomes dos atores primários que interagem com o sistema, possibilitando identificar o perfil de usuário que vai utilizar o software e qual é o seu papel e que tipo de privilégios ele terá.

- **Regra 10 (*SecondaryActorName*)**: quando existir atores secundários, essa regra define o nome dos mesmos.

- **Regra 11 (*MainFlowName*)**: essa regra define o cabeçalho do fluxo principal da especificação do caso de uso. Por meio dela identifica-se os métodos existentes na classe de controle e também é mapeado o nome de cada diagrama de sequência a ser gerado para esse caso de uso, considerando que para cada fluxo, gera-se um diagrama de sequência separado, facilitando assim a leitura e o entendimento dos mesmos.

- **Regra 12 (*MainFlowScope*)**: essa regra estabelece o escopo do fluxo principal da especificação do caso de uso. A regra ***MainFlowScope*** também estabelece que o fluxo principal deve sempre ser iniciado e encerrado por um ator.

- **Regra 13 (*MainFlow*)**: essa regra deriva para a regra ***MainFlowElements***, a qual definirá todos os elementos presente nas sentenças que compõem o fluxo principal e os fluxos alternativos da especificação do casos de uso.

- **Regra 14 (*MainFlowElements*)**: essa regra define quais elementos fazem parte das orações que descrevem as ações dos atores e do sistema.

- **Regra 15 (*MainFlowCore*)**: essa regra define um conjunto de padrões de escrita das orações, as quais descrevem as interações existentes entre o sistema e os atores, ações essas presentes no fluxo principal e nos fluxos alternativos do caso de uso.

- **Regra 16 (*MainFlowTabTransVerb*)**: essa regra complementa a regra ***MainFlowCore*** e possibilita a utilização de verbos transitivos na especificação do caso de uso. Para facilitar a implementação da regra, criou uma tabela de verbos, denominada ***TABTRANSVERB***.

- **Regra 17 (*MainFlowAttibutes*)**: essa regra é uma complementação da regra ***MainFlowCore*** e possibilita a definição dos atributos existentes nas classes. Isso auxilia na geração automática do diagrama de classes e de sequência.

- **Regra 18 (*MainFlowBoundaryClass*)**: essa regra define o nome da classe de fronteira, presente tanto no diagrama de classe como no diagrama de sequência.
- **Regra 19 (*MainFlowActorClass*)**: essa regra é uma complementação da regra ***MainFlowCore*** e auxilia na geração do diagrama de classes e de sequência. Mais especificamente, no diagrama de sequência, estabelece que o ator será o receptor da mensagem enviada pelo sistema.
- **Regra 20 (*MainFlowEntityClass*)**: essa regra é uma complementação da regra ***MainFlowCore*** e define os nomes das classes de entidade, presentes nos diagramas de classes e sequência.
- **Regra 21 (*MainFlowCommunication*)**: essa regra é uma complementação da regra ***MainFlowCore*** e define o tipo de comunicação que o sistema terá no envio de algumas mensagens para o usuário, como por exemplo, uma mensagem enviada por *e-mail*.
- **Regra 22 (*ReturnMessage*)**: essa regra é uma complementação da regra ***MainFlowCore*** e possibilita especificar situações presentes no caso de uso, que indicam interação de mensagens de retorno entre as classes e do sistema para o ator.
- **Regra 23 (*FlowIf*)**: essa regra define uma estrutura condicional para descrever determinadas ações condicionadas, presentes no fluxo do processo. Também servirá para mapear fragmentos combinados no diagrama de sequência.
- **Regra 24 (*FlowLoop*)**: essa regra define uma estrutura que permite executar determinadas ações enquanto uma condição for satisfeita. Também serve para mapear o fragmento combinado ***loop*** no diagrama de sequência.
- **Regra 25 (*FlowConcurrency*)**: essa regra permite descrever ações que concorrentes. Também permite o mapeamento do fragmento combinado no diagrama de sequência.
- **Regra 26 (*InteracionUseCaseID*)**: permite descrever interações entre casos de uso.
- **Regra 27 (*AlternativeFlows*)**: essa regra define o cabeçalho dos fluxos alternativos.
- **Regra 28 (*AlternativeFlowScope*)**: essa define o escopo dos fluxos alternativos presentes na especificação do caso de uso.
- **Regra 29 (*AlternativeFlowCore*)**: Essa regra deriva para a regra ***MainFlow***. Todo padrão de oração aceito no fluxo principal, também será aceito nos fluxos alternativos.

- **Regra 30 (*KeyScenario*)**: essa regra define quais os cenários chaves do caso de uso. Para cada cenário, no diagrama de classes é mapeado e gerado gerado um método na classe de controle.

- **Regra 31 (*PreConditions*)**: essa regra estabelece as pré-condições necessárias para a execução do caso de uso, ou seja, as restrições existentes para que o caso de uso possa ser executado; geralmente, uma pré-condição indica que um caso de uso já foi executado para confirmar a condição.

- **Regra 32 (*PostConditions*)**: essa regra define as pós-condições de execução do caso de uso, ou seja, num caso de uso, identifica os possíveis estados em que o sistema pode ficar ao término da execução do caso de uso. Assim, o sistema deve estar num desses estados ao final da execução do caso de uso. Representa uma condição que será verdadeira quando o caso de uso terminar, independentemente de como ele terminar. Também representa uma condição que será verdadeira quando o caso de uso termina em sucesso, independentemente de qual caminho ele seguiu.

- **Regra 33 (*SpecialRequirements*)**: essa regra estabelece os requisitos especiais para o caso de uso; mais especificadamente, são os requisitos não funcionais, ou seja, são todos os requisitos no caso de uso que não são cobertos pelo fluxo de eventos. Estão relacionados a características do software e não à suas funcionalidades.

- **Regra 34 (*ExtensionPoints*)**: essa regra define os pontos de extensão presentes na especificação do caso de uso. Para Cockburn (2000), um caso de uso pode ter várias extensões; um ponto de extensão é um relacionamento entre um caso de uso de extensão e um caso de uso base, definindo em que parte do caso de uso base o comportamento de um caso de uso de extensão deve ser inserido.

As regras ***SimpleReturnMessage***, ***Text*** e ***ConditionText*** são representadas por "***String***". Na especificação, essas são representadas por textos comuns, entre aspas.

APÊNDICE C – CASO DE USO “CONTA BANCÁRIA”

A seguir serão apresentadas a especificação referente ao caso de uso. Neste o usuários poderá adicionar e desabilitar uma conta, realizar movimentações como saque e transferência.

Listagem C.1 – Especificação do caso de uso - SB_BankAccount

```
1  Use Case: SB_BankAccount.
2  Brief Description
3  "Allows the user perform operations such, as insertion and deleting of
       account data. Withdraw cash and carry out bank transfers.".
4  System: System.
5  Primary and Secondary Actors
6  Primary Actors: Manager.
7  Secondary Actors: Customer.
8  Main Flow: AddAccount.
9  Manager starts Use Case.
10    Customer informs the attributes to the Manager.
11    Manager enters attributes (Int ID, String name, String
         socialSecurity, Date  birthDate) of Customer on MainForm.
12    Manager enters attributes (String agency, String numAccount) of
         Account on MainForm.
13    Manager selects "InsertAccount" on MainForm.
14    System validates attributes of Customer.
15    System validates attributes of Account.
16    If ["Inconsistency"]
17       System returns "Incorrect data" to Manager.
18    Else
19       System saves attributes of Customer.
20       System saves attributes of Account.
21       System returns "Account successfully inserted" to Manager.
22    EndIf
23 Manager finishes Use Case.
24 Alternate Flow 01: CloseAccount.
25    Customer informs the attributes (ID) for the Manager.
26    Manager selects "CloseAccount" on MainForm.
27    System validates the attributes (ID) of Customer.
28
29    If ["Invalid ID"]
30       System returns "Invalid ID" to Manager.
31    Else
32       System retrieves the informations of Customer.
```

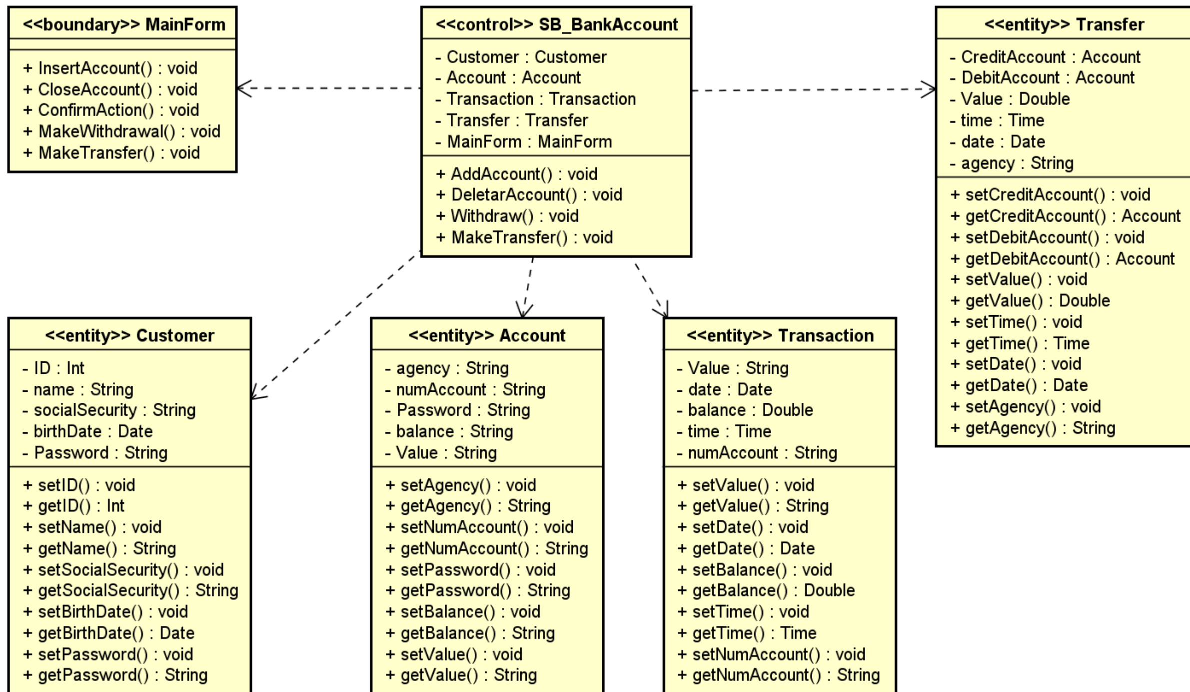
```
33      EndIf
34
35      If ["Customer  nonexistent"]
36         System returns "Customer nonexistent" to Manager.
37      Else
38         System displays the attributes of customer on MainForm.
39      EndIf
40
41      Customer informs the attributes (Password) for Manager.
42      Manager enters the attributes (Password) of Customer on MainForm.
43      System validates the attributes (Password) of Account.
44
45      If ["Invalid Password"]
46         System returns "Invalid password" to Manager.
47      Else
48         Manager selects "ConfirmAction" on MainForm.
49         System disables the informations of Account.
50         System returns "Account Closed Successfully" to Manager.
51      EndIf
52      Alternate Flow 02: Withdraw.
53      Customer selects "MakeWithdrawal" on MainForm.
54      Customer informs the attributes (String numAccount, String  Password)
           of Account on MainForm.
55      System searches for Customer.
56
57      If ["Customer nonexistent"]
58         System returns  "Customer nonexistent" to Customer.
59      Else
60         System validates the attributes (Password) of Customer.
61      EndIf
62
63      If ["Invalid Password"]
64         System returns "Invalid password" to Customer.
65      Else
66         Customer informs the attributes (Value) of Transaction on MainForm
              .
67         Customer selects "MakeWithdrawal" on MainForm.
68         System retrieves the attributes (balance) of Account.
69      EndIf
70
71      If ["Balance Unavailable"]
72         System returns "Balance unavailable" to Customer.
73      Else
74           System saves the informations (Date  date, Double  balance, Time
                 time, String numAccount) of Transaction.
75         System deducts the attributes (Double  balance) of Account.
76         System returns "Booty successfully performed" to Customer.
```

```
77     EndIf
78     Alternate Flow 03: MakeTransfer.
79     Customer selects "MakeTransfer" on MainForm.
80      Customer enters the informations (Account CreditAccount, Account
           DebitAccount, Double Value) of Transfer on MainForm.
81     Customer informs the attributes (Password) of Account.
82     System retrieves the informations of Account.
83     System validates the attributes (Password) of Account.
84
85     If ["Invalid Password"]
86        System returns "Invalid password" to Customer.
87     EndIf
88
89     If ["Balance Unavailable"]
90        System returns "Balance unavailable" to Customer.
91     Else
92        System displays the informations of Transfer on MainForm.
93     EndIf
94
95     Customer selects "ConfirmAction" on MainForm.
96     System deducts the attributes (Value) of Account.
97     System credits the attributes (Value) of Account.
98     System saves the informations (Account CreditAccount, Conta
          DebitAccount, Time  time, Date  date, String agency, Double value)
           of Transfer.
99     System returns "Was successful transfer" to Customer.
100    Key Scenarios
101    Key Scenario 01: AddAccount.
102    Key Scenario 02: DisabledAccount.
103 Preconditions
104    "Before this use case begins the actor has logged onto the system".
105 Postconditions
106    "There are no post conditions associated with this use case".
107 Special Requirements
108    "There are no special requirements associated with this use case".
109 Extension Points
110       "There are no extension points associated with this use case".
```

A seguir é apresentado o diagrama de classes completo gerado automaticamente com base na especificação de caso de uso.

Abaixo, ilustrado pela Figura 14, o diagrama completo referente ao fluxo principal *AddAccount* apresentado na Listagem C.1, no qual o usuário poderá inserir uma nova conta para o cliente. Aqui foi possível explorar os principais recursos presentes num diagrama de sequência, como auto-chamada, mensagens de retorno, validações e fragmentos combinados.

Figura 13 – Diagrama de classes - SB_BankAccount



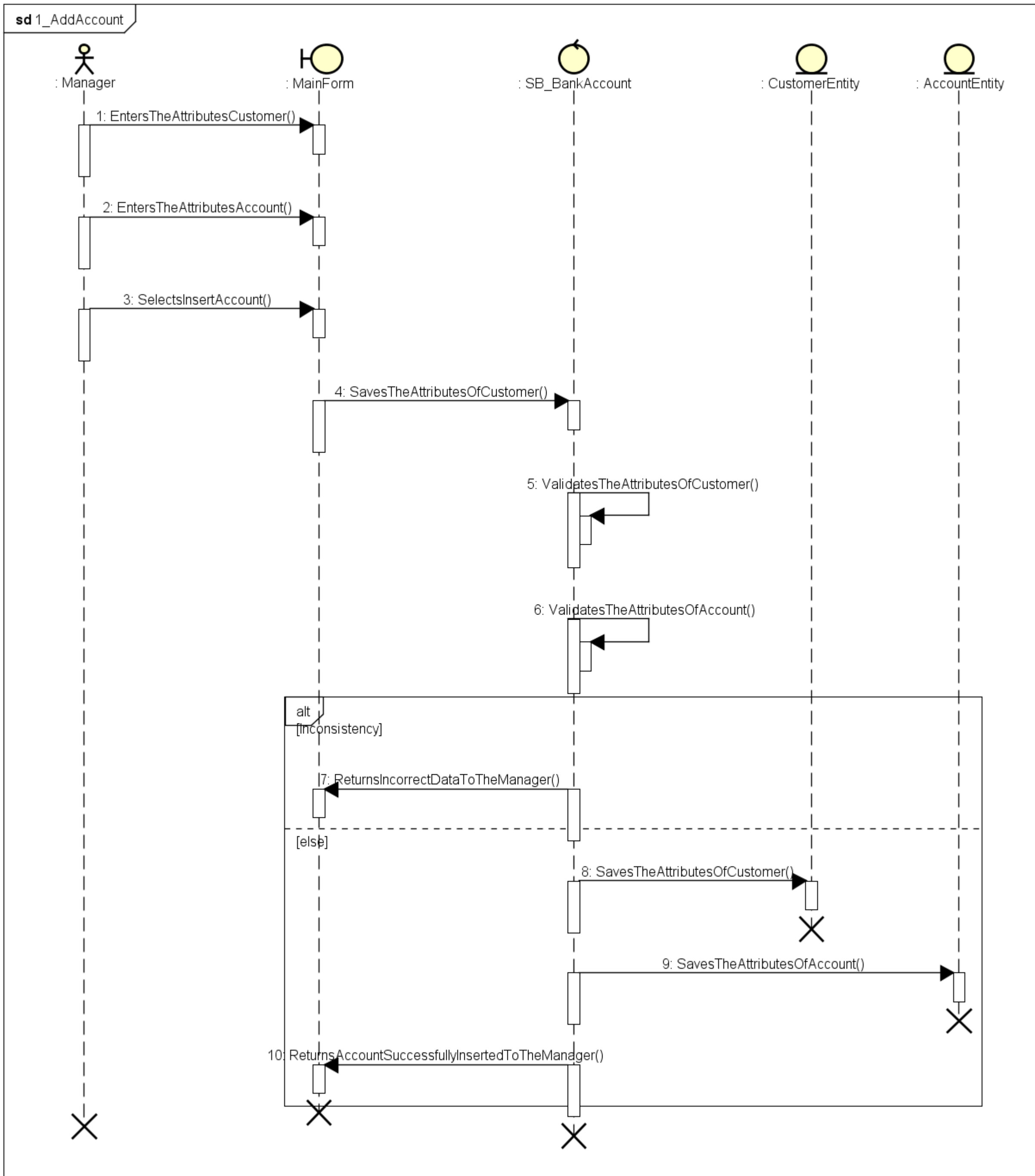
Fonte: Elaborado pelo autor

O diagrama de sequência ilustrado pela Figura 15 é referente ao fluxo alternativo *CloseAccount*, no qual o usuário poderá fechar uma conta bancária caso não exista nenhuma inconsistência, ou seja, após todos os dados necessários sejam validados.

O diagrama de sequência representado pela Figura 16 é referente ao fluxo alternativo *Withdraw*. Nesse fluxo o usuário poderá efetuar saque na conta, se existir saldo disponível e caso não possua nenhuma inconsistência, como, cliente inexistente ou usuário não autenticado.

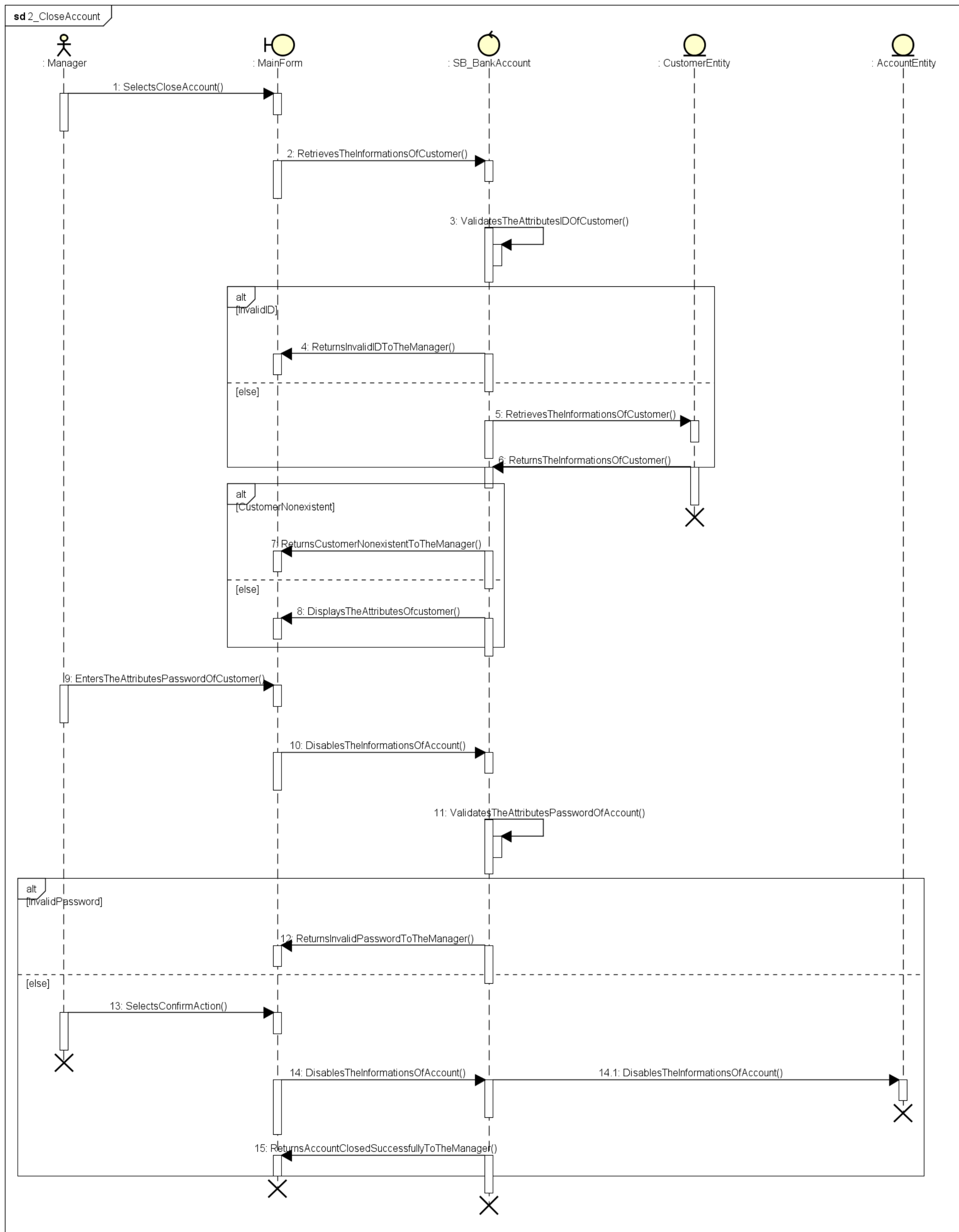
Por meio da Figura 17, é apresentado o diagrama de sequência referente ao fluxo alternativo *MakeTransfer*.

Figura 14 – Diagrama de sequência AddAccount - UC SB_BankAccount



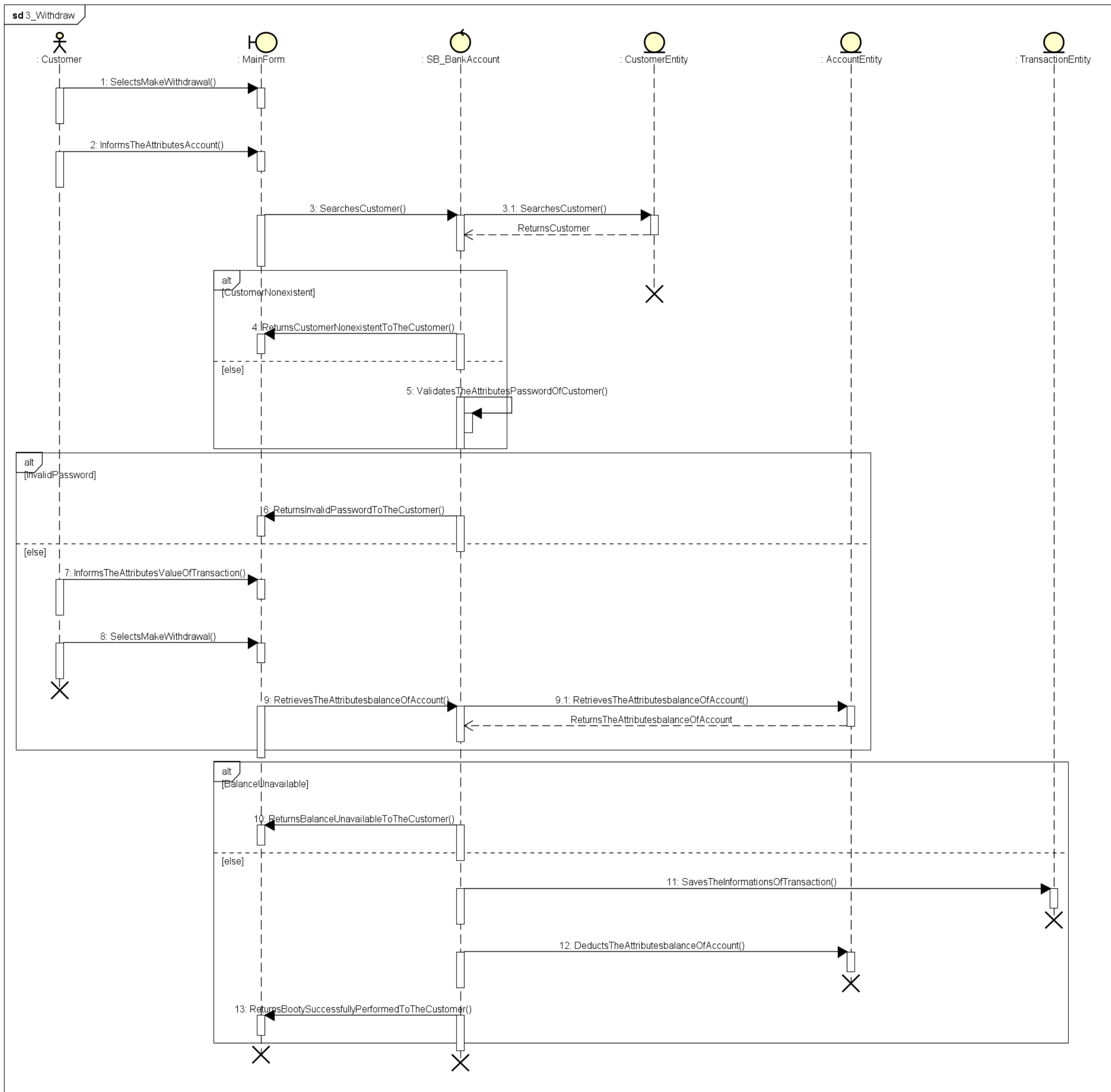
Fonte: Elaborado pelo autor

Figura 15 – Diagrama de sequência CloseAccount - UC SB_BankAccount



Fonte: Elaborado pelo autor

Figura 16 – Diagrama de sequência Withdraw - UC SB_BankAccount



Fonte: Elaborado pelo autor

Figura 17 – Diagrama de sequência MakeTransfer - UC SB_BankAccount

Fonte: Elaborado pelo autor

APÊNDICE D – CASO DE USO “MANTER PACIENTE”

O caso de uso apresentado a seguir é referente ao cenário de uma clínica médica. O atendente é o ator primário e o cliente o ator secundário. O cliente tem como participação, repassar ao atendente seus dados pessoais para que sejam cadastros no sistema. Já o atendente poderá inserir, editar, desativar e pesquisar registros de clientes da clínica.

A seguir, na Listagem D.1, é apresentado o cabeçalho da especificação na linguagem LUCAM para o caso de uso *SCM_CRUD_Patient*.

Listagem D.1 – Cabeçalho - UC SCM_CRUD_Patient

```
1 Use Case: SCM_CRUD_Patient.
2 Brief Description
3 "This use case allows a clerk enter, edit, disable, and search for a
     clinic patient.".
4 System: System.
5 Primary and Secondary Actors
6 Primary Actors: Clerk.
7 Secondary Actors: Patient.
```

O fluxo principal *AddPatient* do caso de uso *SCM_CRUD_Patient* é apresentado na Listagem D.2. Esse fluxo ocorre quando não há desvio no fluxo básico, pois quando há desvios, serão executados os fluxos alternativos ou de exceções.

Listagem D.2 – Fluxo Principal - UC SCM_CRUD_Patient

```
8  Main Flow: AddPatient.
9  Clerk starts Use Case.
10    Clerk selects "MaintainPatient" on MainForm.
11    Clerk selects "AddPatient" on MainForm.
12    System returns "Input mode screen" to Clerk.
13    Loop ["Required fields empty"]
14       Clerk enters attributes (Int id, String name, String socialSNumber
            , String gender, Date birthDate, String homePhone, String
            cellPhone, String email, String status) of Patient.
15       System validates attributes of Patient.
16    EndLoop
17    System searches for the Patient.
18    If ["Does not exist"]
19       System saves attributes of Patient.
20       System returns "Successfully saved" to Clerk.
21    Else
```

```
22        System displays the attributes of Patient on MainForm.
23     EndIf
24  Clerk finishes Use Case.
```

O fluxo principal descreve o processo da inserção de um novo paciente no sistema, onde o usuário seleciona a opção desejada e o sistema exibe a tela em modo de inserção. Ao receber os dados do paciente como entrada, o sistema validará os campos obrigatórios até que todos sejam preenchidos. Posteriormente será verificado se há registro duplicado no banco de dados e exibe uma notificação. Por fim, o registro será salvo e exibe um *feedback* positivo ao usuário.

A especificação dos fluxos alternativos *ModifyPatient*, *DisablePatient* e *ReadPatient* é apresentada a seguir, por meio da Listagem D.3. O atendente poderá modificar, desabilitar e pesquisar um paciente já cadastrado na clínica.

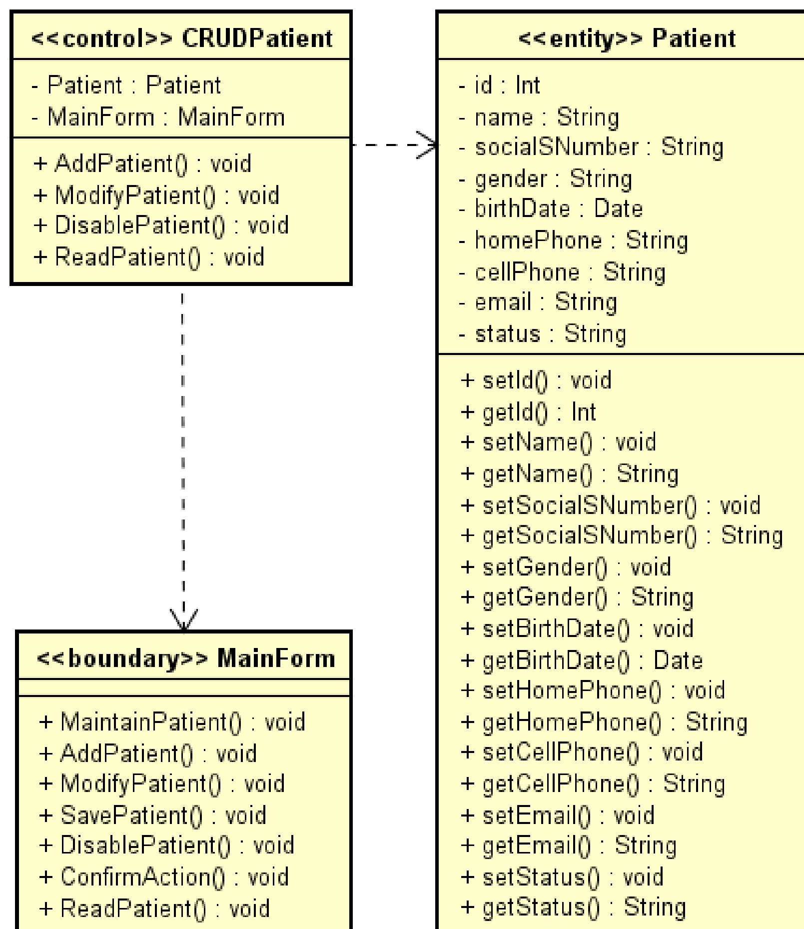
Listagem D.3 – Fluxos Alternativos - UC SCM_CRUD_Patient

```
25  Alternate Flows
26  Alternate Flow 01: ModifyPatient.
27     Clerk selects "ModifyPatient" on MainForm.
28     Clerk types the attributes (id) of Patient on MainForm.
29     System retrieves the attributes of Patient.
30     System displays the attributes of Patient on MainForm.
31     Loop ["Required fields empty"]
32        Clerk modifies the attributes (name, socialSNumber, gender,
             birthDate, homePhone, cellPhone, email, status) of Patient on
             MainForm.
33        System validates attributes of Patient.
34     EndLoop
35     Clerk selects "SavePatient" on MainForm.
36     System updates the informations of Patient.
37     System returns "Successfully modified" to Clerk.
38
39  Alternate Flow 02: DisablePatient.
40     Clerk selects "DisablePatient" on MainForm.
41     Clerk types the attributes (id) of Patient on MainForm.
42     System retrieves the attributes of Patient.
43     System displays the attributes of Patient on MainForm.
44     Clerk selects "ConfirmAction" on MainForm.
45     If ["There is inconsistency"]
46        System returns "There is inconsistency" to Clerk.
47     Else
48        System returns "Successfully disabled" to Clerk.
49     EndIf
50
51  Alternate Flow 03: ReadPatient.
52     Clerk selects "ReadPatient" on MainForm.
```

```
53    Clerk types the attributes (id, name) of Patient on MainForm.
54    System retrieves attributes of Patient.
55    If ["Does not exist patient"]
56       System returns "Does not exist patient" to Clerk.
57    Else
58       System displays the attributes of Patient on MainForm.
59    EndIf
```

Com base na especificação apresentada, foi gerado o diagrama de classes, contendo os relacionamentos entre as classes, os atributos e métodos das mesmas, conforme diagrama da Figura 18. No diagrama foi identificado uma classe do tipo fronteira (*Main-*

Figura 18 – Class Diagram UC CRUDPatient



Fonte: Elaborado pelo autor

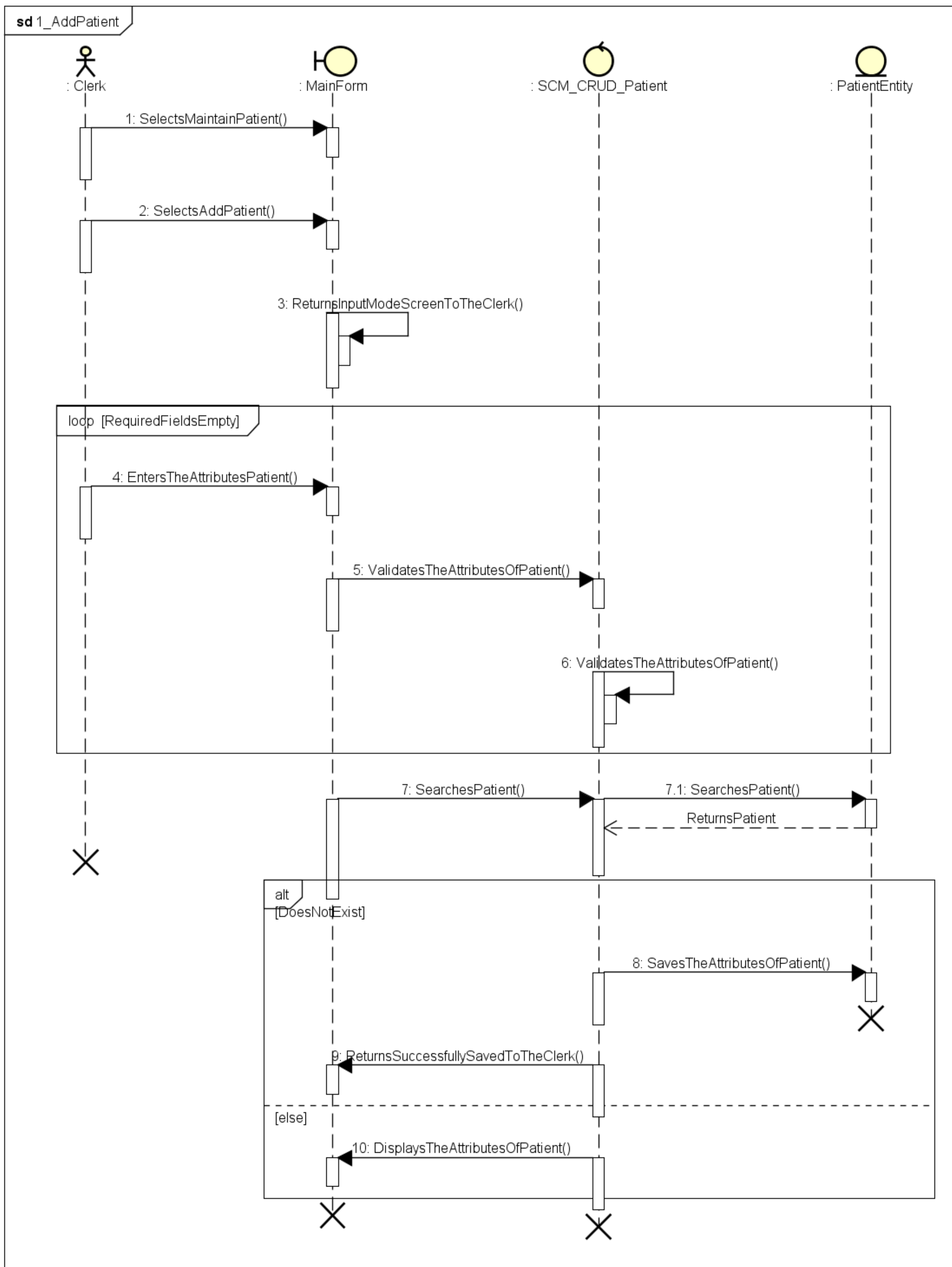
Form), uma classe do tipo controle (*CRUDPatient*) e uma classe do tipo entidade (*Patient*). A nomenclatura de cada classe, como de seus atributos e métodos depende totalmente do que o analista especificar. Observa-se que ao comparar a especificação e os diagramas apresentados, as nomenclaturas definidas tanto na especificação quanto nos

diagramas são iguais.

O primeiro diagrama de sequência apresentado é referente ao fluxo principal *AddPatient*, conforme Figura 19. Observa-se que neste e nos demais diagramas, aplicaram-se os principais recursos presentes no diagrama de sequência, como auto chamada, mensagens de retorno e fragmentos combinados.

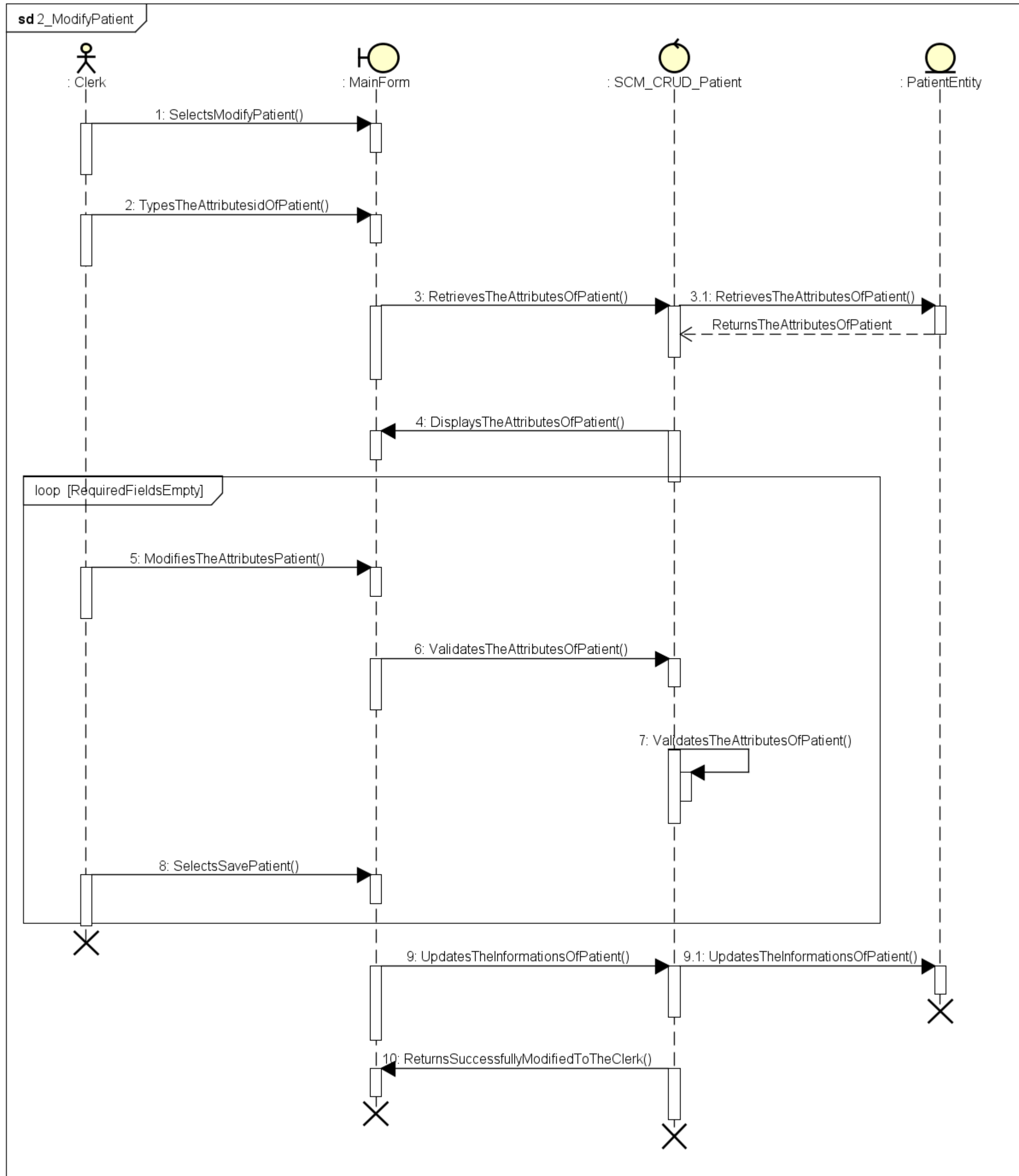
A seguir são apresentados os diagramas de sequência dos demais fluxos (*ModifyPatient, DisabledPatient* e *ReadPatient*), conforme as Figuras 20, 21 e 22.

Figura 19 – Sequence Diagram AddPatient



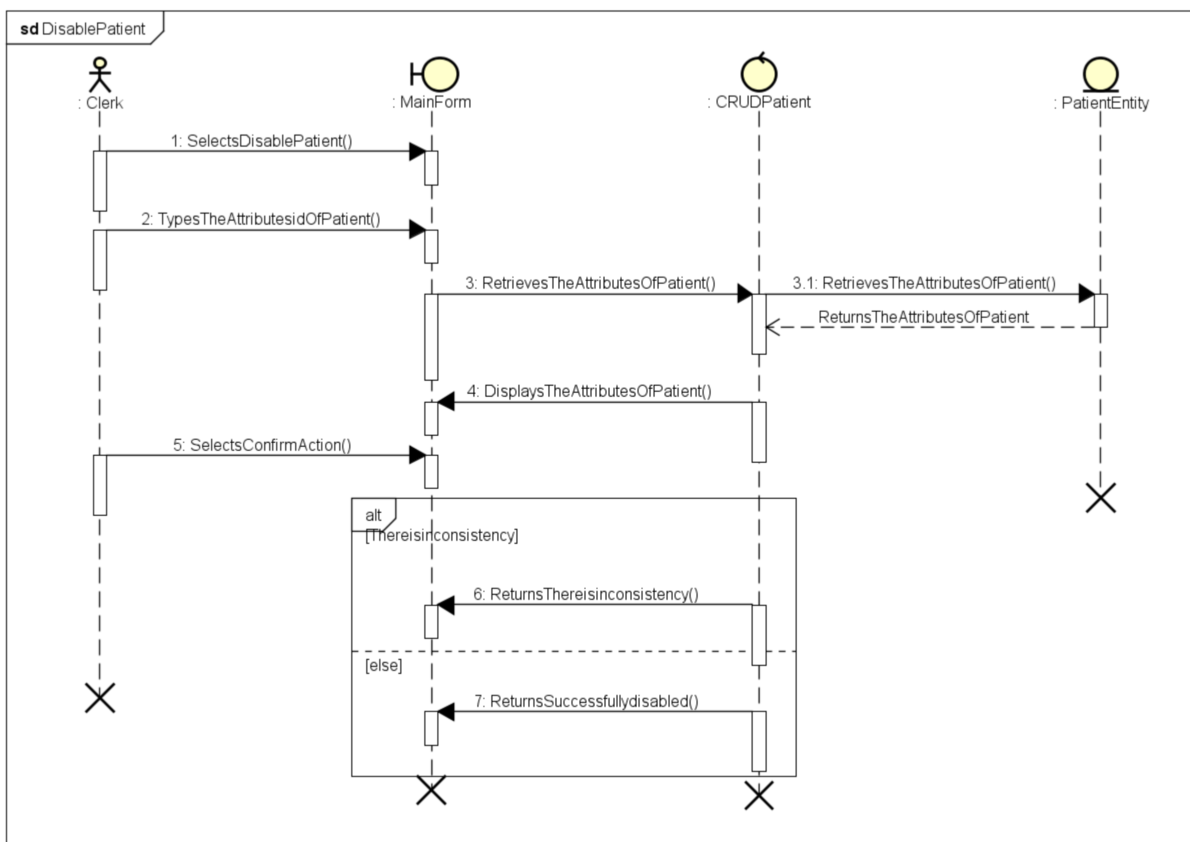
Fonte: Elaborado pelo autor

Figura 20 – Sequence Diagram ModifyPatient



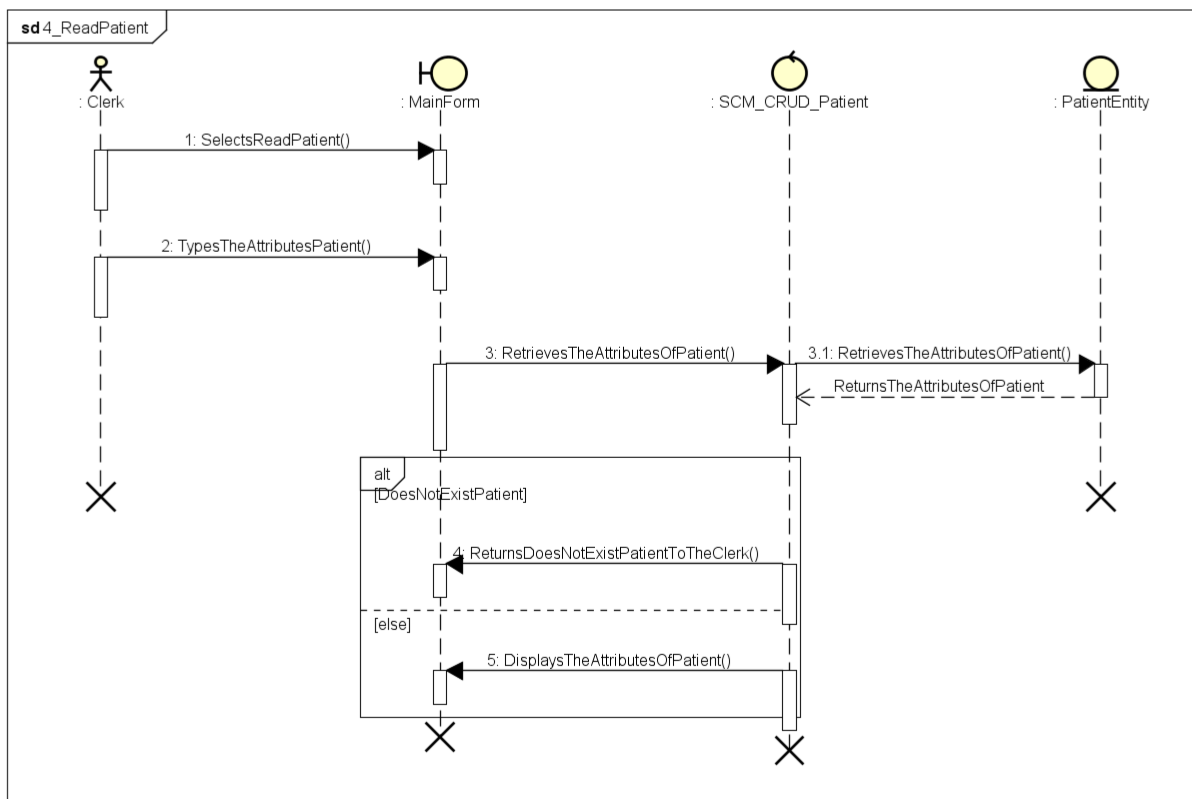
Fonte: Elaborado pelo autor

Figura 21 – Sequence Diagram DisablePatient



Fonte: Elaborado pelo autor

Figura 22 – Sequence Diagram ReadPatient



Fonte: Elaborado pelo autor

APÊNDICE E – CASO DE USO “MATRICULAR ESTUDANTE”

O caso de uso é referente a um sistema acadêmico, para o qual será apresentada a especificação do processo de matrícula, conforme Listagem E.1.

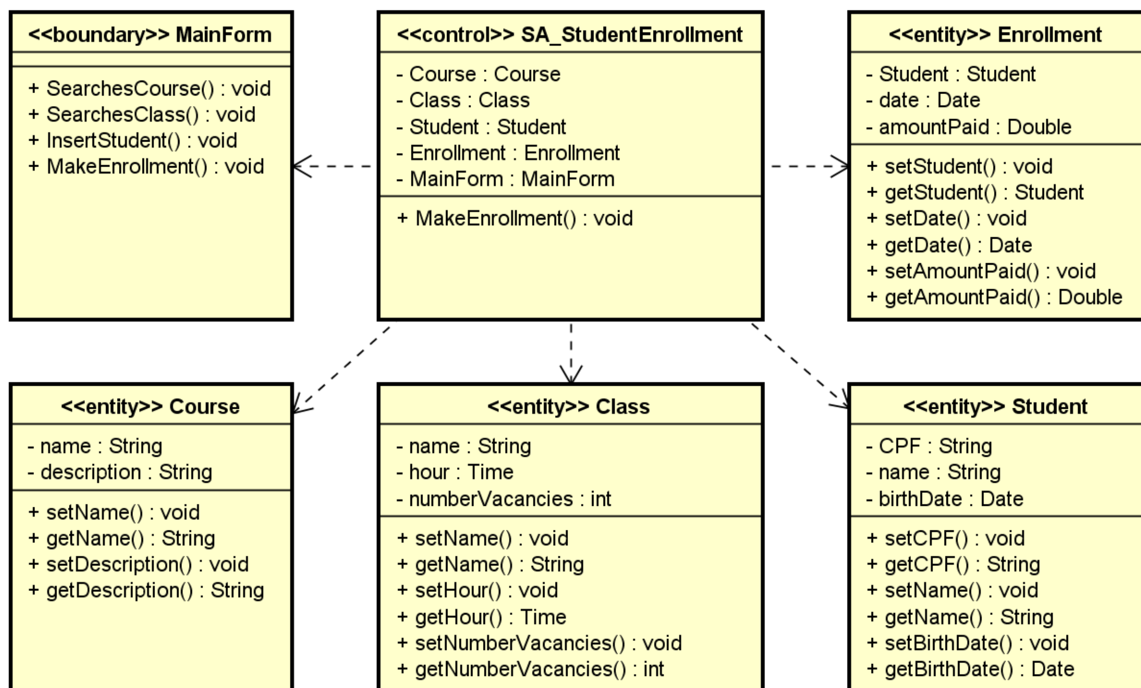
Listagem E.1 – Especificação do caso de uso SA_StudentEnrollment

```
1  Use Case: SA_StudentEnrollment.
2  Brief Description
3  "Allows the inclusion, editing, deleting and searching a enrolment".
4  System: System.
5  Primary and Secondary Actors
6  Primary Actors: Secretary.
7  Secondary Actors: Student.
8  Main Flow: MakeEnrollment.
9  Secretary starts Use Case.
10 Secretary selects "SearchesCourse" on MainForm.
11 System searches the informations of Course.
12 System displays attributes (String name, String  description) of Course
      on MainForm.
13 Secretary appears the informations of Course for the Student.
14 If ["Student Interest"]
15    Secretary selects "SearchesClass" on MainForm.
16 EndIf
17 System searches the informations of Class.
18 System displays the informations (String name, Time hour, int
      numberVacancies) of Class on MainForm.
19 Secretary appears the informations of Class for the Student.
20 If ["Student Interest"]
21    Secretary enters the attributes (CPF) of Student on MainForm.
22    System retrieves the informations of Student.
23    If ["Student Non Registered"]
24       System displays "Student non registered" on MainForm.
25       Secretary selects "Insert Student" on MainForm.
26       System saves the informations (String name, Date  birthDate,
            String CPF) of Student.
27       System returns "Student registered successfully" to Secretary.
28    Else
29       System displays the informations of Student on MainForm.
30       Secretary selects "MakeEnrollment" on MainForm.
31       System saves the informations (Student Student, Date  date, Double
            amountPaid) of Enrollment.
32       System returns "Enrollment registered successfully" to Secretary.
```

```
33      EndIf
34    EndIf
35    Secretary finishes Use Case.
36    Key Scenarios
37    Key Scenario 01: MakeEnrollment.
38    Preconditions
39      "Before this use case begins the Secretary has logged onto the system
          ".
40    Postconditions
41      "There are no post conditions associated with this use case".
42    Special Requirements
43      "There are no special requirements associated with this use case".
44    Extension Points
45      "There are no extension points associated with this use case".
```

O diagrama de classes gerado por meio da LUCAMTool para a especificação apresentada na Listagem E.1 é apresentado abaixo, conforme Figura 23.

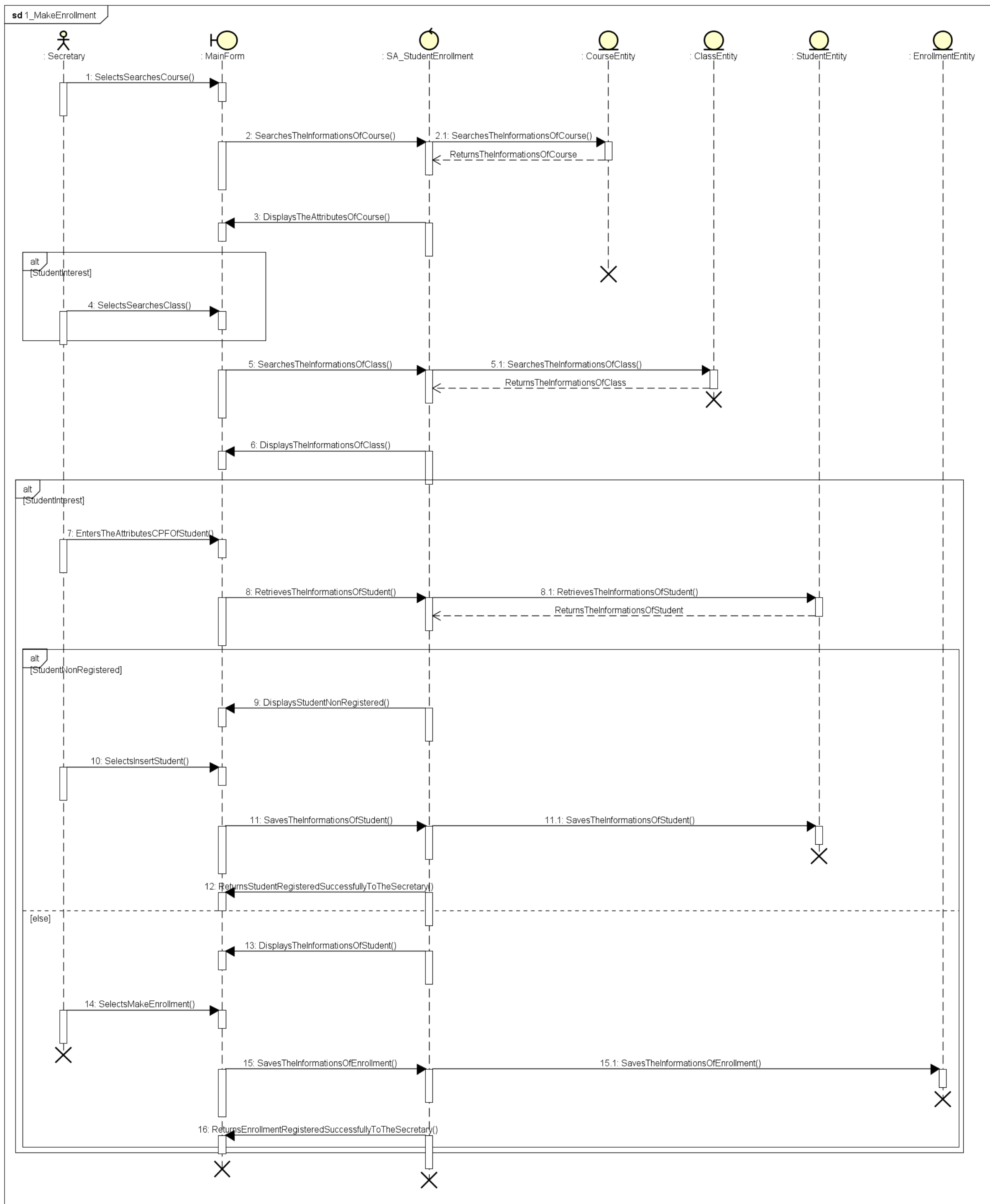
Figura 23 – Diagrama de classes - SA_StudentEnrollment



Fonte: Elaborado pelo autor

A seguir é apresentado o diagrama de sequência do caso de uso especificado na Listagem E.1, conforme Figura 24.

Figura 24 – Diagrama de classes - SA_StudentEnrollment



Fonte: Elaborado pelo autor

APÊNDICE F – CASO DE USO “MANTER OBRAS CIVIS”

Este requisito oferece ao usuário as opções de inserir, editar, desabilitar e pesquisar registros referentes aos dados uma obra civil. No caso do sistema de gestão de concreto, o software deve gerenciar também as informações das obras para as quais determinada porção (traço) de concreto será destinada.

Por meio da Listagem F.1, é apresentada a especificação do caso de uso, que possui 4 fluxos, sendo o primeiro o fluxo principal *AddCivilWorks* e os demais os fluxos alternativos *ModifyCivilWorks, DisabledCivilWorks* e *SearchCivilWorks.*

Listagem F.1 – Especificação do caso de uso PIV_SGC_MaintainCivilWorks

```
1  Use Case: PIV_SGC_MaintainCivilWorks.
2  Brief Description
3  "Allows the inclusion, editing and research a CivilWorks".
4  System: System.
5  Primary and Secondary Actors
6  Primary Actors: Administrator.
7  Main Flow: AddCivilWorks.
8  Administrator starts Use Case.
9     Administrator selects "MaintainCivilWorks" on MainForm.
10    Administrator selects "AddCivilWorks" on MainForm.
11    System returns "Input mode screen" to Administrator.
12
13    Loop ["Required fields empty"]
14       Administrator enters attributes (int Id, string Description,
            Customer Customer, Seller Seller, Address Address, string
            Status) of CivilWorks.
15       System validates attributes of CivilWorks.
16    EndLoop
17
18    System searches for the CivilWorks.
19    Administrator selects "AddCivilWorksItems" on MainForm.
20    Loop ["AddItems = true"]
21       Administrator selects "SearchCivilWorksItems" on MainForm.
22       Administrator types the attributes (IdItem, NameItem) of
            CivilWorksItems on MainForm.
23       System retrieves the attributes of CivilWorksItems.
24       Administrator selects the informations of CivilWorksItems.
25       If ["There is inconsistency"]
26          System returns the message to Administrator.
27       Else
```

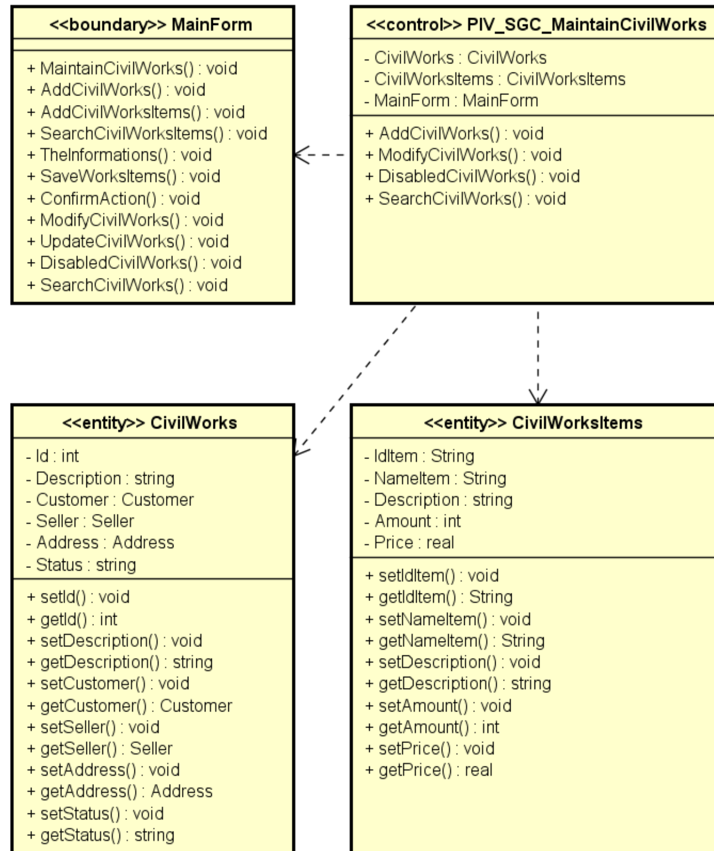
```
28          System displays the attributes of CivilWorksItems on MainForm.
29          Administrator selects the informations of CivilWorksItems.
30          Administrator enters attributes (int IdItem, string Description
               , int Amount, real Price) of CivilWorksItems.
31          Administrator selects "SaveWorksItems" on MainForm.
32          System includes the informations of CivilWorksItems.
33       EndIf
34    EndLoop
35
36    If ["Command == Save"]
37       If ["Is not existent"]
38          System saves attributes of CivilWorks.
39          System returns "Successfully saved" to Administrator.
40       Else
41          System returns the message to Administrator.
42          System displays the attributes of CivilWorks on MainForm.
43       EndIf
44    Else
45       If ["Command = Cancel"]
46          Administrator selects "ConfirmAction" on MainForm.
47          System returns the message to Administrator.
48       EndIf
49    EndIf
50
51 Administrator finishes Use Case.
52
53 Alternate Flows
54 Alternate Flow 01: ModifyCivilWorks.
55    Administrator selects "ModifyCivilWorks" on MainForm.
56    Administrator types the attributes (Id, Description, Customer, Status
         ) of CivilWorks on MainForm.
57    System retrieves the attributes of CivilWorks.
58    System displays the attributes of CivilWorks on MainForm.
59    Administrator modifies the attributes of CivilWorks on MainForm.
60    Administrator selects "UpdateCivilWorks" on MainForm.
61    If ["There is no inconsistency"]
62       System updates the informations of CivilWorks.
63       System returns "Successfully modified" to Administrator.
64    Else
65       System returns the message to Administrator.
66       System displays the attributes of CivilWorks on MainForm.
67    EndIf
68
69 Alternate Flow 02: DisabledCivilWorks.
70    Administrator selects "DisabledCivilWorks" on MainForm.
71    Administrator types the attributes (Id, Description, Customer, Status
         ) of CivilWorks on MainForm.
```

```
72    System retrieves the attributes of CivilWorks.
73    If ["Existing CivilWorks"]
74       System displays the attributes of CivilWorks on MainForm.
75       Administrator selects "ConfirmAction" on MainForm.
76       If ["There is no inconsistency"]
77          System deletes the informations of CivilWorks.
78          System returns "successfully disabled" to Administrator.
79       Else
80          System returns the message to Administrator.
81       EndIf
82    Else
83       System returns the message to Administrator.
84    EndIf
85 Alternate Flow 03: SearchCivilWorks.
86    Administrator selects "SearchCivilWorks" on MainForm.
87    Administrator types the attributes (Id, Description, Customer, Status
         ) of CivilWorks on MainForm.
88    System retrieves attributes of CivilWorks.
89    If ["Existing CivilWorks"]
90       System displays the attributes of CivilWorks on MainForm.
91    Else
92       System returns the message to Administrator.
93    EndIf
94
95 Key Scenarios
96    Key Scenario 01: AddCivilWorks.
97    Key Scenario 02: ModifyCivilWorks.
98    Key Scenario 03: DisabledCivilWorks.
99    Key Scenario 04: SearchCivilWorks.
100 Preconditions
101    "Before this use case begins the Administrator has logged onto the
         system.
102    The system is in registration mode.".
103 Postconditions
104    "There are no post conditions associated with this use case.".
105 Special Requirements
106    "There are no special requirements associated with this use case.".
107 Extension Points
108    "There are no extension points associated with this use case.".
```

O diagrama de classes gerado por meio da LUCAMTool para a especificação apresentada na Listagem F.1 é ilustrado abaixo, conforme Figura 23.

Os diagramas de sequência conforme especificação da Listagem F.1. O primeiro diagrama, representado pela Figura 26 é referente ao fluxo principal *AddCivilWorks*.

Figura 25 – Diagrama de classes do caso de uso PIV_SGC_MaintainCivilWorks



Fonte: Elaborado pelo autor

O diagrama de sequência referente ao fluxo alternativo *ModifyCivilWorks*, é ilustrado pela Figura 27.

O diagrama de sequência referente ao fluxo alternativo *DisabledCivilWorks*, ilustrado pela Figura 28.

Por fim, o diagrama de sequência referente ao fluxo alternativo *SearchCivilWorks*, ilustrado pela Figura 29.

Figura 26 – Diagrama de sequência do fluxo *AddCivilWorks*

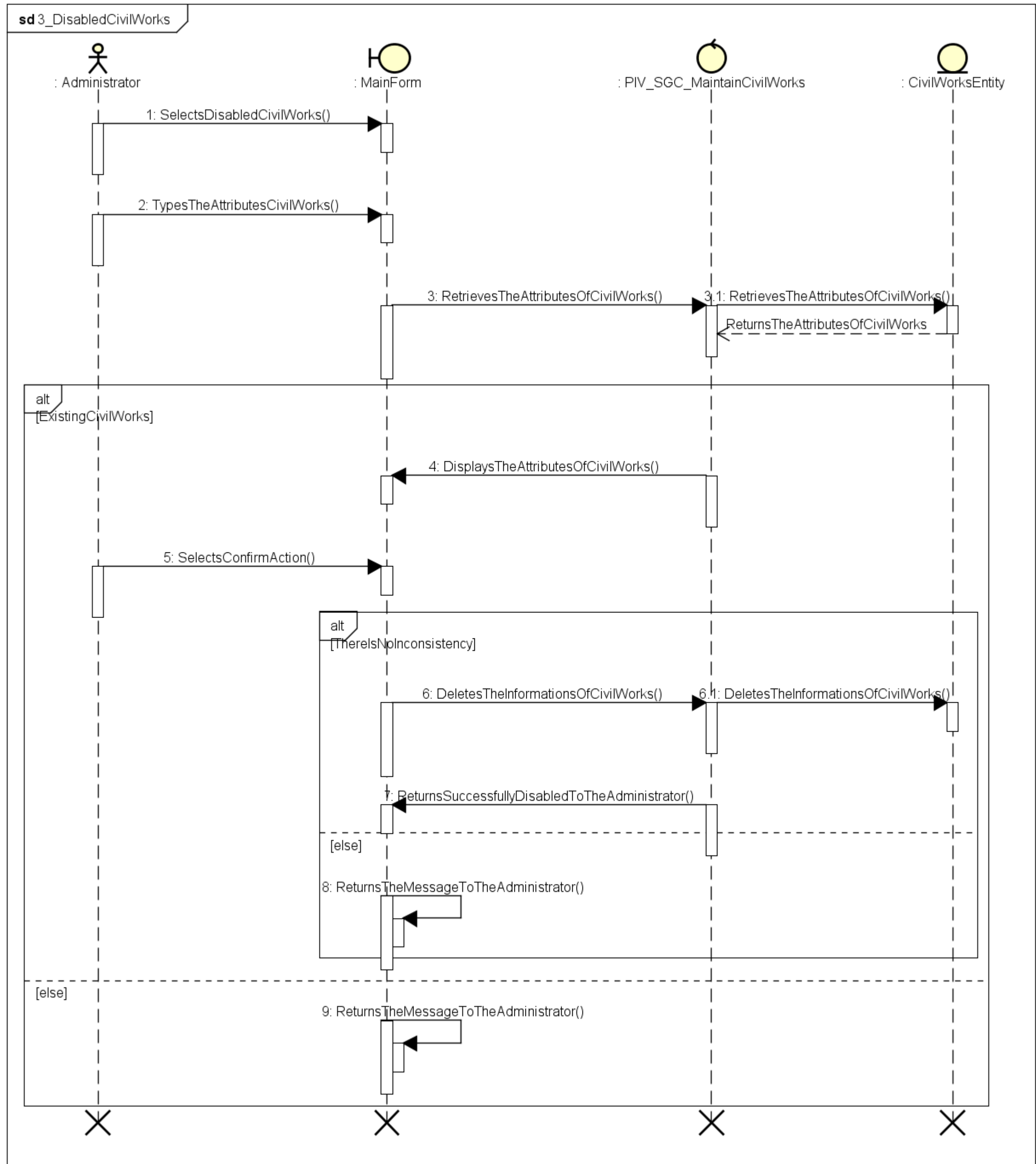
sd 1_AddCivilWorks
: Administrator
: MainForm
: PIV_SGC_MaintainCivilWorks
: CivilWorksEntity
: CivilWorksItemsEntity
1: SelectsMaintainCivilWorks()
2: SelectsAddCivilWorks()
3: ReturnsInputModeScreenToTheAdministrator()
loop [RequiredFieldsEmpty]
4: EntersTheAttributesCivilWorks()
5: ValidatesTheAttributesOfCivilWorks()
6: ValidatesTheAttributesOfCivilWorks()
7: SearchesCivilWorks()
7.1: SearchesCivilWorks()
ReturnsCivilWorks
8: SelectsAddCivilWorksItems()
loop [AddItemsTrue]
9: SelectsSearchCivilWorksItems()
10: TypesTheAttributesCivilWorksItems()
11: RetrievesTheAttributesOfCivilWorksItems()
11.1: RetrievesTheAttributesOfCivilWorksItems()
ReturnsTheAttributesOfCivilWorksItems
12: SelectsTheInformationsCivilWorksItems()
alt
[ThereIsInconsistency]
13: ReturnsTheMessageToTheAdministrator()
[else]
14: DisplaysTheAttributesOfCivilWorksItems()
15: SelectsTheInformationsCivilWorksItems()
16: EntersTheAttributesCivilWorksItems()
17: SelectsSaveWorksItems()
18: IncludesTheInformationsOfCivilWorksItems()
18.1: IncludesTheInformationsOfCivilWorksItems()
alt
[CommandSave]
alt
[IsNotExistent]
19: SavesTheAttributesOfCivilWorks()
19.1: SavesTheAttributesOfCivilWorks()
20: ReturnsSuccessfullySavedToTheAdministrator()
[else]
21: ReturnsTheMessageToTheAdministrator()
22: DisplaysTheAttributesOfCivilWorks()
[else]
alt
[CommandCancel]
23: SelectsConfirmAction()
24: ReturnsTheMessageToTheAdministrator()

Fonte: Elaborado pelo autor

Figura 27 – Diagrama de sequência do fluxo *ModifyCivilWorks*

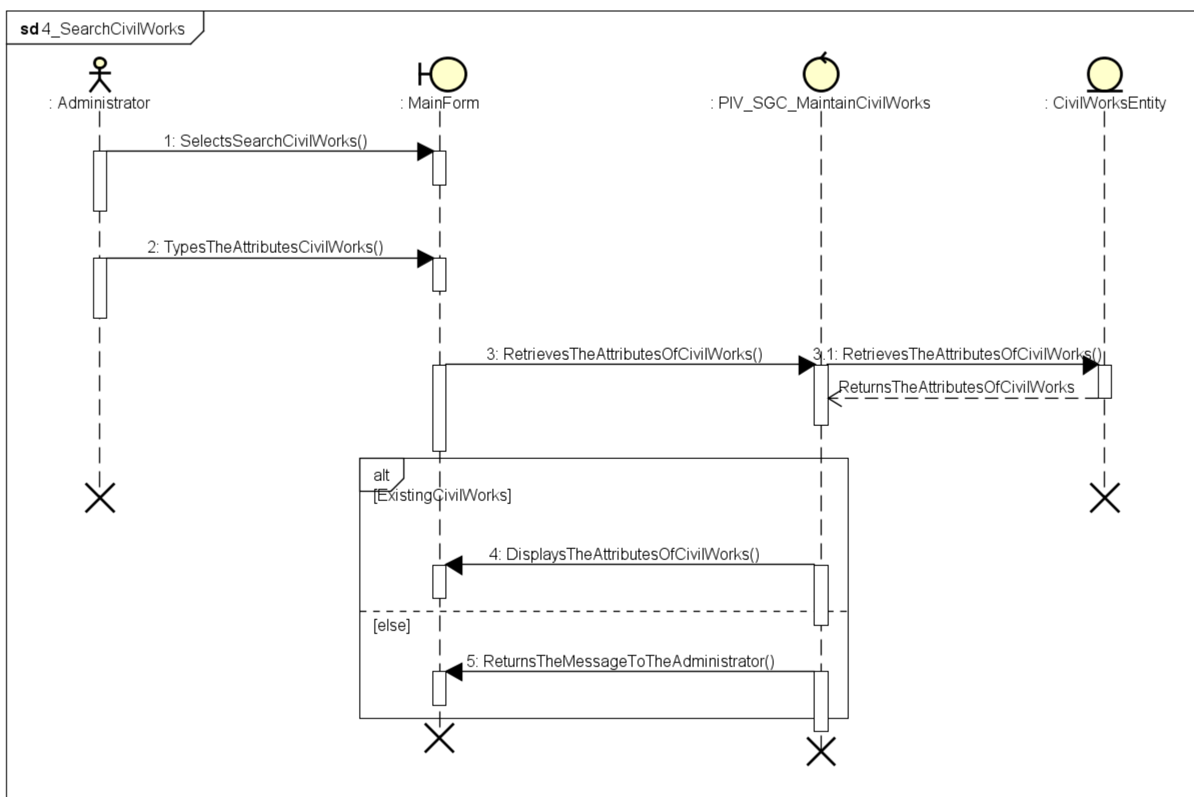
Fonte: Elaborado pelo autor

Figura 28 – Diagrama de sequência do fluxo *DisabledCivilWorks*



Fonte: Elaborado pelo autor

Figura 29 – Diagrama de sequência do fluxo *SearchCivilWorks*



Fonte: Elaborado pelo autor

APÊNDICE G – CASO DE USO “CARREGAMENTO DE CONCRETO”

A seguir é apresentada a especificação do caso de uso “Carregamento de Concreto”. O ator primário é o Operador e por meio desta funcionalidade, ele pode registrar via sistema a pesagem e o carregamento de um traço de concreto para uma determinada obra. Para cada traço de concreto, já são pré-cadastrados no sistema os itens que irão constituí-lo.

De acordo com informações repassadas pela equipe, as configurações dos traços de concreto é cadastrada de acordo com a umidade relativa do ar, temperatura, localização, tipo de solo, quantidade de água, areia, pedra, agregados, cimentos e aditivos. Estas especificações são antes desenvolvidas pelos técnicos de laboratório, que em seguida alimentam o sistema, cadastrando cada porção de concreto, ou seja, para cada porção existe uma fórmula pré-cadastrada.

A especificação completa do caso de uso está na Listagem G.1.

Listagem G.1 – Especificação do caso de uso PIV_SGC_WeighingLoadingConcrete

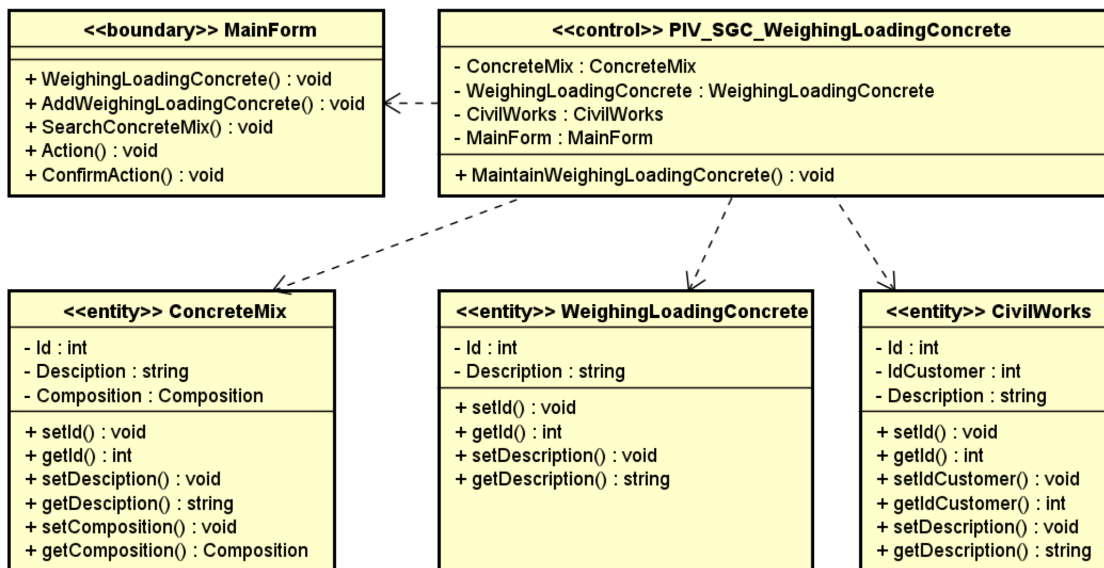
```
1  Use Case: PIV_SGC_WeighingLoadingConcrete.
2  Brief Description
3  "Allows insert and cancel the process of weighing and concrete loading."
     .
4  System: System.
5  Primary and Secondary Actors
6  Primary Actors: Operator.
7  Main Flow: MaintainWeighingLoadingConcrete.
8  Operator starts Use Case.
9     Operator selects "WeighingLoadingConcrete" on MainForm.
10    Operator selects "AddWeighingLoadingConcrete" on MainForm.
11    System returns "Input mode screen" to the Operator.
12    Operator selects "SearchConcreteMix" on MainForm.
13    Operator enters attributes (int Id, string Desciption, Composition
         Composition) of ConcreteMix.
14    System retrieves attributes of ConcreteMix.
15    If ["Existing ConcreteMix"]
16       System displays the attributes of ConcreteMix on MainForm.
17    Else
18       System returns the message to the Operator.
19    EndIf
20    If ["there concrete weighing and loading to CivilWorks"]
```

```
21        Operator enters attributes(int Id, string Description) of
             WeighingLoadingConcrete on MainForm.
22        Operator selects "Action" on MainForm.
23        If ["Action==Start"]
24           Operator performs the attributes(int Id, string Description) of
                 WeighingLoadingConcrete.
25           Operator enters the attributes(int Id, int IdCustomer, string
                Description) of CivilWorks.
26        Else
27           If ["Action==Finalize"]
28              System saves the informations of WeighingLoadingConcrete.
29              System returns "Successfully saved" to the Operator.
30           Else
31              If ["Action==Delete"]
32                    Operator selects "ConfirmAction" on MainForm.
33                    System verifies the dependencies of CivilWorks.
34                    If ["Existing dependencies"]
35                       System deletes the informations of
                            WeighingLoadingConcrete.
36                       System returns "Successfully deleted" to the
                            Operator.
37                    Else
38                       System returns the message to the Operator.
39                    EndIf
40              Else
41                 If ["Action==Cancel"]
42                    Operator selects "ConfirmAction" on MainForm.
43                    System verifies the dependencies of CivilWorks.
44                    If ["Existing dependencies"]
45                       System returns "It is not possible to cancel" to
                            the Operator.
46                    Else
47                       System returns "Successfully canceled" to the
                            Operator.
48                    EndIf
49                 EndIf
50              EndIf
51           EndIf
52        EndIf
53     Else
54        System returns the message to Operator.
55     EndIf
56  Operator finishes Use Case.
57
58  Key Scenarios
59     Key Scenario 01: MaintainWeighingLoadingConcrete.
60  Preconditions
```

```
61      "Before this use case begins the Operator has logged onto the system.
62      The concrete mix to be pre-registered. The system is in weighing
          register mode.
63      Civil work must be pre-registered. The customer must be previously
          registered.".
64  Postconditions
65      "There are no post conditions associated with this use case.".
66  Special Requirements
67      "There are no special requirements associated with this use case.".
68  Extension Points
69      "There are no extension points associated with this use case.".
```

Serão apresentados em seguida os diagramas de classes e sequências gerados automaticamente pela ferramenta LUCAMTool, com base na especificação de entrada. O primeiro diagrama mostrado é o diagrama de classe, conforme Figura 30.

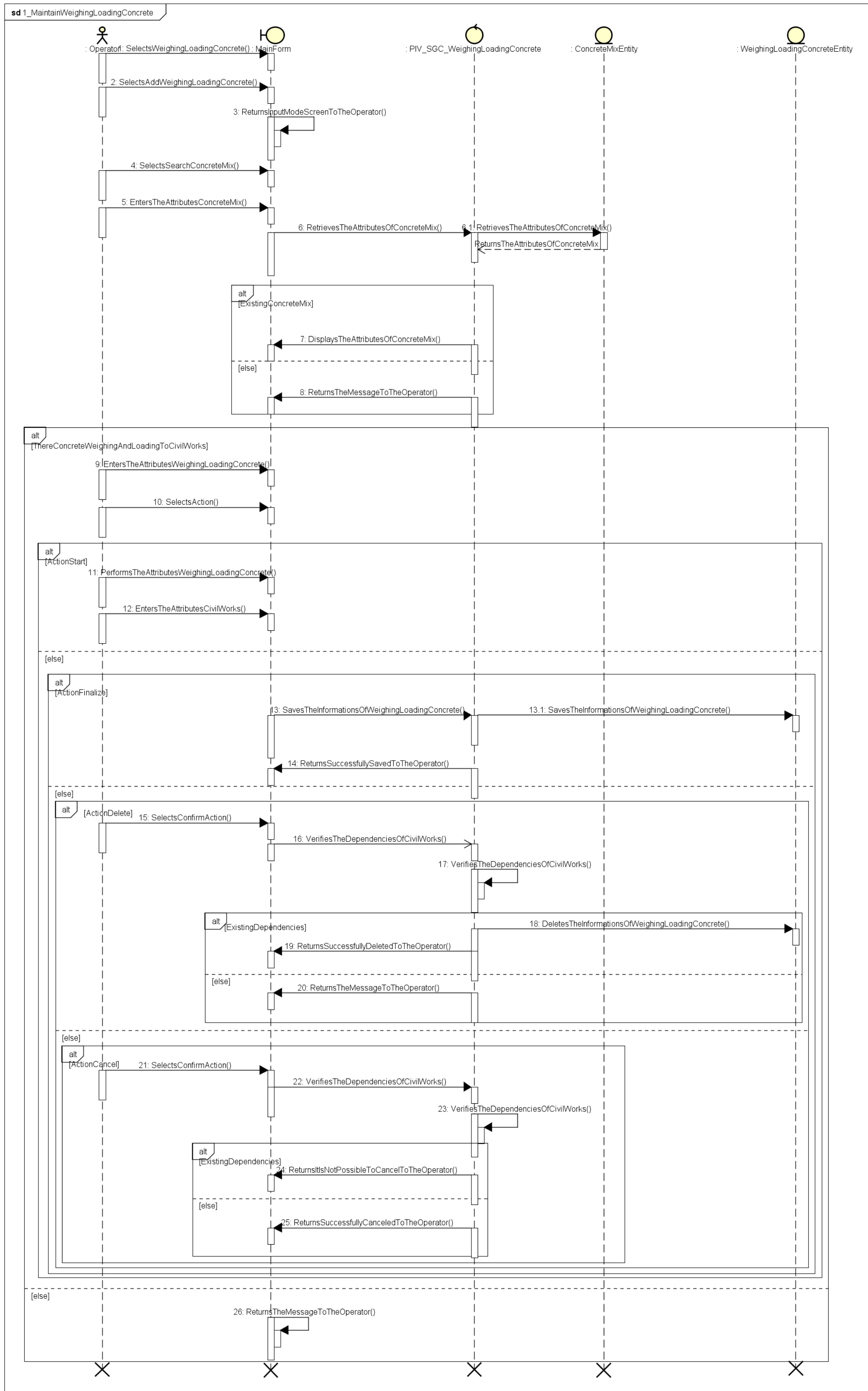
Figura 30 – Diagrama de classe - UC *PIV_SGC_WeighingLoadConcrete*



Fonte: Elaborado pelo autor

Agora será apresentado o diagrama de sequência conforme especificação da Listagem G.1. O diagrama representado pela Figura 31 é referente ao fluxo *MaintainWeighingLoadingConcrete*. Observa-se com este diagrama, que mesmo quando a especificação possui um número elevado de laços aninhado, o mapeamento e geração dos modelos é realizado corretamente, refletindo fielmente o que está descrito na especificação.

Figura 31 – Diagrama de sequência do fluxo *MaintainWeighingLoadingConcrete*



Fonte: Elaborado pelo autor

APÊNDICE H – CASO DE USO “REQUISIÇÃO DE FORMULÁRIO”

Neste caso de uso, o usuário poderá solicitar uma requisição de formulário no sistema. A seguir é apresentada a especificação na linguagem LUCAM.

Listagem H.1 – Especificação do caso de uso CNB_SCOF_FormRequest

```
1  Use Case: CNB_SCOF_FormRequest.
2  Brief Description
3  "Allows the inclusion, attach, approve, finalize and cancel a request
      form.".
4  System: System.
5  Primary and Secondary Actors
6  Primary Actors: Manager.
7  Main Flow: AddFormRequest.
8  Manager starts Use Case.
9     Manager selects "MaintainFormRequest" on MainForm.
10    Manager selects "AddFormRequest" on MainForm.
11    System returns "Input mode screen" to Manager.
12
13    Loop ["Required fields empty"]
14       Manager enters attributes (int Id, string Title, string
            Apresentation, string Note, string Status) of FormRequest.
15       System validates attributes of FormRequest.
16    EndLoop
17
18    System searches for the FormRequest.
19
20    If ["Is not existent"]
21       System saves attributes of FormRequest.
22       System returns "successfully saved" to Manager.
23    Else
24       System displays the attributes of FormRequest on MainForm.
25    EndIf
26
27 Manager finishes Use Case.
28
29 Alternate Flows
30 Alternate Flow 01: AttachStandardizedFile.
31    Manager selects "AttachStandardizedFile" on MainForm.
32    Manager types the attributes (Id, Title, Status) of FormRequest on
         MainForm.
33    System retrieves the attributes of FormRequest.
```

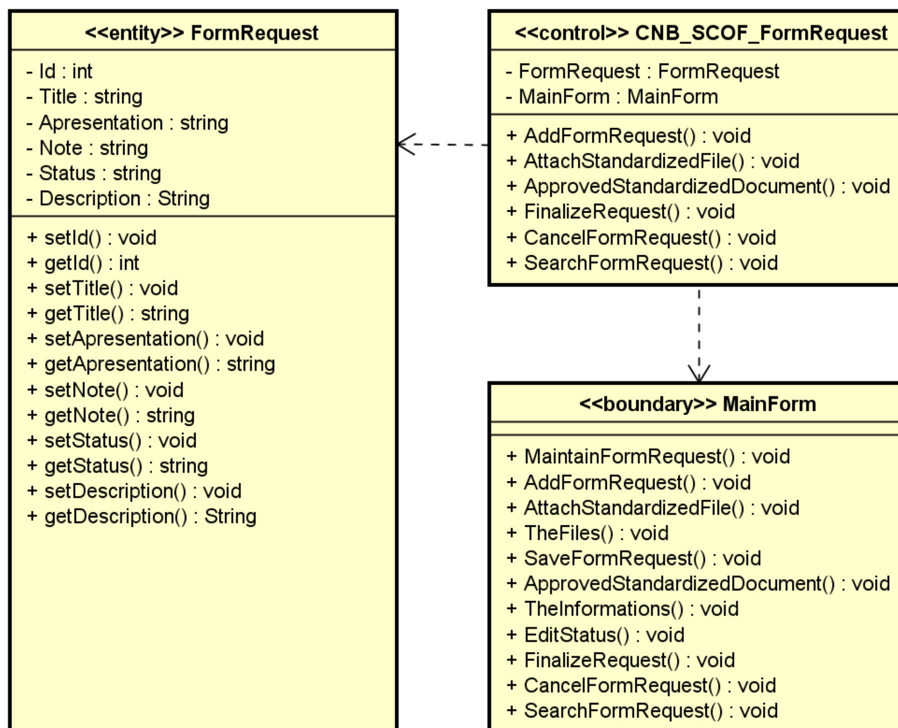
```
34      If ["Request.Status = Opened"]
35         System displays the attributes of FormRequest on MainForm.
36         Manager selects the files of FormRequest.
37         Manager selects "SaveFormRequest" on MainForm.
38
39         If ["There is no inconsistency"]
40            System updates the informations of FormRequest.
41            System returns "successfully modified" to Manager.
42         Else
43            System displays the attributes of FormRequest on MainForm.
44         EndIf
45      Else
46         System returns the message to Manager.
47      EndIf
48
49   Alternate Flow 02: ApprovedStandardizedDocument.
50      Manager selects "ApprovedStandardizedDocument" on MainForm.
51      Manager types the attributes (Id, Title, Status) of FormRequest on
           MainForm.
52      System retrieves the attributes of FormRequest.
53      If ["Request.Status = Opened"]
54         System displays the attributes of FormRequest on MainForm.
55         Manager selects informations of FormRequest.
56         Manager selects "EditStatus" on MainForm.
57         Manager enters the justification of FormRequest on MainForm.
58         If ["There is no inconsistency"]
59            System updates the informations of FormRequest.
60            System returns "Successfully approved" to Manager.
61            System sends the informations (Title, Status) of FormRequest by
                 "e-mail".
62         Else
63            System displays the attributes of FormRequest on MainForm.
64         EndIf
65      Else
66         System returns the message to Manager.
67      EndIf
68
69   Alternate Flow 03: FinalizeRequest.
70      Manager selects "FinalizeRequest" on MainForm.
71      Manager types the attributes (Id, Title, Status) of FormRequest on
           MainForm.
72      System retrieves the attributes of FormRequest.
73      If ["Request.Status = Approved"]
74         System displays the attributes of FormRequest on MainForm.
75         Manager selects informations of FormRequest.
76         Manager selects "EditStatus" on MainForm.
77         If ["There is no inconsistency"]
```

```
78          System updates the informations of FormRequest.
79          System returns "Successfully finalized" to Manager.
80          System sends the informations (Title, Status) of FormRequest by
               "e-mail".
81       Else
82          System displays the attributes of FormRequest on MainForm.
83       EndIf
84    Else
85       System returns the message to Manager.
86    EndIf
87
88 Alternate Flow 04: CancelFormRequest.
89    Manager selects "CancelFormRequest" on MainForm.
90    Manager types the attributes (Id, Title, Status) of FormRequest on
         MainForm.
91    System retrieves the attributes of FormRequest.
92    If ["Request.Status != canceled"]
93       System displays the attributes of FormRequest on MainForm.
94       Manager selects informations of FormRequest.
95       Manager selects "EditStatus" on MainForm.
96       If ["There is no inconsistency"]
97          System updates the informations of FormRequest.
98          System returns "Successfully canceled" to Manager.
99          System sends the informations (Title, Status) of FormRequest by
               "e-mail".
100      Else
101         System displays the attributes of FormRequest on MainForm.
102      EndIf
103   Else
104      System returns the message to Manager.
105   EndIf
106
107 Alternate Flow 05: SearchFormRequest.
108   Manager selects "SearchFormRequest" on MainForm.
109   Manager types the attributes (Id, Description, Status) of FormRequest
         on MainForm.
110   System retrieves the attributes of FormRequest.
111   If ["Existing FormRequest"]
112      System displays the attributes of FormRequest on MainForm.
113   Else
114      System returns the message to Manager.
115   EndIf
116
117 Key Scenarios
118   Key Scenario 01: AddFormRequest.
119   Key Scenario 02: AttachStandardizedFile.
120   Key Scenario 03: ApprovedStandardizedFile.
```

```
121     Key Scenario 04: FinalizeRequest.
122     Key Scenario 05: CancelFormRequest.
123   Preconditions
124     "Before this use case begins the Manager has logged onto the system".
125   Postconditions
126     "There are no post conditions associated with this use case.".
127   Special Requirements
128     "There are no special requirements associated with this use case.".
129   Extension Points
130     "There are no extension points associated with this use case.".
```

O diagrama de classes referente a especificação da Listagem H.1 está ilustrado pela Figura 32.

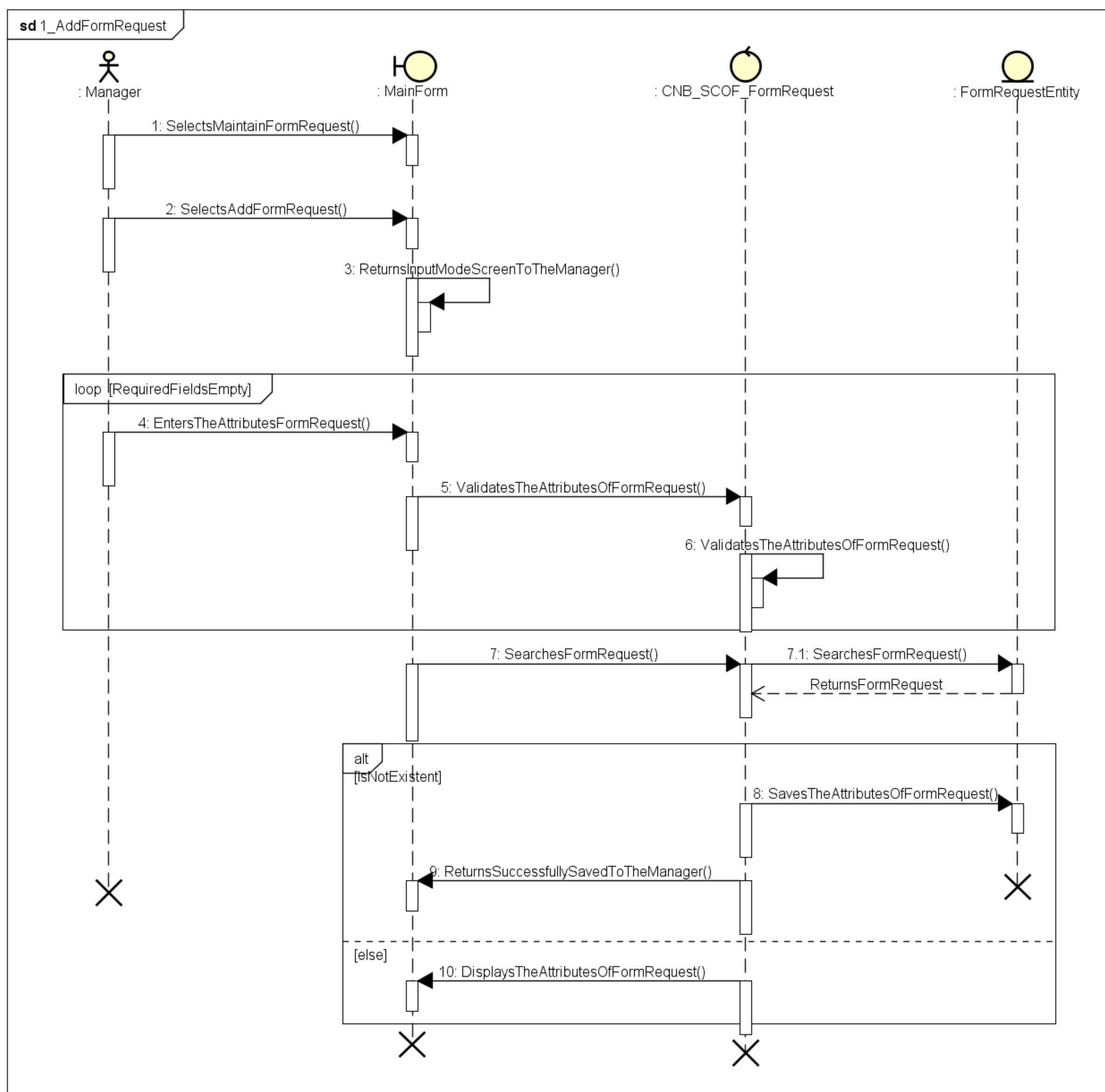
Figura 32 – Diagrama de classes - UC *CNB_SCOF_FormRequest*



Fonte: Elaborado pelo autor

O diagrama de sequência referente ao fluxo principal (*AddFormRequest*) presente na especificação da Listagem H.1 está ilustrado pela Figura 33.

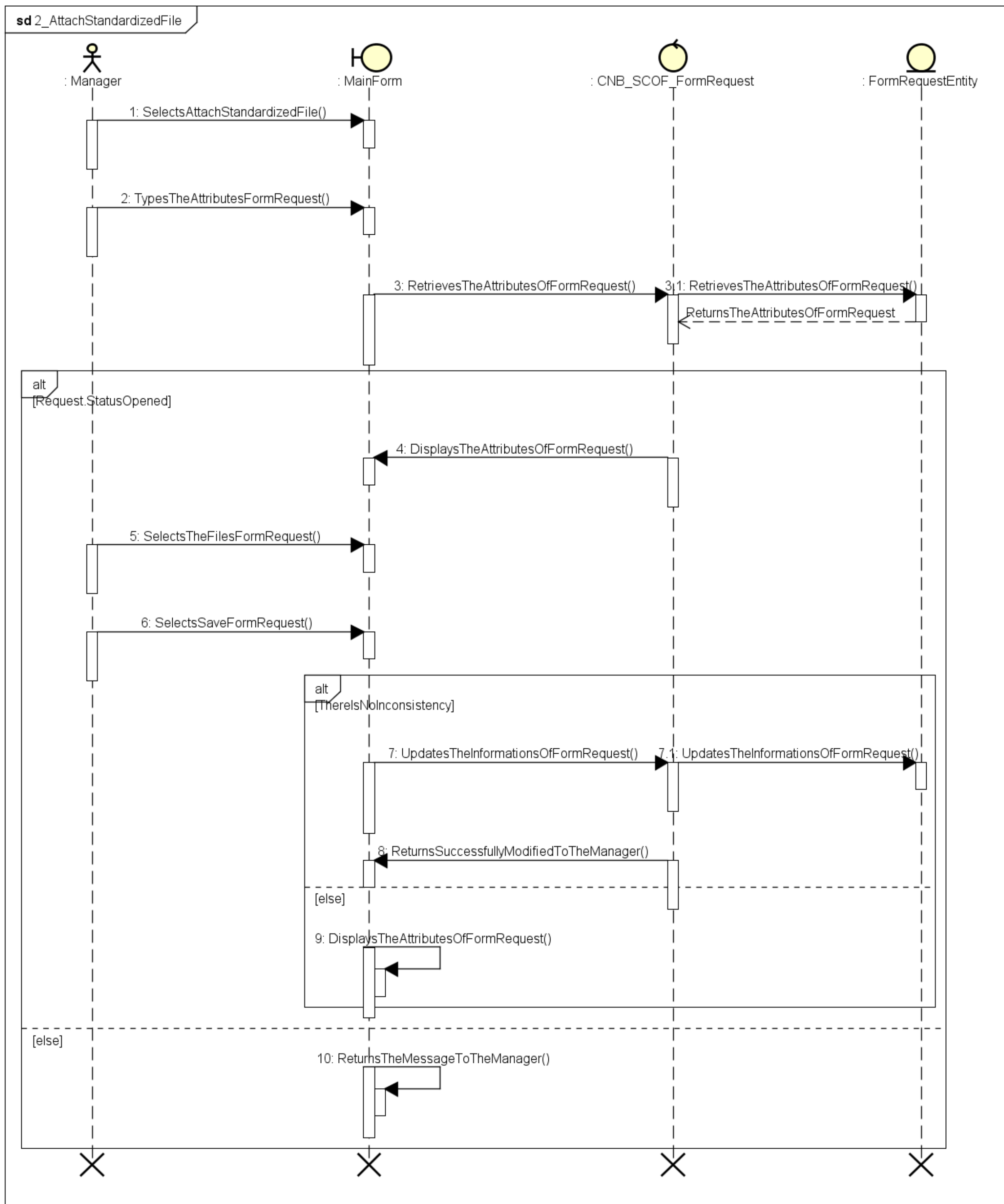
Figura 33 – Diagrama de sequência do fluxo *AddFormRequest*



Fonte: Elaborado pelo autor

Por meio da Figura 34, é apresentado o diagrama de sequência referente ao fluxo alternativo (*AttachStandardizedFile*) presente na especificação da Listagem H.1.

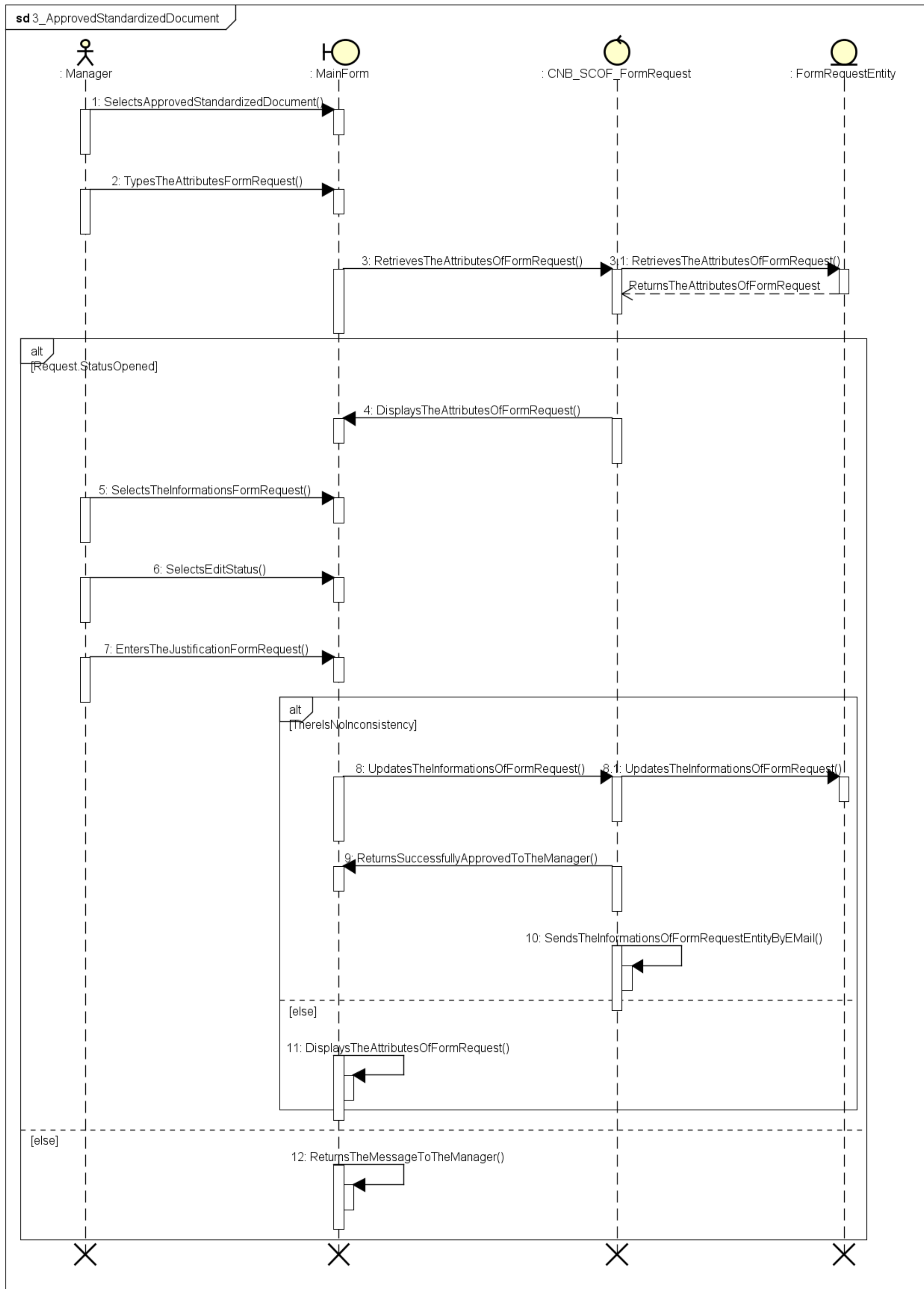
Figura 34 – Diagrama de sequência do fluxo *AttachStandardizedFile*



Fonte: Elaborado pelo autor

O diagrama de sequência referente ao fluxo alternativo (*ApprovedStandardizedDocument*) da especificação da Listagem H.1 está ilustrado pela Figura 35.

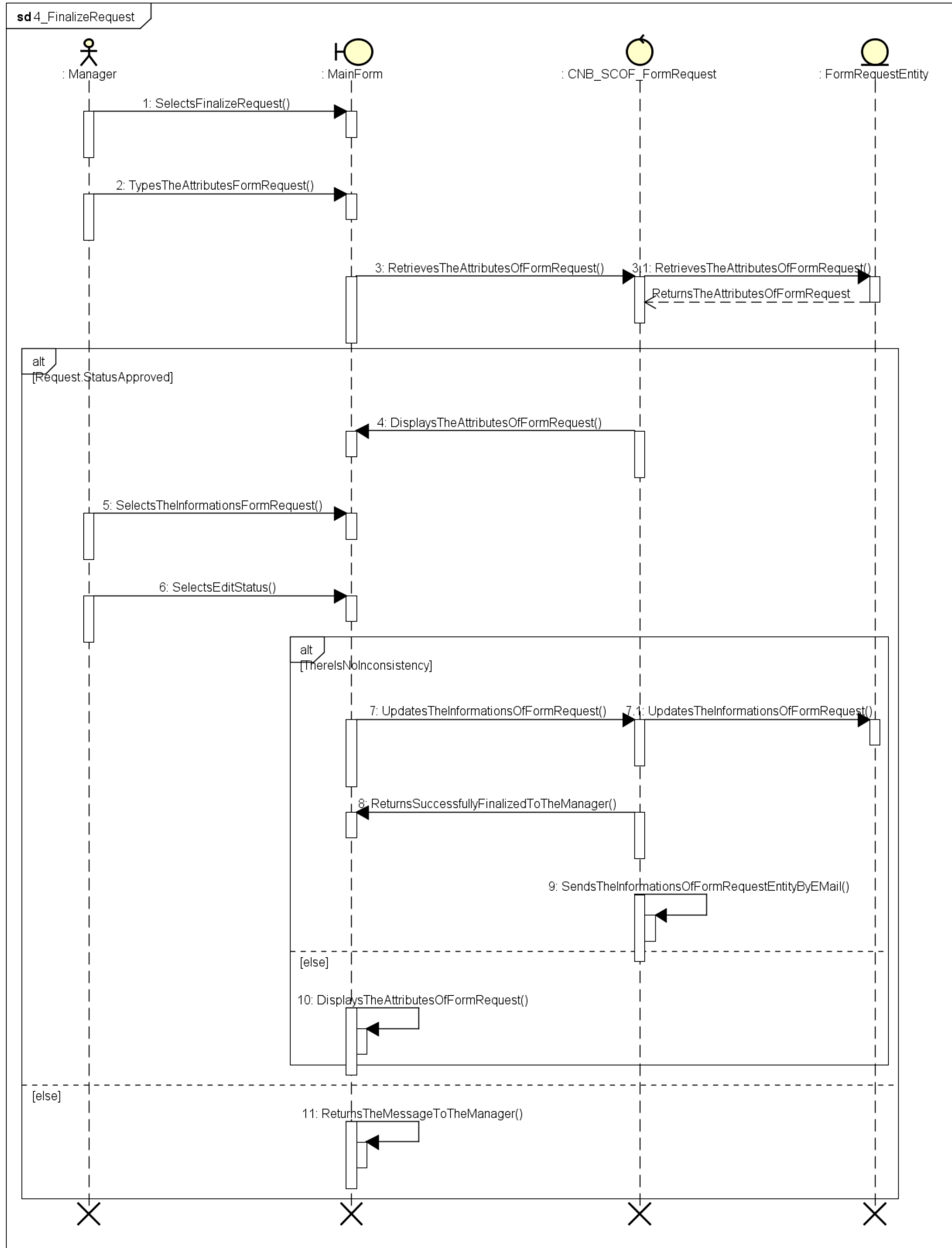
Figura 35 – Diagrama de sequência do fluxo *ApprovedStandardizedDocument*



Fonte: Elaborado pelo autor

O diagrama de sequência referente ao fluxo alternativo (*FinalizeRequest*) da especificação da Listagem H.1 está ilustrado pela Figura 36.

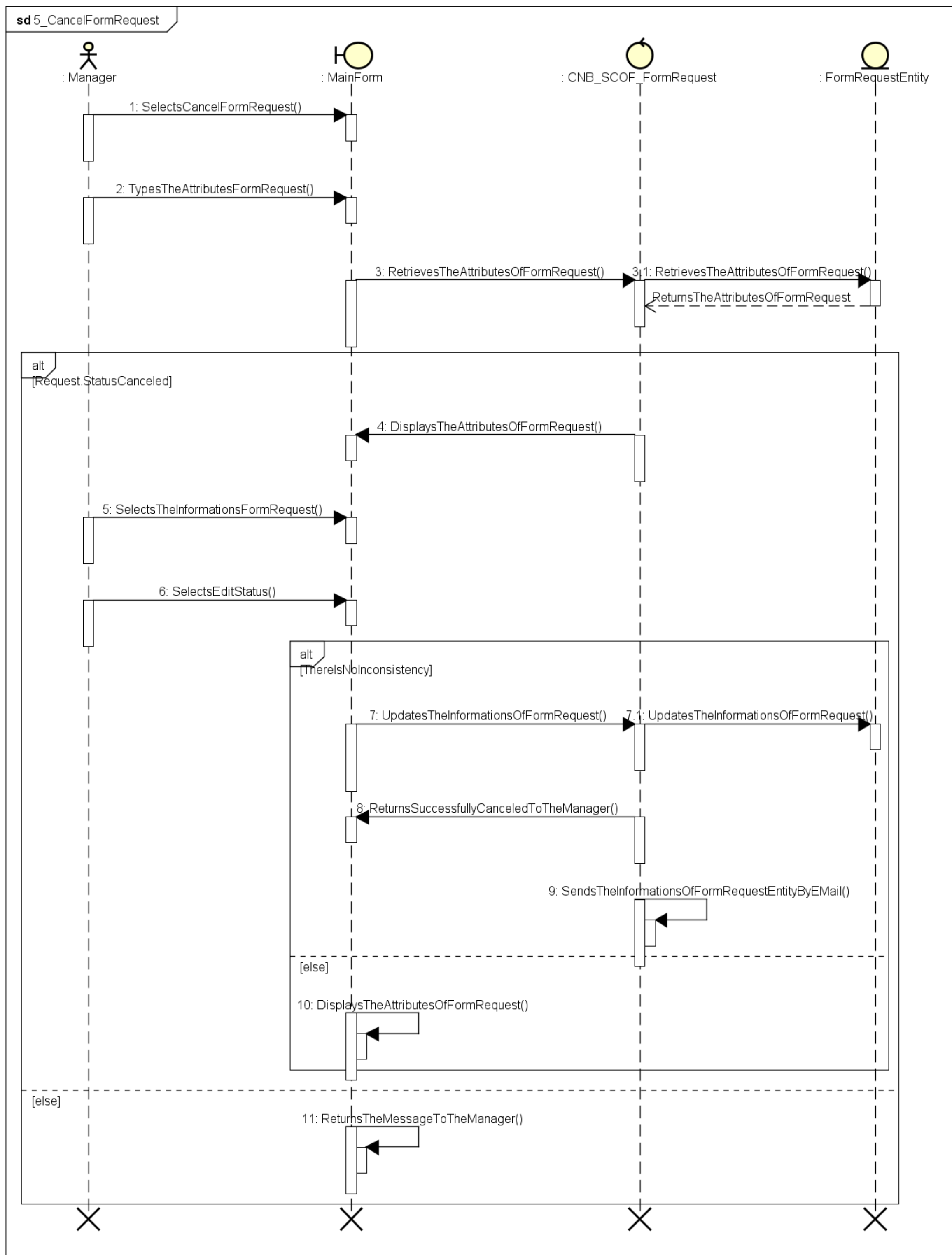
Figura 36 – Diagrama de sequência do fluxo *FinalizeRequest*



Fonte: Elaborado pelo autor

Por meio da Figura 37, é apresentado o diagrama de sequência referente ao fluxo alternativo (*CancelFormRequest*) presente na especificação da Listagem H.1.

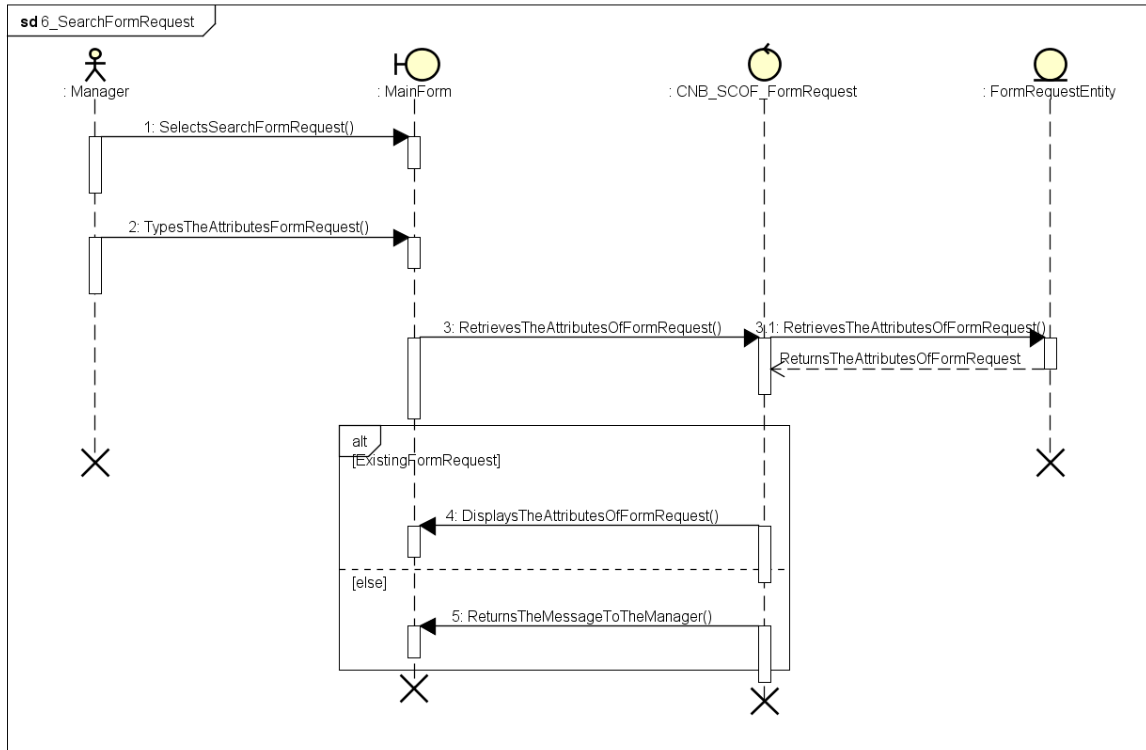
Figura 37 – Diagrama de sequência do fluxo *CancelFormRequest*



Fonte: Elaborado pelo autor

O diagrama de sequência referente ao fluxo alternativo (*SearchFormRequest*) presente na especificação da Listagem H.1 está ilustrado pela Figura 38.

Figura 38 – Diagrama de sequência do fluxo *SearchFormRequest*



Fonte: Elaborado pelo autor

APÊNDICE I – CASO DE USO “RELATÓRIO DE FORNECEDOR”

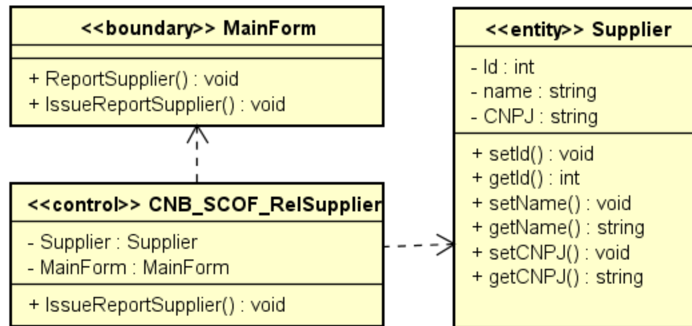
A especificação (Listagem I.1) contempla um cenário de geração de relatório, o qual o usuário tem a opção de definir alguns filtros de pesquisa e o sistema buscará registros na base de dados, retornando dois possíveis resultados: satisfatório caso exista registros; e caso não exista nenhum registro, será retornada apenas uma mensagem notificando o usuário.

Listagem I.1 – Especificação do caso de uso CNB_SCOF_RelSupplier

```
1  Use Case: CNB_SCOF_RelSupplier.
2  Brief Description
3  "Allows the generation of suppliers report.".
4  System: System.
5  Primary and Secondary Actors
6  Primary Actors: Manager.
7  Main Flow: IssueReportSupplier.
8  Manager starts Use Case.
9     Manager selects "ReportSupplier" on MainForm.
10    Manager enters attributes (int Id, string name, string CNPJ) of
         Supplier on MainForm.
11    Manager selects "IssueReportSupplier" on MainForm.
12    System searches attributes of Supplier.
13    If ["Existing Supplier"]
14       System displays the attributes of Supplier on MainForm.
15    Else
16       System returns the message to Manager.
17    EndIf
18 Manager finishes Use Case.
19 
20 Key Scenarios
21    Key Scenario 01: IssueReportSupplier.
22 Preconditions
23    "Before this use case begins the Manager has logged onto the system".
24 Postconditions
25    "There are no post conditions associated with this use case.".
26 Special Requirements
27    "There are no special requirements associated with this use case.".
28 Extension Points
29    "There are no extension points associated with this use case.".
```

De acordo com a especificação, gerou-se o seguinte diagrama de classes.

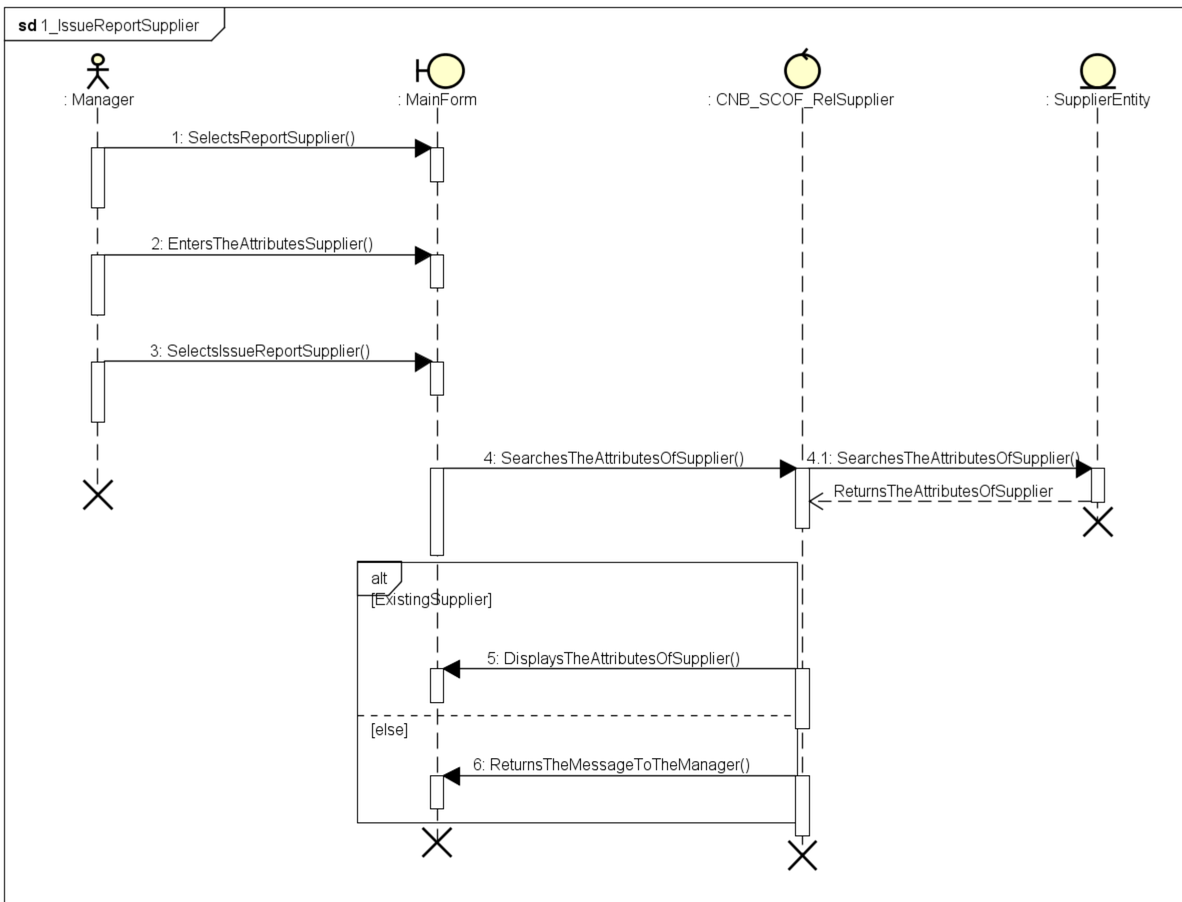
Figura 39 – Diagrama de classes do UC CNB_SCOF_RelSupplier



Fonte: Elaborado pelo autor

A seguir é apresentado o diagrama de sequência referente ao caso de uso.

Figura 40 – Diagrama de sequência do fluxo CNB_SCOF_RelSupplier



Fonte: Elaborado pelo autor

APÊNDICE J – GRAMÁTICA IMPLEMENTADA NO XTEXT

```
grammar org.xtext.example.lucam.Lucam with org.eclipse.xtext.common.Terminals

generate lucam "http://www.xtext.org/example/lucam:/Lucam:"

Lucam:
	useCaseHeader+=UseCaseHeader
	useCaseFlows+=UseCaseFlows
	useCaseFooter+=UseCaseFooter
;

UseCaseHeader:
	useCaseName+=UseCaseName
	useCaseBriefDescription+=UseCaseBriefDescription
	systemName+=SystemName
	primarySecondaryActors+=PrimarySecondaryActors
;

UseCaseFlows:
	mainFlowName+=MainFlowName
	mainFlowScope+=MainFlowScope
	(alternateFlows+=AlternateFlows)?
;

UseCaseFooter :
		'Key Scenarios' (keyScenario+=KeyScenario)*
		preConditions +=PreConditions
		postConditions += PostConditions
		specialRequirements +=SpecialRequirements
		extensionPoints+=ExtensionPoints
;

UseCaseName :
	 'Use Case: ' ControlClassID+=ID point+=POINT
;

UseCaseBriefDescription :
	'Brief Description' descricao+=STRING  point+=POINT
;

SystemName:
	'System: ' SystemID+=ID point+=POINT
;

PrimarySecondaryActors:
		'Primary and Secondary Actors'
		primaryActorName+=PrimaryActorName
		(secondaryActorName+=SecondaryActorName)?
;
PrimaryActorName :
	'Primary Actors: ' ActorID+=ID (',' ActorID+=ID)* point+=POINT
;
```

```
SecondaryActorName:
	'Secondary Actors: ' ActorID+=ID (',' ActorID+=ID)* point+=POINT
;

MainFlowName:
	'Main Flow: ' MethodControlClassID+=ID  point+=POINT
;

MainFlowScope:
	(ActorID+=ID ' starts Use Case' point+=POINT)?
	mainFlow+=MainFlow
	(ActorID+=ID ' finishes Use Case' point+=POINT)?
;

MainFlow:
	(mainFlowElements+=MainFlowElements)+
;

MainFlowElements :
		(mainFlowCore+=MainFlowCore)
		| flowIf+=FlowIf
		| flowLoop+=FlowLoop
		| flowConcurrency+=FlowConcurrency
;

FlowIf:
	'If ' condition+=Condition mainFlow+=MainFlow ('Else' mainFlow+=MainFlow )? 'EndIf'
;

FlowLoop:
	'Loop' condition +=Condition
		mainFlow+=MainFlow
	'EndLoop'
;


FlowConcurrency:
	'StartConcurrency' mainFlow +=MainFlow 'concurrent' mainFlow +=MainFlow
=>'StartConcurrency'
;

Condition :
	'['conditionText+=ConditionText ']'
;


MainFlowCore:
	(ActorSystemID+=ID(

mainFlowTabTransVerb+=MainFlowTabTransVerb
						|tabIntransVerb+=TABINTRANSVERB)
	point+=POINT)
```

```
		| returnMessage+=ReturnMessage point+=POINT
		|interactionUseCase+=InteractionUseCase point+=POINT
;

InteractionUseCase:
   actor+=ID ' executes ' ('use' 'case '|'alternate flow') useCaseName+=STRING ((','| ' and' |' or')
useCaseName+=STRING)*
;



MainFlowTabTransVerb:
		(tabTransVerb+=TABTRANSVERB) ((MethodID+=STRING)? (
mainFlowAttibutes+=MainFlowAttibutes | ' on '
		mainFlowBoundaryClass+=MainFlowBoundaryClass) |
mainFlowSubstantivoSemComplemento+=MainFlowSubstantivoSemComplemento)
;

MainFlowSubstantivoSemComplemento:
		('the ')? TABSUBSTANTIVESEMCOMPLEMENTO
;

MainFlowAttibutes:
		((' the ')? substantive+=TABSUBSTANTIVE ('('(AttributeTypeID+=ID)? AttributeID+=ID (',
'(AttributeTypeID+=ID )? AttributeID+=ID)*')')? )?

		 ('on ' mainFlowBoundaryClass+=MainFlowBoundaryClass
		| (' of ') mainFlowEntityClass+=MainFlowEntityClass
		|  mainFlowsActorClass += MainFlowsActorClass
		| mainFlowProssessingClass+=MainFlowProssessingClass)
;


MainFlowsActorClass:
		(' for ' | ' to ') ('the ')? actor+=ID
;

MainFlowBoundaryClass:
		BoundaryClassID+=ID
;

MainFlowEntityClass	:
		EntityClassID+=ID (' on ' mainFlowBoundaryClass+=MainFlowBoundaryClass
			| mainFlowProssessingClass+= MainFlowProssessingClass
			| mainFlowsActorClass += MainFlowsActorClass
		)?
;

MainFlowProssessingClass:
		' by ' mainFlowCommunication+=MainFlowCommunication
;
```

```
MainFlowCommunication:
			CommunicationID+=STRING
;

ReturnMessage:
		(SystemID=ID verb+=TABTRANSVERB simpleReturnMessage+=SimpleReturnMessage
('to ' | 'for ')('the')?
		ActorSystemID+=ID )
;

AlternateFlows:
		'Alternate Flows'
		(alternateFlowScope+=AlternateFlowScope)+
;

AlternateFlowScope:
		'Alternate Flow ' NUM+=INT ': ' MethodControlClassID+=ID point+=POINT
		alternateFlowCore+=AlternateFlowCore

;
AlternateFlowCore:
		mainFlow+=MainFlow
;

KeyScenario:
		'Key Scenario ' NUM+=INT ': ' MethodControlClassID+=ID point+=POINT
;

PreConditions:
		'Preconditions' (Text+=STRING)? point+=POINT
;

PostConditions:
		'Postconditions' (Text+=STRING)? point+=POINT
;

SpecialRequirements:
		'Special Requirements' (Text+=STRING)? point+=POINT
;
ExtensionPoints:
			'Extension Points' ( texto+=STRING)?  point+=POINT
;

SimpleReturnMessage:
		texto += STRING
;

ConditionText:
		texto+=STRING
;

POINT:
```

```
	'.'
;

TABTRANSVERB:
	'makes' | 'verifies' | ' deletes' | 'generates' | 'enables' | 'validates' | 'authenticates' |
	'accepts' | 'adds' | 'affords' | 'alerts' | 'allows' | 'analyses' | 'answers' | 'appears' |
	'approves' | 'arranges' | 'asks' | 'attaches' | 'backs' | 'bans' | 'begins' | 'calculates' |
	'calls' | 'changes' | 'checks' | 'chooses' | 'cleans' | 'clears' | 'closes' | 'collects' |
	'communicates' | 'compare' | 'completes' | 'connects' | 'consists' | 'contains' | 'copies'
|
	'corrects' | 'counts' | 'decides' | 'delays' | 'describes' | 'detects' | 'enters' |
	'examines' | 'fails' | 'fetches' | 'files' | 'fills' | 'finds' | 'gathers' | 'gets' |
	'gives' | 'guarantees' | 'guards' | 'identifies' | 'includes' | 'informs' | 'interrupts' |
	'introduces' | 'joins' | 'keeps' | 'lists' | 'loads' | 'loses' | 'marks' | 'matches' |
	'meets' | 'needs' | 'obtains' | 'opens' | 'pauses' | 'perexecuta mits' | 'picks' | 'posts' |
	'presents' | 'puts' | 'questions' | 'reads' | 'receives' | 'recognises' | 'records' |
	'releases' | 'removes' | 'repairs' | 'repeats' | 'replaces' | 'replies' | 'reports' |
	'reproduces' | ' requests' | 'rescues' | ' returns' | 'runs' | ' saves' | ' searches' |
	' selects' | ' sends' | 'settles' | ' shows' | 'shuts' | 'signals' | 'starts' | 'stores' |
	'subtracts' | 'supplies' | 'suspends' | 'switches' | 'takes' | ' types' | 'understands' |
	'waits' | 'warns' | 'writes' | ' retrieves' | ' displays' | 'modifies' | 'updates'| ' deducts'| '
credits' |
	'performs' | ' cancels '
;

TABINTRANSVERB:
	'fails' | 'exits' | 'closes' | 'fall' | 'pause'
;

TABSUBSTANTIVE:
	'message' | 'attributes' | 'dependencies' | 'informations' | 'request' | ' response' | 'list'
|
	'schedule' | 'prerequisites'| 'response' | 'file' | 'files' | 'notification' | 'justification' |
'record' | 'privileges' |
	'permission' | 'user interface' | 'interface'

;

TABSUBSTANTIVESEMCOMPLEMENTO:
	'operation'
;
```

ANEXO A – TEMPLATE OPENUP

<project>	
Use-case Specification: <use-case name>	Date: <dd/mmm/yy>

<project>
Use-Case: <use-case name>

1 Brief Description

<brief description of use-case>

2 Actor Brief Descriptions

2.1 <Actor 1 Name>

3 Preconditions

<pre-condition 1>

4 Basic Flow of Events

1. The use case begins when <actor>, <does something>…
2. <basic flow step 1>
3. …
4. <basic flow step n>
5. The use case ends.

5 Alternative Flows

5.1 <alternate flow 1>

If in step <x> of the basic flow the <actor or system does something>, then

1. <describe flow>
2. The use case resumes at step <y>

6 Subflows

6.1 <subflow 1>

1. <subflow 1, step 1>
2. …
3. <subflow 1, step n>

7 Key Scenarios

7.1 <scenario 1>

1. <scenario 1, step 1>
2. …
3. <scenario 1, step n>

8 Post-conditions

8.1 <post-condition 1>

9 Special Requirements

<special requirement 1>

Page 1 of 1