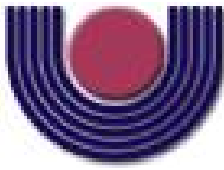


**Unioeste - Universidade Estadual do Oeste do Paraná**
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
***Curso de Bacharelado em Ciência da Computação***

# E4J: Editor *i** para JGOOSE

*Leonardo Pereira Merlin*

**CASCAVEL**
**2013**

**Leonardo Pereira Merlin**

# E4J: Editor *i** para JGOOSE

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Victor Francisco Araya Santander

CASCAVEL
2013

**Leonardo Pereira Merlin**

# E4J: Editor *i** para JGOOSE

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

---

Prof. Victor Francisco Araya Santander
Colegiado de Ciência da Computação,
UNIOESTE

---

Prof. Ivonei Freitas da Silva
Colegiado de Ciência da Computação,
UNIOESTE

---

Prof. Elder Elisandro Schemberger
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 28 de junho de 2013

*Oscar José Merlin Júnior, dedico este trabalho a você, meu irmão e minha referência.*

*"Se você pensa que pode ou se pensa que não pode,*
*de qualquer forma você está certo."*
*Henry Ford*

# AGRADECIMENTOS

Em primeiro lugar, eu gostaria de agradecer ao meu orientador, professor Victor Francisco Araya Santander, pelos conselhos, além do apoio, disponibilidade e interesse no meu trabalho. Apresentou-me oportunidades que eu jamais acreditava que teria novamente. E, além de acreditar nos resultados, sempre me inspirou coragem e ânimo para cumprir meus objetivos.

Também gostaria de agradecer aos outros membros da banca examinadora, os professores Ivonei Freitas da Silva e Elder Elisandro Schemberger. Obrigado por analisar e avaliar cuidadosamente o meu trabalho. Durante as apresentações, contribuiram significativamente com questionamentos e sugestões, de fato, relevantes. E não distante desses, meus sinceros agradecimentos a todos os acadêmicos integrantes do grupo de estudo do Laboratório de Engenharia de Software (LES). Dentre esses, um agradecimento especial aos companheiros Diego Peliser, Leonardo Zanotto Baggio e Rodrigo Trage. Obrigado pelas apresentações e discussões sobre temas de destaque na área de Engenharia de Software. Espero poder retribuir e contribuir para com o grupo.

Quanto aos colegas de moradia, Lucas Inácio, Bruno Belorte, Marcos Schmitt, Nicolas Zaro e Julian Ruiz Diaz, obrigado pela tolerância e bons momentos.

Não posso deixar de agradecer os amigos Fernando Dal Bello, Amadeu Paixão e Felipe Carminati pelas experiências profissionais, oportunidades de um trabalho em equipe bem realizado e um amadurecimento pessoal memorável (nada que uma boa trilha sonora não resolva). Em especial, agradeço ao Carlos Henrique de França, que me ajudou a esclarecer e aguçar minhas pesquisas, além das críticas em numerosas versões de documentos técnicos.

A minha família, pelos conselhos (não ignorados), pelo auxílio e suporte de diversas maneiras. E, Lah (Larissa Torquato de Oliveira e família), este lugar também é de vocês.

A minha companheira, Jamile Merlin, por participar de mais esta fase da minha vida.

À todos, meu sincero agradecimento!

# Lista de Figuras


2.1 Atores e especializações. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 Exemplo de relações entre atores. . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 Tipos de associações entre atores. . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.4 Exemplo de Relação de Dependência (*Depender* -> *Dependum* -> *Dependee*) . 11

2.5 Exemplos de ligação de dependência. . . . . . . . . . . . . . . . . . . . . . . . 13

2.6 Exemplos de fronteira do ator. . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.7 Exemplo de ligação meio-fim. . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.8 Exemplo de ligação de decomposição de tarefa. . . . . . . . . . . . . . . . . . . 15

2.9 Ligações de contribuição. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.10 Exemplo de modelagem com a ferramenta OME . . . . . . . . . . . . . . . . . 19

3.1 Modelo SD da ferramenta E4J. . . . . . . . . . . . . . . . . . . . . . . . . . . 26

3.2 Modelo SR da ferramenta E4J. . . . . . . . . . . . . . . . . . . . . . . . . . . 29

3.3 Casos de Uso gerados pela ferramenta JGOOSE. . . . . . . . . . . . . . . . . . 31

3.4 Casos de Uso importados na ferramenta StarUML. . . . . . . . . . . . . . . . . 32

3.5 Arquitetura da ferramenta JGOOSE demonstrando o seu funcionamento. . . . . 33

3.6 Elementos do Diagrama de Atividades UML. . . . . . . . . . . . . . . . . . . . 34

3.7 Diagrama de atividades em nível contextual da ferramenta JGOOSE. . . . . . . 36

3.8 Diagrama de Atividades da Ferramenta JGOOSE 2011. . . . . . . . . . . . . . 38

4.1 Estrutura central do JGraphX. . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

4.2 Estrutura do modelo do JGraphX. . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.3 Diagrama de Pacotes do projeto E4J. . . . . . . . . . . . . . . . . . . . . . . . 45

4.4 Diagrama de Classes parcial do E4J. Estrutura principal. . . . . . . . . . . . . 47

4.5 Diagrama de Classes parcial do E4J. Principais ações do usuário. . . . . . . . . 48

4.6 Padrão de projeto Mediator adaptado e aplicado entre os projetos E4J e JGOOSE. 49
4.7 Interface Gráfica com menu de chamada ao editor E4J. . . . . . . . . . . . . . 50
4.8 Interface Gráfica Principal do E4J. . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.9 Diagrama de atividades da ferramenta JGOOSE após integração com E4J. . . . 58
4.10 Diagrama de atividades da ferramenta E4J. . . . . . . . . . . . . . . . . . . . . . 59
4.11 Parte da classe "ccistarmlContent". . . . . . . . . . . . . . . . . . . . . . . . . 63

5.1 Arquivo e pasta disponíveis na distribuição do E4J. . . . . . . . . . . . . . . . . 65
5.2 Acessando editor E4J pela interface JGOOSE. . . . . . . . . . . . . . . . . . . . 66
5.3 Tela principal da E4J. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.4 Exemplo: criando uma ligação de dependência. . . . . . . . . . . . . . . . . . . 69
5.5 Exemplo: criando um modelo SR. . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.6 Exemplo: criando uma ligação de dependência. . . . . . . . . . . . . . . . . . . 72
5.7 Exemplo: criando uma ligação de decomposição de tarefa. . . . . . . . . . . . . 72
5.8 Exemplo: criando uma ligação de meio-fim. . . . . . . . . . . . . . . . . . . . . 73
5.9 Salvando diagrama em .mxe . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.10 Conteúdo parcial do diagrama salvo em .mxe . . . . . . . . . . . . . . . . . . . 74
5.11 Salvando diagrama em .istarml . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.12 Conteúdo parcial do diagrama salvo em .istarml . . . . . . . . . . . . . . . . . 75
5.13 Gerar Casos de Uso. Passos (1) e (2). . . . . . . . . . . . . . . . . . . . . . . . 76

# Lista de Abreviaturas e Siglas

| | |
|---|---|
| API | *Application Programming Interface* |
| BSD | *Berkeley Software Distribution* |
| CASE | *Computer-Aided Software Engineering* |
| DTD | *Document Type Definition* |
| EMF | *Eclipse Modelling Framework* |
| ER | Engenharia de Requisitos |
| ES | Engenharia de Software |
| GOOSE | *Goal into Object Oriented Standard Extension* |
| GRL | *Goal-Oriented Requirements Language* |
| GUI | *Graphical User Interface* |
| IDE | *Integrated Development Environment* |
| ITU | *International Telecommunication Union* |
| JAXB | *Java Architecture for XML Binding* |
| JGOOSE | *Java Goal into Object Oriented Standard Extension* |
| LES | Laboratório de Engenharia de Software |
| NFR | *Non-Functional Requirements* |
| OME | *Organization Modelling Environment* |
| OO | Orientado à Objetos |
| ORM | *Object-Relational Mapping* |
| POM | *Project Object Model* |
| POO | Paradigma Orientado à Objetos |
| RNF | Requisitos Não-Funcionais |
| SD | Modelo de Dependências Estratégicas |
| SR | Modelo de Razões Estratégicas |
| UCM | *Use Case Maps* |
| UML | *Unified Modeling Language* |
| URN | *User Requirements Notation* |
| XMI | *Xml Metadata Interchange* |
| XML | *eXtensible Markup Language* |
| UNIOESTE | Universidade Estadual do Oeste do Paraná |

# Sumário

**Lista de Figuras** vii

**Lista de Abreviaturas e Siglas** ix

**Sumário** x

**Resumo** xiii

**1 Introdução** 1

1.1 Contexto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Motivação . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.3 Proposta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.4 Contribuições Esperadas . . . . . . . . . . . . . . . . . . . . . . . . 4

1.5 Estrutura do Trabalho . . . . . . . . . . . . . . . . . . . . . . . . . 5

**2 Framework i*: Variações e Ferramentas** 6

2.1 O Framework i* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1.1 Modelo de Dependências Estratégicas (SD) . . . . . . . . . . . . . . 8

2.1.2 Modelos de Razões Estratégicas (SR) . . . . . . . . . . . . . . . . . 13

2.2 Variações Baseadas no Framework i* . . . . . . . . . . . . . . . . . . 16

2.2.1 i* Wiki . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.2.2 Tropos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.2.3 *Goal-Oriented Requirements Language* (GRL) . . . . . . . . . . . . 17

2.3 Ferramenta OME . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.4 iStarML: uma proposta para intercâmbio entre ferramentas para i* . . . . . . . 19

2.5 Considerações Finais do Capítulo . . . . . . . . . . . . . . . . . . . . 20

**3 JGOOSE** 21

3.1 Visão Geral . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1.1 Diretrizes e Passos . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.1.2 Resumo Histórico . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.2 Usando a Ferramenta JGOOSE . . . . . . . . . . . . . . . . . . . . . . . . 25
3.3 Projeto e Arquitetura . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.4 Considerações Finais do Capítulo . . . . . . . . . . . . . . . . . . . . . . 39
**4 E4J - Editor i* para JGOOSE** **40**
4.1 Visão Geral . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.1.1 JGraphX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.2 Projeto e Arquitetura . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2.1 Diagrama de Pacotes . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.2 Diagrama de Classes . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.3 Desenvolvimento . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.3.1 Definição da Estrutura E4J . . . . . . . . . . . . . . . . . . . . . . 49
4.3.2 Adaptação do JGOOSE . . . . . . . . . . . . . . . . . . . . . . . . 50
4.3.3 Interface Gráfica do Usuário . . . . . . . . . . . . . . . . . . . . . 50
4.3.4 Estrutura do Modelo E4J . . . . . . . . . . . . . . . . . . . . . . . 52
4.3.5 Linguagem iStarML . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3.6 Rotina de Mapeamento . . . . . . . . . . . . . . . . . . . . . . . . 54
4.4 Impacto do editor E4J no JGOOSE . . . . . . . . . . . . . . . . . . . . . 56
4.5 Considerações Finais do Capítulo . . . . . . . . . . . . . . . . . . . . . . 62
4.5.1 Problemas Encontrados . . . . . . . . . . . . . . . . . . . . . . . . 62
**5 Exemplos de Uso** **64**
5.1 Instalação e requisitos necessários . . . . . . . . . . . . . . . . . . . . . 64
5.2 Conhecendo o E4J . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.2.1 Abrindo o E4J . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.2.2 Interface Gráfica do Usuário (GUI) . . . . . . . . . . . . . . . . . . 66
5.3 Construindo Modelos SD e SR . . . . . . . . . . . . . . . . . . . . . . . 67
5.3.1 Ligações de dependências . . . . . . . . . . . . . . . . . . . . . . 71
5.3.2 Ligações de decomposição de tarefa . . . . . . . . . . . . . . . . . 72
5.3.3 Ligações de meio-fim . . . . . . . . . . . . . . . . . . . . . . . . . 73

5.4 Salvando e Exportando . . . 73
5.4.1 Salvando diagrama . . . 73
5.4.2 Exportando para iStarML . . . 74
5.5 Gerando Casos de Uso com a JGOOSE . . . 75
5.6 Considerações Finais do Capítulo . . . 76
**6 Conclusão** **77**
6.1 Resultados . . . 77
6.2 Conclusões . . . 77
6.3 Trabalhos Futuros . . . 78
**A iStarML API: um início** **80**
A.1 Introdução . . . 80
A.1.1 Problemas com o *ccistarml* . . . 80
A.1.2 Solução adotada . . . 81
A.2 Estrutura iStarML . . . 81
**B i* Wiki Questionnaire** **84**
B.1 General Information . . . 84
B.2 i* Modelling Suitability . . . 85
B.3 Usability . . . 86
B.4 Maturity of the tool . . . 86
B.5 Extensibility and Interoperability . . . 87
**C E4J - Atalhos** **88**
**Referências Bibliográficas** **89**

# Resumo

Em discussões pertinentes à engenharia de requisitos ressalta-se que um dos principais desafios da área consiste na integração de modelos organizacionais às demais etapas do processo de engenharia de requisitos. Alguns trabalhos relacionados já apresentaram técnicas e ferramentas para a realização dessa integração. Nesse sentido, em 2006 foi desenvolvida uma solução computacional denominada JGOOSE (*Java Goal Into Object Oriented Standard Extension*), uma ferramenta de auxílio a engenharia de requisitos que proporciona a automatização do mapeamento de modelos e diagramas do framework i* para casos de uso UML. Apesar das atualizações e melhorias realizada por outros pesquisadores e participantes do grupo do Laboratório de Engenharia de Software (LES) da Universidade Estadual do Oeste do Paraná (UNIOESTE - Campus Cascavel/PR), a ferramenta ainda possui uma dependência crítica em relação à elaboração dos dados de entrada. Nas condições atuais é necessário instalar alguma ferramenta específica para produzir o arquivo no formato TELOS. Desta forma, este trabalho apresenta o projeto e desenvolvimento de um editor gráfico de modelos organizacionais i* integrado à ferramenta JGOOSE, visando melhorar suas funcionalidades e diminuir a necessidade de outros softwares para este fim. Essa integração significa que a conversão de estruturas e modelos, do E4J para o JGOOSE, é realizada sem a necessidade de geração de arquivos intermediários. Uma característica importante desse novo editor é o suporte à especificação iStarML - um formato de arquivo baseado em XML para representação e intercâmbio de modelos i*.

**Palavras-chave:** E4J, JGOOSE, iStarML, *Framework* i*, Engenharia de Requisitos.

# Capítulo 1

# Introdução

Este primeiro capítulo tem como objetivo a apresentação geral do trabalho. É realizada a contextualização e delimitação da pesquisa ao escopo da Engenharia de Software, bem como são destacados os principais objetivos da pesquisa. Apresenta-se inicialmente, na seção 1.1, o contexto sobre as ferramentas de modelagem organizacional e suas contribuições na área da Engenharia de Requisitos, destacando a influência dessas ferramentas no desenvolvimento de produtos de qualidade. Na seção 1.2, são apresentadas as principais influências e motivações para a realização do trabalho. Em seguida, na seção 1.3, é apresentada a proposta sob uma visão geral e os objetivos norteadores da pesquisa. Na seção 1.4, descreve-se as contribuições esperadas com a nossa proposta. Por fim, na seção 1.5, é apresentada a estrutura geral e a organização do restante desta monografia.

## 1.1 Contexto

Existem muitas ferramentas e técnicas que visam auxiliar engenheiros de requisitos no processo de construção de modelos organizacionais i* [1, 2]. Classificadas como CASE (*Computer-Aided Software Engineer*) [1] [3], essas ferramentas têm como objetivo o aumento da produtividade no processo de construção de requisitos, melhorando também a qualidade final dos modelos, através da automatização e gerenciamento de várias fases da Engenharia de Software.

A área de Engenharia de Requisitos (ER), subárea da Engenharia de Software, é responsável por diversas atividades que abrangem os processos de análise, elicitação, especificação, avalia-

[1] Ferramentas CASE, é toda e qualquer ferramenta baseada em computador que auxilie nas atividades de desenvolvimento de software.

ção, ajuste, documentação e evolução dos requisitos de um sistema computacional [4]. É uma das áreas mais críticas para o sucesso e qualidade de um projeto de software [5].

Para tentar diminuir os problemas relacionados as fases iniciais do projeto, pesquisas recentes mostram que a comunidade tem buscado estabelecer e utilizar padrões de técnicas, métodos e ferramentas para tratar especificamente da fase inicial de desenvolvimento de software [6]. Pensando nisso, têm-se investido esforços no processo de modelagem organizacional. Este tipo de modelagem visa prover recursos que permitam modelar as intenções, relacionamentos e motivações entre membros de uma organização [7]. Dentre as técnicas de modelagem organizacional, destaca-se a i*, proposto por [8], uma técnica que utiliza a orientação a agentes e a objetivos [9] com enfoque tanto nos desejos e intenções desses agentes, quanto suas dependências [1, 10]. Mais detalhes sobre esta técnica e suas variações serão discutidos no capítulo 2.

Pensando em auxiliar no processo de desenvolvimento de software, alguns trabalhos foram propostos com o intuito de realizar o mapeamento de modelos do *framework* i* para diagramas da UML (*Unified Modeling Language*) [11]. Dentre esses trabalhos, destaca-se o trabalho de Santander [12], que propõe a derivação em casos de uso UML a partir de modelos do *framework* i*.

Para apoiar esse processo de derivação, foi desenvolvida a ferramenta JGOOSE (*Java Goal into Object Oriented Standard Extension*) [13], uma ferramenta que mapeia de forma automática os diagramas i* para casos de uso UML. Essa ferramenta tem como base as diretrizes e passos propostos por Santander [12].

Desta forma, tendo como entrada os modelos i*, no formato de arquivo TELOS [14, 15], a ferramenta consegue gerar conforme o template proposto em [16] os casos de uso UML com um bom nível de detalhamento.

## 1.2 Motivação

A área de Engenharia de Requisitos está em crescimento e destaque por impactar de forma significativa nos resultados finais de um projeto de software [17]. O *framework* i* é a base da GRL (*Goal-oriented Requirements Language*) [18], uma linguagem de modelagem orientada a

objetivos com enfoque nos requisitos não-funcionais, que junto à UCM [2] [19], constituíram a URN [3] [20], que passou a ser adotada como um padrão internacional em novembro de 2008 pela ITU (*International Telecommunication Union*) [21, 22]. No meio industrial, apesar de ainda estar em processo de evolução e incentivo, existem estudos na área de modelagem organizacional. Em [23] são descritos alguns projetos utilizando i* no âmbito industrial. Como exemplo, tem-se projetos na área de BI (*Business Intelligence*), integração de modelos nos serviços de engenharia de software, *Model-Integrated Software Service Engineering*, requisitos não-funcionais (*Developing Non-Functional Requirements for Service-Oriented Software Platforms*), entre outros. Além de se tratar de um padrão internacional, i* é a técnica de modelagem na qual as diretrizes de Santander [12] estão embasadas. Sendo assim, a ferramenta JGOOSE também está associada aos conceitos *framework* i* e as diretrizes citadas.

Inicialmente apresentada como GOOSE (*Goal into Object Oriented Standard Extension*) [11, 24] e em seguida, melhorada e apresentada como JGOOSE [25], passou também por melhorias [26] e, atualmente, está sendo objeto de estudo de pesquisadores do Grupo do Laboratório de Engenharia de Software (LES) da Universidade Estadual do Oeste do Paraná (UNIOESTE) - *campus* Cascavel-PR. Nessa universidade, é frequente o uso por acadêmicos do curso de Ciência da Computação.

A principal motivação para o presente projeto está no fato de que a ferramenta JGOOSE não possui funcionalidades para a criação dos modelos organizacionais nem a produção dos arquivos de entrada da ferramenta. Ainda existe a dependência da ferramenta OME para criar os modelos i* ou, mais especificamente, os modelos devem estar no formato de arquivo TELOS.

Nesse contexto, percebe-se a necessidade de se desenvolver um editor de modelos i* integrado à ferramenta JGOOSE, bem como implementar o suporte à especificação do formato de arquivo iStarML [27], um formato em XML para representação de modelos i* com o propósito de servir como um intercâmbio entre os outros meta-modelos existentes [28].

[2]UCM - *Use Case Maps* , em português: “Mapas de Caso de Uso”. Uma técnica de engenharia de software baseada em cenários para descrever relacionamentos entre um ou mais casos de uso.

[3]URN - *User Requirements Notation*, em portugês: “Notação Requisitos de Usuário”. Notação destinada a elicitação, análise, especificação e validação de requisitos.

## 1.3 Proposta

Neste trabalho, apresenta-se o desenvolvimento de um editor de modelos organizacionais i*, denominado E4J, integrado à ferramenta JGOOSE, com o intuito de facilitar a fase inicial do processo de engenharia de software e melhorar o processo de criação e alteração dos modelos organizacionais i* junto ao processo de mapeamento para Casos de Uso UML do JGOOSE.

É realizado um estudo sobre o *framework* i* e suas variações. Como o mapeamento do JGOOSE está fortemente acoplado às diretrizes propostas por Santander [12] e essas diretrizes, por sua vez, estão fundamentadas nos conceitos do *framework* i*, o estudo desse *framework* se faz necessário visto que os conceitos devem ser compatíveis.

É estudada a ferramenta OME *Organization Modelling Environment*, pois além de possuir problemas de usabilidade e estabilidade, era a única ferramenta geradora dos dados de entrada compatíveis com o JGOOSE. Era necessário criar os modelos do framework i* na OME e, então, exportá-los para o formato de arquivo TELOS (.tel) - único formato interpretado pelo JGOOSE.

Um estudo sobre as versões da ferramenta JGOOSE ao longo dos anos é realizado visando o entendimento da aplicação e o auxílio na implementação de rotinas de integração.

A fim de verificar as principais funcionalidades da ferramenta proposta, um conjunto de exemplos de uso é elaborado.

## 1.4 Contribuições Esperadas

Espera-se com este trabalho contribuir com a área de Engenharia de Requisitos por meio da ferramenta desenvolvida E4J, que fornece um ambiente de desenvolvimento de diagramas do *framework* i* de forma integrada com o JGOOSE. Essa integração visa uma maior independência para a ferramenta JGOOSE e, consequentemente, seus usuários, diminuindo a necessidade de outras ferramentas para realizar a modelagem organizacional.

É uma área de grande destaque na comunidade [1] e espera-se que este trabalho contribua e incentive o uso da ferramenta JGOOSE e do editor E4J para experiências acadêmicas e industriais. Deseja-se uma contribuição significativa diante das ferramentas da comunidade i*, ajudando a divulgar os trabalhos relacionados ao grupo de pesquisa do LES e de todos os en-

volvidos no desenvolvimento da ferramenta JGOOSE. Mais especificamente, este trabalho visa a qualidade a nível de inserção E4J no quadro comparativo de ferramentas do i* Wiki [1]. Com base nos comentários obtidos no referido Wiki, será possível realizar melhorias tanto no editor E4J quanto na ferramenta JGOOSE.

## 1.5 Estrutura do Trabalho

Na sequência, o trabalho encontra-se organizado da seguinte maneira:

- **Capítulo 2**: são apresentados os fundamentos teóricos sob o domínio da modelagem organizacional. Serão mostrados os conceitos básicos e características *framework* i*, as principais variações da especificação original e algumas ferramentas que trabalham com esse *framework*.

- **Capítulo 3**: é realizado um estudo sobre a evolução histórica do software JGOOSE e uma análise e discussão detalhada de sua arquitetura na versão 2013. É discutido sobre um novo diagrama de atividades para representar a ferramenta. Ao final, um exemplo de uso da JGOOSE é apresentado;

- **Capítulo 4**: será detalhada a proposta deste trabalho, mostrando o impacto do novo editor na ferramenta JGOOSE e uma visão geral do projeto e arquitetura da proposta. Os passos seguidos no desenvolvimento do editor e as decisões de projetos adotadas serão justificadas.

- **Capítulo 5**: é apresentado um exemplo de uso, mostrando a aplicação da ferramenta em um domínio específico e analisado os resultados obtidos;

- **Capítulo 6**: reúne, por fim, as análises e considerações finais sobre esta pesquisa e relata sobre os possíveis trabalhos futuros.

# Capítulo 2

# Framework i*: Variações e Ferramentas

Neste capítulo são apresentados os conceitos básicos necessários para o entendimento sobre o framework i* e dos trabalhos baseados no mesmo. Por fim, discute-se sobre algumas ferramentas de modelagem i* e suas variações. Inicialmente, na seção 2.1, são apresentados os conceitos fundamentais do i* através dos seus dois componentes de modelagem. Na seção 2.2, mostra-se algumas variações ou extensões da proposta inicial do i* [29]. Em seguida, na seção 2.3, descreve-se brevemente sobre a ferramenta OME, ferramenta atualmente utilizada para gerar os arquivos de entrada para a ferramenta JGOOSE. Por fim, na seção 2.5, são realizadas as considerações finais do capítulo.

## 2.1 O Framework i*

O framework i* (pronunciado "i-star" [1]), originalmente proposto por Yu [29], é um framework de modelagem organizacional conceitual utilizado no desenvolvimento de modelos que auxiliam a análise de sistemas sob uma visão estratégica e intencional de processos que envolvem vários participantes.

O framework i* preocupa-se principalmente com a análise do contexto organizacional e social de um sistema. O sistema, nesse caso, não consiste somente em componentes técnicos, mas também de elementos humanos e suas relações. Como o framework i* é bastante flexível para representar situações envolvendo interações entre múltiplos participantes, o mesmo pode ser utilizado para representar variados contextos organizacionais.

A seguir, apresenta-se alguns contextos onde a modelagem i* vem sendo aplicada:

[1] O nome i*, pronunciado em inglês "i-star" faz referência ao conceito sobre uma intencionalidade distribuída. No Brasil, são comuns as pronúncias "i-estrela" e "i-star".

- **Engenharia de Requisitos**: é uma das áreas de aplicações mais comuns do i*. Este framework é utilizado principalmente nas fases iniciais do processo de engenharia de requisitos (*Early Requirements*[2]) [10, 30];

- **Modelagem de Negócio (Business Modeling)**: estudos na área apresentaram o uso do i* para visualização explícita da intencionalidade dos processos de negócios. Isso ajuda a obter um melhor entendimento sobre o trabalho, além de facilitar seu replanejamento [31, 32];

- **Desenvolvimento Orientado à Objeto**: alguns trabalhos [33, 34] utilizaram-se da pUML (precise UML) [35] e da *Object Constraint Language* (OCL) [36] para tratar dos requisitos finais (*Late Requirements*[3]), além de usar o framework i* para os requisitos iniciais;

- **Desenvolvimento Orientado à Agentes**: em [37] apresentou-se o uso de agentes com estrutura BDI (*Believe, Desire and Intention*) [38] para realizar análises na fase inicial de requisitos. Já em [39], utilizou-se de Sistemas Multi-Agentes (SMA) para especificar a estrutura organizacional;

- **Segurança, Confiabilidade e Privacidade**: a modelagem i* pode ajudar a lidar com elementos de segurança, confiabilidade e privacidade, através do estudo dos conflitos de intenções de diferentes entidades sociais [40];

Segundo [41], pode-se dizer que o framework i* abrange técnicas de modelagem tanto orientado a agentes quanto orientado a objetivos, pois sua essência é realizada na combinação de conceitos agentes e objetivos. Ambos os paradigmas, Orientação à Agentes [42] e Orientação à Objetivos [18], têm apresentado bons resultados em contextos de modelagem organizacional, principalmente em modelagens da fase inicial (*Early Requirements*) do processo de engenharia de requisitos.

O i* é composto por dois componentes de modelagem: modelo de dependências estratégicas (SD) e modelo de razões estratégicas (SR). Esses componentes auxiliam na representação, respectivamente, das dependências entre atores e dos detalhes por atrás das dependências de

[2]Requisitos Iniciais (*Early Requirements*): estudar a organização visando entender seus problemas.

[3]Requisitos Finais: descrever o sistema sob o contexto do ambiente operacional, com as principais funções e características.

cada ator. É fundamental conhecer as notações e saber aplicar esses conceitos para se construir um bom modelo organizacional [43]. A seguir, são apresentados os conceitos e notações por trás dos modelos [1].

### 2.1.1 Modelo de Dependências Estratégicas (SD)

O modelo SD representa um conjunto de relacionamento estratégicos externos entre os atores organizacionais, formando uma rede de dependências [29]. Fornece uma visão mais abstrata e ampla da organização, sem se preocupar com os detalhes (razões internas) por trás dessas dependências. Segue a descrição dos elementos que compõe este modelo:

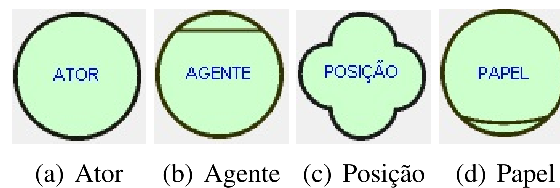
**Atores e Especializações**:

- **Ator** pode ser definido como uma entidade (humana ou computacional) que age sobre o meio que está inserido para conquistar seus objetivos, exercitando seu *know-how* [29]. Atores podem ser vistos como uma referência genérica a qualquer unidade que se possa atribuir dependências intencionais. Os atores possuem relações de dependências com outros atores para um determinado fim. Quando existe uma necessidade de maiores detalhes sobre um modelo organizacional, atores podem ser diferenciados em três especializações: agentes, posições e papéis. A Figura 2.1 exemplifica os tipos de atores, enquanto a Figura 2.2 [44] apresenta um exemplo dos possíveis relacionamentos entre os tipos de atores. A seguir, descreve-se esses tipos:

  - **Agente** é a decomposição de um ator que possui manifestações físicas concretas. Refere-se tanto a humanos quanto a agentes de software ou hardware. Um agente possui dependências independentemente do papel que está executando. As características de um agente normalmente não são fáceis de se transferir para outros atores. São como experiências, habilidades ou mesmo limitações físicas.

  - **Posição** representa uma abstração intermediária entre um agente e um papel. É o conjunto de papéis tipicamente executados por um agente, ou seja, representa uma posição dentro da organização onde o agente pode desempenhar várias funções (papéis). Diz-se que um agente ocupa uma posição e uma posição cobre um papel.

– **Papel** é a caracterização abstrata do comportamento de um ator dentro de determinados contextos sociais ou domínio de informação. Essas características devem ser facilmente transferíveis a outro ator social. As dependências associadas a um papel são aplicáveis independentemente do agente que desempenha o papel.




(a) Ator (b) Agente (c) Posição (d) Papel

Figura 2.1: Atores e especializações.

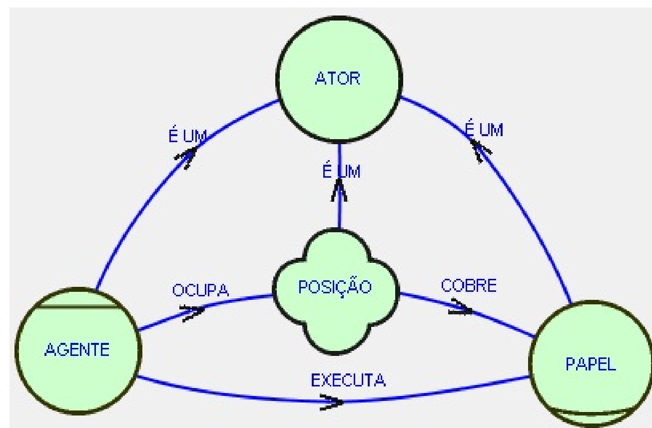



Figura 2.2: Exemplo de relações entre atores.

**Associações entre Atores**: As associações entre os atores são descritas através de links de associação, conforme apresentado na Figura 2.3 [1]. Essas associações podem ser de seis tipos:

- ***IS PART OF*** (faz parte de) - Nessa associação cada papel, posição e agente pode ter sub-partes. Em *IS PART OF* há dependências intencionais entre o todo e sua parte. Por exemplo, a dependência do todo sobre suas partes para manter a unidade na organização.

- ***ISA*** (é um) - Essa associação representa uma generalização, com um ator sendo um caso especializado de outro ator. Ambas, *ISA* e *IS PART OF*, podem ser aplicadas entre quaisquer duas instâncias do mesmo tipo de ator.

- ***PLAYS*** (executa) - A associação *PLAYS* é usada entre um agente e um papel, com um agente executando um papel. A identidade do agente que executa um papel não deverá ter efeito algum nas responsabilidades do papel ao qual está associado, e similarmente, aspectos de um agente deverão permanecer inalterados mesmo associados a um papel que este desempenha.

- ***COVERS*** (cobre) - A associação *COVERS* é usada para descrever uma relação entre uma posição e os papéis que esta cobre.

- ***OCCUPIES*** (ocupa) - Esta associação é usada para mostrar que um agente ocupa uma posição, ou seja, o ator executa todos os papéis que são cobertos pela posição que ele ocupa.

- **INS** - Esta associação é usada para representar uma **INS**tância específica de uma entidade mais geral. Por exemplo, quando se deseja representar um agente que é uma instanciação de outro agente.

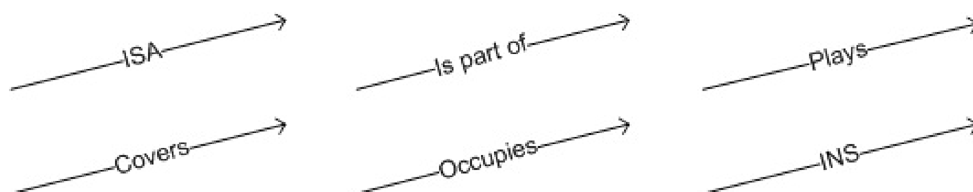



Figura 2.3: Tipos de associações entre atores.

**Relação de Dependência**: Uma relação de dependência pode ser definida como um acordo entre dois atores. Os elementos que compõe uma relação de dependência são:

- ***Depender***: é o ator dependente, ou seja, o ator que precisa que o acordo (*Dependum*) seja realizado. Esse ator não se importa como o outro ator (*Dependee*) irá satisfazer a necessidade da dependência.

- ***Dependum***: é o elemento intermediário, objeto de questionamento e validação, da relação de dependência.

- ***Dependee***: é o ator que tem a responsabilidade de satisfazer a relação de dependência.

Dessa forma, pode-se classificar o tipo de uma relação de dependência com base em um dos seguintes tipos de *Dependum*:

- **Objetivo** (*Goal*) - é uma declaração de afirmação sobre um certo estado do mundo. Deve ser de fácil verificação. O *Dependee* é livre para tomar qualquer decisão para satisfazer o objetivo e é esperado que ele o faça. Não importa para o *Depender* como o *Dependee* irá alcançar esse objetivo.

- **Tarefa** (*Task*) - é uma atividade a ser realizada pelo *Dependee*. Tarefas podem ser vistas como a realização de operações, processos, etc. Porém, não deve ser uma descrição passo-a-passo ou uma especificação completa de execução de uma rotina.

- **Recurso** (*Resource*) - é entidade (física ou informativa) a ser entregue para o *Depender* pelo *Dependee*. Satisfazendo-se esta dependência, o *Depender* está habilitado a usar essa entidade como um recurso.

- **Objetivo-Soft** (*Softgoal*) - é semelhante ao Objetivo, porém os critérios de avaliação e verificação são mais subjetivos. O *Depender* pode decidir sobre o que constitui a realização satisfatória do objetivo.

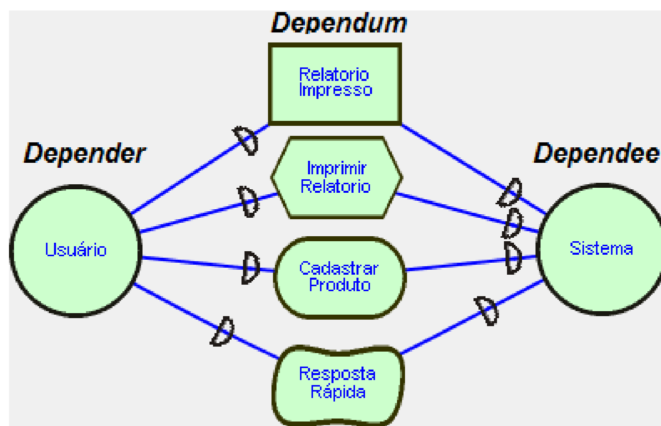


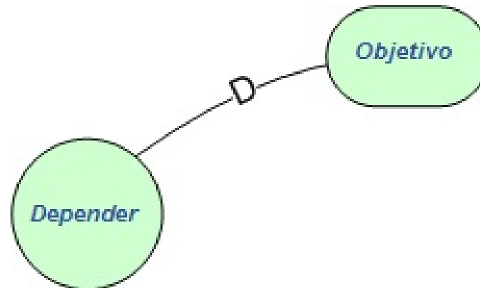

Figura 2.4: Exemplo de Relação de Dependência (*Depender* -> *Dependum* -> *Dependee*)

A Figura 2.4 apresenta alguns exemplos de relações de dependências. Nesta figura podemos observar dois atores denominados "Usuário" e "Sistema" que possuem quatro relações de dependências:

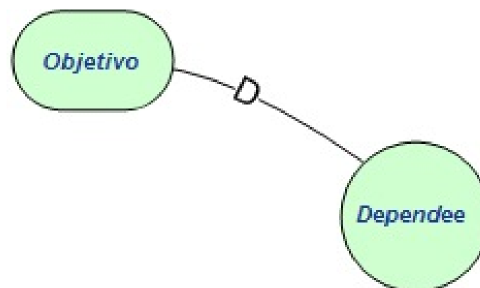
- “Relatório Impresso”: representa um **recurso** concreto de relatórios impressos;
- “Imprimir Relatório”: é uma **tarefa** que o ator “Usuário” depende do ator “Sistema” para realizá-la;
- “Cadastrar Produto”: representa um **objetivo** do ator “Usuário” sobre o ator “Sistema” para que o cadastro de produto seja realizado;
- “Resposta Rápida”: é um **objetivo-soft** cuja a satisfação é relativa aos critério do ator “Usuário”.

**Ligação de dependência**: É uma conexão direcionada entre dois elementos. No modelo SD, pode-se ter somente duas opções dessa conexão: ou do *Depender* para o *Dependum* ou do *Dependum* para o *Dependee*. Uma ligação de dependência é definida por um segmento contínuo e direcionada da origem para o destino, com a letra “D” sobrescrita, conforme os exemplos apresentados na Figura 2.5.

(a) Ligação de Dependência.




(b) Ligação de Dependência: do *Depender* para o *Dependum*.




(c) Ligação de Dependência: do *Dependum* para o *Dependee*.

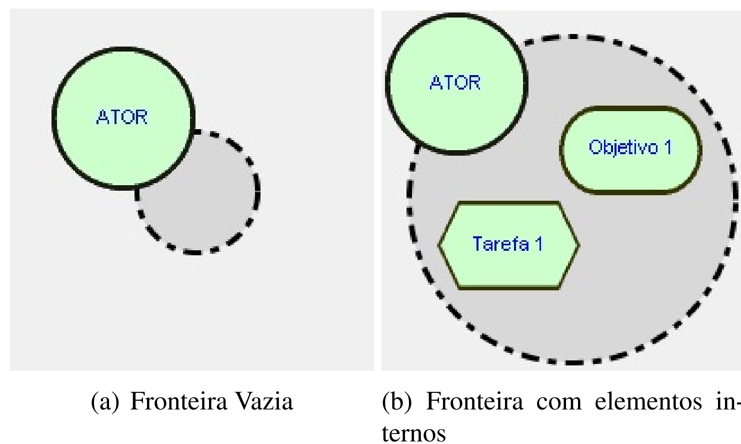
Figura 2.5: Exemplos de ligação de dependência.

### 2.1.2 Modelos de Razões Estratégicas (SR)

O modelo SR representa os detalhes das razões internas que estão por trás das dependências entre atores [29]. Visa retratar os interesses, preocupações e motivações específicas de um ator. Os elementos desse refinamento de motivações internas são agrupados e envolvidos por um limite espacial conhecido como *fronteira* do ator.

A seguir, são descritos os elementos de um modelo SR.

**Fronteira**: Uma fronteira indica os limites intencionais de um determinado ator. Todos os elementos dentro dos limites de um ator, são explicitamente desejos ou pretenções desse ator. Na ferramenta OME [45], uma fronteira é representada por um círculo tracejado e o elemento do ator fica posicionado acima do tracejado, conforme apresentado na Figura 2.6.




(a) Fronteira Vazia (b) Fronteira com elementos internos

Figura 2.6: Exemplos de fronteira do ator.

**Ligação de meio-fim (*means-end*)**: É representada graficamente por uma seta direcionada ao nó fim, significando o meio (objetivo, recurso, *softgoal*, ou uma tarefa) para atingir um fim (objetivo). A Figura 2.7 exemplifica este tipo de ligação.

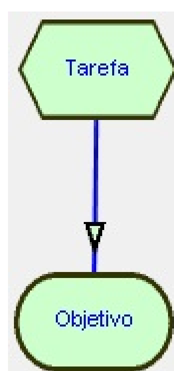



Figura 2.7: Exemplo de ligação meio-fim.

**Ligação de decomposição de tarefa (*task-decomposition*)**: É responsável por detalhar uma determinada tarefa, através da decomposição em sub-elementos ligados a tarefa principal. É representado por um segmento de reta sobreposto perpendicularmente ao segmento de ligação, conforme ilustrado na figura 2.8. Os sub-elementos podem ser: metas, tarefas, recursos e objetivos-soft.

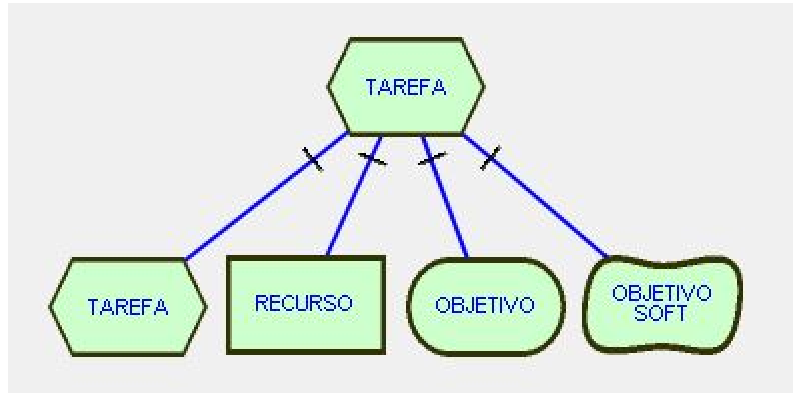



Figura 2.8: Exemplo de ligação de decomposição de tarefa.

**Ligações de Contribuição (*contribution*)**: As ligações de contribuição são para ligar elementos à exclusivamente um objetivo-soft (*softgoal*). Essa ligação ajuda a modelar a forma como os elementos contribuem para a satisfação desse objetivo-soft (*softgoal*). Essas ligações de contribuição [1], ilustradas na Figura 2.9, podem ser:

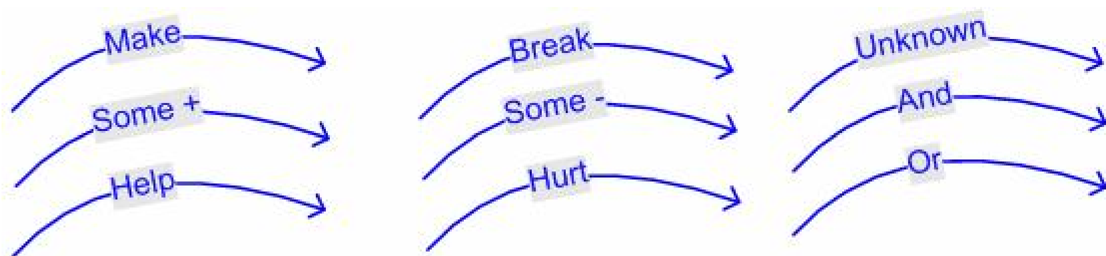



Figura 2.9: Ligações de contribuição.

- ***Make***: é uma contribuição positiva, suficientemente forte para satisfazer o objetivo-soft;
- ***Some +***: é uma contribuição positiva, mas cuja força de influência é desconhecida. Pode equivaler a um *make* ou a um *help*;
- ***Help***: é uma contribuição positiva fraca, pois não é suficiente para que ela sozinha satisfaça o objetivo-soft;
- ***Unknown***: é uma contribuição cuja influência é desconhecida.
- ***Hurt***: é uma contribuição negativa fraca, porém não é suficiente para que ela sozinha recuse a satisfação de um objetivo-soft;

- ***Some -***: é uma contribuição negativa, mas a força de sua influência é desconhecida. Pode equivaler a um *hurt* ou a um *break*;
- ***Break***: é uma contribuição negativa, suficientemente forte para rejeitar a satisfação do objetivo-soft;
- ***Or***: é uma contribuição onde o objetivo-soft é satisfeito se algum dos descendentes for satisfeitos;
- ***And***: é uma contribuição onde o objetivo-soft é satisfeito se todos os descendentes forem satisfeitos;

Cabe ressaltar que os elementos descritos nesta seção sobre o *framework* i*, consideram a proposta original deste framework apresentada em [29].

## 2.2 Variações Baseadas no Framework i*

O processo de modelagem i* é um dos mais conhecidos entre as metodologias de modelagem orientada à agentes.

Várias linguagens já foram propostas para a construção de modelos orientados à agentes.

### 2.2.1 i* Wiki

O i* Wiki, é um projeto criado com o intuito de reunir trabalhos relativos ao i*, de forma colaborativa [1, 46]. Com isso, a comunidade incentiva a colaboração dos usuários do framework, por meio de *feedback* ou mesmo inserção de conteúdo em site oficial. Além disso, esses usuários podem sugerir alternativas ou extensões sintáticas e semânticas em relação a linguagem utilizada.

Apesar da ampla visão que a comunidade pode ter com os trabalhos divulgados no site, a intenção é fornecer e evoluir uma única versão semântica do i*. Dessa forma, o i* Wiki funciona sobre duas versões do guia para o i* [1]: uma versão estável, servindo de referência para os usuários; outra versão aberta a discussão, acessível aos usuários registrados no site e passível de comentários e sugestões individuais. Além disso, o site reúne um conjunto de Estudos de Casos, Publicações e Eventos relacionados a área de i*.

Entre as principais

### 2.2.2 Tropos

Metodologias de desenvolvimento de software existentes (orientadas à objetos, estruturadas ou outra) normalmente são inspiradas por conceitos da programação, focando no software em si ao invés dos problemas ou necessidades da organização [47]. Tropos é uma metodologia de desenvolvimento de sistemas orientada a agentes que tem como objetivo dar suporte ao processo de engenharia de software focado nas reais necessidades de uma organização [47]. Esta metodologia visa aproximar o sistema de software do ambiente operacional de uma organização. Em resumo, a metodologia Tropos pode ser dividida em quatro fases [47]:

1. Requisitos Iniciais (*Early Requirements*): estudar a organização visando entender seus problemas. Esta fase visa extrair um modelo organizacional que inclui os principais atores, seus respectivos objetivos e suas interdependências.

2. Requisitos Finais (*Late Requirements*): descrever o sistema sob o contexto do ambiente operacional, com as principais funções e características. Nesta etapa, o modelo organizacional da etapa anterior deve ser melhor detalhado.

3. Projeto Arquitetural (*Architectural Design*): define, em termos de subsistemas, fluxo de dados, controle e outras dependências, a arquitetura global do sistema. Ao final desta fase, um modelo de arquitetura deve ser definido.

4. Projeto Detalhado (*Detailed Design*): a estrutura e o comportamento de cada componente arquitetural é definido detalhadamente nesta fase. O resultado desta fase é a geração de modelos bases para a implementação.

### 2.2.3 *Goal-Oriented Requirements Language* (GRL)

A GRL é uma linguagem de apoio à modelagem orientada a agentes e objetivos. Assim como o i*, a GRL foca na modelagem dos relacionamentos estratégicos entre atores e seus objetivos. Pode-se pensar como uma alternativa que concentra recursos das metodologias NFR (*Non-Functional Requirements*) [48], i* e Tropos [49]. Outro ponto interessante é que a GRL é escalável, permitindo trabalhar com diferentes níveis de granularidade, em múltiplos diagramas ou visões de um mesmo modelo.

Além disso, uma combinação da GRL com a *Use Case Map* (UCM) [19] deu origem a *User Requirement Notation* (URN) - um padrão internacional do *International Telecommunication Union* (ITU) para notação de requisitos de usuário [22].

## 2.3 Ferramenta OME

Atualmente, existem várias ferramentas de modelagem para o framework i*. Pode-se encontrar mais de 20 ferramentas referenciadas no site do i* Wiki [1]. Além disso, alguns trabalhos já realizaram comparações sobre ferramentas do framework i* [44, 1].

O OME (*Organization Modelling Environment* ou Ambiente de Modelagem Organizacional) é um editor gráfico de propósito geral para dar suporte à modelagem orientada a objetivo e orientada a agentes. É uma aplicação Java para *desktop* desenvolvida na Universidade de Toronto [45].

A ferramenta possui recursos que auxiliam o usuário no desenvolvimento e manipulação de modelos i* e NFR (*Non-Functional Requirements*) [48]. Em 2004, o desenvolvimento foi parado e seu código foi portado para a plataforma Eclipse [50], dando origem a sua versão em código aberto chamada OpenOME [51].

Apesar do seu desenvolvimento ter sido finalizado, ainda existem usuários da OME. Além disso, a ferramenta possui um manual do usuário online[4] e é de fácil utilização. A maioria dos recursos i*, por exemplo, estão de acordo com [29].

Como exemplo, têm-se na Figura 2.10 um dos modelos de exemplos já contidos na ferramenta OME junto à instalação (arquivo: “Meeting-Schedule.tel”).

[4]http://www.cs.toronto.edu/km/ome/docs/manual/manual.html

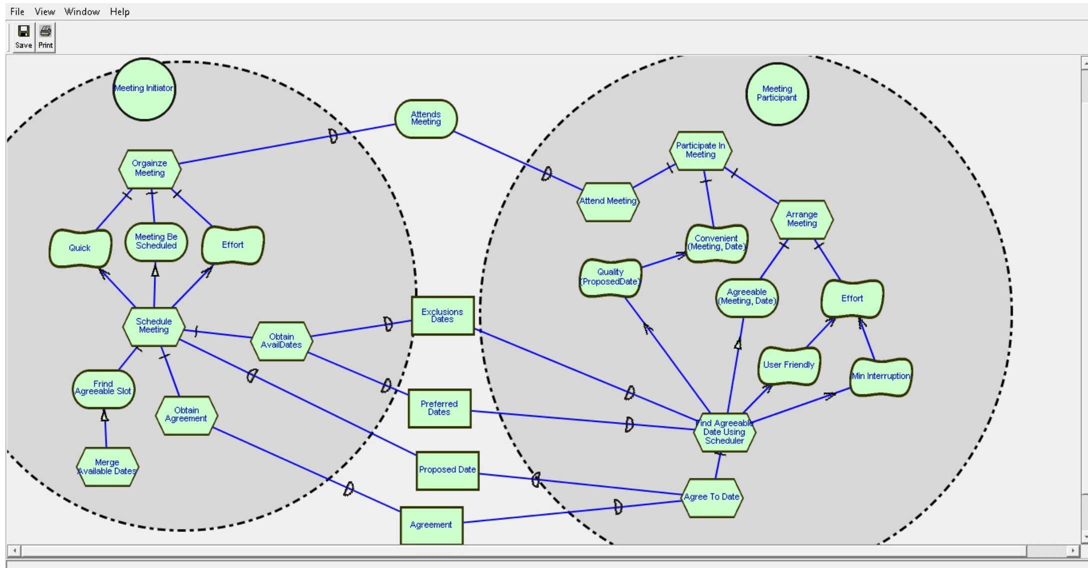



Figura 2.10: Exemplo de modelagem com a ferramenta OME

Cabe ressaltar que a ferramenta OME juntamente com as ferramentas iStarTool [44] e OpenOME [51] foram estudadas para auxiliar no processo de definição das características que o novo editor de i* E4J (descritos no capítulo 4) deveria conter ou suportar.

Vale relembrar que a OME gera os modelos i* no formato TELOS, o único formato atualmente aceito pela ferramenta JGOOSE. Além disso, a ferramenta OME apresenta problemas de estabilidade e seu processo de desenvolvimento foi descontinuado. Acredita-se que a JGOOSE deva adotar uma nova solução para este fim, conforme proposto neste trabalho. Isto daria maior autonomia no uso do JGOOSE permitindo criar os modelos diretamente e facilitaria também na alteração desses modelos, atingindo os objetivos com maior praticidade e rapidez.

## 2.4 iStarML: uma proposta para intercâmbio entre ferramentas para i*

Muitas ferramentas foram criadas com base nos conceitos do framework i* ou de variações desse [52, 53]. A maioria dessas ferramentas possui estruturas e modelos próprios, dificultando o intercâmbio de modelos entre essas ferramentas. Essa incompatibilidade entre ferramentas dificulta o compartilhamento de modelos e bases de conhecimentos comuns.

Pensando nisso, foi desenvolvida a iStarML, um metamodelo baseado em XML (*Extensible Markup Language*) usado para representar modelos i* [54]. O principal objetivo desse meta-

modelo é proporcionar um formato de intercâmbio entre os outros formatos de modelos do i*. É uma especificação de linguagem que suporta todas as outras definições e especificações das variações de modelos i* já propostos. Como exemplo, tudo o que for possível de especificar no formato TELOS é possível também no formato iStarML. Em um endereço na internet[5] pode ser encontrada uma aplicação *Java Applet* que converte do formato TELOS para o formato iStarML.

Como prova de conceitos, foi realizado um estudo onde se utilizou a iStarML para realizar o intercâmbio entre duas ferramentas diferentes [28] - jUCMNav [55] e HiME [56]. Nesse estudo foi realizado um mapeamento entre os modelos e as transformações necessárias para esse mapeamento. Foram identificados os casos que geraram conflitos e detalhou-se a solução adotada.

Existe ainda a preocupação sobre a real adoção da especificação iStarML. Os desenvolvedores da ferramenta OpenOME relataram estar trabalhando para implementar rotinas de importação e exportação em iStarML [51, 57].

Conforme já mencionado no capítulo 1, faz parte da proposta da presente pesquisa, a implementação e adoção do iStarML como especificação do formato de arquivo padrão. Detalhes mais específicos sobre a linguagem iStarML serão descritos no capítulo 4.

## 2.5 Considerações Finais do Capítulo

Neste capítulo foram apresentados os conceitos gerais do framework i* tradicional [29]. A importância do estudo desses conceitos é justificável pelo fato da incorporação com o JGOOSE - ferramenta que está fundamentada em diretrizes propostas com base no i* tradicional [12].

Além disso, a GRL e Tropos, variações do i*, foram brevemente estudadas, visando em trabalhos futuros, suportar outros modelos, além do i* tradicional. Essas variações auxiliaram na definição de estruturas para suportar a compatibilidade e intercomunicação entre os metamodelos dessas metodologias. Neste sentido, buscando permitir a interoperabilidade entre as várias ferramentas e metamodelos, cabe destacar a importância da linguagem iStarML, uma solução de intercâmbio entre diferentes formatos e ferramentas. A linguagem iStarML é adotada pelo editor proposto e os detalhes serão discutidos no capítulo 4.

[5]http://www.essi.upc.edu/ ccares/index.php?section=ometranslator

# Capítulo 3

# JGOOSE

Neste capítulo, é apresentado o estudo realizado sobre a ferramenta JGOOSE (*Java Goal into Object Oriented Standard Extension*) com foco na investigação sobre as características da ferramenta bem como nas características que o novo editor E4J, proposto neste trabalho, deverá conter.

Inicialmente, na seção 3.1, são apresentados os principais conceitos, objetivos e as diretrizes que norteiam os processos de mapeamento de modelos organizacionais i* para casos de uso UML suportados pela ferramenta JGOOSE. Ainda nessa seção, também é apresentado um resumo histórico das versões ao longo dos anos e as principais contribuições de outros autores. Em seguida, na seção 3.3, é analisada a organização arquitetural da ferramenta JGOOSE. Para exemplificar o uso da ferramenta JGOOSE, na seção 3.2 serão apresentados os modelos SD e SR elaborados para expressar as intencionalidades dos interessados no uso da ferramenta E4J, proposta deste trabalho, e a aplicação do JGOOSE para gerar os Casos de Uso UML. Finalmente, na seção 3.4 são realizadas as considerações finais do capítulo.

## 3.1 Visão Geral

A JGOOSE é uma ferramenta de auxílio no mapeamento de modelos organizacionais para modelos funcionais [13]. Essa ferramenta implementa seus processos guiados pelas diretrizes propostas por Santander [12] e é com base nessas diretrizes que a ferramenta interpreta os modelos organizacionais do framework i* e gera os casos de uso UML, apresentando-os no *template* proposto por Cockburn [16]. Com essa ferramenta, é possível derivar casos de uso com base nas intencionalidades associadas aos atores de um ambiente organizacional.

Na subseção a seguir, apresentaremos as diretrizes e passos da proposta de Santander [12], para posteriormente compreender melhor o funcionamento da ferramenta JGOOSE. Na subseção seguinte (3.1.2), será apresentado um resumo histórico sobre as principais mudanças já ocorridas no projeto JGOOSE.

### 3.1.1 Diretrizes e Passos

O conjunto de diretrizes e passos propostos por Santander [12] são a essência dos processos realizados pela ferramenta JGOOSE. É com base nessas diretrizes e passos que a ferramenta realiza o mapeamento de modelos organizacionais i* para modelos funcionais de Caso de Uso UML. A seguir, são descritas brevemente essas diretrizes e passos [26]. Cabe ressaltar que a última versão destas diretrizes está disponível em [41]. Um exemplo apresentando o uso destas diretrizes é descrito na seção 3.2.

**Passo 1**: Descobrir os atores do sistema

**Diretriz 1**: Todo ator em i* é um *candidato* a ser mapeado para um ator em caso de uso.

**Diretriz 2**: O ator *candidato* i* deve ser externo ao sistema computacional pretendido. Isto implica em que atores que representam o sistema computacional, ou partes do mesmo, não são candidatos a atores de casos de uso.

**Diretriz 3**: O ator *candidato* i* deve ter pelo menos uma dependência com o sistema computacional pretendido. Caso contrário, esse ator não pode ser mapeado para um ator de caso de uso.

**Diretriz 4**: Atores em i*, relacionados através da associação *ISA* e mapeados individualmente para atores em casos de uso (após aplicação das diretrizes 1, 2 e 3), serão associados no diagrama de casos de uso através do relacionamento do tipo "generalização".

**Passo 2**: Descobrir casos de uso para os atores

**Diretriz 5**: Para cada ator do sistema descoberto no passo 1, devemos analisar todas as dependências entre o sistema pretendido e esse ator, na qual esse ator é o *dependee* da relação de dependência, buscando-se por casos de uso para esse ator.

**Subdiretriz 5.1**: As dependências do tipo *objetivo* podem ser mapeadas diretamente para casos de uso.

**Subdiretriz 5.2**: As dependências do tipo *tarefa* podem ser mapeadas diretamente para casos de uso.

**Subdiretriz 5.3**: As dependências do tipo *recurso* devem ser analisadas com o seguinte questionamento: "por que este recurso é requerido?". Se para esta resposta existir um objetivo, esse objetivo será candidato a ser um caso de uso para este ator.

**Subdiretriz 5.4**: As dependências do tipo *objetivo-soft* normalmente são requisitos não-funcionais associadas ao sistema pretendido. Portanto, um *objetivo-soft* não representa um caso de uso do sistema, e sim um requisito não-funcional de um caso de uso específico ou do sistema como um todo.

**Diretriz 6**: Analisar situações especiais na qual um ator (descoberto no passo 1) possui dependências (como *depender*) em relação ao ator no modelo i* que representa o sistema pretendido ou parte dele (ator $\rightarrow$ *dependum* $\rightarrow$ sistema pretendido).

**Diretriz 7**: Classificar cada caso de uso de acordo com seu tipo de objetivo associado: de negócio, contextual, de usuário ou de subfunção.

**Passo 3**: Descobrir e determinar cenários dos casos de uso.

**Diretriz 8**: Analisar cada ator e seus relacionamentos no modelo SR para extrair informações que possam conduzir à descrição do cenário do caso de uso para o ator. É importante ressaltar que os diagramas SR representam as razões internas associadas aos objetivos do ator. Por isso, devemos considerar os elementos internos que são usados para o ator para conquistar os objetivos e objetivos-soft, realizar as tarefas e obter os recursos. O ator possui a responsabilidade de satisfazer esses elementos e a decomposição em um diagrama SR mostra como o ator irá fazer isso. Normalmente, as dependências associadas ao ator são satisfeitas internamente por meio de dois tipos de relacionamentos usados no SR: meio-fim e decomposição de tarefa. Devemos observar esses relacionamentos a fim de obter os passos dos cenários dos casos de uso. Os subcomponentes em relações de decomposição de tarefa normalmente são mapeados para passos (atividades) do cenário de caso de

uso associado à tarefa. Note que se a tarefa que está sendo decomposta cumpre alguma dependência (com outros atores) previamente mapeada para um caso de uso, os subcomponentes são mapeados para atividades (passos) do cenário principal do caso de uso. Por outro lado, em uma relação meio-fim, os meios representam alternativas para atingir um fim. Este fim pode ser um *objetivo* a ser alcançado, uma *tarefa* a ser realizada, um *recurso* a ser produzido, ou um *objetivo-soft* a ser satisfeito. Se este fim é um *objetivo* ou *tarefa* que cumpre alguma dependência previamente mapeada para um caso de uso, essas alternativas (meios) são descritas como extensões do cenário de um caso de uso («*extend*», mecanismo de estruturação em UML).

Além disso, também podemos associar *objetivos-soft* representados no diagrama SR com casos de uso. Se um subcomponente em uma relação de decomposição de tarefa é um objetivo-soft e a decomposição de tarefa cumpre alguma dependência mapeada para um caso de uso, este objetivo-soft deve ser associado com o caso de uso como um requisito não funcional no cenário primário.

**Diretriz 9**: Cada caso de uso deve ser analisado a fim de refinar e derivar novos casos de uso a partir da observação dos cenários.

**Diretriz 10**: Construir o diagrama de casos de uso utilizando os casos de uso e atores descobertos, bem como os seguintes relacionamentos de caso de uso UML: *include*, *extend* e *generalization*.

### 3.1.2 Resumo Histórico

Desde a sua primeira versão [13], a ferramenta JGOOSE passou por várias melhorias e aprimoramentos. Estas mudanças têm variado desde a refatoração de código fonte (Classes e *Packages* Java) até alterações na interface gráfica do usuário [58]. A seguir, é apresentado um resumo sobre a origem do JGOOSE e as principais alterações já realizadas na ferramenta.

- **GOOSE** - A *Goal into Object Oriented Standard Extension* (GOOSE), foi a ferramenta que antecedeu à JGOOSE. Foi implementada na linguagem *Rational Rose Scripting* por Marcelo Brischke [59] como uma extensão da *Rational Rose*. Além da dependência da

*Rational Rose*, existia também a dependência da ferramenta OME (*(Organization Modelling Environment*) para criar o modelo i* e o arquivo telos.

- **JGOOSE versão 2006** - Desenvolvida por André Abe Vicente [13]. A nova implementação passou a ser na linguagem Java e foi atribuído o nome de JGOOSE (*Java Goal into Object Oriented Standard Extension*). Por ser em uma nova linguagem de programação, todo o projeto teve que ser re-implementado em Java. Entre outros aspectos, a solução continuou dependente da ferramenta OME, contudo não utilizava mais a solução proprietária *Rational Rose*.

- **JGOOSE versão 2011** - Melhorada por Mauro Brischke [58], a nova versão contempla a implementação de três diretrizes faltantes nas versões anteriores: 8, 9 e 10. Também foi fruto desse trabalho a implementação da exportação dos casos de uso no formato XMI [60], melhorando a comunicação com outras ferramentas como a StarUML [61]. Nessa versão, foi implementado soluções que permitiram o usuário da ferramenta realizar um refinamento manual dos casos de uso gerados, bem como visualizar graficamente os casos de uso na forma de imagens estáticas.

- **JGOOSE versão 2013** - Essa versão está em fase de desenvolvimento por Diego Peliser, em seu projeto de iniciação científica, e visa a correção de alguns *bugs* na aplicação das diretrizes e a otimizações de código (algoritmos), bem como melhorias na interface gráfica e na documentação.

## 3.2 Usando a Ferramenta JGOOSE

Nesta seção será demonstrado o uso da ferramenta JGOOSE no desenvolvimento deste projeto, utilizando a mesma para gerar os casos de uso essenciais que devem ser cobertos pelo editor de i* proposto (E4J) neste trabalho. Primeiramente, foram criados os modelos organizacionais SD e SR apresentados nas figuras 3.1 e 3.2, respectivamente.

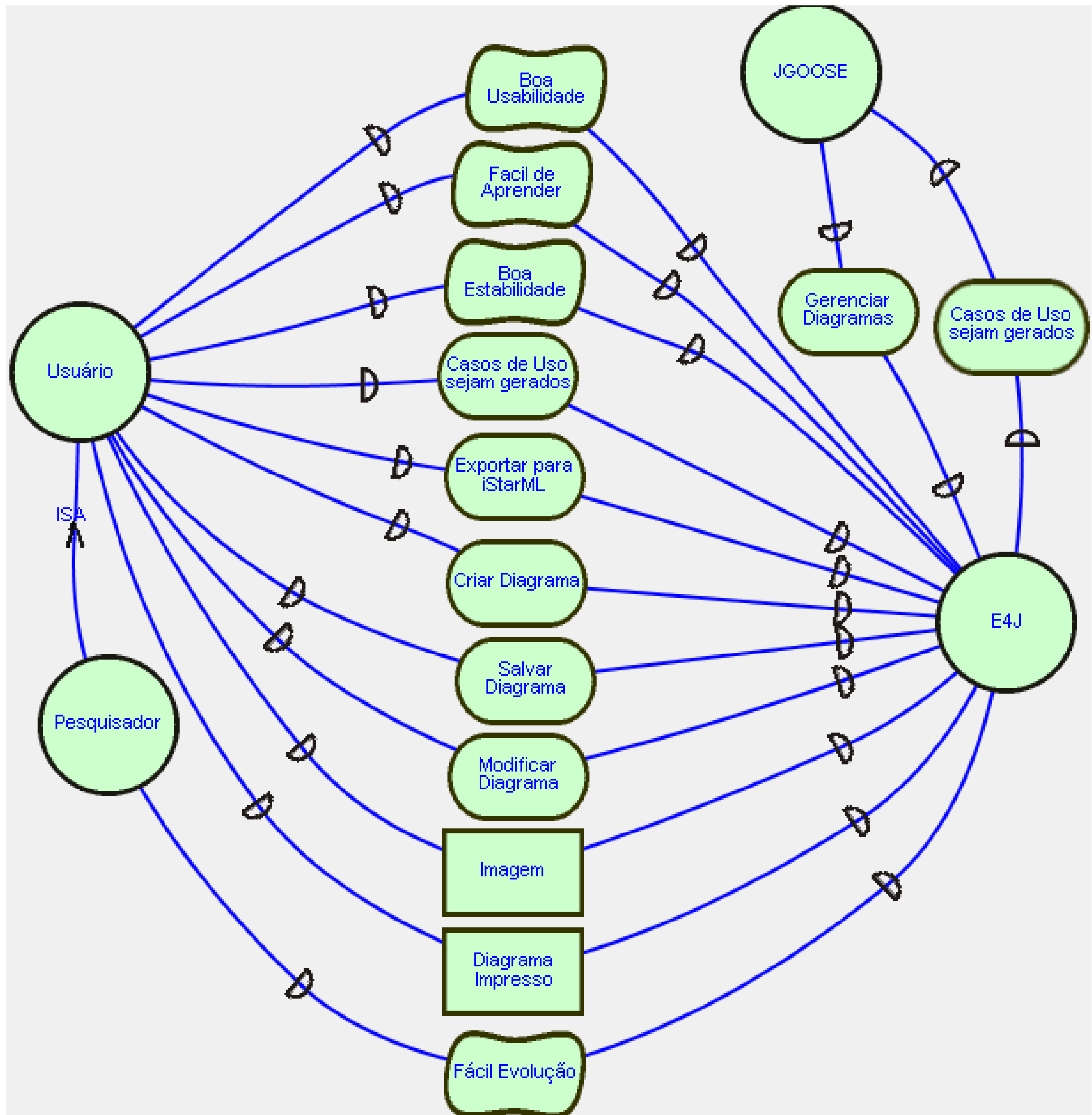



Figura 3.1: Modelo SD da ferramenta E4J.

Considerando as diretrizes propostas em [12] e brevemente descritas na subseção 3.1.1, verificamos que os modelos SD (figura 3.1) e SR (figura 3.2) devem ser construídos para o ambiente envolvendo a utilização do editor E4J. Assim, do modelo SD pode-se obter as seguintes informações sobre os atores (figura 3.1):

- *E4J*: Ator (Sistema) principal desta análise. Depende do JGOOSE para que os *Casos de Uso sejam gerados* (Objetivo).

- *Usuário*: É o ator que fará uso do *E4J*. Segundo o modelo, este ator depende do ator *E4J* para *Criar Diagrama*, *Modificar Diagrama*, *Salvar Diagrama* e *Exportar para iStarML*. Este ator também deseja que: *Casos de Uso sejam gerados*; apresente uma *Boa Usabilidade*; que seja *Fácil de Aprender*; e tenha uma *Boa Estabilidade*. Utilizando o *E4J*, o *Usuário* deseja gerar um arquivo da *Imagem* e obter o *Diagrama Impresso*.

- *Pesquisador*: Também fara uso do E4J com as mesmas intenções do ator *Usuário* (devido a relação *ISA*). Mas, além disso, o ator *Pesquisador* deseja que a ferramenta seja de *Fácil Evolução* (objetivo-soft).

- *JGOOSE*: Após a integração das ferramentas JGOOSE e E4J, o JGOOSE dependerá da E4J para *Gerenciar Diagramas* (Objetivo).

- *Casos de Uso sejam Gerados*: o usuário deseja que o sistema (*E4J*) crie os casos de uso automaticamente, não importando ao usuário "como" a ferramenta E4J irá satisfazer esse objetivo. Este objetivo será mapeado para um caso de uso conforme as diretrizes 5 e 5.1.

- *Criar Diagrama*: o ator *Usuário* deseja *Criar Diagrama* (objetivo) utilizando o sistema (*E4J*). Este objetivo será mapeado para um caso de uso conforme as diretrizes 5 e 5.1.

- *Modificar Diagrama*: o ator *Usuário* deseja *Modificar Diagrama* (objetivo) utilizando o sistema (*E4J*). Este objetivo será mapeado para um caso de uso conforme as diretrizes 5 e 5.1.

- *Salvar Diagrama*: o ator *Usuário* deseja *Salvar Diagrama* (objetivo) utilizando o sistema (*E4J*). Este objetivo será mapeado para um caso de uso conforme as diretrizes 5 e 5.1.

- *Exportar para iStarML*: o ator *Usuário* deseja *Exportar para iStarML* (objetivo) utilizando o sistema (*E4J*). Este objetivo será mapeado para um caso de uso conforme as diretrizes 5 e 5.1.

- *Boa Usabilidade*: o ator *Usuário* espera que seja um programa que facilite algumas tarefas como: copiar e colar, arrastar e soltar, etc. Este *objetivo-soft* será mapeado para um requisito não-funcional conforme as diretrizes 5 e 5.4.

- *Fácil de Aprender*: o ator *Usuário* espera que tenha ícones e menus intuitivos. Este *objetivo-soft* será mapeado para um requisito não-funcional conforme as diretrizes 5 e 5.4.

- *Boa Estabilidade*: o ator *Usuário* espera que o programa não "trave" enquanto uma modelagem é realizada. Este *objetivo-soft* será mapeado para um requisito não-funcional conforme as diretrizes 5 e 5.4.

- *Imagem*: o ator *Usuário* deseja obter uma *Imagem* (recurso) em arquivo do diagrama criado no sistema (*E4J*). Este recurso será mapeado para um caso de uso conforme as diretrizes 5 e 5.3.

- *Diagrama Impresso*: o ator *Usuário* deseja obter um *Diagrama Impresso* (recurso) do diagrama criado no sistema (*E4J*). Este recurso será mapeado para um caso de uso conforme as diretrizes 5 e 5.3.

- *Fácil Evolução*: o ator *Pesquisador* espera que o sistema (*E4J*) seja fácil de atualizar e evoluir. Este *objetivo-soft* será mapeado para um requisito não-funcional conforme as diretrizes 5 e 5.4.

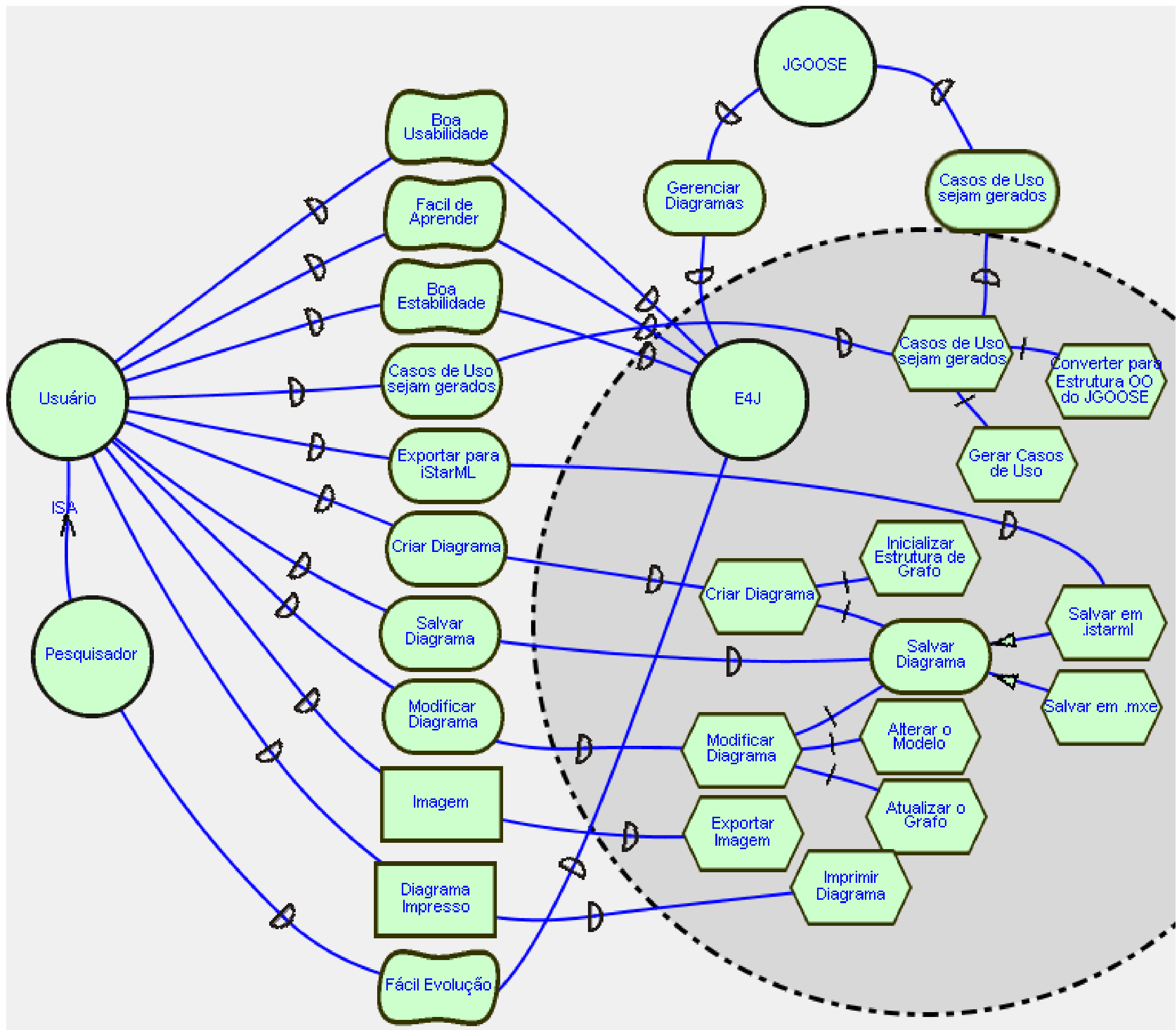



Figura 3.2: Modelo SR da ferramenta E4J.

O modelo SR, apresentado na figura 3.2, contém o detalhamento do sistema (ator *E4J*) e, deste ator, podemos extrair as as seguintes informações:

- *Criar Diagrama*: esta tarefa reflete o objetivo *Criar Diagrama* especificado no SD. Esta tarefa foi decomposta em: *Inicializar Estrutura do Grafo* (tarefa) e *Salvar Diagrama* (objetivo). Conforme as diretrizes 5, 5.1 e 8, os elementos da decomposição são mapeados para o cenário principal do caso de uso gerado.

- *Modificar Diagrama*: esta tarefa reflete o objetivo *Modificar Diagrama* especificado no SD. Esta tarefa foi decomposta em: *Atualizar o Grafo* (tarefa), *Alterar o modelo* (tarefa)

e *Salvar Diagrama* (objetivo). Conforme as diretrizes 5, 5.1 e 8, os elementos da decomposição são mapeados para o cenário principal do caso de uso gerado.

- *Exportar Imagem*: esta tarefa é uma rotina para entregar, na forma de arquivo, a *Imagem* (recurso) do diagrama criado.

- *Imprimir Diagrama*: esta tarefa é uma rotina para executar a impressão do diagrama a fim de entregar ao usuário o *Diagrama Impresso* (recurso).

- *Salvar Diagrama*: este objetivo reflete o objetivo *Salvar Diagrama* especificado no SD. Este objetivo pode ser atingido realizando uma das duas tarefas: *Salvar em .istarml* ou *Salvar em .mxe*. Estas duas opções resultam em dois cenários, um principal (*Salvar em .mxe*) e outro secundário (*Salvar em .istarml*), conforme descrito na diretriz 8.

- *Casos de Uso sejam gerados*: esta tarefa reflete o objetivo *Casos de Uso sejam gerados* especificado no SD. Esta tarefa foi decomposta em: *Gerar Casos de Uso* (tarefa) e *Converter para Estrutura OO do JGOOSE* (tarefa). Conforme as diretrizes 5, 5.1 e 8, os elementos da decomposição são mapeados para o cenário principal do caso de uso gerado.

Cabe ressaltar a importância de um objetivo (*Salvar Diagrama*) do ator E4J ser comum à outras tarefas (*Criar Diagrama* e *Editar e Modificar Diagrama*). Isso impacta na geração de um "«include»" no diagrama de casos de uso (conforme as figuras abaixo). Os casos de uso gerados pela ferramenta JGOOSE, para o ator Usuário, são apresentados na figura 3.3. Foram exportados os casos de uso de todos os atores para arquivos XMI. Esses arquivos foram importados na ferramenta StarUML e o resultado é apresentado na figura 3.4.

Figura 3.3: Casos de Uso gerados pela ferramenta JGOOSE.

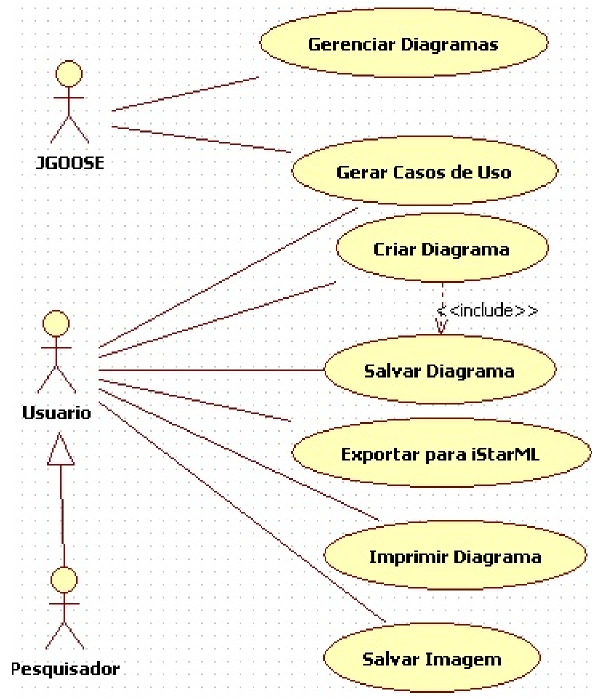



Figura 3.4: Casos de Uso importados na ferramenta StarUML.

## 3.3 Projeto e Arquitetura

Conforme o resumo histórico, a JGOOSE sofreu várias alterações ao longo dos anos. Entretanto, nenhuma dessas alterações influenciaram de forma significativa as etapas de processamentos tradicionais da ferramenta, mantendo como base as diretrizes e passos propostos por Santander [12]. A figura 3.5 foi apresentada por Brischke [58] como sendo a representação da arquitetura da ferramenta JGOOSE.

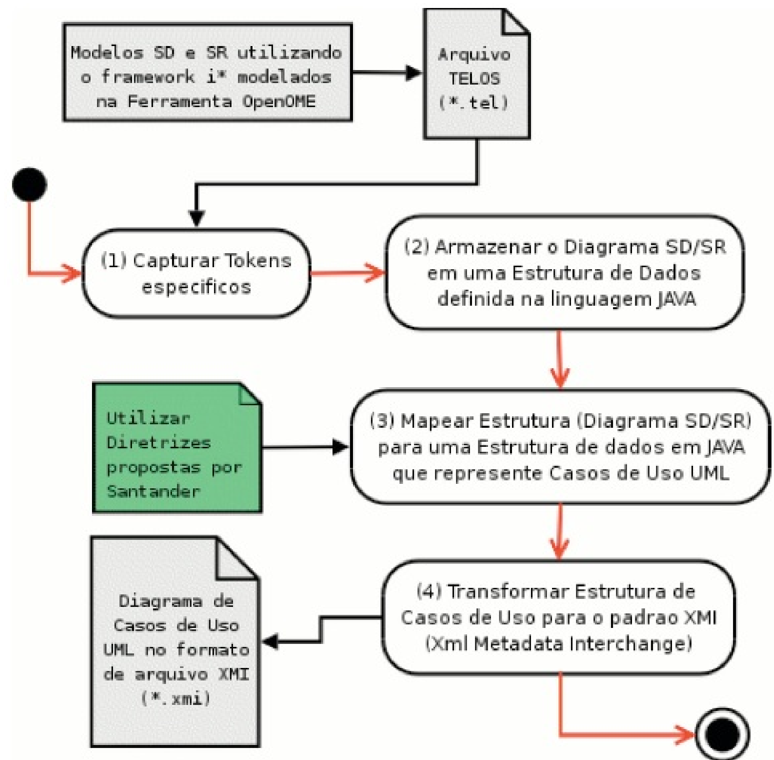



Figura 3.5: Arquitetura da ferramenta JGOOSE demonstrando o seu funcionamento.

A figura 3.5 foi analisada e refeita sobre os conceitos de diagramas de atividades UML [62], pois além de não ser uma arquitetura e sim um fluxo, o novo diagrama de atividades ajudará a entender melhor o impacto do E4J sobre o JGOOSE. Esse tipo de diagrama UML tem como foco principal a representação gráfica de modelos que coordenam as sequências e condições de comportamentos de um sistema [63]. Também são chamados de fluxo de controle ou de modelo de fluxo de objetos. É uma abstração em alto nível que permite diferentes fluxos de execuções e controles simultâneos, assim como é possível realizar o sincronismo desses fluxos e garantir que as atividades executarão em uma ordem específica [62].

Antes de apresentar o diagrama de atividades do JGOOSE, precisa-se entender a notação básica e conhecer os elementos usados nesse tipo de diagrama. Conforme a figura 3.6, os elementos presentes em diagramas de atividades UML são os seguintes:

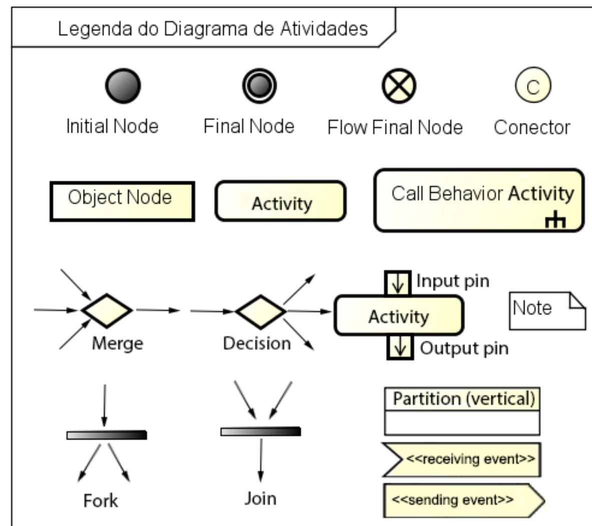



Figura 3.6: Elementos do Diagrama de Atividades UML.

- **Início (*Initial*)**: Um círculo preenchido indica o ponto inicial de um fluxo de atividades. Não é um elemento obrigatório, porém usá-lo torna mais fácil a leitura do diagrama.

- **Fim (*Final*)**: Um círculo preenchido com uma borda indica que a atividade terminou. Um diagrama de atividades pode ter zero ou mais elementos "fim". Se existirem *ações* ou fluxos paralelos em andamento, estes serão interrompidos e a atividade inteira é declarada terminada.

- **Fim de Fluxo (*Flow Final*)**: É um círculo com um "X" no meio indicando que o fluxo terminou. Se existirem fluxos paralelos, apenas os fluxos que chegarem neste elemento são finalizados.

- **Atividade (*Activity*)**: Retângulos com cantos arredondados representam atividades, conjunto de tarefas ou ações que são realizadas. Pode concentrar um conjunto de ações, como exemplo: acessar dados, transformar dados e testar dados.

- **Objeto (*Object*)**: Retângulos com cantos retos representam objetos ou uma instanciação de alguma classe específica, possivelmente em um estado particular.

- **Fluxo (*Flow*)**: São setas ou arestas no diagrama. Dão sentido, seguimento ou transições entre os elementos. Essas arestas podem conter objetos ou dados passando por elas.

- **Bifurcação (*Fork* ou *Split*)**: É uma barra preta com um único *fluxo* “entrando” e mais de um *fluxo* “saindo”. Divide o fluxo em múltiplos fluxos concorrentes. Isso significa o início de fluxos paralelos.

- **Junção (*Join*)**: É uma barra preta com mais de um *fluxo* “entrando” e um único *fluxo* “saindo”. Ao contrário da *bifurcação*, significa o fim de fluxos paralelos. Todos os fluxos devem chegar a junção antes que o processamento continue. É o sincronismo de fluxos paralelos.

- **Decisão (*Decision*)**: É um losango em que existe um único *fluxo* “entrando” e mais de um *fluxo* “saindo”. A diferença da *bifurcação* é que os fluxos que saem devem ter o texto de *condição*, a menos que a *condição* seja trivial.

- **Fusão (*Merge*)**: É um losango em que existe mais de um *fluxo* “entrando” e um único *fluxo* “saindo”. Diferentemente de sincronizar múltiplos fluxos, o *merge* aceita um dos possíveis fluxos. A diferença é que pelo menos um dos fluxos deve chegar ao losango para que o processamento continue.

- **Partição (*Partition*)**: É uma região de domínio indicando “quem” ou “o que” está executando as atividades.

- **Chamada ao Comportamento da Atividade (*Call Behavior Activity*)**: É representada como uma *atividade*, com um ícone no canto inferior direito, significando que essa atividade pode ser descrita mais detalhadamente em outro diagrama de atividade.

- **Nota (*Note*)**: São descrições textuais que podem estar relacionadas com algum elemento do diagrama ou não.

- **Pin**: São elementos de entrada (*InputPin*) e saída (*OutputPin*) para as atividades.

- **Aceitação de Eventos (*receiving events*)**: É uma ação que espera a ocorrência de um evento específico.

- **Envio de Eventos (*sending events*)**: É uma ação que cria uma instância de sinal e envia aos objetos destinatários.

Estabelecidos os principais conceitos e notações dos elementos de um diagrama de atividades, criamos os diagramas a seguir para melhor expressar alguns comportamentos do JGOOSE (figuras 3.7 e 3.8).

No primeiro diagrama (figura 3.7), em uma visão mais abstrata e simples, mostra-se o fluxo a nível de usuário. Inicia-se o diagrama com uma *condição*, avaliando a existência de um prévio modelo organizacional i* em arquivo Telos. Caso o usuário já possua esse modelo em arquivo telos, o fluxo indica ir direto para as atividades do JGOOSE, indicando que o usuário pode abrir o modelo direto pela JGOOSE.

Nos casos em que o usuário não possua o modelo em arquivo telos, deve-se utilizar separadamente a ferramenta OME3 para desenvolver o modelo organizacional e, assim, criar o arquivo telos necessário para o uso do JGOOSE. Esta é uma dificuldade no uso do JGOOSE, já que o usuário precisa usar de forma independente a ferramenta OME3 para construir os modelos i*.

Por fim, pode-se fechar a ferramenta JGOOSE ou exportar em XMI os casos de uso gerados. Esse XMI pode ser aberto no programa StarUML [61] ou por outras ferramentas que permitem a importação deste tipo de arquivo.

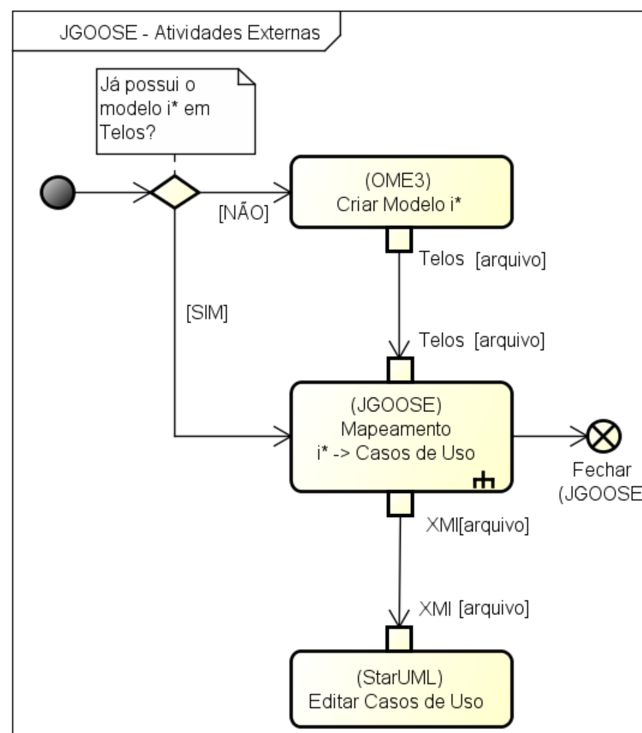



Figura 3.7: Diagrama de atividades em nível contextual da ferramenta JGOOSE.

Para melhor representar as sequências de atividades do JGOOSE, um outro diagrama foi construído e apresentado na figura 3.8.

Inicialmente, tem-se um elemento de decisão para verificar se o usuário já possui um modelo i* no formato telos em um arquivo com extensão ".tel". Caso o usuário não tenha um modelo, deverá ser usado o OME para criar o modelo e gerar o arquivo em telos. Caso contrário, o usuário segue as demais etapas do diagrama:

- **Carregar Arquivo Telos**: o usuário será solicitado para escolher um único arquivo com extensão ".tel". O JGOOSE ira ler este arquivo e interpretar como uma única cadeia de caracteres;

- **Converter para Estrutura OO**: nesta etapa, o JGOOSE ira interpretar a cadeia de caracteres e criará os objetos correspondentes as informações contidas no arquivo;

- **Modelo Organizacional (Estrutura JGOOSE)**: representa a estrutura orientada à objetos do JGOOSE criada para manipulação do modelo organizacional. Mais especificamente, esta estrutura é composta por: uma lista contendo os atores, agentes, posições e papéis; uma lista para cada tipo de relacionamento, por exemplo uma lista chamada "decompositions" armazena todas as relações de decomposição de tarefa de um modelo;

- **Aplicar Diretrizes**: nesta etapa, o JGOOSE aplica as diretrizes com base no modelo organizacional (estrutura anterior) para gerar os casos de uso (próxima estrutura);

- **Casos de Uso (Estrutura JGOOSE)**: representa a estrutura orientada à objetos do JGOOSE criada para manipulação de casos de uso. Basicamente, esta estrutura engloba uma lista de casos de uso, uma lista de atores e uma lista de ligações e extensões;

- **Apresentar Diagrama de Casos de Uso estático**: nesta atividade o JGOOSE mostra ao usuário uma imagem estática dos casos de uso mapeados;

- **Descrição textual dos Casos de Uso**: conforme o template Cockburn [16], os casos de uso são apresentados nesta atividade;

- **Salvar em arquivo XMI**: ação realizada pelo usuário para gravar a estrutura de casos de uso mapeado para um arquivo XMI.

Esse diagrama também apresenta os fluxos paralelos das atividades "Descrição textual dos Casos de Uso", "Salvar em arquivo XMI" e "Apresentar diagrama de Casos de Uso estático".

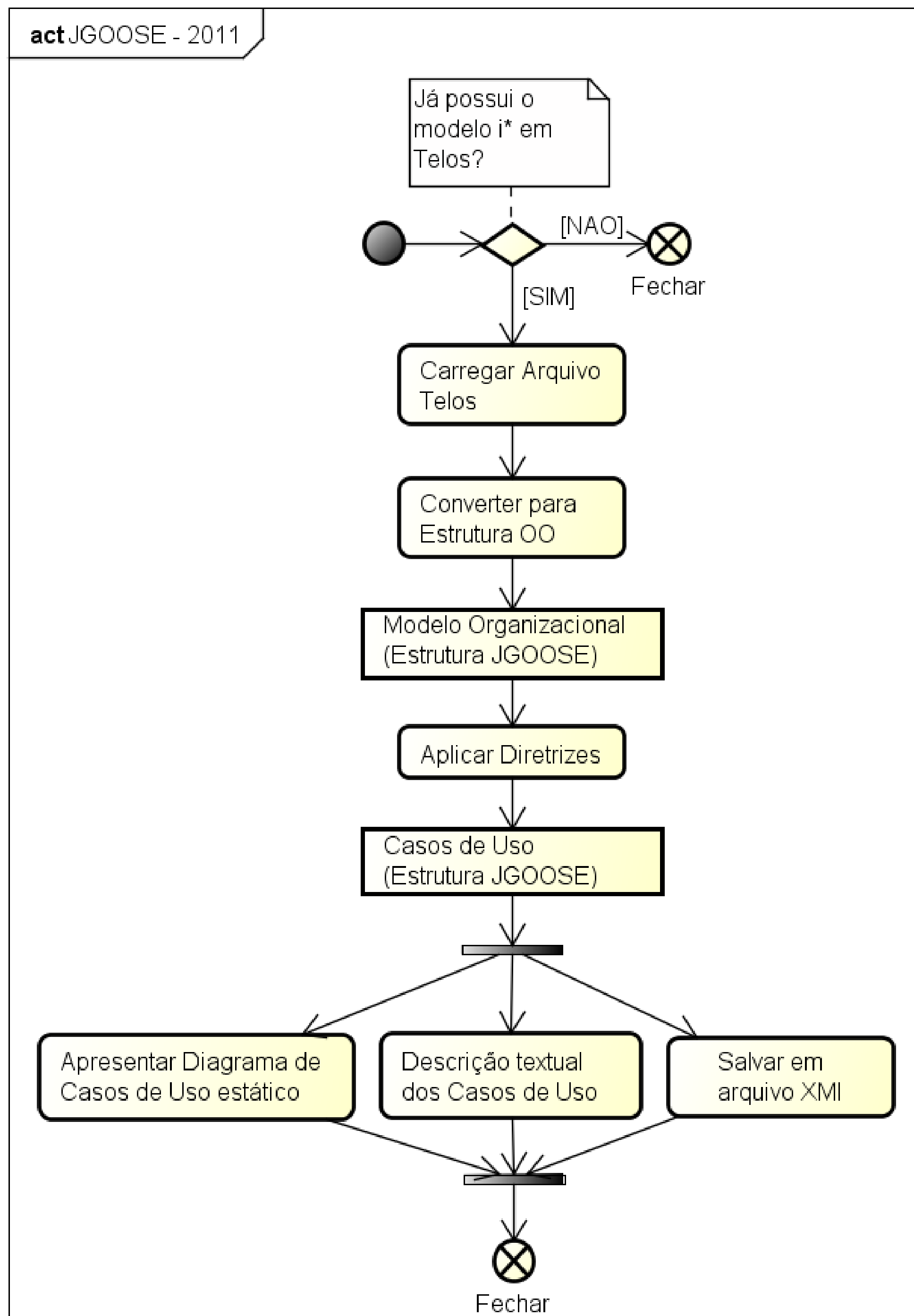



Figura 3.8: Diagrama de Atividades da Ferramenta JGOOSE 2011.

## 3.4 Considerações Finais do Capítulo

Neste capítulo foi apresentada a ferramenta JGOOSE e mostrado o seu funcionamento para um cenário específico. Considerando as atividades que serão de responsabilidade da E4J, é conveniente pensar na incorporação da E4J à JGOOSE, pois o foco principal da ferramenta JGOOSE é a geração dos casos de uso a partir de modelos i*, enquanto o **foco da E4J está no gerenciamento dos diagramas e modelos**. Dessa forma, a JGOOSE passará a cuidar especificamente das rotinas (diretrizes e passos) de mapeamento da estrutura (objetos e classes) i* para a estrutura de casos de uso UML, enquanto a E4J trata dos processos iniciais de manipulação dos diagramas organizacionais.

Em trabalhos futuros, existe a possibilidade de se estudar a viabilidade da alteração de algumas estruturas e rotinas do JGOOSE para usar as mesmas estruturas (objetos e classes) que a E4J. Por enquanto, é responsabilidade do E4J entregar ao JGOOSE o modelo organizacional no formato legado (mais detalhes de como isso é realizado será visto no capítulo 4). Vale ressaltar que a integração **não afetará** a forma com a qual a JGOOSE realiza suas funções de mapeamento. Também cabe ressaltar que as figuras 3.1 e 3.2 da ferramenta E4J, representam uma visão arquitetural de modelagem organizacional da referida ferramenta. Também a figura 3.8 representa outra perspectiva arquitetural representando os casos de uso a serrem cobertos pela ferramenta [64].

# Capítulo 4

# E4J - Editor i* para JGOOSE

Em capítulos anteriores foram apresentados os fundamentos teóricos e a motivação de pesquisas na área de modelagem organizacional. Também foi alvo de estudos a ferramenta e a técnica de geração de casos de uso a partir de modelos organizacionais do *framework* i*. No capítulo 3 foi apresentada a ferramenta JGOOSE e o seu papel fundamental no processo de mapeamento de modelos i* para casos de uso UML. Neste capítulo, é apresentado o projeto e o desenvolvimento do E4J, uma ferramenta de suporte a modelagem gráfica do framework i* para o JGOOSE.

Previamente, na seção 4.1, uma visão geral da proposta contextualiza a ferramenta sob o domínio da modelagem organizacional e o uso de estruturas de grafos para a representação desses modelos. A biblioteca *JGraphX* é apresentada como a solução adotada para o auxílio a manipulação de grafos e a representação dos diagramas. Na seção 4.2 são apresentadas visões arquiteturais na forma de diagrama de pacotes, diagrama de classes e o diagrama de atividades. O modelo organizacional e o diagrama de casos de uso já foram apresentados no capítulo anterior. Em seguida, na seção 4.3 é detalhado o processo de desenvolvimento, mostrando a implementação das principais funcionalidades da ferramenta. Na seção 4.4 é apresentada o impacto do editor E4J sobre o fluxo de atividades do JGOOSE. Finalmente, na seção 4.5 são realizadas as considerações finais do capítulo, apresentando os principais problemas encontrados e as limitações do editor nesta primeira versão.

## 4.1 Visão Geral

O E4J[1] é um ambiente de desenvolvimento de diagramas de modelos organizacionais i*. Ele provê recursos e funcionalidades para criação e manipulação de diagramas SD e SR e a conversão desses diagramas para a estrutura de modelo do JGOOSE. Considerando que o E4J está integrado ao JGOOSE, isso significa que a conversão de estruturas e modelos, do E4J para o JGOOSE, é realizada via rotinas internas, sem a necessidade de geração de arquivos intermediários.

Segundo [29], os diagramas SD e SR são estruturas de grafos com diversos tipos de vértices e ligações que, em conjunto, expressam as razões por trás de processos. Como o JGraphX [65] está embasado na teoria dos grafos, o E4J utiliza os recursos do JGraphX para criar uma estrutura i* e adaptá-la à estrutura do JGraphX.

### 4.1.1 JGraphX

A JGraphX é uma biblioteca Java para visualização de grafos [65]. Consiste em um conjunto de estruturas e funcionalidades que facilitam a produção de Aplicações *Java Swing*. Com essa biblioteca é possível criar aplicações interativas voltadas principalmente para manipulação de diagramas.

O núcleo da biblioteca JGraphX está fundamentada na Teoria dos Grafos [65]. Um grafo consiste em um conjunto de *vértices* e *arestas*. Um vértice pode ser chamado também de *nodo* ou *nó*. As arestas são conexões entre os nós. Na estrutura da JGraphX existe também o conceito de *célula* que representa um elemento do grafo: uma aresta, um vértice ou um grupo destes. Sabe-se que os modelos organizacionais i* podem ser construídos sobre as estruturas de grafos [29] e isso também influenciou na escolha da biblioteca JGraphX.

Além disso, a JGraphX está sob a licença BSD [66] e possui boa representatividade na comunidade, contanto com:

- Um repositório público no GitHub[2] com 38 *Forks* (cópias de outros usuários para novos trabalhos ou melhorias);

[1] E4J é uma abreviação de *Editor 4 JGOOSE* ou *Editor for JGOOSE* ou ainda, em português, “Editor para JGOOSE”.

[2] https://github.com/jgraph/jgraphx.

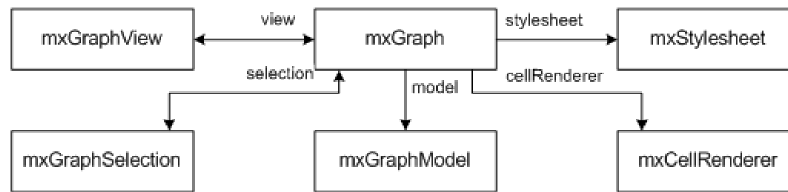



Figura 4.1: Estrutura central do JGraphX.

- Um tópico específico no StackOverflow[3], um fórum de discussão reunindo especialistas em áreas específicas do conhecimento;
- Um antigo fórum[4], com 1602 perguntas e 1822 respostas dos usuários e desenvolvedores;

A estrutura arquitetural da JGraphX, segundo [67], assemelha-se com uma arquitetura MVC (Model-View-Controller) [68]. Conforme a figura 4.1, pode-se observar que a estrutura do grafo (*mxGraph*) possui um *model* (*mxGraphModel*), uma *view* (*mxGraphView*) e os elementos *mxStylesheet*, *mxCellRenderer* e *mxGraphSelection* atuando como o *controller* da arquitetura.

Um *mxGraphModel* é a essência da estrutura de grafo do JGraphX. Conforme a figura 4.2, pode-se observar que o *mxCell* é a estrutura que representa um vértice (*vertex*), uma aresta (*edge*) ou um conjunto destes. E é na propriedade *value* do *mxCell* que podemos armazenar os dados desejados. No caso da presente proposta, ficam armazenadas as informações dos elementos do modelo i* referentes as seguintes propriedades:

- *id*: identificação única do elemento no modelo.
- *title*: título ou rótulo do elemento no modelo, como o nome de um ator ou de um objetivo.
- *type*: o tipo do elemento do modelo. Como exemplo: *actor*, *agent*, *goal*, *dependency* e etc.

## 4.2 Projeto e Arquitetura

A etapa de construção da arquitetura do E4J está contemplada pela seguintes visões: Visão Organizacional, expressando os envolvidos com o sistema; Diagrama de caso de uso, apresentando alguns requisitos do sistema; Diagrama de atividades, mostrando os processos e fluxo de

[3]http://stackoverflow.com/questions/tagged/jgraphx.
[4]http://forum.jgraph.com/

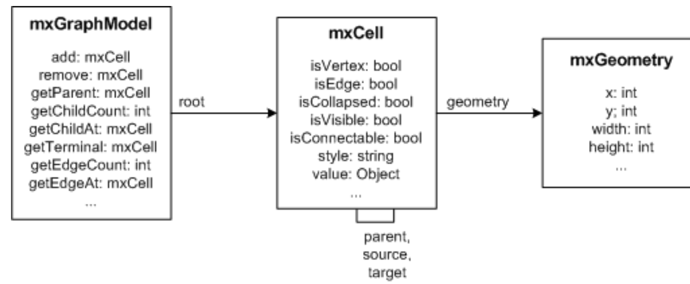



Figura 4.2: Estrutura do modelo do JGraphX.

objetos; Diagrama de pacotes, com uma visão mais estrutural (apresentado abaixo); e Diagrama de classes, apresentando uma visão de implementação;

A visão organizacional, já apresentada na figura 3.2 (seção 3.2 do capítulo 3), ajuda a identificar as principais relações entre os envolvidos com o sistema. Já o diagrama de casos de uso, apresentado na figura 3.3 (seção 3.2 do capítulo 3), permite extrair os principais requisitos funcionais do sistema proposto. O diagrama de pacotes e o diagrama de classes são apresentados em subseções seguintes.

### 4.2.1 Diagrama de Pacotes

Sob o contexto de diagramas arquiteturais, foi desenvolvido um diagrama de pacotes conforme apresentado na figura 4.3. Um pacote é um mecanismo de propósito geral para auxiliar a organização do sistema desenvolvido. O "jgoose-maven" é o pacote principal. Os pacotes internos ao "jgoose-maven" formam as dependências do pacote principal. O pacote "jgoose-maven" possui as seguintes dependências:

- **jgoose**: representa a versão projeto do JGOOSE 2013 refatorado para a estrutura de projetos Maven [69]. A refatoração foi necessária para criar compatibilidade entre os projetos do JGOOSE e o do E4J. Sobre as classes desse pacote, destacamos: a classe *JanelaPrincipal*, que é responsável pela interface gráfica do JGOOSE; a classe *Telos*, que contém a estrutura de dados para representar o modelo organizacional; e a classe *UseCases*, que contém a estrutura dos casos de uso gerados pelo JGOOSE após a aplicação das diretrizes de mapeamento.

- **istarml**: representa a API iStarML e contém um conjunto de estruturas e rotinas para manipulação de arquivos e estruturas no formato iStarML[54]. A classe "jaxb.Model"

representa um conjunto de classes que são mapeadas para aquivos, no formato XML, pelas *annotations* do *framework* JAXB (*Java Architecture for XML Binding*) [70]. A classe "adapter.model" realiza uma agregação da classe "jaxb.Model" para diminuir a possibilidade de inconsistência no modelo. Quando se utiliza apenas a classe "jaxb.Model", o conteúdo das variáveis são todos do tipo *String*, enquanto que ao se utilizar o adapter ("adapter.Model"), os tipos dos atributos passam a referenciar os objetos de fato.

Como exemplo, existe uma classe *ActorTag* (subclasse de "jaxb.Model") que possui o atributo "aref" do tipo *String* e este atributo armazena o *ID* de um outro objeto do *ActorTag*. Porém, nada impede que, por algum motivo, uma nova *String* seja atribuída ao "aref". No entanto, no *adapter* desta classe, este atributo passa a ser do tipo *ActorTag* e passa a armazenar a referência certa para outro objeto *ActorTag*.

- **e4j**: representa toda a estrutura da aplicação do editor E4J. Contém vários elementos de gerenciamento da interface gráfica e de tratamento dos eventos do usuário. Boa parte desses elementos são heranças de classes do pacote *jgraphx*. É neste pacote que ficam também as funções de mapeamento entre as estruturas do E4J para o JGOOSE e a rotina de exportação para a estrutura iStarML, do pacote *istarml*.

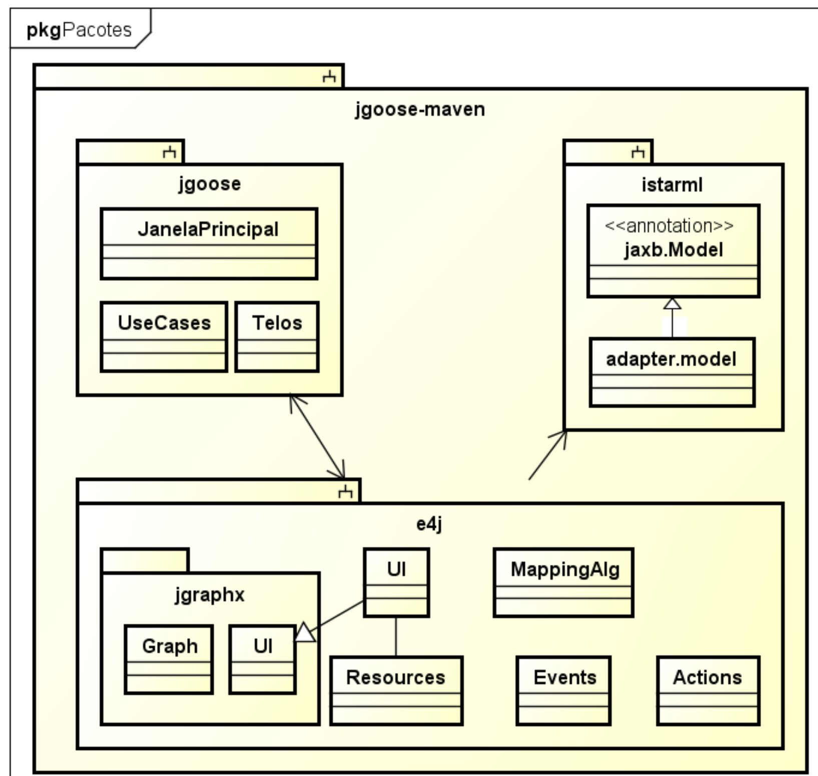



Figura 4.3: Diagrama de Pacotes do projeto E4J.

Cabe destacar que, neste trabalho, cada pacote citado nesta seção é o equivalente a um projeto Maven [69]. Mantido pela *Apache Software Foundation*, o Maven ajuda a gerenciar e organizar melhor o processo de construção de software [71]. O principal motivo pela adoção do Maven foi o gerenciamento de dependências dos pacotes, fazendo com que outros projetos e bibliotecas ficassem separados do projeto principal. Essa separação, dentre outros fatores, auxiliou o versionamento de código fonte, pois somente o que estava sendo desenvolvido era registrado no histórico de versões.

### 4.2.2 Diagrama de Classes

Diagramas de classe são os mais comuns na área de modelagem sistemas orientado à objetos [62]. Esses diagramas mostram um conjunto de classes, interfaces e relacionamentos, representando uma visão mais estática do projeto do sistema. Os diagramas de classe não só ajudam na visualização, especificando e documentando os modelos estruturais, como também servem para a construção de sistemas por meio de técnicas de geração de código automática. Neste trabalho,

foram desenvolvidos os seguintes diagramas de classe: estrutura principal (figura 4.4), conjunto de ações (figura 4.5) e o padrão de projeto mediator [72] (figura 4.6).

Descrevemos a seguir o diagrama de classe da estrutura principal do E4J:

- **EditorPalette**: classe para gerenciamento dos componentes das paletas "elementos" e "conectores". Esta classe trata a seleção dos tipos de conexões (via método "setSelectionEntry") e a funcionalidade de *arrastar e soltar* dos elementos (via "listeners"). O ato de *arrastar e soltar* os elementos reflete na alteração do grafo (*CustomGraph*).

- **EditorKeyboardHandler**: é a classe responsável por tratar os eventos do teclado. É nesta classe que são especificados os atalhos (presentes no apêndice C). Um evento gerenciado por esta classe normalmente reflete na alteração do grafo (*CustomGraph*).

- **AbstractAction**: classe abstrata para prover funcionalidades comuns as outras classes de ações do usuário. As classes que estendem esta abstração são apresentadas na figura 4.5.

- **CustomGraph**: é a classe responsável por gerenciar a estrutura grafo, suas alterações e restrições. É nesta classe que são criados os tratamentos de validação das ligações entre elementos do modelo (método "createEdge").

- **CustomGraphComponent**: é uma classe intermediária entre a visão do usuário (*BasicGraphEditor*) e o modelo em grafo (*CustomGraph*). É responsável por mapear o conteúdo do valor de um elemento do grafo (retornado pelo método "convertValueToString" do modelo) para a visão.

- **BasicGraphEditor**: é a classe principal do projeto. Estende a classe *JFrame* para apresentar os elementos gráficos (visão), além de agregar os elementos acima descritos (modelo e eventos). Esta classe também agrega o gerenciador de histórico ("UndoManager"), tornando possível o uso de "Undo/Redo".

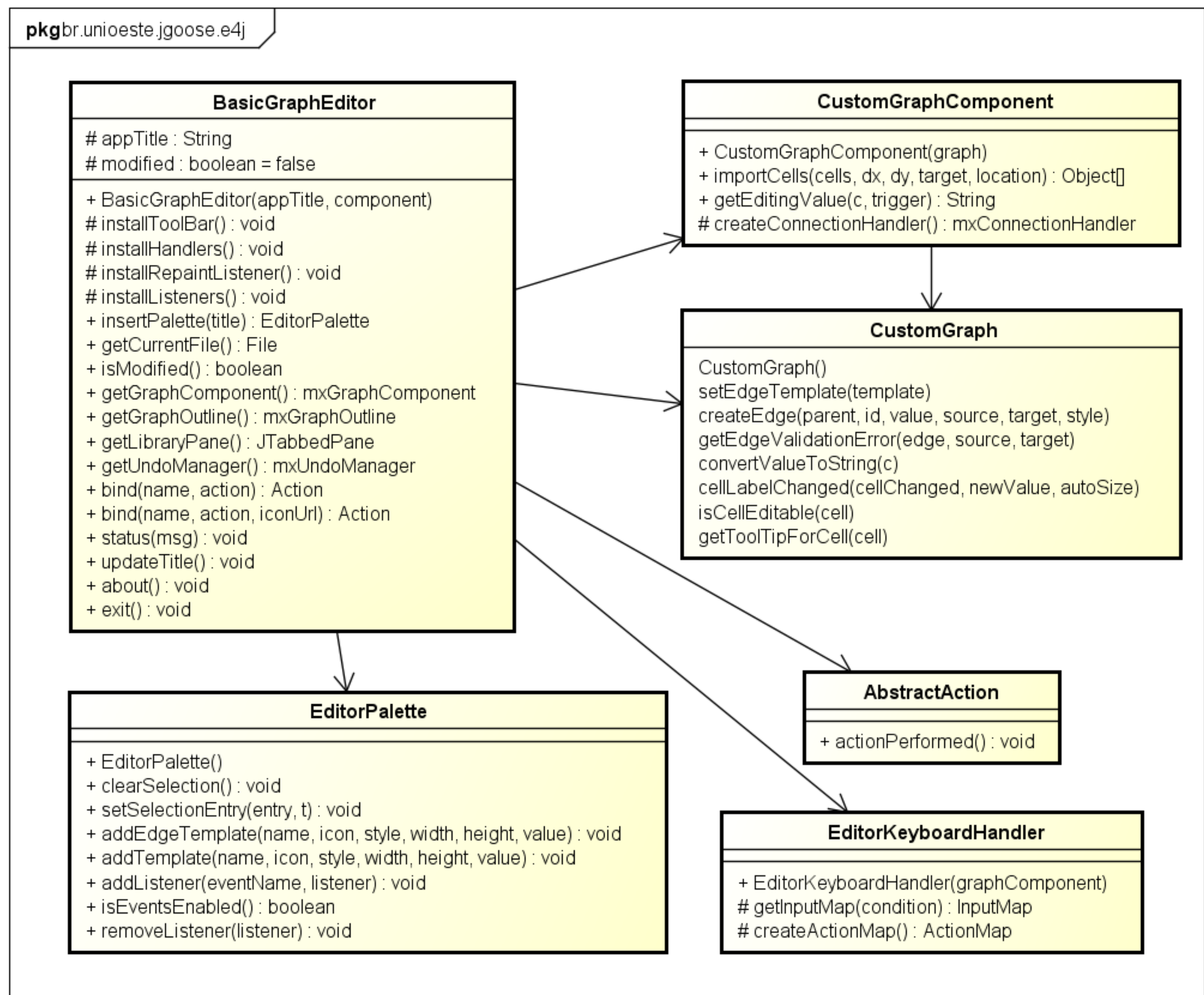



Figura 4.4: Diagrama de Classes parcial do E4J. Estrutura principal.

A seguir, na figura 4.5, temos um diagrama de classes para tratamento e execução das principais ações do usuário. Essas ações estão normalmente acopladas aos itens de menus, menus de contexto ou atalhos do teclado. Observa-se que todas são extensões da classe *AbstractAction*, classe utilizada para implementar ações relativas aos componentes do Java Swing [73].

Figura 4.5: Diagrama de Classes parcial do E4J. Principais ações do usuário.

A figura 4.6 representa uma implementação do padrão de projeto Mediator [72] utilizada para troca de contexto entre o JGOOSE e o E4J. Por exemplo, quando o usuário que está utilizando o E4J deseja gerar os casos de uso, a classe *Mediator* é acionada para tratar a troca de contexto entre as aplicações. O mesmo acontece quando o usuário está utilizando o JGOOSE

e deseja abrir o editor E4J. O papel principal do *mediator*, neste caso, é gerenciar as duas interfaces, alternando a visibilidade entre as elas. Este padrão de projeto foi necessário pois o gerenciador de janelas não permite a visualização simultânea de mais de um *JFrame*.

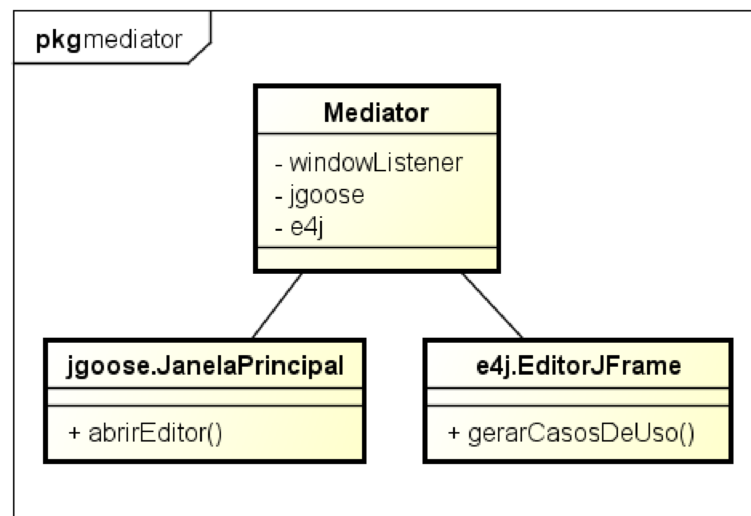



Figura 4.6: Padrão de projeto Mediator adaptado e aplicado entre os projetos E4J e JGOOSE.

## 4.3 Desenvolvimento

O processo de desenvolvimento foi iterativo e incremental, com algumas apresentações e discussões junto ao Grupo LES (Laboratório de Engenharia de Software). Foram realizados encontros com o cliente (professor orientador Victor Francisco Araya Santander) para avaliar as versões após o término de cada iteração, atingindo um processo ágil de *feedback*.

O código fonte foi dividido em vários projetos. Utilizando a especificação de projetos Maven [69] foi possível realizar um desenvolvimento modular conforme o modelo apresentado no diagrama de pacotes (seção 4.2).

### 4.3.1 Definição da Estrutura E4J

Uma das estruturas mais usadas para representação de diagramas de modelos organizacionais é a estrutura em grafos [41]. Nessa estrutura, os vértices podem representar elementos como *atores* ou *dependum*, enquanto que as arestas dos grafos podem representar as ligações entre os elementos do modelo organizacional.

### 4.3.2 Adaptação do JGOOSE

Conforme a figura 4.7, foi adicionado um item de menu à interface gráfica do JGOOSE para realizar a chamada ao E4J. Esse menu realiza a função de carregar a interface gráfica do E4J e apresentá-la ao usuário. Após a execução do evento gerado por esse menu, a linha de execução principal do programa passa a ser de responsabilidade das classes do E4J. Retornando à JGOOSE apenas em situações apresentadas no diagrama de atividades apresentadas no capítulo 3 (figura 4.10).

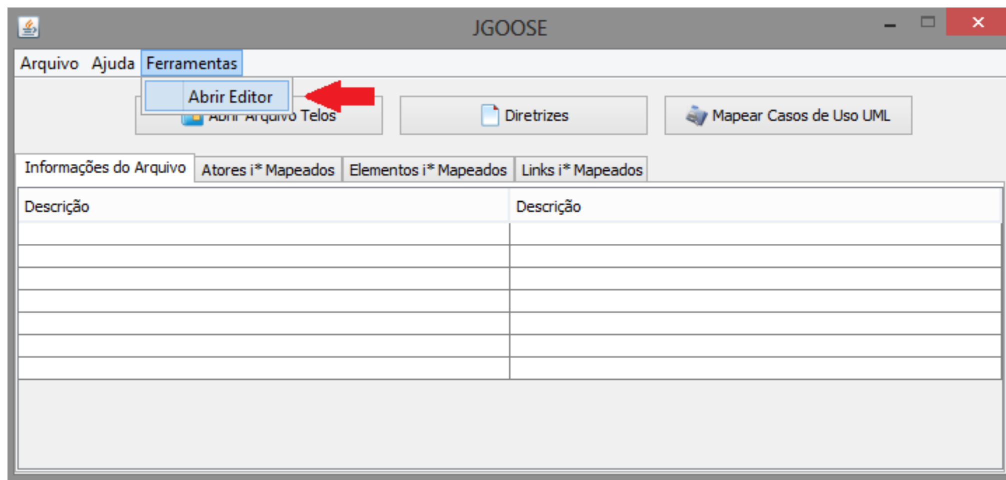



Figura 4.7: Interface Gráfica com menu de chamada ao editor E4J.

Também foi necessário deixar o JGOOSE compatível com o padrão de projeto mediator (já apresentado na figura 4.6). Para isso, bastou tratar os eventos de janelas da classe *br.unioeste.jgoose.view.MainView* implementando a interface *java.awt.event.WindowListener*.

### 4.3.3 Interface Gráfica do Usuário

A interface gráfica do usuário é composta por uma tela principal e é nesta tela que o usuário terá a sua disposição as principais funcionalidades do editor E4J. A figura 4.8 apresenta a tela principal do editor E4J. Esta tela foi planejada visando usabilidade e produtividade. Os principais elementos dessa interface são apresentados no capítulo 5.

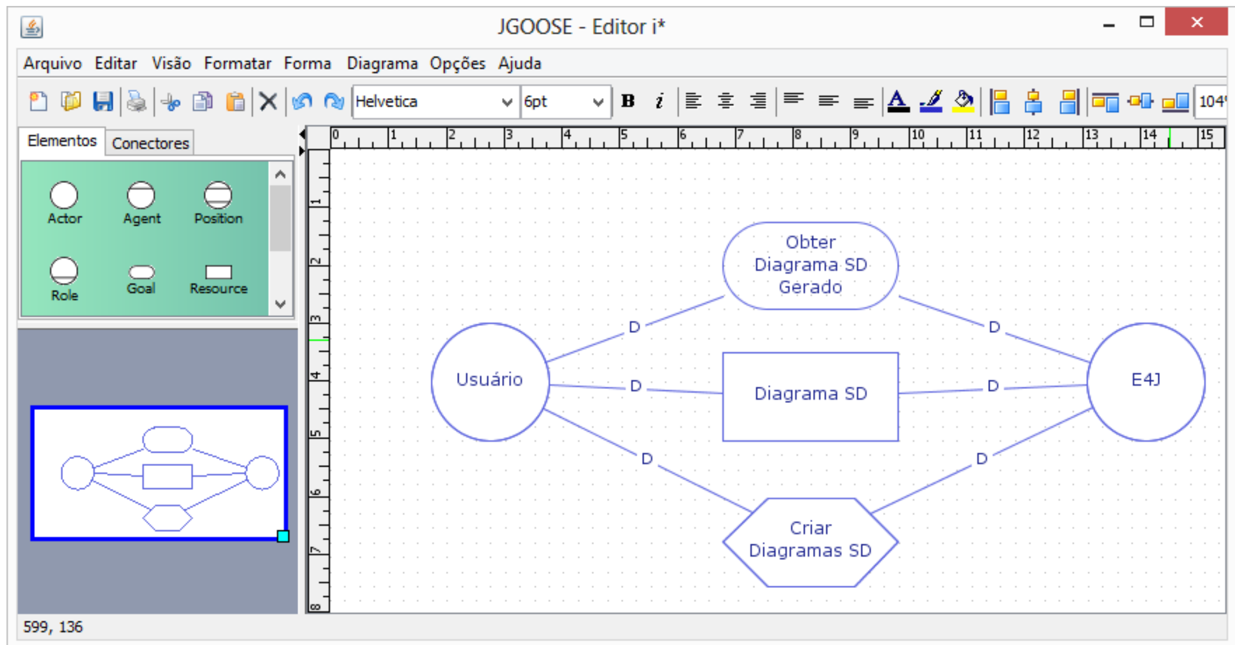



Figura 4.8: Interface Gráfica Principal do E4J.

Para gerenciar as ações do usuário, o E4J conta com 43 arquivos no *package* “br.unioeste.jgoose.e4j.actions” responsáveis por executarem alguma rotina muito específica, como exemplo a rotina de *fechar a aplicação* realizada pela classe “br.unioeste.jgoose.e4j.actions.ExitAction”. Algumas ações executas via interface gráfica podem ser acessadas pelos atalhos de teclado. O apêndice C contém uma lista dessas ações e seus respectivos atalhos.

Em ferramentas com interface gráfica, durante a interação entre o usuário e o E4J, vários eventos pode ser gerados. As principais origens desses eventos são o mouse, teclado e o próprio sistema. Para gerenciar esses eventos, o E4J conta com 8 arquivos no *package* “br.unioeste.jgoose.e4j.swing.listeners”, responsáveis pela captura dos eventos e tratamento dos principais eventos do sistema.

Outros eventos foram desenvolvidos em modo anônimo (ou “inline”), classe criada e instanciada dentro de outra classe ou na passagem de parâmetros, não possuindo um arquivo específico com a sua declaração.

### 4.3.4 Estrutura do Modelo E4J

Os recursos do E4J, em sua maioria, estão implementados com base no JGraphX. Isto inclui a estrutura principal que representa o modelo organizacional SD ou SR. Várias classes do JGraphx foram estendidas assim como as interfaces necessárias foram implementas, sempre seguindo os padrões recomendados pelo manual [67]. Ao se utilizar dessas classes do JGraphX, uma atenção especial deve ser dada a classe *mxCell*. É no atributo *value* desta classe que o conteúdo da estrutura deve ser armazenado. Na implementação deste trabalho, o atributo *value* da classe *mxCell* armazenou uma estrutura *Element* (do *package* "org.w3c.dom") com as seguintes propriedades:

- ***id***: propriedade que identifica única e exclusivamente um elemento no diagrama.
- ***tagname***: dado relativo à equivalência na estrutura da linguagem iStarML. Em iStarML um ator é definido pela *tag* "<actor>", portanto esta propriedade deve possuir o valor "actor", por exemplo.
- ***type***: especifica o tipo do elemento, quando necessário. São exemplos de tipos de elementos: "agent", "role", "goal", "is_a", "plays", etc.
- ***label***: propriedade que armazena o nome atribuído pelo usuário.

A tabela 4.1 apresenta cada elemento da estrutura E4J e suas respectivas propriedades.

Tabela 4.1: Elementos da estrutura E4J e suas propriedades.

| **Elemento** | ***tag*** | ***type*** |
|---:|:---:|:---:|
| Ator | actor | actor |
| Agente | actor | agent |
| Papel | actor | role |
| Posição | actor | position |
| Objetivo | ielement | goal |
| Tarefa | ielement | task |
| Recurso | ielement | resource |
| Objetivo-Soft | ielement | softgoal |
| IS A | actorLink | is_a |
| IS PART OF | actorLink | is_part_of |
| INS | actorLink | instance_of |
| PLAYS | actorLink | plays |
| OCCUPIES | actorLink | occupies |
| COVERS | actorLink | covers |
| Decomposição | ielementLink | decomposition |
| Meio-Fim | ielementLink | means-end |
| Contribuição | ielementLink | contribution |
| Dependência | dependency | - |

### 4.3.5 Linguagem iStarML

A linguagem iStarML [54] foi desenvolvida com o objetivo de promover o intercâmbio entre metamodelos de diferentes ferramentas. Cabe destacar que a proposta dessa linguagem é ser uma auxiliar comum entre as ferramentas e não a linguagem definitiva delas [27]. Já foi demonstrado que esta linguagem pode ser utilizada para representar qualquer metamodelo já desenvolvido na área de modelos i* [53].

A linguagem define um conjunto de *tags* XML que são uma abstração dos conceitos de modelos i* [27]. Além disso, visando suportar funcionalidades adicionais, a linguagem especifica atributos extras para um conjunto de *tags*. A tabela 4.2 é uma adaptação da tabela 1.1 de [27] e apresenta os principais conceitos das linguagens de modelagem baseadas em i* e sua *tag* XML correspondente.

Tabela 4.2: Principais conceitos das linguagens baseadas em i* e a *tag* XML correspondente.

| **Conceito** | **Significado** | ***Tag*** |
|---|---|---|
| Ator | Representa uma entidade que pode ser um elemento da organização, um humano ou um software. Também pode representar abstrações como: *papel* e *posição*. | <actor> |
| Elemento Intencional | Representa uma entidade que permite o relacionamento entre diferentes atores conforme a relação social. Também pode representar uma razão interna de um determinado ator. Normalmente os tipos de elementos intencionais são: objetivo, tarefa, recurso e objetivo-soft. | <ielement> |
| Dependência | É um relacionamento que representa explicitamente a dependência de um ator (*depender*) em relação a outro ator (*dependee*). Uma dependência é expressa por meio de um elemento intencional | <dependency><br><depender><br><dependee> |
| Limite | Representa um grupo de elementos intencionais. O tipo mais comum de limite é o "limite do ator" que agrupa esses elementos com base no escopo do ator. Entretanto, outros tipos de limites também podem ser usados. | <boundary> |
| Ligação de Elementos Intencionais | Uma ligação de elementos intencionais representando um relacionamento entre os elementos intencionais (tanto dentro quanto fora dos limites dos atores). Os exemplos mais comuns dessas ligações são: decomposição, meio-fim e contribuição. Conceitos como "rotinas" ou "capacidades" também pode ser representadas utilizando este relacionamento. | <ielementLink> |
| Ligação de Associação entre Atores | Representa um relacionamento entre dois atores. Os exemplos mais comuns são: are is_a, is_part_of, instance_of (INS), plays, occupies e covers. | <actorLink> |

### 4.3.6 Rotina de Mapeamento

Para transformar a estrutura do E4J em uma estrutura do JGOOSE ou do iStarML foram implementadas duas rotinas de mapeamento. O algoritmo 1 é uma abstração dessas duas rotinas de mapeamento e serve de base para a implementação de outros mapeamentos. É importante analisar a ordem de mapeamento dos elementos do grafo: primeiro mapeia-se todos os vértices e

depois todas as arestas. Essa restrição na ordem existe em virtude de que, em algumas estruturas, só é possível criar arestas após a existência dos vértices de origem e destino. Os métodos "map_vertex" e "map_edge" devem ser implementados conforme os conceitos comuns entre as estruturas de origem e destino.

**Algoritmo 1** Mapeamento de modelos na estrutura E4J

**Parâmetros:** $graph, destiny_structure$
**Saída:** $destiny_structure$ com os elementos mapeados

**para todo** $v$ onde $v$ é um vértice de $graph$ **faça**
  $element \Leftarrow v.value$
  $type \Leftarrow element.type$
  $label \Leftarrow element.label$
  $graphic \Leftarrow element.graphic$
  $destiny_structure.map_vertex(type, label, graphic)$
**fim para**
**para todo** $e$ onde $e$ é uma aresta de $graph$ **faça**
  $element \Leftarrow e.value$
  $type \Leftarrow element.type$
  $label \Leftarrow element.label$
  $graphic \Leftarrow element.graphic$
  $source \Leftarrow element.source$
  $target \Leftarrow element.target$
  $destiny_structure.map_edge(type, label, graphic, source, target)$
**fim para**
**return** $destiny_structure$

Como os conceitos utilizados pelas ferramentas E4J e JGOOSE são equivalentes aos conceitos de Yu'95 [29], a rotina de mapeamento foi, basicamente, encontrar uma classe na estrutura do JGOOSE que fosse equivalente à estrutura do E4J. Para auxiliar e enfatizar esta análise, foram construídas relações de equivalência as quais são apresentadas na tabela 4.3. Esta tabela é um complemento da tabela 4.1 que acrescenta as colunas "Classe JGOOSE" e "Coleção JGOOSE" correspondentes a estrutura mapeada e a coleção dessas estruturas (em *ArrayList*), respectivamente.

Tabela 4.3: Elementos da estrutura E4J e suas propriedades.

| **Elemento no E4J** | ***tag*** | ***type*** | **Classe JGOOSE** | **Coleção JGOOSE** |
|---|---|---|---|---|
| Ator | actor | actor | IStarActorElement | actors |
| Agente | actor | agent | IStarActorElement | actors |
| Papel | actor | role | IStarActorElement | actors |
| Posição | actor | position | IStarActorElement | actors |
| Objetivo | ielement | goal | IStarElement | goals |
| Tarefa | ielement | task | IStarElement | tasks |
| Recurso | ielement | resource | IStarElement | resources |
| Objetivo-Soft | ielement | softgoal | IStarElement | softgoals |
| IS A | actorLink | is_a | IStarLink | isas |
| IS PART OF | actorLink | is_part_of | IStarLink | ispartofs |
| INS | actorLink | instance_of | IStarLink | inss |
| PLAYS | actorLink | plays | IStarLink | playss |
| OCCUPIES | actorLink | occupies | IStarLink | occupiess |
| COVERS | actorLink | covers | IStarLink | coverss |
| Decomposição | ielementLink | decomposition | IStarLink | decompositions |
| Meio-Fim | ielementLink | means-end | IStarLink | meansEnds |
| Contribuição | ielementLink | contribution | IStarLink | contributions |
| Dependência | dependency | - | IStarLink | dependencies |

Nesta tabela, podemos observar que os diferentes tipos de atores (ator, agente, papel e posição) são mapeados para a mesma classe (*IStarActorElement*) e para a mesma coleção (*actors*) do JGOOSE. Isso significa dizer que existe um atributo na classe *IStarActorElement* que permite diferenciar os tipos de atores. Porém, o mesmo não ocorre para as outras duas classes (*IStarElement* e *IStarLink*) que difere os elementos com base na coleção deles. Esta diferença de estruturação dos elementos impactou aumentando número de comparações realizadas no processo de mapeamento de elementos das classes *IStarElement* e *IStarLink*. Uma sugestão sobre a solução foi passada para a versão que está em desenvolvimento: adicionar um atributo nas classes *IStarElement* e *IStarLink* para diferenciar os dados, ao invés de utilizar várias coleções.

## 4.4 Impacto do editor E4J no JGOOSE

Para realizar a integração da E4J com a JGOOSE, foram necessárias algumas modificações no fluxo de atividades da JGOOSE. O impacto é mais especificamente nas condições iniciais do

diagrama de atividades (primeiro elemento de *condição* da figura 4.9). Antes a única opção era abrir um arquivo em telos. Agora, o usuário pode abrir arquivos .mxe ou criar um novo modelo manualmente. Conforme a proposta deste trabalho, que trata da manipulação e criação de diagramas SD e SR, o usuário poderá utilizar os recursos da E4J para criar o modelo organizacional necessário à aplicação das diretrizes.

Após a integração entre as duas ferramentas, continuará sendo possível executar as funcionalidades legadas da JGOOSE, sem a necessidade de uso da E4J, desde que se trabalhe apenas com o formato de arquivo TELOS. Todo o processo de mapeamento do arquivo TELOS para a estrutura O.O. (Orientado à Objeto) da JGOOSE e dessa estrutura O.O. para casos de uso UML continuará sendo realizado pela JGOOSE.

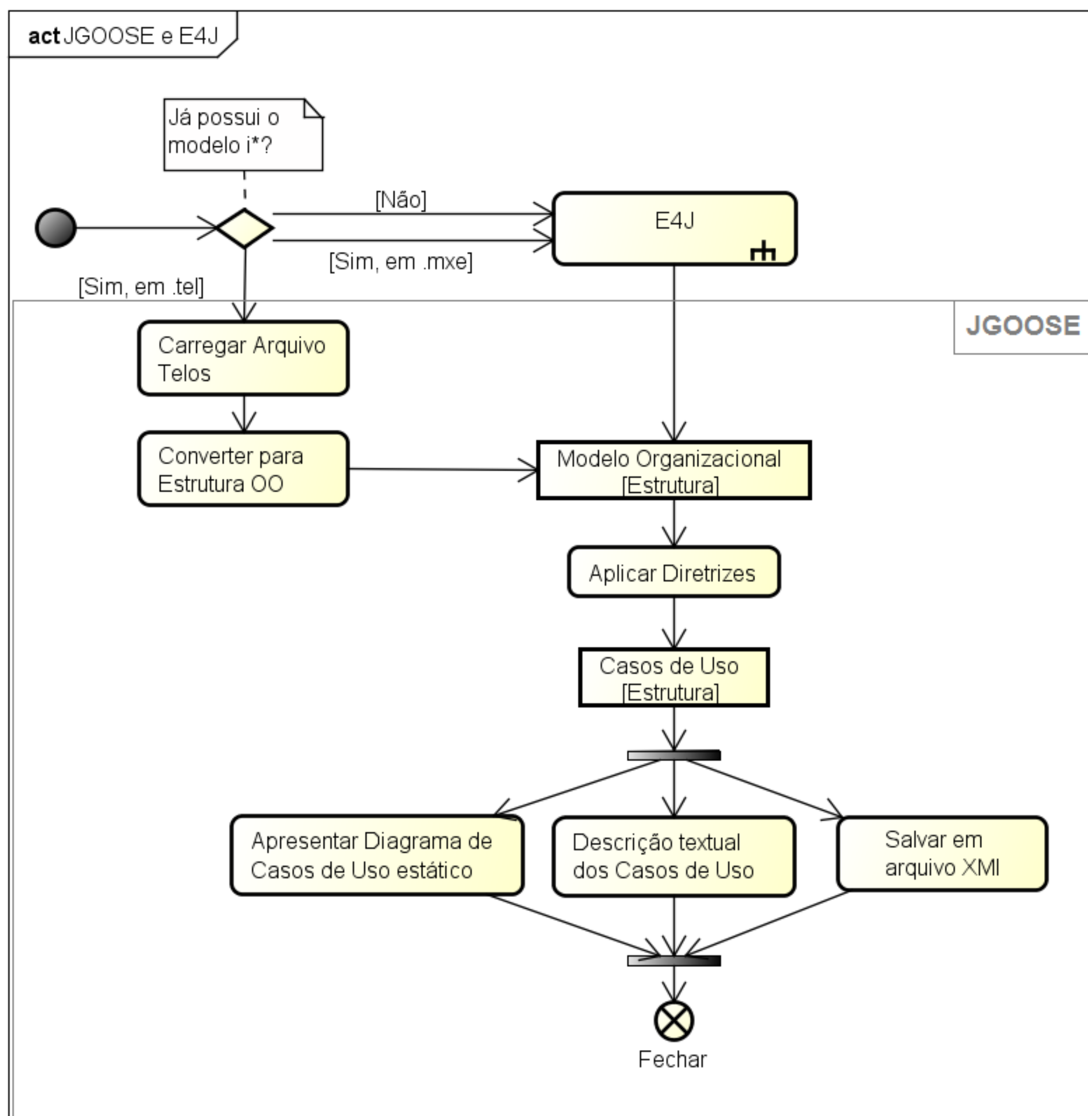



Figura 4.9: Diagrama de atividades da ferramenta JGOOSE após integração com E4J.

Na figura 4.10 são apresentadas as *partições E4J*, *JGraphX* e a *API IStarML* que, em conjunto, coordenam as principais atividades do editor E4J. Iniciando pela partição *E4J*, as configurações e elementos da interface gráfica do usuário são carregados e apresentados. A partir disso, o fluxo passa do *conector* “A” da partição *E4J* para o *conector* “A” da partição *JGraphX*. Em paralelo, são realizadas as rotinas de envio e recebimento de sinais (*«signal sending»*, *«signal receipt»*) ou eventos.

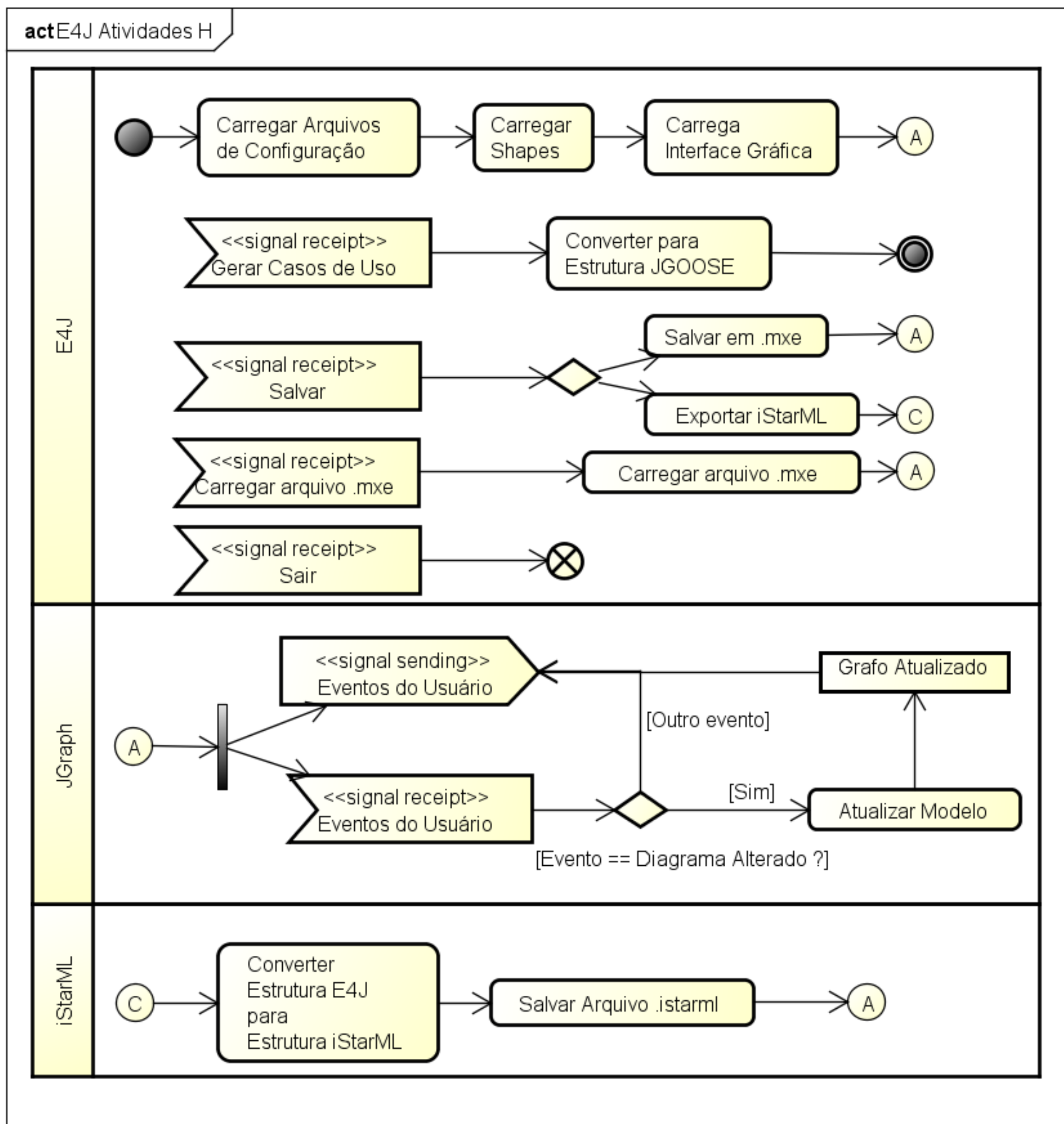



Figura 4.10: Diagrama de atividades da ferramenta E4J.

No diagrama de atividades da ferramenta E4J, apresentada na figura 4.10, são representados os seguintes elementos:

- E4J (Partição): esta partição concentra as principais funcionalidades de responsabilidade do E4J.
  - Nodo inicial: é onde começa a aplicação E4J. É dado início ao carregamento dos recursos necessários para utilização da ferramenta.

– Carregar Arquivos de Configuração: nesta ação, os arquivos de idioma e de configuração do *Logger* são carregados. Os arquivos de idioma interferem diretamente nos nomes dos *shapes* e nos elementos da interface gráfica, e por isso deve ser a primeira ação do E4J.

– Carregar Shapes: os *shapes* são a definição da representação gráfica dos elementos. Esta ação efetua o carregamento dos *shapes* e a adição de cada um na paleta de elementos.

– Carrega Interface Gráfica: é a ação que constrói toda a interface gráfica. Após esta ação, os elementos da interface gráfica são apresentados e já podem enviar e receber eventos do usuário.

– Conector (A): indica que o fluxo é transferido para o *Conector (A)* da partição *JGraph*. Este, por sua vez, começa a tratar (enviar e receber) os eventos do usuário.

– Gerar Casos de Uso (evento recebido): evento monitorado pelo E4J e gerado quando o usuário clica no item de menu "Gerar Casos de Uso". Este evento leva a ação "Converter para Estrutura JGOOSE".

– Converter para Estrutura JGOOSE: esta ação realiza o mapeamento da estrutura E4J para a estrutura do JGOOSE. Após isto, o fluxo passa ao nodo final.

– Nodo final: o fluxo deve chegar a este elemento com uma estrutura do JGOOSE que representa o modelo organizacional projetado. Este fluxo deixa de ser do subcomponente E4J e passa para o fluxo do JGOOSE (conforme a figura 4.9).

– Salvar (evento recebido): é o evento gerado para salvar o modelo projetado em arquivo. As opções de arquivo são: *mxe* e *istarml*. A primeira opção é um formato nativo do JGraph. A segunda opção é uma estrutura iStarML (apresentada no capítulo 2.4).

– Salvar em .mxe: ação para salvar, em arquivo, a estrutura E4J no formato padrão do JGraph. Após esta ação, o E4J volta a tratar os eventos do usuário pelo *Conector (A)*.

– Exportar iStarML: ação para gravar, em arquivo, a estrutura E4J no formato iStarML. Após esta ação, o fluxo é direcionado à partição iStarML que irá fazer os

tratamentos necessários e salvar em arquivo *istarml*.

  - Conector (C): indica que o fluxo é transferido para o *Conector (C)* da partição *iStarML*.
  - Carregar arquivo .mxe: evento que, quando recebido, gera a ação de mesmo nome. Esta ação é a rotina de carregamento do arquivo e sua apresentação na interface gráfica. Após esta ação, o editor volta a tratar os eventos do usuário.
  - Sair: evento recebido quando o usuário clica no item de menu "Sair" ou quando pressiona as teclas de atalho "Alt + F4". Após este evento, o fluxo de execução da aplicação é finalizada.

- JGraph (Partição): esta partição se resume no tratamento (envio e recebimento) de eventos do usuário realizado pelo JGraph.
  - Conector (A): elemento de conexão com as outras partições. Pode ser encarada como nodo inicial desta partição.
  - *Split*: divide o fluxo entre as atividades de envio e recebimento de eventos da interação com o usuário.
  - Eventos do Usuário (*signal sending*): são os eventos **gerados** pelo o usuário. Esses eventos são enviados para toda a aplicação. Por exemplo, caso o usuário clique no item de menu "Sair", via interface gráfica, este sinal de evento será enviado para toda a aplicação e o recebimento deste evento é tratado pelo elemento *Sair* (da partição *E4J*).
  - Eventos do Usuário (*signal receipt*): são os eventos captados pelo JGraph. O principal evento apresentado no diagrama é o de atualização do modelo. Por exemplo, a adição de um ator no diagrama é um evento que implica na atualização do modelo.
  - Condição: este elemento verifica se o evento que foi gerado é um evento de alteração do diagrama. Caso positivo, o fluxo de atividade passa para a execução da ação *Atualizar Modelo*. Caso contrário, o evento é enviado para o restante da aplicação.
  - Atualizar Modelo: ação de manipulação do modelo. Todas as alterações são aplicadas na estrutura do modelo e o resultado é a estrutura do *Grafo Atualizado*.

- Grafo Atualizado: objeto resultante da atualização do modelo. Este objeto, já atualizado, é enviado como um sinal de evento para tratamento em outras partes da aplicação, como exemplo, a verificação da consistência do modelo final. Estas atividades mais detalhadas não foram apresentadas neste diagrama por motivos de relevância.

- iStarML (Partição):
  - Conector (C): elemento de conexão com as outras partições. Pode ser entendido como nodo inicial desta partição.
  - Converter Estrutura E4J para Estrutura JGOOSE: Esta é operação de mapeamento da estrutura E4J para a estrutura iStarML. Este mapeamento é realizado com base nos algoritmos e tabelas apresentadas (subseção 4.3.6).
  - Salvar Arquivo .istarml: rotina que tem como entrada a estrutura em iStarML e realiza a escrita em arquivo por meio das funcionalidades do JAXB.
  - Conector (A): conexão de retorno para o tratamento dos eventos da interface gráfica do JGraph.

## 4.5 Considerações Finais do Capítulo

É importante ressaltar os benefícios que os modelos e diagramas podem agregar ao projeto de software. Eles auxiliaram na organização do sistema e na tomada de decisão sobre a criação dos elementos estruturais e interfaces compõe o sistema. Além disso, facilitam o entendimento geral do *software* e ajudam na especificação do comportamento da aplicação.

### 4.5.1 Problemas Encontrados

No início do desenvolvimento deste trabalho, foi realizada uma breve pesquisa sobre o estado da arte do iStarML no contexto de implementação Java. A intenção era encontrar códigos, rotinas ou bibliotecas que auxiliassem na manipulação de estruturas e arquivos no formato da especificação do iStarML. Carlos Cares, autor da versão 1.0 da especificação do iStarML, disponibilizou em sua página na internet[5] códigos-fonte na linguagem Java, um tutorial mostrando como utilizar esses códigos e um arquivo DTD (*Document Type Definition*) para validação de

[5]http://www.essi.upc.edu/ ccares/

arquivos XML. Porém, durante um estudo mais aprofundado desses arquivos foram encontrados diversos fatores que impactaram significativamente na decisão sobre o seu uso. Por exemplo, na figura 4.11 apresenta um trecho da classe “ccistarmlContent” que realiza a implementação de um autômato finito de estados (repare na matriz do atributo “afd”), uma estrutura complexa e de difícil depuração, inviabilizando a manutenção do código. Esses e outros problemas, bem como a solução adotada, foram descritas no apêndice A.

```
private static char[] l_ = {'l'};
private static char[] underscore_ = {'_'};
private static char[] number_ = {'0', '1', '2', '3', '4', '5', '6', '7', '8'
private static char[][] alphabet = {excla_, initag_, endtag_, quest_, x_, m_
// 0   ,   1   ,  2   ,3 ,4 ,5 ,  6   ,  7   ,  8   ,    9   ,  10  ,  11  ,
private static short afd[][] = {
    //00,01,02,03,04,05,06,07,08,09,10,11,12,13,14,15,16,17,18,19,20
    {40, 01, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40}, // state 0
    {41, 41, 41, 02, 41, 41, 41, 41, 41, 41, 41, 41, 41, 41, 41}, // state 1
    {42, 42, 42, 42, 3, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42}, //state 2
    {42, 42, 42, 42, 42, 4, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
```

Figura 4.11: Parte da classe “ccistarmlContent”.

Outro problema encontrado durante o desenvolvimento deste trabalho foi o forte acoplamento de estruturas e rotinas do JGOOSE, dificultando a adaptação. Houveram até chamadas de *Dialogs* dentro de construtores padrões, impossibilitando o uso direto por outros código sem a necessidade da interação com o usuário. Um outro problema permaneceu: o usuário, ao iniciar o JGOOSE deve clicar no botão “Abrir Arquivo Telos” antes de começar a utilizar o E4J. Estes problemas foram relatados ao desenvolvedor da ferramenta JGOOSE e os mesmos estão sendo sanados.

Do editor E4J, uma das principais limitações é que só foi implementada a rotina de exportação para o formato iStarML. Devido ao escopo de tempo e projeto, a rotina de importação não foi finalizada. Portanto, uma estrutura iStarML não é mapeada para a estrutura E4J.

# Capítulo 5

# Exemplos de Uso

Este capítulo apresenta exemplos de uso da ferramenta E4J, apresentada e descrita no capítulo 4 deste trabalho. O objetivo é exemplificar o uso da ferramenta e mostrar como modelos organizacionais i*, desenvolvidos no E4J, são mapeados para a estrutura de classes e objetos do JGOOSE. Após esse mapeamento, o usuário poderá gerar os casos de uso a partir dos modelos desenvolvidos no E4J.

## 5.1 Instalação e requisitos necessários

O E4J não necessita de nenhum processo de instalação. O programa é distribuído contendo um arquivo executável "jgoose-with-e4j.jar" e uma pasta com os recursos necessários (como ícones e arquivos de configuraçã). Apenas a execução de seu arquivo principal "jgoose-with-e4j.jar" é o suficiente para dar início à aplicação. Cabe ressaltar que, por ser uma aplicação desenvolvida na linguagem de programação Java, é necessário que uma plataforma de execução Java compatível com os recursos da aplicação esteja instalada no computador.

O *Java Runtime Environment* (JRE) - é o Ambiente de Execução Java que contém uma *Java Virtual Machine* (JVM), bibliotecas de classes Java e um lançador de aplicações Java - recursos necessários para executar programas desenvolvimentos na linguagem de programação Java [74].

Conforme apresentado no capítulo 4, a aplicação foi desenvolvida utilizando o JDK 7. Isso implica na versão do JRE, que deve ser também a versão 7 (JRE 7). Esse JRE pode ser baixado gratuitamente no site oficial da Oracle[1].

Cabe destacar que durante o desenvolvimento dos exemplos deste capítulo foi utilizada a

[1]Endereço para download do JRE: *http://www.oracle.com/technetwork/java/javase/downloads/index.html*.

JVM na versão “1.7.0_21” em um ambiente Windows 8. A aplicação também foi testada em ambientes como Windows XP e Ubuntu 10.04. Todos os ambientes se apresentaram estáveis durante todo o processo de execução e testes.

## 5.2 Conhecendo o E4J

O E4J é um programa voltado totalmente para interação com o usuário por meio de uma interface gráfica rica em recursos de manipulação e edição de diagramas. Com base nisso, a exemplificação a seguir é um conteúdo explicativo sobre as interfaces, telas, janelas e diálogos, da ferramenta.

### 5.2.1 Abrindo o E4J

Como o objetivo do E4J é ser integrado com o JGOOSE, para chegar tela principal do E4J é necessário, primeiramente, iniciar a aplicação do JGOOSE. A figura 5.1 mostra o arquivo e a pasta que devem estar disponíveis juntos na distribuição do E4J. Para iniciar a aplicação, execute o arquivo “jgoose-with-e4j.jar”. A aplicação será iniciada e será mostrado a interface principal do JGOOSE.

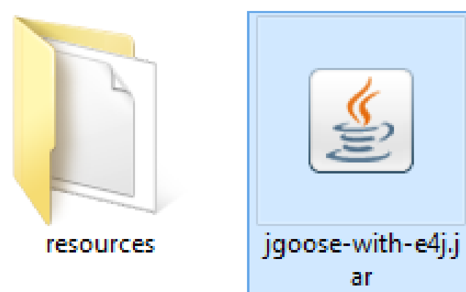



Figura 5.1: Arquivo e pasta disponíveis na distribuição do E4J.

Na interface do JGOOSE foi inserido um menu “Ferramentas” com um submemu “Abrir Editor”, indicado na figura 5.2 pelos números “(1)” e “(2)”. Para acessar o editor E4J, clique no submenu “Abrir Editor” e, então, interface gráfica do E4J será apresentada ao usuário.

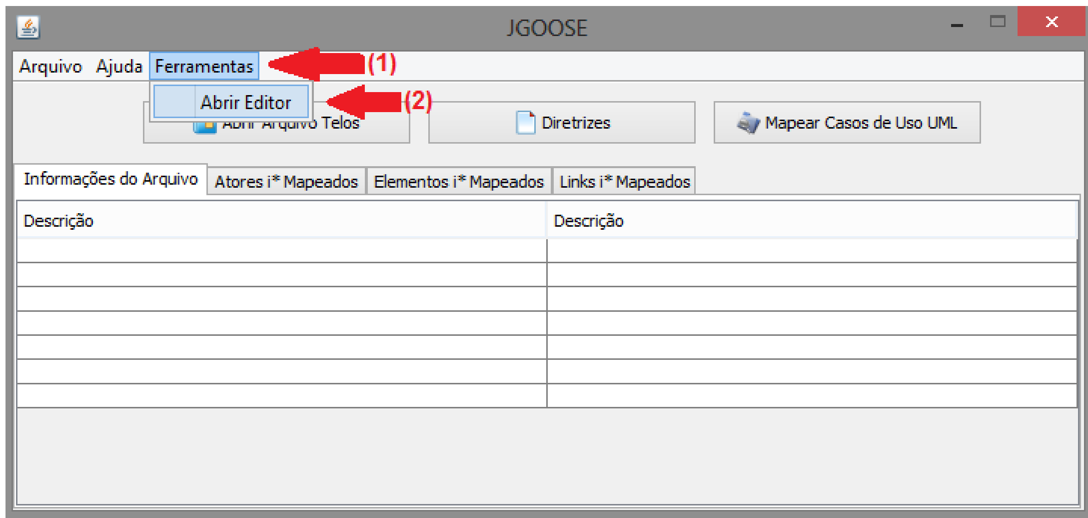



Figura 5.2: Acessando editor E4J pela interface JGOOSE.

### 5.2.2 Interface Gráfica do Usuário (GUI)

A Interface Gráfica do Usuário (GUI - *Graphical User Interface*) é apresentada a seguir. A figura 5.3 apresenta a tela principal da E4J de duas formas: a primeira (a) é a ilustração original da interface, enquanto que a segunda (b) foi dividida em 6 áreas comentadas a seguir.

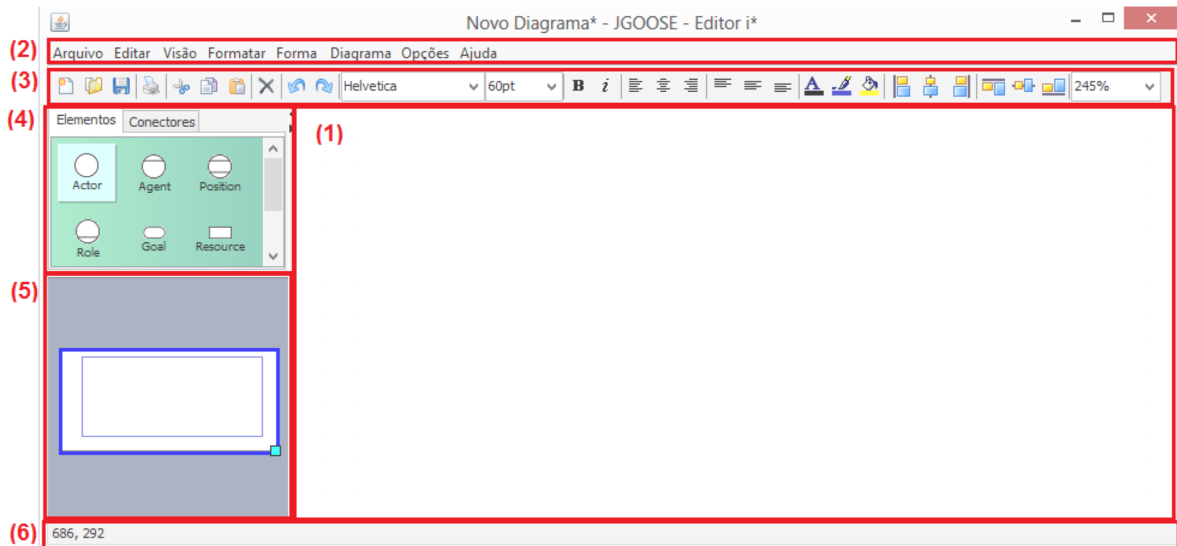



Figura 5.3: Tela principal da E4J.

(1) Área de desenho: é a área de trabalho do usuário na qual é construído o modelo. É uma área de visualização e manipulação do diagrama. Nesta área ficam visualmente os elementos do modelo (ex. atores e objetivos) e suas ligações (ex. ligação de decomposição).

(2) Barra de menus: dá o acesso a todas as ações gerais sobre a aplicação como: fechar aplicação, abrir ou salvar arquivo, configurar área de desenho, etc.

(3) Barra de ferramentas: é um conjunto de atalhos para as funções mais comuns de uma aplicação ou funções usadas com frequência durante a criação ou edição de diagramas.

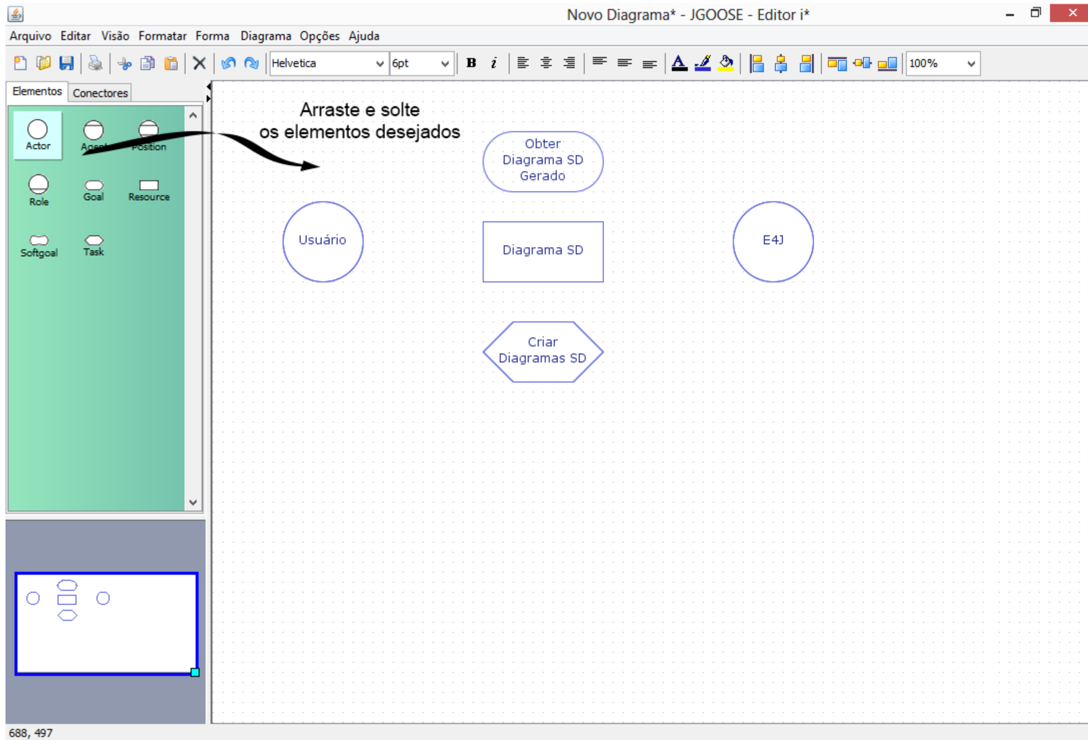
(4) Paleta de elementos: contém duas abas que agrupam os elementos dos modelos SD e SR (vértices e arestas). A aba "elementos" contém os vértices (Ator, Agente, Objetivo, etc.) enquanto que a aba "conectores" contém as possíveis ligações dos modelos (dependência, ISA, decomposição, etc).

(5) Minimapa: é uma miniatura do diagrama inteiro. Sua função é ajudar o trabalho com diagramas muito grandes. Contém um retângulo azul que representa uma porção do diagrama total correspondente ao que está sendo exibido na área de desenho.

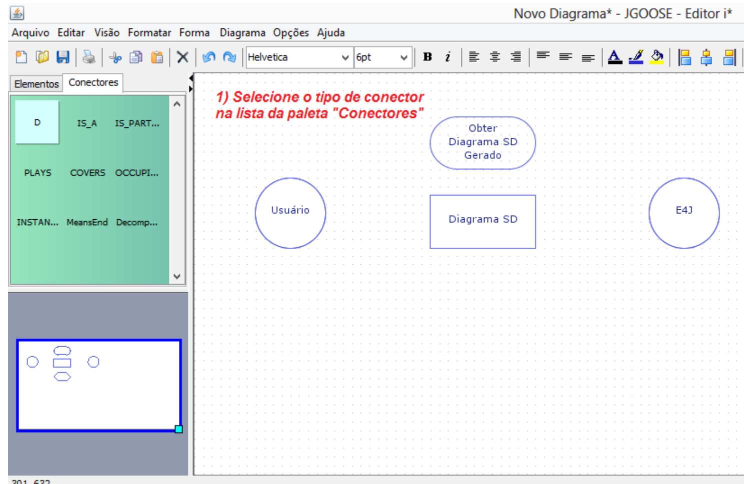
(6) Barra de informações: basicamente apresenta ao usuário a posição do mouse ou fornece algumas informações sobre o estado da área de desenho.

## 5.3 Construindo Modelos SD e SR

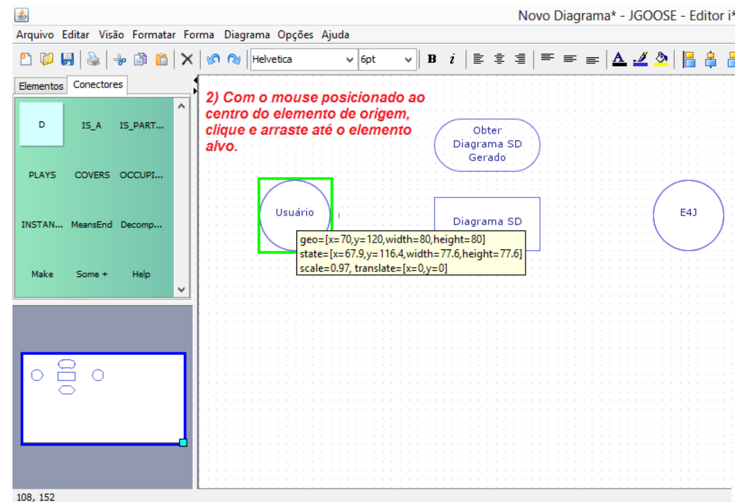
O processo de construção dos modelos, tanto SD quanto SR, pode ser resumido em ações de adição de elementos da paleta de elementos e da ligação entre esses elementos. Para adicionar um elemento à área de trabalho, clique no elemento desejado na paleta de elementos e arraste para a área de desenho (conforme ilustrado na figura 5.3).



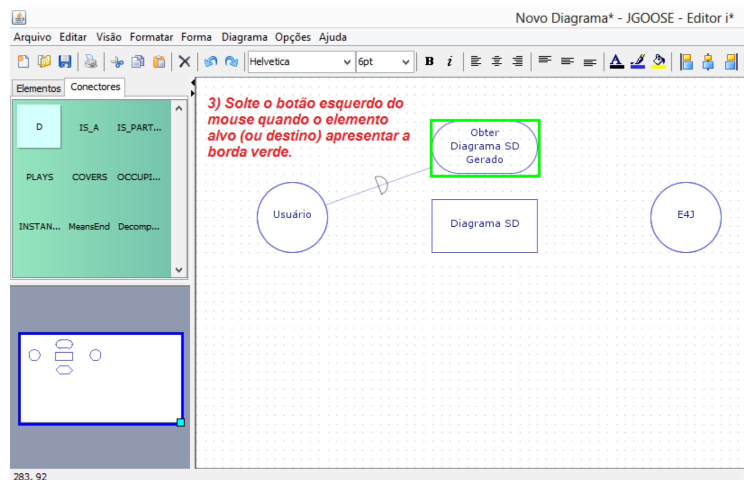

Para criar as ligações, primeiro selecione o tipo de ligação desejada na paleta de conectores e, depois, clique no centro do elemento de origem e arraste até o elemento de destino. A figura 5.4 ilustra este processo exemplificando a criação de uma ligação de dependência entre o ator "Usuário" e o objetivo "Obter Diagrama SD Gerado". Com estes passos, o usuário do E4J é capaz de criar qualquer diagrama SD.




(a) Selecione o tipo da ligação.



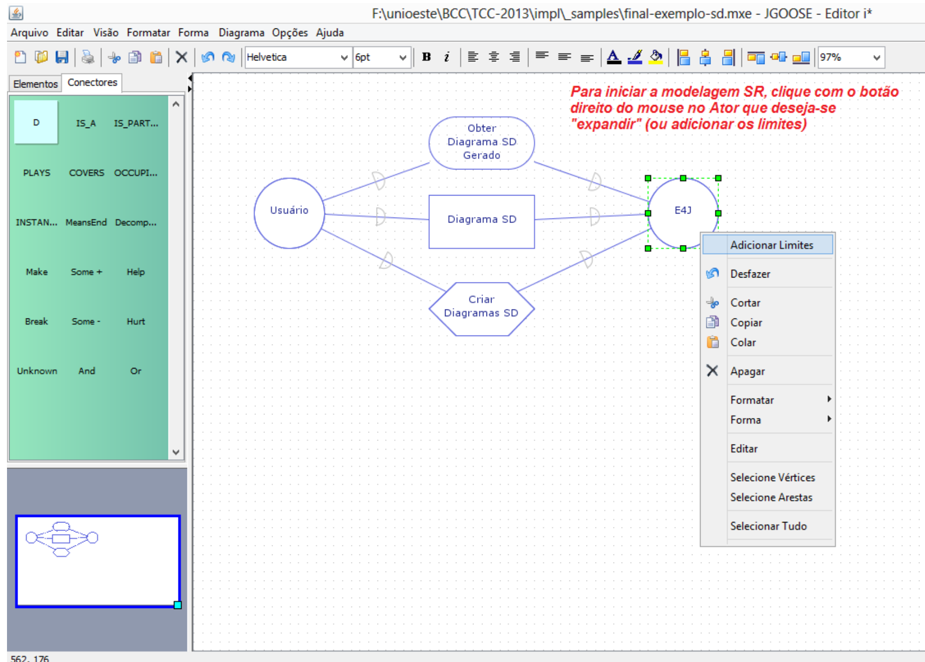

(b) Clique no elemento de origem.



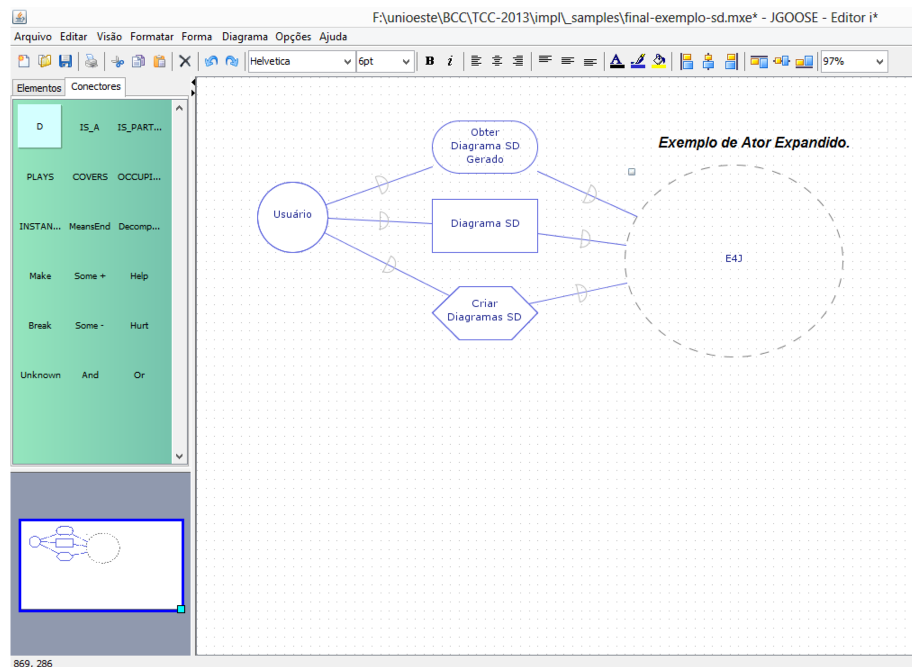

(c) Arraste até o elemento de destino.

Figura 5.4: Exemplo: criando uma ligação de dependência.

Considerando um diagrama SD já criado, para criar um diagrama SR basta selecionar um ator (normalmente o que representa o sistema de software), clicar com o botão direito e selecionar “Adicionar Limites” (conforme ilustrado na figura 5.5). Após adicionado os limites, o ator aparecerá no estado expandido.




(a) Adicionando limites.



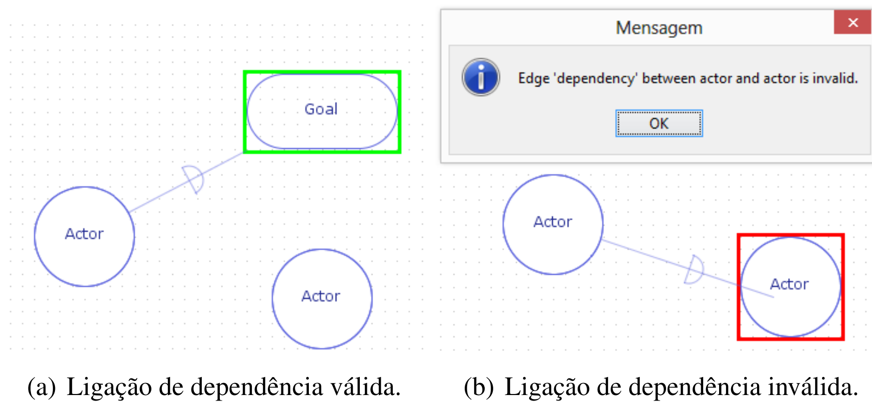

(b) Ator expandido, após a adição dos limites.

Figura 5.5: Exemplo: criando um modelo SR.

### 5.3.1 Ligações de dependências

Ligações de dependências possui uma rotina de verificação que indica se a ligação é válida ou não. Uma ligação de dependência é válida somente se for entre um ator e um *dependum* ou

entre dois *dependum*. A figura 5.6 ilustra uma ligação de dependência válida e uma inválida. Cabe ressaltar que na ligação de dependência inválida um retângulo vermelho é mostrado previamente no elemento de destino e, caso a tentativa de criação persista, uma mensagem de erro é apresentada ao usuário.

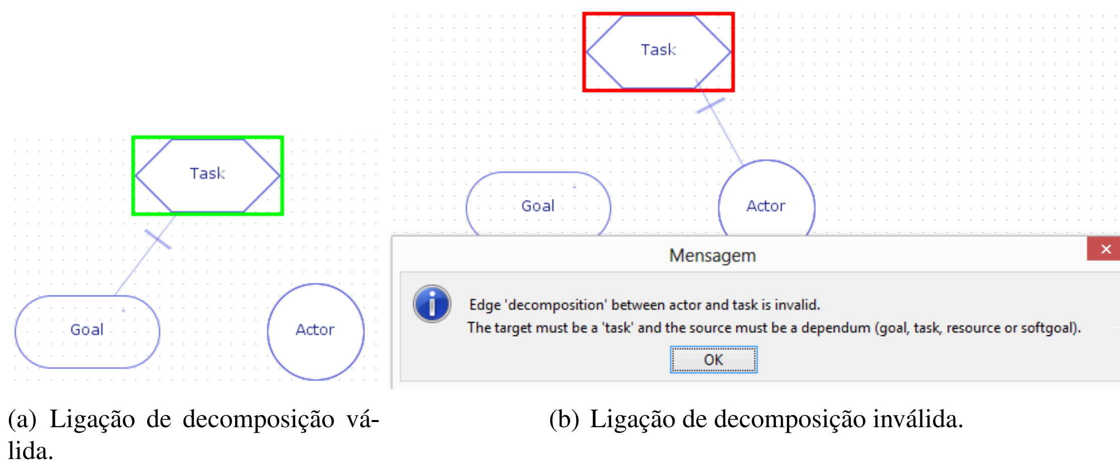



(a) Ligação de dependência válida. (b) Ligação de dependência inválida.

Figura 5.6: Exemplo: criando uma ligação de dependência.

### 5.3.2 Ligações de decomposição de tarefa

De forma semelhante as ligações de dependências, as ligações de decomposição de tarefa também possuem uma rotina de verificação para validação. Neste caso, uma ligação de decomposição de tarefa só pode ser realizada de um *dependum* (que pode ser um objetivo, um objetivo-soft, uma tarefa ou um recurso) para uma tarefa (conforme ilustrado na figura 5.7)

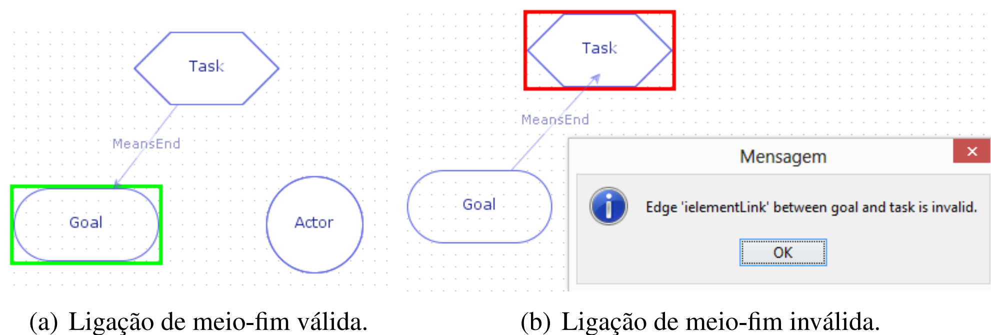



(a) Ligação de decomposição válida. (b) Ligação de decomposição inválida.

Figura 5.7: Exemplo: criando uma ligação de decomposição de tarefa.

### 5.3.3 Ligações de meio-fim

De forma semelhante as ligações de decomposição de tarefa, as ligações de meio-fim também possuem uma rotina de verificação para validação. Neste caso, uma ligação de meio-fim, o meio pode ser de qualquer tipo de *dependum* (um objetivo, um objetivo-soft, uma tarefa ou um recurso) para um objetivo (conforme ilustrado na figura 5.8)




(a) Ligação de meio-fim válida. (b) Ligação de meio-fim inválida.

Figura 5.8: Exemplo: criando uma ligação de meio-fim.

## 5.4 Salvando e Exportando

A seguir, é apresentado como salvar o diagrama e como exportá-lo para o formato iStarML. Para isso, são utilizados os formatos de extensão ".mxe" e ".istarml", respectivamente. A primeira extensão (".mxe") é específica da estrutura do JGgraphX e já vem embutida nos arquivos de exemplo dessa biblioteca. Neste formato de arquivo é salvo toda a estrutura e conteúdo do diagrama e modelo. Cabe destacar que com este formato de arquivo é possível "abrir" um diagrama previamente salvo. Já a segunda extensão (".istarml") é possível apenas exportá-la, pois, por questões de escopo de tempo, não foi implementada a rotina de mapeamento da estrutura iStarML para a estrutura E4J.

### 5.4.1 Salvando diagrama

Considerando que um diagrama qualquer foi construído na ferramenta, para salvar na extensão ".mxe", basta seguir os passos indicados na figura 5.9: (1) clique no atalho "Salvar" (disquete); (2) escolha a pasta e o nome do arquivo; (3) selecione a extensão; (4) clique no botão "salvar". O passo (1) também pode ser realizado executando o atalho "ctrl + S", conforme

a lista de atalhos do apêndice C, ou ainda acessando o menu “Arquivo” e clicando no submenu “Salvar”.

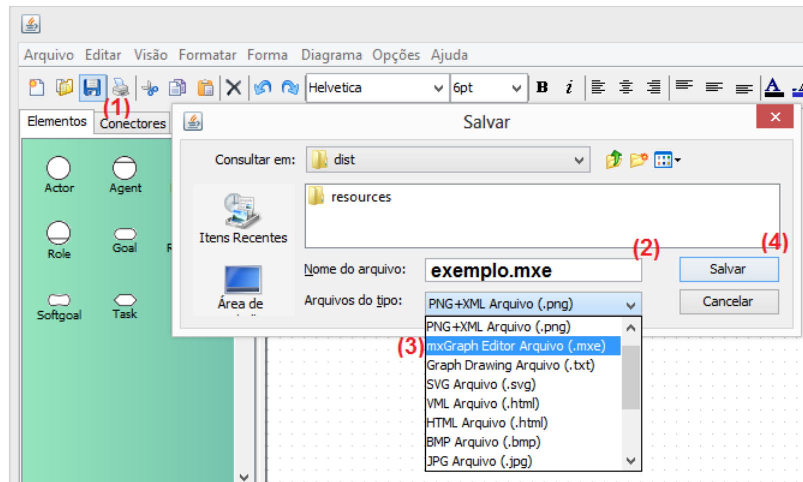



Figura 5.9: Salvando diagrama em .mxe

O resultado é um arquivo com extensão “.mxe”. Este arquivo é estruturado no formato XML, conforme exemplificado na figura 5.10.

```
<mxGraphModel>
     <root>
          <mxCell id="0"/>
          <mxCell id="1" parent="0"/>
          <dependency id="10" label="D" type="">
               <mxCell edge="1" parent="1" source="7"
style="straight;fontSize=10" target="4">
```

Figura 5.10: Conteúdo parcial do diagrama salvo em .mxe

### 5.4.2 Exportando para iStarML

Considerando que um diagrama qualquer foi construído na ferramenta, para exportá-lo para o formato iStarML, basta seguir os passos indicados na figura 5.9: (1) clique no menu “Arquivo”; (2) clique no submenu “Exportar iStarML”; (3) escolha a pasta e o nome do arquivo; (4) clique no botão “salvar”.

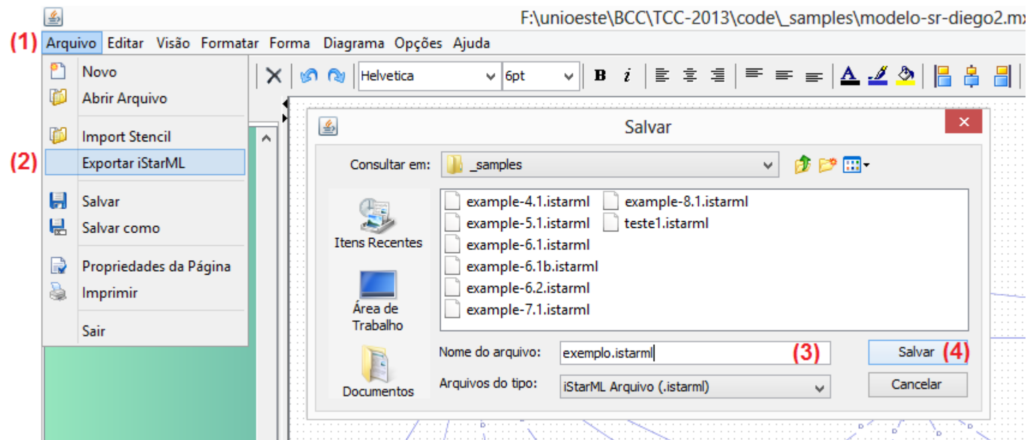



Figura 5.11: Salvando diagrama em .istarml

O resultado é um arquivo com extensão ".istarml" estruturado no formato XML, conforme exemplificado na figura 5.12.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<istarml version="1.0">
    <diagram id="" name="Exported from E4J">
        <actor id="31" name="Cliente" type="actor"/>
        <actor id="30" name="SGBD" type="actor"/>
        <actor id="28" name="Funcionário" type="actor"/>
        <actor id="2" name="Fornecedor" type="actor"/>
        <actor id="29" name="Sistema" type="actor">
            <boundary>
                ...
```

Figura 5.12: Conteúdo parcial do diagrama salvo em .istarml

## 5.5 Gerando Casos de Uso com a JGOOSE

O processo de geração de casos de uso conta principalmente com as funções do JGOOSE. Apesar do diagrama ser desenvolvido no E4J, a geração de casos de uso continua sob o controle do JGOOSE. Tudo que é feito é apenas uma conversão entre estruturas (do E4J para o JGOOSE). Para gerar os casos de uso a partir de um diagrama no E4J, basta seguir os passos indicados na figura 5.13: (1) clique no menu "Diagrama"; (2) clique no submenu "Gerar Casos de Uso". Após isto, a interface do JGOOSE será ativa.

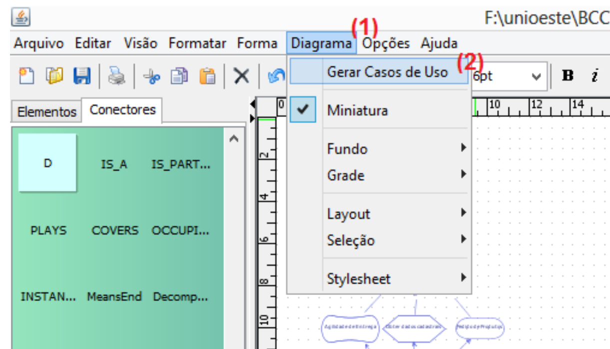



Figura 5.13: Gerar Casos de Uso. Passos (1) e (2).

## 5.6 Considerações Finais do Capítulo

Os exemplos de uso da ferramenta E4J, desenvolvida e descrita no capítulo 4 deste trabalho, foram apresentados neste capítulo. Na prática, durante todo o processo de uso e exemplificação, a ferramenta se comportou de forma apropriada. Porém, algumas questões de usabilidade ainda precisam ser analisadas e refletidas em trabalhos futuros. Mais especificamente, a interação com o usuário e a paleta de elementos ou a paleta de conectores precisa ser modificada e seguir um padrão. Enquanto na paleta de elementos a interação é realizada pela técnica de arraste-e-solte (*drag-and-drop*), a paleta de conectores é utilizada a técnica de seleção simples. Uma solução seria adotar a mesma abordagem realizada pela ferramenta OME: para adicionar um elemento, basta um clique no elemento e um clique na área de desenho; para criar uma ligação, basta clicar na ligação desejada, clicar no elemento de origem e clicar no elemento de destino.

Outro problema presente na interface do E4J é a apresentação dos conectores na paleta de conectores. Eles não possuem um ícone ou uma representação. São apenas descritos em texto puro. Uma solução para isto seria criar os *shapes* para melhorar a aparência e visualização desses conectores.

# Capítulo 6

# Conclusão

Neste trabalho de conclusão de curso foi desenvolvido o E4J, um editor de diagramas do *framework* i* integrado à ferramenta JGOOSE. Neste capítulo são apresentados os resultados desta pesquisa e as conclusões com base nesses resultados. Por fim, são propostos alguns tópicos para a realização de pesquisas e trabalhos futuros.

## 6.1 Resultados

A integração do E4J ao JGOOSE transformou o JGOOSE numa aplicação auto-suficiente (*standalone application*), pois não é mais necessário o uso de softwares auxiliares, como o OME, para criar o modelo i*. Todo o processo de criação e edição de diagramas pode ser realizado de dentro do JGOOSE, chamando o E4J apenas quando necessário.

Os exemplos de uso do E4J, conforme apresentado no capítulo 5, mostrou que a ferramenta contempla as principais funcionalidades necessárias para a criação e edição de diagramas e a transformação internas das estruturas do E4J para o JGOOSE.

## 6.2 Conclusões

Analisando os resultados obtidos, podemos afirmar que um editor gráfico de modelos i* integrado ao JGOOSE, como o E4J, melhora o processo de desenvolvimento de modelos organizacionais com a ferramenta. A criação e edição desses modelos é realizada em conformidade com o mapeamento para casos de uso UML, complementando as funcionalidades do JGOOSE. Esse mapeamento é realizado internamente, sem a necessidades do uso de arquivos auxiliares para tal fim.

Com o questionário i* Wiki preenchido (apêndice B), será possível que a nova ferramenta seja analisada pela comunidade envolvida com o projeto i* Wiki. Espera-se obter avaliações, críticas e comentários, advindos de diversos pesquisadores da área de modelagem organizacional.

Além disso, apesar de existirem várias ferramentas no i* Wiki [1], a maioria são *plugins* de outros *softwares*, como o Eclipse. Um dos problemas de se utilizar o Eclipse é que ele não permite mais de uma instancia de execução do programa ao mesmo tempo. Realizando uma comparação entre os tamanhos (em megabytes) das soluções na forma de *plugins* do Eclipse e as soluções ditas "standalones", percebe-se que muitos recursos do Eclipse não são utilizados ou mesmo necessários para se trabalhar com modelos organizacionais.

Conforme [53], a adoção de um modelo de dados comum como a linguagem iStarML também facilita a interação entre os diferentes componentes do sistema. Neste trabalho, mostramos que a ferramenta suporta a estrutura iStarML e, além disso, possui uma rotina de exportação dessa estrutura interna para arquivo com extensão ".istarml".

## 6.3 Trabalhos Futuros

Após analisar os resultados deste trabalho e as conclusões realizadas, é possível apontar os seguintes estudos futuros:

- Realizar uma pesquisa empírica, buscando validar os resultados obtidos do E4J, auxiliada por metodologias específicas de validação de software.

- Implementar uma rotina de mapeamento da estrutura interna do iStarML para a estrutura de grafos do E4J. Como a importação de arquivo no formato da iStarML é realizada pelo *package* iStarML, bastará varrer a estrutura interna do iStarML mapeando cada elemento para o seu equivalente na estrutura E4J.

- Elaborar o **Manual do Usuário**, contendo informações de cada elemento da interface gráfica que é possível de ser acessada pelo usuário.

- Construir o **Manual do Desenvolvedor** visando auxiliar principalmente os desenvolvedores que desejam colaborar com o projeto. Deve ser especificado as decisões de projeto

como: padronizar nomenclaturas [75], padronizar o idioma usado no código-fonte, etc. Também deve ser especificado políticas de desenvolvimento e fluxo de trabalho condizente com as boas práticas de desenvolvimento colaborativo em projetos *open-source* [76, 77].

- Melhorar as rotinas de testes unitários. Nas rotinas de mapeamento de estruturas ou conversão de formatos são imprescindíveis a automatização dos testes. Também é recomendado o gerenciamento de testes em APIs, acompanhando a versão e incrementando documentação do sistema. Incorporar o XMLUnit [78] ou outra solução que facilite os testes com estruturas e arquivos XML.

- Disponibilizar em um local público e de fácil acesso.

- Reflexão das alterações nas estruturas internas do JGOOSE para a estrutura do E4J, atualizando os diagramas SD e SR e melhorando o processo de mudança ou atualização dos modelos organizacionais.

# Apêndice A

# iStarML API: um início

Uma API (*Application Programming Interface*), no contexto deste trabalho, pode ser definida como uma especificação, documentação e implementação de um conjunto de rotinas e estruturas que auxiliam desenvolvedores a lidar com um determinado problema da aplicação. Sabe-se que uma API pode trazer diversos benefícios quando bem desenvolvida e planejada [79] [80]. Seu principal objetivo é facilitar o trabalho do programador sobre um determinado problema.

Este apêndice foi o fruto da necessidade de se utilizar um metamodelo comum entre as aplicações JGOOSE e E4J para se trabalhar com modelos do framework i*. O objetivo deste apêndice é divulgar o que foi desenvolvido e iniciar uma discussão sobre o projeto e desenvolvimento de uma API para o metamodelo iStarML.

## A.1 Introdução

O iStarML é uma especificação de um conjunto de elementos XML que visa a representação de diferentes conceitos da modelagem organizacional i* [54].

### A.1.1 Problemas com o *ccistarml*

O *ccistarml* é um pacote Java para criação, importação e manipulação de arquivos iStarML [81]. Foi desenvolvido usando JDK 1.5.0_11 e NetBeans IDE 5.5 [81]. Permite a verificação de sintaxe XML e a especificada iStarML. Porém, após uma análise sobre o código-fonte e a documentação do *ccistarml*, observou-se os seguintes itens:

- Foi implementado um interpretador de texto que, através de um AFD (Autômato Finito

de Estados), visa analisar, validar e construir uma estrutura iStarML. O problema é que o AFD utiliza uma matriz de inteiros para armazenar os estados e suas transições. Essa estrutura dificulta o entendimento do interpretador e a manutenção do seu código-fonte.

- Não foi utilizado nenhuma política de depuração. Todos os arquivos usavam as funções do “System.out.*”, ao invés de *Loggers*, para imprimir os erros, ferindo as boas práticas de programação em Java [82].

- Ao invés de sobrescrever o método “toString” da classe *Object*, algumas classes possuem métodos específicos para transcrever um objeto em “String”.

- Um gerenciador de identificadores em “String” hexadecimal com incrementos tratados de forma manual (pela classe “IDManager”), pode prejudicar a qualidade do projeto.

- Ao invés de se estender a classe *Exception* (ou similares), a classe *ccFileError* manualmente indica os problemas por meio de “System.out.*”.

- Algumas das classes que possuem documentação nos métodos e atributos não utilizam o padrão de documentação JavaDoc, dificultando a análise durante o desenvolvimento e uso desses itens.

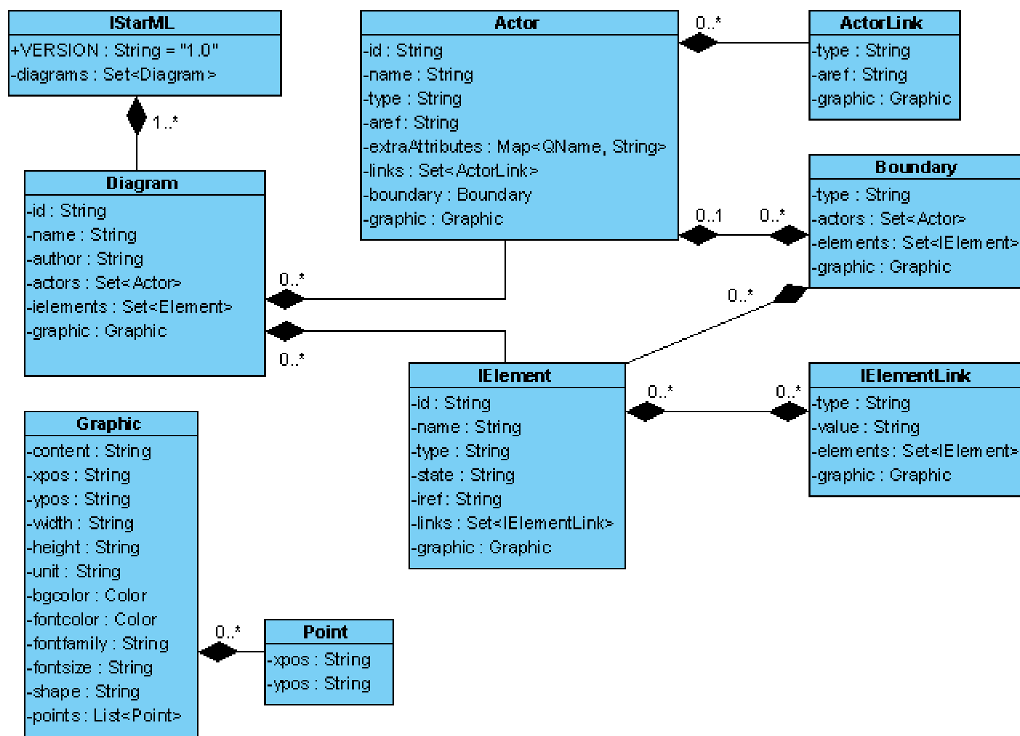
### A.1.2 Solução adotada

O iStarML é uma especificação de linguagem baseada na estrutura XML [54]. É uma solução para o intercâmbio entre metamodelos baseados no *framework* i*.

O iStarML possui sua estrutura baseada em XML e hoje em dia existem diversas soluções para a se trabalhar com este tipo de estrutura [70] [83]. São soluções consolidadas no mercado e que, além de facilitar o trabalho do desenvolvedor, agregam qualidade ao projeto. Desta forma, neste trabalho adotamos uma estrutura alternativa, utilizando a solução da biblioteca de mapeamento entre XML e objetos Java, JAXB [70].

## A.2 Estrutura iStarML

A figura A.2 apresenta um diagrama de classes da estrutura iStarML desenvolvida. Ela representa uma equivalência à estrutura XML definida no guia de referência da versão 1.0 [27].




Cada classe do diagrama de classe apresentado na figura A.2 é descrita a seguir:

- IStarML: classe mapeada para a *tag* “<istarml>”, é o elemento raiz de toda a estrutura iStarML. Possui um atributo para indicar a versão da especificação iStarML [27] e um conjunto de diagramas representando os subelementos deste elemento raiz.

- Diagram: classe mapeada para a *tag* “<diagram>”. Esta representação permite que mais de um diagrama seja representado em um mesmo modelo ou arquivo.

- Actor: classe mapeada para a *tag* “<actor>”. Representa um ator do modelo. Os diferentes tipos de atores são manipulados pelo atributo “type”.

- ActorLink: classe mapeada para a *tag* “<actorlink>”. Representa os elementos de ligações entre os elementos da classe *Actor*.

- Boundary: classe mapeada para a *tag* “<boundary>”. Representa as condições de estado interno de um ator, e essas condições podem ser representadas pela aglomeração de elementos dentro do escopo do ator. Esta aglomeração também pode ser chamada de limite (ou *boundary*).

- IElement: classe mapeada para a *tag* “<ielement>”. Representa uma abstração dos elementos intencionais (objetivo, tarefa, recurso, objetivo-soft). Algumas variações do i* aceitam tipos adicionais como crenças (*belief*) ou restrição (*constraint*). Como a proposta iStarML considera todos estes tipos de elementos intencionais [27], eles podem ser representados pela classe *IElement*.

- IElementLink: classe mapeada para a *tag* “<ielementlink>”. Representa os elementos de ligações entre os elementos da classe *IElement*.

- Graphic: classe mapeada para a *tag* “<graphic>”. Esta classe permite a especificação de elementos gráficos específicos para cada um dos elementos descritos acima (exceto o elemento raiz IStarML).

- Point: esta é a única classe do diagrama que não é mapeada para nenhuma *tag* XML. Porém, a sua ligação de agregação facilita a manipulação dos pontos dos elementos gráficos.

Para um rápido desenvolvimento e validação da estrutura, foi utilizada a metodologia de desenvolvimento orientada à testes, construindo uma bateria de testes que comparava a estrutura implementada e os arquivos de entrada e saída em XML. Para facilitar este processo de teste, foi utilizado o XMLUnit [78], uma biblioteca compatível com a conhecida biblioteca de testes unitários Java - JUnit [84].

# Apêndice B

# i* Wiki Questionnaire

## B.1 General Information

**Tool Name**: E4J: Editor for JGOOSE.

**Version**: 0.4.7

**Group**: Laboratório de Engenharia de Software (LES) da UNIOESTE, *campus* Cascavel, PR - Brasil.

**Date Tool Template Last Updated**: n/a.

**Web page**: to be defined.

**Main Purpose of the Tool**: A graph editor to support goal-oriented and/or agent-oriented modeling.

**i* framework supported**: Yu PhD thesis.

**Availability of the tool**: Both (for modelling and development).

**Programming Language**: Java.

**Platform Requirements**: Windows, Linux or Mac.

**Other technology needed**: Java Runtime Environment 7.

**Current state of the tool**: Under development.

**Ongoing work**:

- Improve model validation rules of the tool.
- Improve draws of the tool.
- Improve the usability and functionality of the tool.
- Improve interoperability with iStarML language.

## B.2 i* Modelling Suitability

1. **Does the tool allow SD modelling?** Yes.

2. **Does the tool allow SR modelling?** Yes.

3. **Does the tool allow working with SD & SR models jointly?** Yes. By means of expandable elements (as in OME).

4. **Does the tool allow the construction of the models graphically?** Yes. Dragging and dropping the elements into a drawing page. To construct an element, click an element symbol on the "Elements" palette and drag to the position in the drawing page where the element should locate. To draw a link element, select a link symbol on the "Connectors" palette, click in the center of source element and drag to the center of target element at drawing page.

5. **Does the tool allow the construction of the models textually?** No.

6. **Describe how the elements are modelled and their flexibility**: Users can drag-and-drop elements from "Elements" palette to the drawing page. Users can change the name and size of an element. The element can also be moved, deleted, copied, pasted, grouped and styles can be added. All these actions are done graphically and by commands either on the menu bar or pop-up command window.

7. **Describe how the dependency links are modelled and their flexibility**: dependencies are modelled with straight lines that can not be redirected.

8. **Does the tool allow automatic organization of the elements?** Yes. The tool provides a basic graph layout function.

9. **Other modelling facilities provided by the tool**: work separately with expandable elements; Undo/Redo Manager; Can push an element into or move it from an expandable element;

10. **Does the tool check SD models?** Yes. Checks for unrecognised connections and checks for invalid connections.

11. **Does the tool check SR models?** Yes. Checks for unrecognised connections and checks for invalid connections.

12. **Other checks provided by the tool**: none.

13. **Does the tool allow working with two or more models at the same time?** No. But the user can run multiples instances of the tool.

14. **Does the tool allow to group models in projects?** No.

15. **Does the tool allow working with two or more projects at the same time?** No.

16. **What are the other functionalities that the tool provides?**

    - Export the i* model to iStarML;
    - Generate Use Cases;
    - Export the Use Case to XMI;

## B.3 Usability

17. **Rate the understandability of the user interface**: Internal use. Is not all in English.

18. **Rate the quality of the user manual**: Internal use. Is not in English and incomplete.

19. **Does the tool provides i* learning facilities?** Yes. Because the syntax checker alert the user about the common errors by using popup.

20. **Does the tool provide any examples for the users?** Yes. In the folder "Samples".

21. **Rate the difficulty of installing the tool**: copy files.

## B.4 Maturity of the tool

22. **Rate the maturity of the tool from the user point of view:** Prototype. Because it has a occasional testing, a non-exhaustive testing and the interface can be improved.

23. **Has the tool been used for any case study?** No.

24. **Has the tool been tested in large models?** No.

25. **Has the tool any drawback when working with very large models?** Not tested.

26. **Which is approximately the maximum size of the model (in terms of actors and dependencies) the tool has been used for?** Not tested.

## B.5 Extensibility and Interoperability

27. **Does the tool allow importing files?** Yes. i* model in ".tel" and ".mxe".

28. **Does the tool allow exporting files?** Yes. i* model in ".istarml" and ".mxe". Use Case model in ".xmi".

29. **Does the tool allow importing/exporting the data through an XML format?** No. Only export ".istarml" and a new DTD must be provided.

30. **Is the architecture of the tool published?** Yes. In this thesis.

31. **Does the tool allow the addition of other elements outside the i* framework of the tool?** No.

32. **New functionalities can be added to the tool by means of:** open-Source code.

33. **Rate the maturity of the tool from for open development:** under development; no help provided; non-exhaustive testing;

34. **Is there any internal documentation for programmers?** No.

# Apêndice C: E4J - Atalhos

Todos os comandos aqui listados podem ser acessados via interface gráfica.

## Arquivo

| Atalho | Ação |
|---|---|
| Ctrl n | Novo arquivo |
| Ctrl o | Abrir arquivo |
| Ctrl s | Salvar |
| Ctrl ⇑ s | Salvar como |
| Alt F4 | Sair |

## Edição

| Atalho | Ação |
|---|---|
| Ctrl x | Recorta os elementos selecionados |
| Ctrl c | Copia os elementos selecionados |
| Ctrl v | Cola os elementos selecionados |
| F2 | Renomeia o elemento selecionado |

## Seleção

| Atalho | Ação |
|---|---|
| Ctrl a | Seleciona todos |
| Ctrl ⇑ v | Seleciona todos os vrtices |
| Ctrl ⇑ e | Seleciona todos as arestas |

# Referências Bibliográficas


[1] AACHEN, R. *i* Wiki*. 2013. Acessado em: 04 de junho de 2013. Disponível em: <http://istarwiki.org/>.

[2] GRAU, G. et al. A comparative analysis of i* agent-oriented modelling techniques. In: *Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering, SEKE*. [S.l.: s.n.], p. 5–7. 2006.

[3] CASE, A. F. Computer-aided software engineering (case): technology for improving software development productivity. *ACM SIGMIS Database*, ACM, v. 17, n. 1, p. 35–43, 1985.

[4] THAYER, M. D. R. H. Software requirements engineering. In: IEEE. *IEEE Computer Society Press*. [S.l.]. 1993.

[5] KOTONYA, G.; SOMMERVILLE, I. *Requirements engineering: processes and techniques*. [S.l.]: J. Wiley. (Worldwide series in computer science). ISBN 9780471972082. 1998.

[6] SCHNEIDER, F.; BERENBACH, B. A literature survey on international standards for systems requirements engineering. *Procedia Computer Science*, Elsevier, v. 16, p. 796–805, 2013.

[7] MASON, G. L. A conceptual basis for organizational modelling. *Systems Research and Behavioral Science*, Wiley Online Library, v. 14, n. 5, p. 331–345, 1997.

[8] YU, E. Modeling organizations for information systems requirements engineering. In: IEEE. *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*. [S.l.], p. 34–41. 1993.

[9] YU, E. Agent orientation as a modelling paradigm. *Wirtschaftsinformatik*, Springer, v. 43, n. 2, p. 123–132, 2001.

[10] YU, E. Towards modelling and reasoning support for early-phase requirements engineering. In: IEEE. *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*. [S.l.], p. 226–235. 1997.

[11] PEDROZA, F. P. et al. Ferramentas para suporte do mapeamento da modelagem i* para a uml: eXtended GOOD XGOOD e GOOSE. In: *Proceedings of the VII Workshop on Requirements Engineering-WER*. [S.l.: s.n.], v. 4, p. 164–175. 2004.

[12] SANTANDER, V. F. A. *Integrando Modelagem Organizacional com Modelagem Funcional*. Tese (Tese de Doutorado) — Universidade Federal de Pernambuco, Recife - PE, Dezembro 2002.

[13] VICENTE, A. A. *JGOOSE: Uma ferramenta de Engenharia de Requisitos para Integração da Modelagem Organizacional i* com a Modelagem Funcional de Casos de Uso UML*. Cascavel - PR: [s.n.], Dezembro 2006.

[14] MYLOPOULOS, J. et al. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, ACM, v. 8, n. 4, p. 325–362, 1990.

[15] KOUBARAKIS, M. et al. *Telos: Features and formalization*. [S.l.]: Computer Science Institute, Foundation of Research and Technology, Hellas. 1989.

[16] COCKBURN, A. *Writing effective use cases*. [S.l.]: Addison-Wesley Reading. 2001.

[17] CARDOSO, A. d. O. J. H.; ALENCAR, F. Gerenciando a engenharia de requisitos como um processo de negÓcio: Uma revisÃo sistemÁtica. In: RSC. *Revista de Sistemas e Computação, v.2 n.2*. [S.l.]. 2012.

[18] LAMSWEERDE, A. van. Goal-oriented requirements enginering: a roundtrip from research to practice [enginering read engineering]. In: IEEE. *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*. [S.l.], p. 4–7. 2004.

[19] FENG, Y.; LEE, L. S. The importance analysis of use case map with markov chains. *arXiv preprint arXiv:1002.1692*, 2010.

[20] AMYOT, D.; MUSSBACHER, G. User Requirements Notation: The first ten years the next ten years. *Journal of Software*, v. 6, n. 5, p. 747–768, 2011.

[21] AMYOT, D. Introduction to the user requirements notation: learning by example. *Computer Networks*, Elsevier, v. 42, n. 3, p. 285–301, 2003.

[22] AMYOT, D.; MUSSBACHER, G. *URN: Towards a new standard for the visual description of requirements*. 2003.

[23] YU, E. *Professor Eric Yu - Home Page*. 2013. Acessado em: 04 de junho de 2013. Disponível em: <http://www.cs.toronto.edu/ eric/>.

[24] BRISCKE, M.; SANTANDER, V. F. A.; CASTRO, J. Goose: Uma ferramenta para integrar modelagem organizacional e modelagem funcional. *Jornadas Chilenas de Computación-V Workshop Chileno de Ingeniería de Software*, Valdivia, Chile, 2005.

[25] VICENTE, A. A. et al. JGOOSE:a requirements engineering tool to integrate i* organizational modeling with use cases in uml JGOOSE: Una herramienta de ingenierÍa de requisitos para la integraciÓn del modelado organizacional i* con el modelado de casos de uso en uml. *Ingeniare. Revista chilena de ingeniería*, SciELO Chile, v. 17, n. 1, p. 6–20, 2009.

[26] BRISCHKE, M.; SANTANDER, V. F. A.; SILVA, I. F. da. Melhorando a ferramenta JGOOSE. In: *XV Workshop de Engenharia de Requisitos*. [S.l.: s.n.]. 2012.

[27] CARES, C. et al. *iStarML Reference's Guide*. [S.l.], 2007.

[28] COLOMER, D. et al. Model interchange and tool interoperability in the i* framework: a proof of concept. In: *Proc. of the 14th Workshop on Requirements Engineering*. [S.l.: s.n.], p. 369–381. 2011.

[29] YU, E. S.-K. *MODELLING STRATEGIC RELATIONSHIPS FOR PROCESS REENGINEERING*. Tese (Doutorado) — University of Toronto, 1995.

[30] MAIDEN, N. A. et al. Model-driven requirements engineering: synchronising models in an air traffic management case study. In: SPRINGER. *Advanced Information Systems Engineering*. [S.l.], p. 368–383. 2004.

[31] YU, E. S.; MYLOPOULOS, J.; LESPÉRANCE, Y. Al models for business process reengineering. *IEEE expert*, IEEE, v. 11, n. 4, p. 16–23, 1996.

[32] KOLP, M.; GIORGINI, P.; MYLOPOULOS, J. Organizational patterns for early requirements analysis. *Lecture Notes in Computer Science*, Springer, v. 2681, p. 617–632, 2003.

[33] CASTRO, J.; ALENCAR, F.; CYSNEIROS, G. Closing the gap between organizational requirements and object oriented modeling. *Journal of the Brazilian Computer Society*, SciELO Brasil, v. 7, n. 1, p. 05–16, 2000.

[34] CASTRO, J. F. et al. Integrating organizational requirements and object oriented modeling. In: IEEE. *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. [S.l.], p. 146–153. 2001.

[35] EVANS, A.; KENT, S. Core meta-modelling semantics of uml: the puml approach. In: *«UML»'99—The Unified Modeling Language*. [S.l.]: Springer. p. 140–155. 1999.

[36] WARMER, J.; KLEPPE, A. *The object constraint language: getting your models ready for MDA*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc. 2003.

[37] BRESCIANI, P. et al. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, Springer, v. 8, n. 3, p. 203–236, 2004.

[38] RAO, A. S.; GEORGEFF, M. P. et al. BDI agents: From theory to practice. In: SAN FRANCISCO. *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*. [S.l.], p. 312–319. 1995.

[39] BASTOS, L. R.; CASTRO, J. F. Enhancing requirements to derive multi-agent architectures. In: *Proceedings of WER*. [S.l.: s.n.], p. 127–139. 2004.

[40] YU, E.; LIU, L. Modelling trust for system design using the i* strategic actors framework. In: *Trust in Cyber-societies*. [S.l.]: Springer. p. 175–194. 2001.

[41] YU, E.; GIORGINI, P.; MAIDEN, N. *Social modeling for requirements engineering*. [S.l.]: Mit Press. 2011.

[42] MAO, X.; YU, E. Organizational and social concepts in agent oriented software engineering. In: *Agent-Oriented Software Engineering V*. [S.l.]: Springer. p. 1–15. 2005.

[43] WEBSTER, I. et al. A survey of good practices and misuses for modelling with i* framework. In: *Proc. of the VIII Workshop on Requirements Engineering, WER*. [S.l.: s.n.], v. 5, p. 148–160. 2005.

[44] SANTOS, B. S. *IStar Tool-Uma proposta de ferramenta para modelagem de i**. Tese (Dissertação de Mestrado) — Centro de Informática, Recife, PE, outubro 2008.

[45] YU, E.; YU, Y. *Organization Modelling Environment*. 2013. Consultado na INTERNET: http://www.cs.toronto.edu/km/ome/, 2013.

[46] LEUF, B.; CUNNINGHAM, W. The wiki way: quick collaboration on the web. Addison-Wesley Professional, 2001.

[47] MYLOPOULOS, J.; KOLP, M.; CASTRO, J. Uml for agent-oriented software development: The tropos proposal. In: *The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. [S.l.]: Springer. p. 422–441. 2001.

[48] CHUNG, L. et al. Non-functional requirements. *Software Engineering*, 2000.

[49] REGEV, G.; WEGMANN, A. Where do goals come from: the underlying principles of goal-oriented requirements engineering. In: IEEE COMPUTER SOCIETY. *Proceedings of the 13th IEEE International Conference on Requirements Engineering*. [S.l.], p. 253–362. 2005.

[50] FOUNDATION, E. *Eclipse*. 2013. Consultado na INTERNET: http://www.eclipse.org/, 2013.

[51] HORKOFF, J.; YU, Y.; YU, E. Openome: an open-source goal and agent-oriented model drawing and analysis tool. In: *CEUR proceedings of the 5th international i* workshop (iStar 2011)*. [S.l.: s.n.], p. 154–156. 2011.

[52] CARES, C.; FRANCH, X. Towards a framework for improving goal-oriented requirement models quality. 2009.

[53] CARES, C. et al. Towards interoperability of *i** models using istarml. *Computer Standards & Interfaces*, Elsevier, v. 33, n. 1, p. 69–79, 2011.

[54] CARES, C. et al. istarml: An xml-based model interchange format for i*. In: *Proc. 3rd Int. i* Workshop, Recife, Brazil*. [S.l.: s.n.], v. 322, p. 13–16. 2008.

[55] KEALEY, J. et al. Integrating an eclipse-based scenario modeling environment with a requirements management system. In: IEEE. *Electrical and Computer Engineering, 2006. CCECE'06. Canadian Conference on*. [S.l.], p. 2432–2435. 2006.

[56] LóPEZ, L.; FRANCH, X.; MARCO, J. Hime: hierarchical i* modeling editor. *Revista de Informática Teórica e Aplicada*, v. 16, n. 2, p. 57–60, 2009.

[57] LAUE, R.; STORCH, A. Adding functionality to openome for everyone. In: *CEUR Proceedings of the 5th International i* Workshop (iStar 2011)*. [S.l.: s.n.], v. 766, p. 169–171. 2011.

[58] BRISCHKE, M. *Melhorando a Ferramenta JGOOSE*. Cascavel - PR: [s.n.], Dezembro 2012.

[59] BRISCHKE, M. *Desenvolvimento de uma Ferramenta para Integrar Modelagem Organizacional e Modelagem Funcional na Engenharia de Requisitos*. Cascavel - PR: [s.n.], Novembro 2005.

[60] JECKLE, M. Omg's xml metadata interchange format xmi. *XML4BPM*, p. 25–42, 2004.

[61] WONG, S. *StarUML Tutorial*. Connexions Web site, Sep 2007. Acessado em: 04 de junho de 2013. Disponível em: <http://cnx.org/content/m15092/1.1/>.

[62] OMG. *Unified Modeling Language: Superstructure*. [S.l.]: Object Management Group, Fevereiro 2007.

[63] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. [S.l.]: Addison-Wesley Professional. ISBN 0321267974. 2005.

[64] CLEMENTS, P. et al. *Documenting software architectures: views and beyond*. [S.l.]: Pearson Education. 2002.

[65] ALDER, G. Design and implementation of the JGraph swing component. *Techical Report*, v. 1, n. 6, 2002.

[66] PROJECT, T. L. I. *BSD License Definition*. 2005. Acessado em: 04 de junho de 2013. Disponível em: <http://www.linfo.org/bsdlicense.html>.

[67] JGRAPH. *JGraphX (JGraph 6) User Manual*. 2013. Acessado em: 04 de junho de 2013. Disponível em: <http://jgraph.github.io/mxgraph/docs/manual_javavis.html>.

[68] PRESSMAN, R. *Software Engineering: A Practitioner's Approach*. 7. ed. [S.l.]: McGraw-Hill Science/Engineering/Math. ISBN 0073375977. 2009.

[69] MAVEN, A. *Apache Maven Project*. 2013. Consultado na INTERNET: http://maven.apache.org/, 2013.

[70] FIALLI, J.; VAJJHALA, S. The Java architecture for XML binding (JAXB). *JSR, JCP, January*, 2003.

[71] ANDERSEN, T. J.; AMDOR, L. E. Leveraging maven 2 for agility. In: IEEE. *Agile Conference, 2009. AGILE'09.* [S.l.], p. 383–386. 2009.

[72] PREE, W. Design patterns for object-oriented software development. Addison Wesley Longman, 1994.

[73] ECKSTEIN, R.; LOY, M.; WOOD, D. *Java swing*. [S.l.]: O'Reilly & Associates, Inc. 1998.

[74] ORACLE. *JRE 7 - README*. Oracle, April 2013. Acessado em: 04 de junho de 2013. Disponível em: <http://www.oracle.com/technetwork/pt/java/javase/jre-7-readme-430162.html>.

[75] BINKLEY, D. et al. To camelcase or under_score. In: IEEE. *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. [S.l.], p. 158–167. 2009.

[76] SPINELLIS, D. Git. *Software, IEEE*, v. 29, n. 3, p. 100–101, 2012. ISSN 0740-7459.

[77] RODRIGUEZ-BUSTOS, C.; APONTE, J. How distributed version control systems impact open source software projects. In: IEEE. *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. [S.l.], p. 36–39. 2012.

[78] BACON, T.; MARTIN, J. *XMLUnit - JUnit and NUnit testing for XML*. April 2013. Acessado em: 04 de junho de 2013. Disponível em: <http://xmlunit.sourceforge.net>.

[79] HENNING, M. API design matters. *Queue*, ACM, New York, NY, USA, v. 5, n. 4, p. 24–36, may 2007. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1255421.1255422>.

[80] KRAMER, D. Api documentation from source code comments: a case study of javadoc. In: ACM. *Proceedings of the 17th annual international conference on Computer documentation*. [S.l.], p. 147–153. 1999.

[81] CARES, C. *CCISTARML (v0. 6): A Java Package for handling iStarML files*. [S.l.].

[82] BLOCH, J. *Effective JAVA*. [S.l.]: Addison-Wesley Professional. 2008.

[83] HAROLD, E. R. *Processing XML with Java*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN 0201771861. 2002.

[84] HUNT, A.; THOMAS, D.; PROGRAMMERS, P. *Pragmatic unit testing in Java with JUnit*. [S.l.]: Pragmatic Bookshelf. 2004.