UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Automatizando as Regras de Mapeamento entre a Modelagem *i** e a Modelagem UML usando XMI para Implementação de um Simulador de Rede Ópticas.

por

Flávio Pereira Pedroza

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para a obtenção do grau de Mestre em Engenharia Elétrica.

ORIENTADORA: Profa. Fernanda Maria Ribeiro Alencar, Doutora

Recife, Fevereiro de 2005.

P372a **Pedroza, Flávio Pereira.**

Automatizando as regras de mapeamento entre a Modelagem i* e a Modelagem UML usando XMI para implementação de um Simulador de Rede Ópticas. – Recife: O Autor, 2005.

xii, 103 folhas. : il. ; fig., tabs.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CTG. Engenharia Elétrica, 2005.

Inclui bibliografia.

1. Engenharia elétrica. 2. Engenharia de software. 3. Modelagem i* - Modelagem UML – Mapeamento. 4. Simulador de Redes Ópticas. I. Título.

621.3 CDD (22.ed.)

UFPE
BCTG/2007-009

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE DISSERTAÇÃO DE MESTRADO ACADÊMICO DE

FLÁVIO PEREIRA PEDROZA

TÍTULO

"AUTOMATIZANDO AS REGRAS DE MAPEAMENTO ENTRE A MODELAGEM i* E A MODELAGEM UML USANDO XMI PARA IMPLEMENTAÇÃO DE UM SIMULADOR DE REDES ÓPTICAS"

A comissão examinadora composta pelos professores: FERNANDA MARIA RIBEIRO DE ALENCAR, DES/UFPE; FREDERICO DIAS NUNES, DES/UFPE e JAELSON FREIRE BRELAZ DE CASTRO, CIN/UFPE, sob a presidência do primeiro, consideram o candidato **Flávio Pereira Pedroza APROVADO**.

Recife, 25 de fevereiro de 2005.

JOAQUIM FERREIRA MARTINS FILHO
Coordenador do PPGEE

FERNANDA MARIA RIBEIRO DE ALENCAR
Orientador e Membro Titular Interno

JAELSON FREIRE BRELAZ DE CASTRO
Membro Titular Externo

FREDERICO DIAS NUNES
Membro Titular Interno

AGRADECIMENTOS

Primeiramente, eu gostaria de agradecer a minha orientadora, Professora Fernanda Maria Ribeiro Alencar, pelo icentivo, interesse e aconselhamento dado ao meu trabalho. Aos professores Frederico dias Nunes e Jaelson Freire Brelaz de Castro pela sua participação na minha banca examinadora.

Ao professor Joaquim Ferreira Martins Filho e aos alunos Carmelo Bastos Filho e Eric de Albuquerque Jorge Arantes do Grupo de Fotônica pelo auxílio na especificação do simulador de redes ópticas. Ao CAESER e o CNPq pelo apoio financeiro e pla utlização de sua infra-estrutura para o desenvolvimento deste trabalho. Aos demais colegas do CAESER, pelas críticas e sugestões a este trabalho.

Finalmente, eu gostaria de agradecer à minha família pelo amor, paciência e apoio. A minha mãe, Maria das Neves Pereira Pedroza, e ao meu pai Dinilson Pedroza pela dedicação, compreensão e encorajamento.

Resumo da Dissertação apresentada à UFPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Engenharia Elétrica.

Automatizando as Regras de Mapeamento entre a Modelagem *i** e a Modelagem UML usando XMI para Implementação de um Simulador de Rede Ópticas.

Flávio Pereira Pedroza

Fevereiro/2005

Orientadora: Fernanda Maria Ribeiro Alencar, Doutora.

Área de Concentração: Engenharia de Software.

Palavras-chave: Engenharia de requisitos, XMI, Integração *i** e UML, Ferramenta de apoio, Integração de ferramentas CASE, Simulador de redes ópticas.

Número de Páginas: 111.

RESUMO: O bom entendimento dos requisitos organizacionais é vital para o sucesso do desenvolvimento de aplicações na área de engenharia de software. Com a popularização do paradigma da orientação à objeto, a linguagem de modelagem UML (Unified Modeling Language) se tornou padrão para este tipo de desenvolvimento. Porém, a UML ainda não está suficientemente estruturada para suportar a modelagem dos requisitos organizacionais. Faz-se necessário a utilização de outras técnicas de modelagem. A técnica de modelagem *i** supre esta deficiência, sendo uma técnica utilizada para a modelagem de requisitos de negócios, bem difundida e aceita. Com a utilização dessas duas técnicas, faz-se necessário um meio de mapearmos os elementos *i** em elementos UML. O mapeamento entre as duas técnicas foi realizado através de um conjunto diretrizes que ditam as regras pelas quais os elementos são mapeados. Neste trabalho apresentamos uma ferramenta de apoio às regras de mapeamento entre as técnicas *i** e UML: o eXtended GOOD (Goals into Object Oriented Development) ou simplesmente, XGOOD. Essa ferramenta realiza esse mapeamento de forma automática possibilitando a troca dos modelos gerados entre várias ferramentas de modelagem orientada a objetos existentes no mercado, através do uso do XMI (XML Metadata Interchange). Com o objetivo de testar e validar a nova ferramenta foi desenvolvido um simulador de redes ópticas como estudo de caso.

Abstract of Dissertation presented to UFPE as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

Automatizing the Rules of Mapping between Modeling i * and Modeling UML using XMI for Implementation of an Optic Network Simulator.

Flávio Pereira Pedroza

Fevereiro/2005

Supervisor: Fernanda Maria Ribeiro Alencar, Doutora.
Area of Concentration: Software Engeneering.
Keywords: Requirements Engineering, XMI, Integration *i** and UML, Suport tooll, CASE tools integration, Optical network simulator.
Number of Pages: 111.

ABSTRACT: The good understanding of the organizational requirements is vital for the success of the development of applications in the area of Software Engineering. With the popularization of the object oriented paradigm, the modeling language UML (Unified Modeling Language) has became standard for this type of development. However, the UML still is not enough equipped to support the modeling of the organizational requirements. The use of other modeling technique becomes necessary. The modeling technique *i** covers this deficiency, being one technique used for business-oriented modeling very well spread out and accepted. With the use of this two distinct techniques it is necessary a way to map *i** elements in UML elements. The mapping between the two techniques is carried through a set of guidelines that determine how the elements are mapped. In this work, we discuss a tool to support these mapping rules between the techniques *i** and UML: eXtended GOOD (Goals into Object Oriented Development) that carries through this mapping automatically and makes possible the exchange of the models between several objected oriented modeling tools of the market, through the use of the XMI (XML Metadata Interchange). To test and to validate the new tool, a optical network simulator will be developed as a case study.

SUMÁRIO

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

1. API - Application Program Interface
2. ATM – Asynchronous Transfer Mode
3. CASE - Computer-Aided Software Engineering
4. CORE - Controlled Requirements Expression
5. CML - Conceptual Modelling Language
6. DDL - Data Definition Language
7. DTD - Document Type Definition
8. DWDM - Dense Wavelenght Division Multiplexing
9. GOOD - Goals into Object Oriented Development
10. GUI - Graphical User Interface
11. HTML - HyperText Markup Language
12. IDE - Integrated Development Environment
13. IEEE – Institute of Electrical and Electronics Engineers
14. KAOS - Knowledge Acquisition in autOmated Specification
15. MAN - Mentropolitan Area Network
16. MFC - Microsoft Foundation Classes
17. MOF - Meta Object Facility
18. OCL - Object Constraint Language
19. OME - Organization Modeling Environment
20. OMG - Object Management Group
21. REI - Rose Extensibility Interface
22. SADT - Structured Analysis and Design Technique
23. SD – Strategic Dependency
24. SDH - Syncronous Digital Hierarchy
25. SONET - Synchronous Optical Network
26. SR – Strategic Rationale
27. UML - Unified Modeling Language
28. VORD - Viewpoint-oriented Requirements Definition.
29. VOSE - Viewpoint-oriented System Engineering
30. W3C - World Wide Web Consortium
31. WWW - World Wide Web
32. WAN - Wide Area Network
33. WDM - Wavelenght Division Multiplexing

34. XGOOD - eXtended Goals into Object Oriented Development
35. XML - eXtensible Markup Language
36. XMI - XML Metadata Interchange
37. XSD - XML Schema Definition
38. XSLT - eXtensible Stylesheet Language

1 Introdução

Hoje em dia, grande parte da população mundial depende dos aplicativos de *software* para realizar suas tarefas diárias [1]. Estes aplicativos de *software* possuem um grande potencial para trazer benefícios àqueles que os utilizam. Atividades antes executadas manualmente por pessoas, hoje são realizadas, em parte ou no todo, através de computadores, resultando na ampla melhoria da qualidade e rapidez dos serviços e produtos oferecidos pelas organizações a seus usuários.

Todavia, os aplicativos de *software* também tem grande potencial para trazer prejuízos. Problemas tais como: prazos e orçamentos estourados; sistemas que não fazem o que os usuários realmente querem, etc., vêm assolando o desenvolvimento de sistemas de *software* desde 1960 [2]. A qualidade do produto final, seja um aplicativo de *software* ou não, está fortemente ligada ao modo como foi desenvolvido.

A Engenharia de *Software* surgiu em meados dos anos 70 numa tentativa de contornar a crise do *software* e dar um tratamento de engenharia (mais sistemático e controlado) ao desenvolvimento de sistemas de software complexos e para obter softwares mais confiáveis, eficientes e passíveis de certificação [2]. Um sistema de software complexo se caracteriza por um conjunto de componentes abstratos de software (estruturas de dados e algoritmos) encapsulados na forma de procedimentos, funções, módulos, objetos ou agentes e interconectados entre si, compondo a arquitetura do software, que deverão ser executados em sistemas computacionais.

Segundo SOMMERVILEE [2], um dos fatores que contribuem para a baixa qualidade do produto final está relacionado com a elicitação e controle dos requisitos do sistema. Os requisitos do sistema definem que serviços o sistema deve prover e quais são as limitações do mesmo. Erros oriundos de uma má definição de requisitos, nos estágios iniciais do processo de desenvolvimento, podem resultar em altos custos na manutenção dos sistemas, em total rejeição do sistema e/ou em perdas econômicas, sociais ou ambientais. Alguns dos problemas bastante comuns relacionados aos requisitos do sistema são:

- os requisitos não refletem as reais necessidades dos clientes;
- os requisitos são inconsistentes e/ou incompletos;
- as mudanças nos requisitos são difíceis de serem feitas, devido a falta de rastreabilidade (quem solicitou o requisito; porque o requisito existe; que outros requisitos são afetados, etc.) dos requisitos;
- o desentendimento entre os *stakeholders* (clientes, desenvolvedores, gerentes, etc., ou seja, todos aqueles envolvidos de alguma forma com o desenvolvimento do sistema);

A Engenharia de Requisisto [2] é uma das áreas da Engenharia de *Software* que cobre todas as atividades envolvidas em descobrir, documentar e manter um conjunto de requisitos para um sistema de *software* qualquer. Ela auxiliada no projeto, implementação e manutenção de sistemas de *software.*

1.1 Motivação

A captura dos requisitos tem sido reconhecida como uma fase crítica do processo da Engenharia de Software. Nesta fase inicial, podemos determinar se o sistema proposto é viável e identificar os riscos do mesmo. Uma falha nessa etapa inicial pode acarretar um produto de baixa qualidade ou um produto entregue fora do prazo e por um custo maior que o esperado. A Engenharia de Requisitos surge como meio de minimizar esses efeitos. Na atividade de **análise e negociação** do processo de Engenharia de Software temos associada a modelagem dos requisitos. O objetivo da modelagem é obter um documento formal contendo uma definição oficial do que seja necessário aos desenvolvedores do sistema, o **documento de requisitos**. Diversas são as técnicas para a modelagem de requisitos existentes.

A modelagem de requisitos realiza importantes papéis [3]:

1. ajuda no entendimento da informação, da função e do comportamento de um sistema de *software*, tornando a análise de requisitos mais fácil e mais sistemática;
2. é realizada, através do modelo de requisitos, a revisão de requisitos, ou seja, são verificados a completeza, consistência e precisão dos requisitos;
3. é a base para o projeto do sistema a ser desenvolvido, fornecendo uma representação essencial do software e auxiliando o desenvolvedor a visualizar o sistema a ser desenvolvido;

O modelo de requisitos deve ser capaz de representar as informações que o sistema transforma; as funções do sistema que realizam estas transformações e o comportamento do sistema enquanto realiza estas transformações [3]. Os modelos apresentam uma notação gráfica, algumas vezes complementados com descrições em linguagem natural (linguaguem falada no dia a dia) ou alguma linguagem formal (uma linguagem de especificação), de modo a permitir um maior entendimento do modelo.

A técnica a ser utilizada para modelar os requisitos vai depender de seu tipo (funcional, não funcional, organizacional). Os requisitos funcionais definem as funções que o sistema deve executar. Os não funcionais definem as restrições do sistema, tais como, segurança, confiabilidade, desempenho, etc. Os requisitos organizacionais dizem respeito às metas da empresa, suas políticas estratégicas adotadas, os empregados da empresa com seus respectivos objetivos; enfim toda a estrutura da organização. Além disso, a técnica utilizada também pode variar de acordo com o

estágio de desenvolvimento do sistema. Tipicamente, os estágios iniciais necessitam de um modelo mais abstrato, que forneça uma visão geral e um entendimento do sistema a ser desenvolvido. Os estágios finais necessitam de um modelo mais concreto, que possibilite a geração de código fonte.

Em [3, 4] foi levantando o problema de se usar técnicas distintas para tipos de requisitos e diferentes fases do desenvolvimento. Foram propostas regras de mapeamento que possibilitem a transformação entre as diferentes técnicas utilizadas para modelar os requisitos. Essas regras utilizam como base duas técnicas: a i-estrela (*i**) [5, 6] e a Modelagem Unificada [7, 8, 9] (UML). A *i** se ajusta melhor aos estágios iniciais do desenvolvimento e a captura dos requisitos não funcionais e organizacionais. A UML se ajusta melhor aos estágios finais do desenvolvimento e para a captura dos requisitos funcionais. Foi proposta também uma ferramenta que auxilia na realização desse mapeamento de forma automática, com base nas regras de mapeamento propostas: o GOOD (*Goals into Object Oriented Development*) [10]. O GOOD foi construído de forma a ser integrada em uma única ferramenta comercial específica, de modo que é necessário que o usuário possuísse esta ferramenta comercial para utilizar o GOOD.

O objetivo desse trabalho é a construção de uma nova ferramenta que irá realizar o mapeamento entre os modelos *i** e UML de forma automática. Sendo esta nova ferramenta uma extensão da ferramenta GOOD demos o nome de XGOOD (*eXtended Goals into Object Oriented Development*). Assim, essa nova ferramenta apresentará algumas vantagens em relação à versão anterior:

1. opção de selecionar as diretrizes de mapeamento adequadas, ou seja, quais elementos serão mapeados - possibilita ao usuário excluir certos elementos capturados segundo a técnica *i**, mas que não são computacionais no modelo UML;
2. inclusão de novas diretrizes de mapeamento - o XGOOD suportará mais diretrizes (as diretrizes foram revistas em ALENCAR [11]) relacionadas a novos elementos capturados pelos modelos *i** (ex.: papel, posição e agente);
3. adoção de um padrão aberto para representar os modelos gerados – adotando um padrão aberto e reconhecido pela OMG [12] facilita o compartilhamento dos modelos entre diversas ferramentas comerciais disponíveis;

Como subobjetivo temos o desenvolvimento de um simulador para a especificação topológica e funcional de uma rede de comunicação que faz uso de meios ópticos. O diagrama de classes desse simulador será modelado segundo as regras de mapeamento em [11] e utilizando o XGOOD. Isso servirá como meio de validarmos o XGOOD.

1.2 Metodologia

Para o desenvolvimento da ferramenta, foi usado como base os trabalhos de desenvolvimento da ferramenta GOOD [10]. Inicialmente, as sugestões lançadas por CYSNEIROS [10] como trabalhos futuros (suporte a XMI, permitir a seleção dos elementos mapeados, etc.) foram analisadas e uma nova ferramenta aprimorada foi proposta. Em seguida, além dessas sugestões, trabalhou-se também no aprimoramento das diretrizes de mapeamento [11]. Nesse aprimoramento buscou-se dar suporte a mais elementos do modelo i* (atores, posição, papel) e do modelo UML (interfaces). Para realizar a persistência dos modelos gerados, foi utilizado o formato XMI (XML Metadata Interchange), que é um padrão proposto pela OMG [9], e tem se tornado um padrão de fato para troca de dados entre ferramentas CASE. Seguiu-se, então, um estudo do formato XMI, visualizando como os vários elementos do diagrama de classes são representados neste formato através de *tags* e atributos.

Para o desenvolvimento da ferramenta em si, buscou-se uma solução que permitisse maior flexibilidade, ou seja, que seu funcionamento e seu uso não estão amarrados a uma ferramenta CASE especifica, como a anterior [10]. Sendo assim, a solução adotada foi o uso da linguagem de programação C++, linguagem esta bastante robusta e utilizada há vários anos no mercado em desenvolvimento de *softwares* profissionais.

Atualmente, a ferramenta suporta cinco diretrizes, de um total de sete. Além disso, permite selecionar os elementos a serem mapeados e a diretriz de mapeamento mais adequada, pois alguns elementos do modelo i* possuem mais de uma opção de mapeamento. O diagrama de classes é salvo nos formatos XMI [13, 14] 1.0 ou 1.1, dependendo da escolha do usuário. Para o desenvolvimento do simulador de redes ópticas, primeiramente modelamos o sistema através da técnica i* e posteriormente geramos o diagrama de classes com o uso da ferramenta XGOOD. Este diagrama foi importando em uma ferramenta CASE comercial para realizarmos um refinamento do mesmo. O simulador também foi implementado utilizando-se a linguagem C++.

1.3 Estrutura

Esta dissertação está dividida em oito capítulos, sendo o primeiro esta introdução. Ao final de cada capítulo temos uma seção dedicada para comentários finais relativos ao mesmo.

No capítulo dois, introduziremos conceitos sobre engenharia de requisitos e sua importância no desenvolvimento de sistemas de *software*. Apresentaremos também a modelagem de requisitos e algumas técnicas de modelagem de requisitos. No capítulo três, veremos com mais

detalhes as técnicas de modelagem utilizadas neste trabalho. A técnica *i**, utilizada para a modelagem de negócio e captura inicial dos requisitos e a UML, utilizada para a modelagem dos requisitos funcionais, com ênfase no diagrama de classes. Também são discutidas as diretrizes de mapeamento, que são regras pelas quais é possível migrar de um modelo *i** para o diagrama de classes UML. No capítulo quatro, veremos as ferramentas de apoio que são utilizadas para a modelagem *i** e a modelagem UML, assim como uma ferramenta utilizada para auxiliar o mapeamento entre esses modelos: a ferramenta GOOD (*Goal Into Object Oriented Development*). No capítulo cinco, apresentaremos o padrão que permite expressarmos o diagrama de classes UML através do uso do padrão XMI (*XML Metada Interchange*) [13, 14]. Discutiremos o "porquê" do uso deste padrão e quais as suas vantagens. No capítulo seis, apresentaremos a nova ferramenta proposta neste trabalho: XGOOD (*eXtended Goal Into Object Oriented Development*), uma versão aprimorada da ferramenta GOOD que realiza de forma automática o mapeamento do modelo *i** para o diagrama de classes UML. São discutidas as vantagens em relação à ferramenta anterior e a justificativa da construção desta nova ferramenta. No capítulo sete, exemplificaremos o uso da nova ferramenta proposta através de um estudo de caso, onde um sistema proposto será modelo utilizando-se a técnica *i**. A partir deste modelo *i** iremos desenvolver o sistema final. O sistema desenvolvido é um simulador de redes óticas. Por fim, no capítulo oito teremos as conclusões finais e propostas de trabalhos futuros.

2 Modelagem de Requisitos

A Engenharia de *Software* é uma disciplina da Engenharia que se ocupa de todos os aspectos da produção de *sofwtare*, desde os estágios iniciais da especificação do sistema até a manutenção desse sistema [15].

Um produto de *sofware* tem certos atributos que refletem a sua qualidade. Esses atributos estão relacionados com: i) o seu comportamento quando em funcionamento (tempo de resposta rápido, facilidade de uso, uso dos recursos do sistema de modo eficiente); ii) a estrutura e a organização do código fonte; iii) a documentação associada (facilidade de manutenção, facilidade de entendimento do programa por outros desenvolvedores). A Engenharia de *Software* possui técnicas e métodos que direcionam para produção de um *software* com essas qualidades.

A definição dos requisitos é um passo primário e fundamental para o sucesso, em qualquer modelo de desenvolvimento de software. Um sistema mal especificado, mesmo que seja bem projetado e codificado, com certeza causará prejuízos e transtornos para o cliente e para os desenvolvedores. Por isso, devemos utilizar as metodologias oferecidas pela Engenharia de Requisitos, que veremos na seção adiante, para a definição e análise de requisitos.

2.1 Engenharia de Requisitos

Para a produção de produtos de *software,* os Engenheiros de *Software* executam um conjunto de atividades, com resultados associados, denominado processo de *software*. Existem quatro atividades fundamentais de processo comuns a todos os processo de *software*.

1. **especificação**: as funcionalidades do sistema a ser desenvolvido e suas restrições de operação são definidas;
2. **desenvolvimento**: o *software* é desenvolvido de acordo com a sua especificação definida;
3. **validação**: o *software* é validado para garantir que o mesmo realiza tudo aquilo que o cliente deseja;
4. **evolução**: o *software* deve evoluir para acomodar mudanças nas necessidades dos clientes;

Existem vários modelos de processo de desenvolvimento de *software*: modelo em cascata; desenvolvimento evolucionário; desenvolvimento em espiral, etc [15]. Mas todos começam com a especificação do *software* ou definição dos **requisitos.** Requisitos são descrições das funções e das restrições do sistema de *software* que será desenvolvido. De acordo com SOMMERVILEE [15] os requisitos devem descrever:

- facilidades a nível do usuário; por exemplo, um corretor de gramática e ortografia;

- propriedades muito gerais do sistema, por exemplo: o sigilo de informações;
- restrições específicas no sistema; por exemplo, o tempo de varredura de um sensor;
- restrições no desenvolvimento do sistema; por exemplo: a linguagem JAVA deverá ser utilizada para o desenvolvimento do sistema;

Os requisitos são frequentemente classificados como requisitos funcionais, não funcionais e organizacionais [15]. Os requisitos funcionais definem quais os serviços que o sistema deve oferecer, como o sistema deve se comportar a certas entradas e como deve se comportar em determinadas situações. Já os requisitos não funcionais definem as restrições de operação do sistema, como, por exemplo, restrições de velocidade, segurança, etc. Os requisitos organizacionais definem como o sistema a ser desenvolvido irá satisfazer aos objetivos da organização.

Para descobrir, analisar, verificar, documentar e gerenciar as funções e restrições temos a Engenharia de Requisitos. Segundo o IEEE [16], Engenharia de Requisitos corresponde ao processo de aquisição, refinamento e verificação das necessidades do cliente para um sistema de software, objetivando-se ter uma especificação completa e correta dos requisitos de software. Para SOMMERVILEE [15], a Engenharia de Requisitos é um processo que envolve todas as atividades exigidas para criar e manter o documento de requisitos do sistema. Umas das razões principais para se estudar e aplicar a Engenharia de Requisitos é a correlação existente entre os erros e inadequações do produto final, com as falhas ocorridas durante as fases de análise e especificação dos requisitos.

O Processo da Engenharia de Requisitos compreende as atividades de **elicitação**, **análise e negociação**, **documentação**, **validação** e **gerenciamento**. Na **elicitação** dos requisitos, os membros da equipe de desenvolvimento trabalham junto ao cliente e os usuários finais do sistema para descobrir informações sobre as funcionalidades do sistema, restrições de operação, etc. Na **elicitação** estão envolvidas toda e qualquer pessoa que, de forma direta ou indireta, exerca alguma influência sobre os requisitos do sistema. O termo *stakeholder* é utilizado para referir-se a esse tipo de pessoa.

Na **análise e negociação,** os requisitos são classificados, os conflitos entre os *stakeholders* são encontrados e solucionados e os requisitos são verificados para descobrir se eles são completos e consistentes. Se esses requisitos estão de acordo com o que os *stakeholders* esperam que o sistema a ser desenvolvido realize então ficam acordados.

Na **validação,** os requistos são verificados para descobrir se eles definem realmente o sistema que o cliente deseja. A diferença em relação a **análise e negociação** reside no fato de que a mesma trabalha com requisitos incompletos. A atividade de **validação** tem como saída um esboço completo do documento de requisitos.

O **gerenciamento** de requisitos ocupa-se com os problemas que podem surgir devido à mudanças nos requisitos. Mudanças de requisitos sempre podem ocorrer em sistemas de *software*,

seja devido a um problema que, inicialmente, não foi totalmente compreendido pelos desenvolvedores, seja pelo surgimento de novas tecnologias ou aperfeiçoamento nos sistemas relacionados

A fases de **análise** e de **elicitação** dos requisitos são fundamentais no processo de desenvolvimento de sistemas. Na verdade, a análise deve ser intercalada com a elicitação, pois, em geral, problemas são descobertos quando os requisitos estão sendo elicitados. O custo de se consertar um erro identificado nessas fases de **elicitação** e **análise** de requisitos (etapas iniciais do processo de *software*) é muito menor do que aquele em que o erro é identificado nas etapas finais do desenvolvimento [15]. Deve existir, então, grande concentração de esforços nessa etapa do desenvolvimento, melhorando de forma contínua os processos formais a serem executados durante essa fase, como forma de aumentarmos a qualidade do software desenvolvido, bem como, minimizar o custo desse desenvolvimento.

Associada à análise tem-se a modelagem dos requisitos. De forma geral, um modelo é, normalmente, fruto de uma linguagem de representação de requisitos ou combinação de linguagens de representação de requisitos (linguagens naturais, rigorosas ou formais). Os requisitos do sistema devem ser expressos de maneira o mais técnica possível, de modo a eliminar ambigüidades.

2.2 Modelagem

Modelos são como plantas arquiteturais para os sistemas de *software.* Um modelo ajuda o desenvolvedor a planejar um sistema antes de construí-lo. Ele auxilia na confecção do projeto, na verificação do atendimento dos requisitos e possibilita que o sistema possa suportar mudanças nos requisitos sem maiores problemas [15]. Os modelos também auxiliam na análise dos requisitos elicitados.

À medida que os requisitos do sistema são capturados, as necessidades dos usuários pdem ser mapeadas em requisitos que possam ser entendidos pelos desenvolvedores de forma não ambígua. Para isso se utiliza de recursos gráficos em uma linguagem padrão universal. Os requisitos então são refinados e trabalhados para, nas fases de implementação do processo, possam finalmente ser transformados em código fonte. Deste modo, assegura-se que todos os requisitos são atendidos e que o código fonte gerado pode ser rastreado de volta para os requisitos. Este processo é chamado de **modelagem**.

Modelagem é o processo de tomar a informação do modelo e mostrá-la graficamente utilizando um conjunto padrão de elementos gráficos. A adoção de um padrão assegura a comunicação entre usuários, desenvolvedores, analistas, testadores, gerentes, etc. A modelagem visual ajuda facilita o entendimento de sistemas complexos, tanto por partes dos usuários como por parte dos desenvolvedores.

O processo de modelagem possibilita o rastreamento das necessidades do negócio para requisitos, dos requisitos para o modelo e do modelo para o código. Ou seja, podemos determina qual foi o requisitos que originou determinada função ou classe no código fonte e qual o impacto que uma alteração nesses requisito irá causar no código fonte. Além disso, possibilita também o processo inverso, sem a existência de perdas no caminho.

O tipo de técnica utilizada para a modelagem dos requisitos depende do tipo de requisito (**funcional**, **não funcional** ou **organizacional**) e também do estágio de desenvolvimento do *software*. Nos estágios iniciais temos um modelo em que os requisitos fornecem uma visão geral (*early requirements*) e um entendimento do sistema a ser desenvolvido. Os estágios finais necessitam de um modelo mais concreto (*late requirements*), que possibilita a geração de código fonte. Nas seções seguintes veremos algumas técnicas de modelagem de requisitos, cada uma com um perfil que se ajusta melhor a determinado tipo de requisito. Começaremos com a técnica SADT de 1972, mostrando uma evolução das técnicas de modelagem até as utilizadas hojem em dia.

2.2.1 Técnica de Análise e Projeto Estruturado - SADT

A SADT (*Structured Analysis and Design Technique*) [17, 18] foi originalmente desenvolvida em 1972 pela Softech Inc. Foi desenvolvida para descrever um sistema complexo e controlar o desenvolvimento de softwares complexos através de um método sistemático de definição de requisitos [19]. No SADT, a definição de requisitos é realizada através de uma série de passos que determinam o porquê do sistema a ser necessário, o que o sistema irá fazer quando for construído e como o sistema será construído. Os requisitos são definidos através da identificação das funções necessárias e desenvolvendo uma implementação que realiza estas funções, de modo que os requisitos sejam satisfeitos.

A técnica tem sido adotada para descrever sistemas complexos que podem incluir uma combinação de *hardware*, *software,* e pessoas. Um modelo SADT é uma coleção de diagramas organizados em uma estrutura em forma de árvore. O diagrama no topo da árvore representa a descrição mais geral do sistema. O diagrama no fim da árvore contém a descrição mais detalhada do sistema. Cada diagrama SADT é composto de caixas representando todas as funções do sistema (atividades, ações, processos, operações). As *interfaces* das caixas são representadas por setas (Figura 1). No diagrama, à esquerda da caixa estão, representados por setas entrantes, os recursos que são entradas da função. À direita da caixa temos a saída das funções (os recursos gerados), representada por setas que saem da caixa. Na parte de baixo estão localizadas as pessoas ou equipamentos que irão realizar a função (setas entrantes). Na parte de cima estão localizados os controles, que definem as condições para a realização da função (setas entrantes).

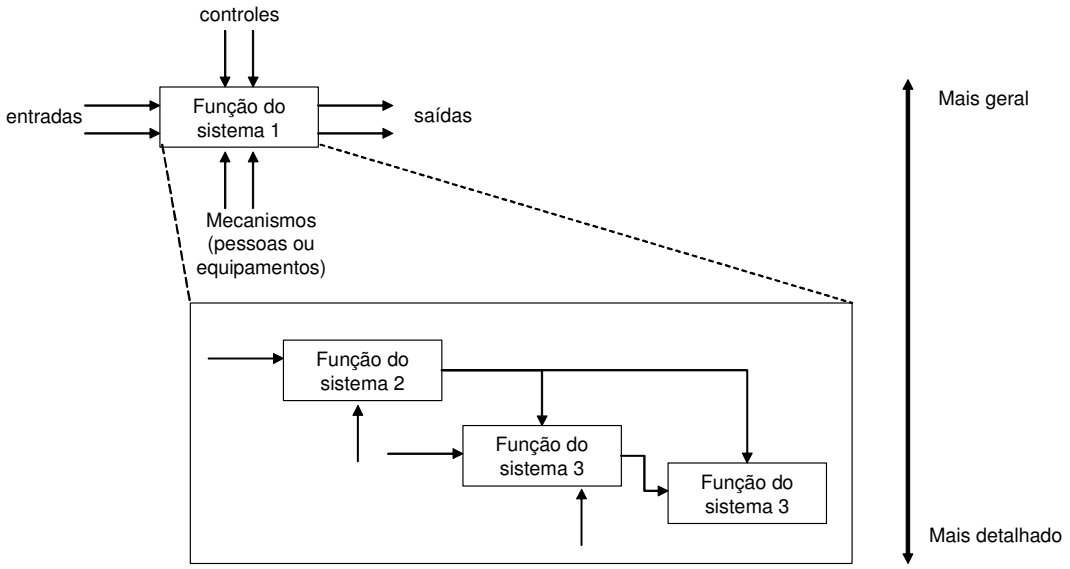

Figura 1 - Modelo de estrutura do SADT

No SADT, a análise do sistema começa com uma única caixa, mostrando as *interfaces* para funções e recursos que estão localizadas fora do sistema a ser desenvolvido. Esta caixa é expandida em módulos que fazem parte deste módulo "pai", representados também como caixas com *interfaces*. Cada módulo por sua vez é decomposto em outros módulos. Os módulos são decompostos em outros módulos e assim sucessivamente, proporcionando a cada decomposição um incremento cada vez maior no nível de detalhamento. Na Figura 1 o módulo *Função do sistema 1* foi decomposto em três outros módulos. As regras para a decomposição dos módulos estão definidas no SATD. Como regra geral, é recomendado que um modulo seja dividido em não mais que seis e não menos que três outros módulos.

O modelo final contém a especificação de requisitos do sistema a ser desenvolvido, mostrando as funções do sistema, as entradas, saídas e controles das funções, e fluxo lógico de informação e recursos entre as funções. Esta técnica se ajusta melhor ao estágio inicial de desenvolvimento de um projeto, fase de definição dos requisitos. Infelizmente esta só mostra o fluxo de entrada e saída dos recursos entre funções. Ela não permite mostrar como é feita a execução das funções no tempo (fluxo seqüencial das funções). Além da SADT, Existem outras técnicas similares que se possibilitam a modelagem dos requisitos de forma estrutural: O CORE (*Controlled Requirements Expression*) [20], VOSE (*Viewpoint-oriented System Engineering*) [21], VORD (Viewpoint-oriented Requirements Definition) [22].

2.2.2 Modelagem Orientada a Objetos

No paradigma de orientação a objetos as aplicações são vistas de uma forma diferente da forma tradicional (programação procedural). Com o uso da orientação a objetos, uma aplicação é

dividida em vários pedaços pequenos, chamados de objetos. Um aplicativo pode ser construído juntando-se esses objetos. Uma das vantagens do paradigma de orientação a objetos é a capacidade de construir componentes uma vez e utilizá-los várias vezes.

Em sistemas orientados a objetos, as informações são combinadas com um comportamento específico que age sobre essas informações. Isto é chamado de **encapsulamento.** No **encapsulamento** a aplicação é dividida em pequenas partes. Por exemplo, se tivermos informações relativas a uma conta bancária, como número da conta, saldo, nome do cliente, endereço, tipo de conta, etc; também temos comportamentos para esta conta bancária: abrir, fechar, depositar, retirar, mudar o tipo de conta, mudar dados do cliente. O comportamento e as informações da conta bancária são encapsulados em um objeto **conta.** Com o **encapsulamento** podemos restringir acesso não autorizado a determinados itens de informação ou a características de funcionamento da classe. Podemos definir níveis de visibilidade a esses elementos (visíveis a todos, apenas internamente ao objeto). Desse modo podemos proteger informações que devem ser secretas ou que não podem ser modificadas de forma aleatória.

Outro conceito fundamental da orientação a objeto é a **herança**. Em sistemas orientados a objetos, **herança** é o mecanismo que nos permite criar novos objetos baseados em outros já existentes. Um objeto **filho** herda as informações de um objeto **pai** (como no mundo real), podendo acrescentar qualidades adicionais.

Entre os principais benefícios do uso da **herança** temos o reuso. O reuso acontece quando queremos implementar um novo objeto que é semelhante a uma já existente. Podemos aproveitar a implementação anterior para evitar que tenhamos que reescrever tudo a partir do zero. Além disso, o usa da herança facilita a manutenção. No caso de termos um sistema grande com centenas de objetos diferentes e quisermos adicionar algo a todos eles, se todos eles tiverem um objeto **pai** em comum, bastaria modificar somente este objeto uma única vez e todos os objetos **filhos** irão **herdar** esta modificação. Sem o uso da **herança**, teríamos que modificar cada um dos objetos **filhos** em separado.

A orientação a objetos possui ainda outros conceitos como **polimorfismo** (possibilidade de termos várias implementações diferentes de uma mesma funcionalidade)**,** ***overriding*** (quando uma classe **filha** possui uma operação ou função com a mesma assinatura da classe **pai**, mas implementada de uma maneira diferente), ***overload*** (possibilidade de termos várias operações com o mesmo nome, porém com assinaturas diferentes).

A primeira linguagem orientada a objetos foi a Simula-67, desenvolvida em 1967. Esta linguagem nunca teve grande aceitação, embora tenha influenciado muita das linguagens orientadas a objetos que vieram a seguir. O paradigma de orientação a objetos só veio a ganhar impulso com o surgimento da linguagem Smaltalk, no início da década de 80. Seguindo seu rastro vieram outras

linguagens como C++, Objective C, Eiffel e CLOS. Logo foram publicados os primeiros métodos de desenvolvimento voltados para a orientação a objetos.

Com o passar dos anos, surgiram vários livros sobre metodologias orientadas a objetos, cada um com o seu próprio conjunto de conceitos, notações, terminologias e processos. Em geral, os autores possuíam conceitos chaves comuns, porém com algumas discrepâncias e expressos de forma ligeiramente diferente. Esta heterogeneidade dificultava e desencorajava a adoção desta nova tecnologia de orientação a objetos por novos usuários.

Houve tentativas iniciais de se unificar os conceitos. A primeira tentativa com sucesso aconteceu quando Rumbaugh [23] juntou-se a Booch [24] na Rational Software Corporation em 1994. Eles começaram a combinar os conceitos dos métodos do OMT [23] e de Booch [24], resultando numa primeira proposta em 1995. Um ano depois Jacobson [25] se associou a Rational e se uniu a Booch e Rumbaugh. O resultado desse trabalho conjunto ficou conhecido como UML - Unified Modeling Language.Specification.

Em 1996, a OMG (*Object Management Group*) [12] lançou uma requisição por propostas para um método padrão para modelagem orientada a objetos. Os autores da UML, Jacobson, Booch e Rumbauch, começaram a trabalhar com metodologistas e desenvolvedores de outras companhias para tornar a UML atrativa aos membros da OMG. Finalmente, a proposta final da UML, que foi o produto da colaboração de várias pessoas, foi enviada a OMG em setembro de 1997. A UML foi aceita pela OMG como padrão em novembro de 1997 [7], que assumiu a responsabilidade pelo futuro desenvolvimento da mesma. A UML é a notação padrão para o desenvolvimento de sistemas orientados a objetos.

2.2.3 Modelagem Organizacional

Durante o processo de modelagem organizacional, examinamos a estrutura da organização e olhamos os papéis dentro da organização e como eles se relacionam [8]. Examinamos também os fluxos da organização, identificamos quais os principais processos, como eles trabalham e se são realmente eficientes. Também examinamos as entidades ou indivíduos que não pertencem a organização, mas que interagem com a mesma, e as implicações desta interação.

A modelagem organizacional nos ajuda a entender a estrutura e a dinâmica da organização representando os seus objetivos, processos, recursos, regras de negócio; e a garantir que os clientes, usuários finais e desenvolvedores tenham um entendimento comum da organização. Ela também auxilia na elicitação dos requisitos do sistema, necessários para suportar a organização.

Existem várias técnicas de modelagem que nos auxiliam a modelar e elicitar os requisitos organizacionais, tais comos:

- KAOS (*Knowledge Acquisition in autOmated Specification*) [26, 27]: é uma técnica para aquisição de requisitos orientados a objetivos, que fornece uma linguagem formal, rica para a captura de requisitos funcionais, não-funcionais e organizacionais, capaz de fornecer facilidades para a descrição de objetivos, agentes, alternativas, eventos, ações, modalidade de existência e responsabilidades O KAOS é projetada para suportar todo o processos de elaboração dos requisitos – desde os objetivos a serem alcançados até os requisitos, objetos e operações a serem atribuídas aos vários agentes do sistema. A técnica KAOS fornece uma linguagem de especificação formal para captura dos requisitos, um método de elaboração dos requisitos e uma ferramenta de suporte;
- a técnica proposta por Bubenko [28] afirmar que para o modelo da organização (*enterprise model*) ser entidido pelos diversos gurpos de *stakeholders* (gerentes, clientes, programadores, etc.) o mesmo deve ser constítuido de componentes de conhecimento que abragem os diversos aspectos e atividades da Engenharia de Requisitos. O modelo da organização é constituído de diferentes submodelos: Modelo de Objetidos (*Objectives Model* – OM), Modelo de Atividas e Usos (*Activities & Usage Model* - AUM), Modelo de Atores (*Actors Model* – AM), Modelo de Conceito (*Concept Model* – CM) e Modelo de Requisitos do Sistema de Informação (*Information System Requirements Model* – IRSM);
- a técnica *i** [5, 6] é um método de modelagem de requisitos orienta a agentes. Possui dois modelos: O modelo de Dependência Estratégica (*Strategic Dependency* - SD), que mostra uma rede de dependência entre os atores do sistema, e o modelo de Razão Estratégica (*Strategic Rationale* - SR), que olha o ator internamente, mostrando as razões por trás dos relacionamentos de dependência entre os atores;
- a UML (*Unified Modeling Language*) [7, 8, 9] suporta a modelagem organizacional através de do **diagrama de casos de uso** (um dos vários diagramas da UML). Esse diagrama é utilizado para criar o **modelo de casos de uso de negócio**. O modelo consiste dos **atores do negócio** (*business actors* – representam papéis externos a organização, por exemplo, clientes), **trabalhadores do negócio** (*business workers* – representam um mais papéis em um negócio), **entidades do negócio** (*business entities* – representam alguma coisa que o trabalhador acessa, inspeciona, manipula, produz ou usa num caso de negócio) e **casos de uso do negócio** (*business use cases*) da organização. Esse modelo descreve como cada **trabalhadores do negócio** utiliza uma ou mais entidades do negócio para realizar um **caso de uso de negócio**;

A modelagem de negócio se ajusta melhor aos estágios iniciais do desenvolvimento. Ela deve vir como primeiro passo no processo de desenvolvimento, pois o modelo de negócios irá ditar o contexto do sistema a ser desenvolvido durante o resto do projeto.

2.3 Conclusões

Vimos que existem diferentes técnicas utilizadas para a modelagem dos requisitos funcionais, não funcionais e organizacionais e que uma determinada técnica pode ser ajustar melhor aos estágios finais ou iniciais do processo de desenvolvimento de *software*. Nesse trabalho utilizamos a técnica *i** para a modelagem dos requisitos não funcionais e dos requisitos organizacionais, os chamados requisitos nos estágios iniciais do desenvolvimento (*early requirements*). Para a modelagem dos requisitos funcionais nos estágios posteriores do desenvolvimento (*late requirements*), utilizamos a UML. No capítulo seguinte, vermos em maior detalhes essas duas técnicas e como realizar o mapeamento entre uma modelagem de requisitos através da técnica *i** e uma modelagem de requisitos utilizando a UML.

3. Técnicas de Modelagem

Vimos no capítulo anterior que existem diferentes tipos de requisitos: requisitos funcionais (que definem as funções que o sistema deve executar), não funcionais (que definem as restrições do sistema, tais como, segurança, confiabilidade, desempenho, etc.) e requisitos organizacionais (que dizem respeito à toda a estrutura da organização). Existem técnicas de modelagem de requisitos diferentes para os diferentes tipos de requisitos e também que se ajustam melhor a uma determinada etapa do desenvolvimento.

Neste capítulo iremos ver com maior detalhe duas técnicas de modelagem que foram introduzidas no capítulo anterior. A técnica *i**, utilizada para a modelagem dos requisitos organizacionais ou requisitos iniciais, e a UML, utilizada para a modelagem dos requisitos funcionais.

3.1 Técnica i-estrela (i)*

Na técnica de modelagem *i** as organizações são vistas como se consistissem de unidades semi-autônomas denominadas atores, ou seja, é uma técnica orientada a agentes [5, 6] , que foca nos relacionamentos entre os atores e suas dependências. Consiste de dois modelos: o modelo de Dependência Estratégica (SD - *Strategic Dependency*), que é utilizado para descrever os relacionamento de dependência entre os vários atores em um contexto organizacional; e o modelo de Razões Estratégicas (SR - *Strategic Rationale*), que serve para descrever os interesses e preocupações dos atores envolvidos e as razões por trás da adoção de uma determinada configuração.

Os atores são vistos como tendo propriedades intencionais, tais como objetivos, opiniões, habilidades e compromissos. Os atores dependem uns dos outros para cumprir objetivos, realizar tarefas e fornecer recursos. Através da cooperação de outros, um ator pode alcançar objetivos que serão difíceis de serem alcançados sozinho. Os atores são estratégicos no sentido em que os mesmos não estão interessados em apenas alcançar seus objetivos, mas também estão preocupados com as oportunidades e vulnerabilidades do meio ambiente.

Enquanto a maioria das outras técnicas de modelagem se concentra nos aspectos comportamentais do processo, deixando de lado as razões ou motivações que estão associadas aos comportamentos, a *i** fornece conceitos para modelar e responder questões como:

- por que atores executam os processos?
- quem deseja que eles façam isso?
- quais são as formas alternativas de executar um processo?
- por que os atores possuem ou recebem informação?

A *i** permite obter uma melhor compreensão sobre os relacionamentos da organização, entre os vários atores do sistema e entender as razões envolvidas nos processos de decisões. Essa técnica possui várias características de modelagem que podem ser apropriadas às etapas iniciais do desenvolvimento (especificação dos requisitos), como, por exemplo, a possibilidade de modelar dos requisitos não funcionais. Pode ser aplicada [6] nas áreas de Engenharia de Requisitos, reengenharia de processos de negócio, modelagem de processos de software e desenvolvimento de sistemas orientados a agentes. Nas próximas seções serão vistos os dois modelos cobertos pela técnica i*.

3.1.1 Modelo de Dependências Estratégicas - SD

O modelo de Dependência Estratégica é uma rede de relacionamentos de dependência entre atores. O modelo SD foca na captura da estrutura intencional de um processo, ou seja, na captura das motivações e intenções por trás das atividades e fluxos em um processo. Possui um nível alto de abstração na caracterização de um processo, pois captura somente os interesses dos atores, deixando de lado os detalhes não essenciais.

O modelo consiste numa série de nós e *links*. Cada nó representa um ator. Um ator é uma entidade ativa que realiza ações para atingir determinados objetivos. Cada *link* entre dois atores indica que um ator depende do outro para, de alguma forma, cumpri algum objetivo, tarefa ou fornecer um recurso. O ator que depende de outro é chamado de *depender* e o ator responsável por cumpri a dependência é chamado de *dependee*. O objetivo da dependência é denominado *dependum*.

Através da dependência, o *depender* é capaz de alcançar objetivos ou realizar tarefas que não seria capaz de alcançar sozinho. Porém, o *depender* se torna vulnerável, pois se o *dependee* não for capaz de fornecer o *dependum* ele poderá não alcançar o objetivo.

Figura 2 - Dependência do tipo objetivo.

Existem quatro tipos de dependências: dependência de **objetivo** (*goal*), dependência de **tarefa** (*task*), dependência de **recurso** (*resource*) e dependência de **objetivo *soft*** (*sof goal)*.

A dependência de **objetivo** e uma dependência do tipo afirmação (*assertions*). Uma afirmação expressa um estado ou condição sobre o meio ambiente. Nesse tipo de dependência, o

depender depende do *dependee* para alcançar um objetivo, sendo o *dependum* o objetivo a ser alcançado. Na Figura 2 temos um exemplo deste tipo de dependência. O *depender Usuario* possui uma relação de dependência com o *dependee Logica*. O ator *Usuario* necessita do ator *Logica* para cumprir o objetivo *Realizar simulacao*. O ator *Usuario* está interessado apenas em cumprir a sua meta, dos detalhes de como esse objetivo será alcançado diz respeito apenas ao ator *Logica*.

Na noção de dependência de **objetivo** descrita acima, os objetivos possuem dois estados bem distintos: objetivo alcançado e objetivo não alcançado. Muitas vezes os objetivos a serem atingidos podem não ser muitos bem definidos. Nesse caso, não existe acordo entre os atores envolvidos sobre o que seria realmente o cumprimento deste objetivo, ou seja, não se sabe ao certo quando o objetivo é antigido ou não. A satisfação desse tipo de objetivo é medida por meios subjetivos. Esse tipo de objetivo é chamado de **objetivo *soft***. Na Figura 3 temos um exemplo deste tipo de relacionamento. O ator *Usuario* necessita cumprir o **objetivo *soft*** *Facil usabilidade* (ou seja, o sistem deve possuir uma interface amigável para o usuário). O ator *Simulador* é responsável por cumprir esse objetivo.

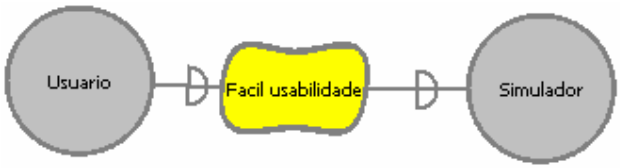

Figura 3- Dependência do tipo objetivo *soft*.

Em uma dependência de **tarefa**, o *depender* necessita que o *dependee* realize alguma tarefa. Essa é uma dependência do tipo atividade (*activity*). Uma atividade produz uma mudança no meio ambiente. A dependência especifica como a tarefa será executada, mas não a razão de sua execução. No exemplo da Figura 4, o *Simulador* necessita que o *Usuario* execute a tarefa *Montar topologia*.

Figura 4 - Dependência do tipo tarefa.

Em uma dependência do tipo **recurso**, o ator *depender* precisa que o ator *dependee* forneça uma entidade (*entitie*). Entidades são usadas para modelar objetos em um meio ambiente. Podem ser algo físico ou apenas uma informação, (Figura 5). Na Figura 5 ator *Visao* (*depender*) depende que o ator *Logica* (*dependee*) forneça o recurso *Resultados da Simulacao*. Não são explicados o porquê desta dependência ou como o recurso será gerado.

Figura 5 - Dependência do tipo recurso.

No modelo SD podemos definir também a "força" da dependência. Quanto maior o grau de dependência, mais vulnerável o *depender* se torna dependente do *dependum* (recurso, objetivo ou tarefa). Para o *dependee*, um grau maior de dependência implica em um maior esforço para disponibilizar o *dependum*. Existem três graus de dependência possíveis: aberta (*open*), comprometida (*committed*) e crítica (*critical*).

Uma dependência aberta para o *depender* significa que a falha em obter o *dependum* pode afetar negativamente os seus objetivos, mas não de forma crítica. Para o *dependee,* uma dependência aberta significa que o mesmo não terá problemas em disponibilizar o *dependum*. Já em uma dependência comprometida, o *depender* será significativamente afetado se houve falhar em obter o *dependeum*. No caso do *dependee*, uma dependência comprometida indica que o mesmo fará o melhor possível para entregar o *dependum*. Esse é o grau de dependência padrão. Finalmente, uma dependência crítica indica que o *depender* possui objetivos que serão seriamente prejudicados caso o *dependee* falhe no fornecimento do *dependum*. Graficamente, são utilizados os símbolos "**O**" para indicar uma dependência aberta e "**X**" para indicar uma dependência crítica.

O modelo SD distingue três tipos de atores: **agente**, **papel** e **posição**. Cada uma dessas subunidades é uma especialização ou refinamento do ator genérico. **Papel** é uma caracterização abstrata do comportamento de um ator social dentro de um contexto especializado. Essas características são facilmente transferíveis para outros atores. Um **agente** é um ator com uma manifestação física concreta, como, por exemplo, um indivíduo ou um sistema de *hardware* e/ou *software*. Essas características tipicamente não podem ser transferidas facilmente para outros. Uma **posição** possui um nível de abstração intermediário entre um papel e um agente. Denota um conjunto de papéis que são desempenhados por um agente. Dizemos que um agente ocupa uma posição e que uma posição cobre um papel (Figura 6).

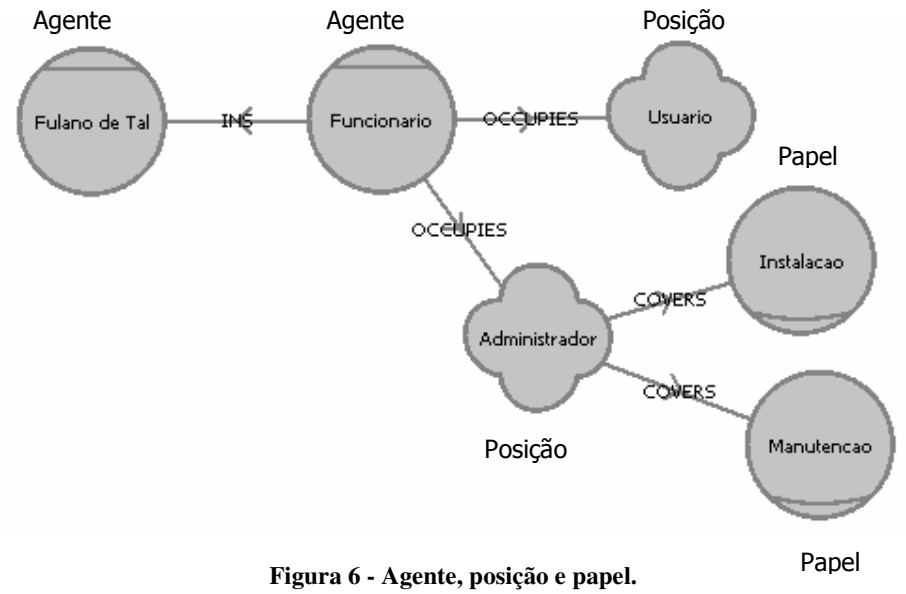

Figura 6 - Agente, posição e papel.

Na Figura 6 temos um agente *Funcionario*, que pode ocupar (*occupies)* duas posições: *Usuario,* e *Administrador*. A posição *Administrador* por sua vez cobre (*covers*) *dois* papéis: *Instalacao* e *Manutencao*. Vemos pela figura acima que as especializações do ator possuem símbolos gráficos diferenciados. Além das relações *occupies* entre agente e posição e *covers* entre posição e papel, temos a relação *ins*, entre agentes, posições e papéis e a relação *plays* entre um agente e um papel. A relação *ins* indica uma instanciação. No nosso caso, *Fulano de Tal* é um instancia do agente *Funcionario*. Na Figura 7 temos um exemplo de uma relação do tipo *plays*. Este tipo de relacionamento indica que um agente (*Fulano de Tal*) está desempenhado um determinado papel (*Gerente*) dentre da organização.

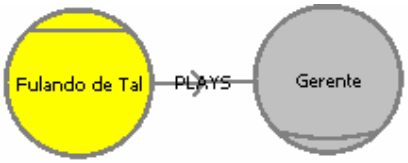

Figura 7 - Relação *plays* .

Papéis, agentes e posições podem possuir sub partes. Isso é indicado por uma relação do tipo *is-part-of*, (Figura 8). A posição *Simulador* é composta de outras três posições: *Locigl, Documento* e *Visao*. Os atores, quer sejam parte de um todo, quer sejam compostos de outros, são intencionais. Um ator intencional não executa apenas atividades e consome/produz entidades, mas também possui motivações, intenções e razões associadas a suas ações. Os atores envolvidos no relacionamento *is-part-of* possuem liberdade e são, portanto, imprevisíveis. Podem existir relações de dependência entre o todo e suas partes.

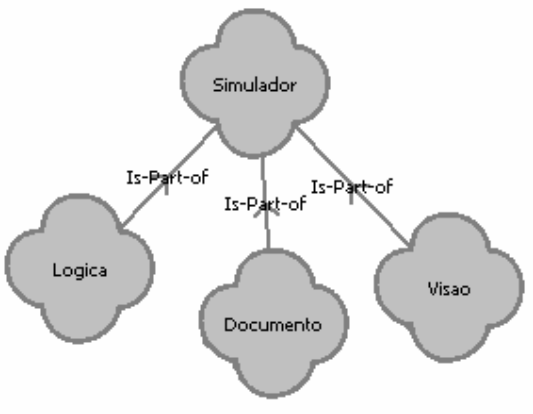

Figura 8 - Relação *is-part-of*.

Por fim, temos o relacionamento IS-A, que indica uma generalização ou especialização de atores (Figura 9). Na Figura 9, vemos que agentes, papel e posição são especializações de um ator.

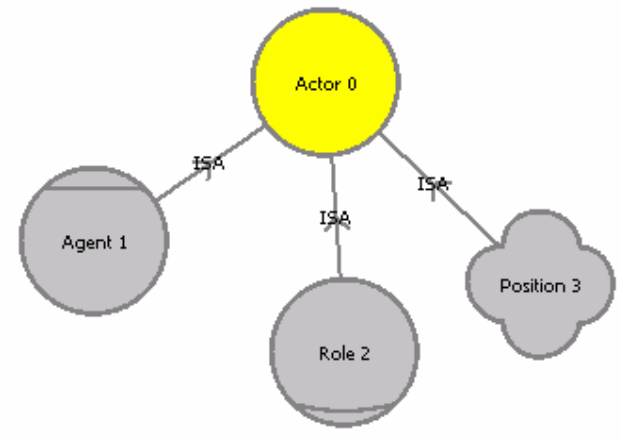

Figura 9 - Relação *IS-A*

3.1.2 Modelo de Razões Estratégicas - SR

O modelo de dependência estratégica mostra somente os relacionamentos externos entre os atores. No modelo de razão estratégica (*Strategic Rationale model – SR)*, temos uma representação e uma racionalização mais explícita sobre os interesses dos atores, e de como estes interesses podem ser atendidos ou afetados pelos diversos sistemas. Portanto, o modelo SD possui um nível de abstração maior, enquanto que o modelo SR diminua esta abstração de modo a permitir um melhor entendimento sobre os atores estratégicos.

No modelo SR temos uma visão interna dos atores, proporcionando um nível de detalhamento maior do modelo. Os elementos intencionais (**objetivos**, **tarefas**, **recursos** e

objetivos *soft*) aparecem no modelo SR como elementos internos ligados por relacionamentos meio-fim e decomposição de tarefas. Estes relacionamentos proporcionam uma representação explicita das razões por trás das dependências entre os atores e quais são alternativas dos processos.

Como no modelo SD, no modelo SR também temos um grafo com nós e *links*. Existem quatro tipos de nós, da mesma forma que no modelo SD: **objetivos**, **tarefas**, **recursos** e **objetivos *soft***. Porém, são introduzidas novas classes de ligações: ligações meio-fim, decomposição de tarefas e contribuição para ***objetivos soft***.

Uma ligação meio-fim indica um relacionamento entre um fim (que pode ser um **objetivo** a ser alcançado, uma **tarefa** a ser cumprida, um **recurso** a ser produzido ou um **objetivo *soft*** a ser satisfeito) e um meio para alcançar este fim. O meio é geralmente expresso na forma de uma tarefa, que incorpora a noção de como fazer alguma coisa. Por exemplo, na Figura 10 podemos cumprir o objetivo *Acesso ao sistema* de duas formas: acessando via terminal na própria máquina (realizando a tarefa *Acessar via terminal*) ou realizar um acesso remoto via *WEB* (realizando a tarefa *Acessar via WEB*).

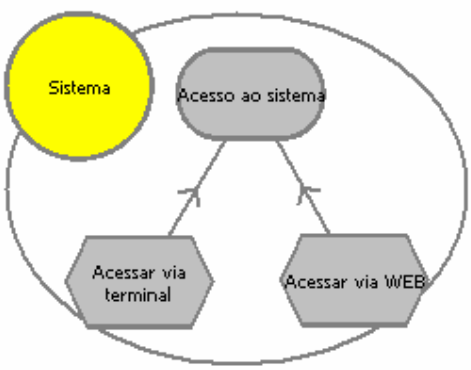

Figura 10 - Relacionamento de meio-fim

Uma ligação de decomposição de tarefa é utilizada para descrever os componentes de uma tarefa. Uma tarefa pode ser decomposta em objetivos, tarefas, recursos e objetivos *soft*. Esses elementos também podem ser conectados com ligações de dependência no modelo de Dependência Estratégica.

Quando temos um objetivo como um componente de uma tarefa, isto significa que uma certa condição deve ser alcançada para permitir a execução desta tarefa. No caso de termos uma tarefa como um componente de outra tarefa, essa tarefa dever ser executada para que a outra seja executada. Se o componente for um recurso, o mesmo deve ser fornecido para que a tarefa seja executada (Figura 11). Finalmente, se tivermos um objetivo *soft* como um componente de uma tarefa, o mesmo serve como um objetivo de qualidade para a tarefa, guiando a seleção das alternativas disponíveis. Observando a Figura 11, vemos que a tarefa *Simular* necessita para a sua execução o fornecimento de três recursos. *Arquivo Topologia da Rede, Arquivo Configuracao da*

Simulacao, Arquivo dos Comprimento de Onda. A falha no fornecimento de um desses recursos impede a execução da tarefa.

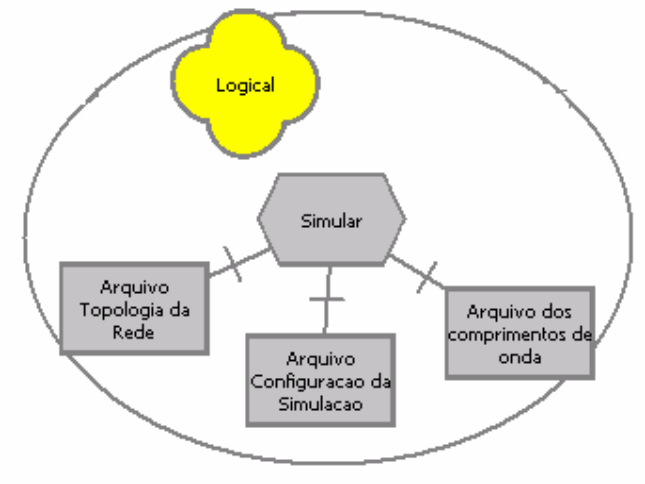

Figura 11 - Decomposição de tarefa.

As contruibuições para ***objetivos soft*** indicam se a execução de uma tarefa afeta de forma positiva (indicado por um sinal "+") ou negativa (indicada por um sinal "-") o cumprimento de um ***objetivos soft***. Podemos ter, por exemplo na Figura 12, dois ***objetivos soft*****:** *Velocidade de acesso* e *Seguranca dos dados.* Uma ***tarefa*** denominada *Criptografia dos dados* pode contribuir de forma positiva para alcançarmos o ***objetivo soft*** *Seguranca dos dados,* mas pode contribuir de forma negativa para o ***objetivo soft*** *Velocidade de acesso.*

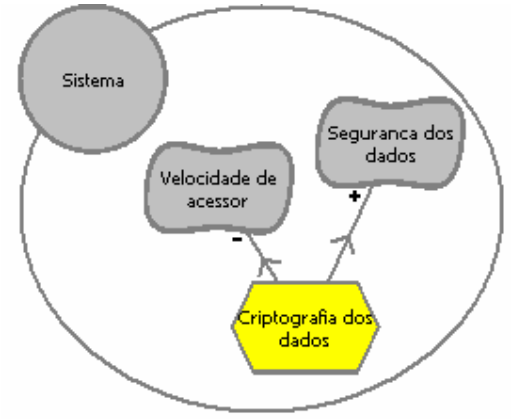

Figura 12 - Relacionamento de contribuição.

Nesta seção foram apenas mostrados alguns dos os elementos principais do *i**, que serão referenciados no decorrer da dissertação.

A técnica *i** é indicada para modelagem organizacional e captura dos requisitos iniciais do sistema. Para os estágios posteriores do desenvolvimento de *software* temos a *Unified Modeling Language* (UML), uma linguagem de modelagem visual bastante dinfundida e utilizada.

3.2 A Linguagem de Modelagem Unificada

A UML (Unified Modeling Language Specification) [7, 8, 9] é uma linguagem de modelagem visual para especificar, visualizar, construir e documentar os artefatos de um sistema de *software*. É utilizada para configurar, descrever, navegar, manter e controlar informações relevantes ao desenvolvimento de sistemas de *software*. Entre seus principais objetivos temos:

- fornecer aos usuários uma linguagem de modelagem visual expressiva, para desenvolver e trocar modelos entre si;
- suportar especificações que são independentes de uma linguagem de programação e processo de desenvolvimento particular;
- suportar conceitos de desenvolvimento de alto nível, como componentes, colaborações e *frameworks*;
- auxiliar o crescimento de ferramentas de desenvolvimento orientadas a objetos no mercado;

Um sistema será modelado como um conjunto de objetos que interagem entre si para realizar alguma função ou fornecer algum serviço, segundo o paradigma da orientação a objetos. A UML captura informações sobre a estrutura e o comportamento estático e dinâmico de um sistema. A estrutura estática define quais os objetos são importantes para um sistema e como será sua implementação. A estrutura dinâmica descreve o comportamento destes objetos ao longo do tempo e como se processa a comunicação entre eles. As informção dos modelos são representadas através de diagramas. Ao todo são nove diagramas que representam os diversos aspectos de um sistema. Nas seções a seguir discutiremos os diagramas da UML, com ênfase no diagrama de classes.

3.2.1 Diagramas da UML

A UML possui vários diagramas que representam os diversos aspectos de um sistema:

1. **Diagrama de classes:** mostra uma visão da estrutura estática do modelo. Os elementos do modelo podem ser classes, *interfaces*, pacotes. No diagrama de classes são mostrados a estrutura interna desses elementos e os relacionamentos entre eles, representados por *links* que conectam os elementos. Os elementos do diagrama podem ser agrupados em pacotes.
2. **Diagrama de casos de uso:** mostra as interações entre casos de uso do sistema e os atores. Os casos de uso representam a funcionalidade do sistema. Os atores representam pessoas ou outros sistemas que interagem com o sistema que está sendo desenvolvido. Fazem parte do diagrama de casos de uso: atores, casos de usos, *interfaces* e as relações entre estes elementos.

3. **Diagrama de estado**: representa uma máquina de estados. Fornecem um meio para descrever o comportamento das instâncias de um elemento do modelo (um objeto, por exemplo). O diagrama de estado descreve as possíveis seqüências de estados e ações das instâncias dos elementos do modelo, durante os seus ciclos de vida. Enquanto o diagrama de classes descreve a estrutura estática do sistema, o diagrama de estado é utilizado para modelar o comportamento dinâmico do sistema.
4. **Diagrama de atividades**: é uma variação de uma máquina de estados, no qual os estados representam a realização de alguma ação e as transições entre os estados são acionadas pelo termino das ações no estado anterior. O diagrama de atividades é um caso especial do diagrama de estado. O propósito deste diagrama é mostrar o fluxo da funcionalidade do sistema que está sendo desenvolvido;
5. **Diagrama de seqüência:** mostra um conjunto de mensagens, dispostas em uma seqüência temporal, que são trocadas entre objetos. Um dos seus usos é mostrar o comportamento seqüencial de um caso de uso. O diagrama de seqüência enfatiza a organização temporal das mensagens trocadas;
6. **Diagrama de colaboração:** mostra a mesma informação que o diagrama de seqüência, porém de uma maneira nova e com um propósito diferente. O diagrama de colaboração mostra os objetos e atores interagindo sem referência ao tempo. Seu propósito é enfatizar a organização estrutural dos objetos que recebem e enviam mensagens;
7. **Diagrama de objetos:** é um gráfico de instâncias dos elementos do modelo, incluindo objetos e valores dos dados. Um diagrama de objeto é uma instância de um diagrama de classes. Mostra uma fotografia detalhada do estado de um sistema em um determinado instante de tempo.
8. **Diagrama de componentes:** mostra as dependências entre os componentes de *software*. Componentes são unidades físicas de implementação com *interfaces* bem definidas. Os componentes são partes do sistema que podem ser substituídas sem afetar as demais partes.
9. **Diagrama de implantação:** descreve como o sistema sendo desenvolvido será implantando fisicamente. Descreve todo os nós da rede, as conexões entre eles e os processos que irão executar em cada nó. O diagrama de implantação inclui o *layout* da rede e onde os componentes deverão estar localizados dentro da rede. Além disso, pode incluir restrições da rede no qual o sistema irá funcionar, como largura de banda disponível e quanto usuário poderão utilizar o sistema simultaneamente.

No nosso trabalho, é utilizado fundamentalmente o diagrama de classes para representar os requisitos que serão mapeados da técnica *i** para a UML. A seguir veremos com mais detalhe a

notação utilizada neste diagrama, muito embora exista definições para mapeamento entre a técnica *i** e o diagrama de casos de uso [49].

3.2.1.1 Diagrama de Classes

Os diagramas de classes são utilizados para descrever os tipos de objetos de um sistema e as relações entre eles, utilizando elementos como classes, pacotes e *interfaces*.

Uma classe é algo que encapsula alguma informação e/ou comportamento. A informação encapsulada é modelada através de **atributos**, enquanto o comportamento é modelado através de **operações** ou **métodos**. Uma classe é composta de **nome**, **atributo**s e **operações** (Figura 13)**.**

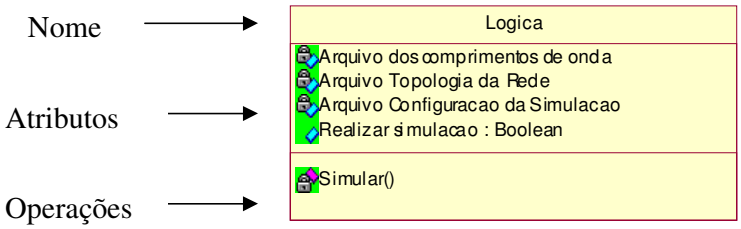

Figura 13 - Classe UML.

O diagrama de classes também pode representar o relacionamento entre as classes. Existem cinco tipos de relacionamentos possíveis entre classes: **associações**, **dependência**, **agregação**, **realização** e **generalização**.

A forma mais comum de relacionamento é a **associação**. Uma **associação** descreve o relacionamento entre instâncias das classes e representa a troca de mensagens entre elas. Podem ser unidirecionais (direção indicada por uma seta) ou bidirecionais (Figura 14). É possível nomear a **associação** para melhor identificá-la. Podemos indicar a multiplicidade nos extremos de uma associação. A multiplicidade indica o número de instâncias de uma classe possíveis que podem ser relacionadas a uma outra instância de outra classe. Por exemplo, na Figura 14, a classe *Documento* pode se relacionar com um ou mais objetos (instancias) da classe *Elementos.*.

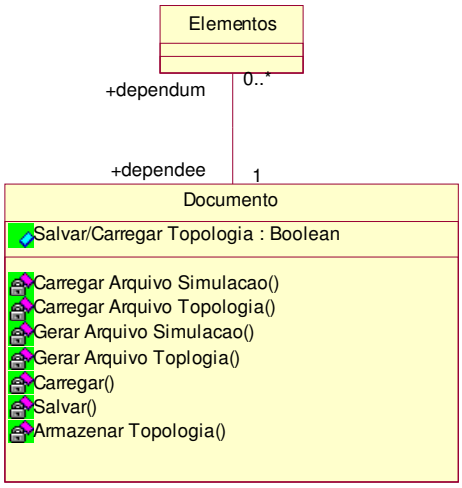

Figura 14 - Relacionamento associação com multiplicidade.

Dependências também conectam duas classes. São sempre unidirecionais e indicam que uma mudança em uma classe pode causar uma mudança em outra classe (Figura 15). Também indicam que um objeto precisa enviar mensagens para outro. A diferença em relação à associação reside no fato de que uma classe não instancia a outra. Na Figura 15 a classe *Janela de Desenho* depende da classe *Visao*, como indica o sentido da seta. Esta depência pode ser, por exemplo, um método da classe *Janela de Desenho* que possui como parâmetro um objeto da classe *Visao*. Porém, a classe *Janela de Desenho* não instancia um objeto da classe *Visao*. Este objeto é fornecido por uma terceira classe.

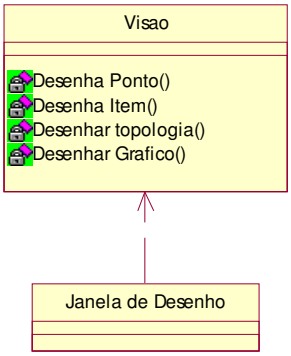

Figura 15 - Relacionamento de dependência.

Agregações são uma forma mais forte de associação. Indica um relacionamento entre o todo e suas partes (Figura 16), ou seja, quando uma classe é constituída de uma coleção de uma ou mais classes. A Figura 16 mostra que o objeto do tipo *Simulador* é fisicamente contsituído de um ou mais objetos do tipo *Logica* ou que um objeto do tipo *Simulador* contém um ou mais objetos do tipo *Logica*.

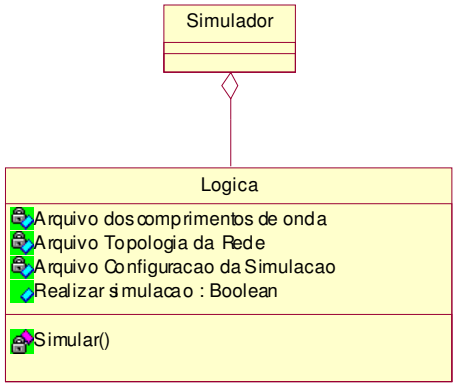

Figura 16 - Relacionamento de agregação.

Os relacionamentos de **realização** são usados para mostrar o relacionamento entre a classe e sua *interface* (Figura 17). Uma *interface* é basicamente uma classe com todos os seus métodos declarados, mas sem implementa-los (especifica o comportamento da classe, mas não sua estrutura). A classe deve implementar todos as operações declaradas na *interface*. Na Figura 171, a *interface* é representa pelo círculo. A classe *Sistema* deve implementar todos métodos da interface *SistemaUsuario.*.

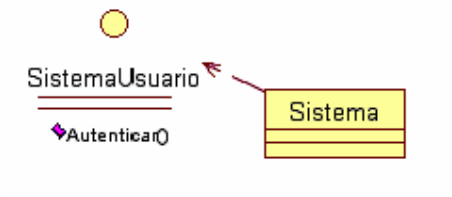

Figura 17 - Relacionamento de realização entre uma classe e um interface.

Outra forma bastante comum de relacionamento é a **generalização**, que é um relacionamento de **herança** entre dois elementos do modelo (Figura 18). Várias classes podem compartilhar uma estrutura comum através da **generalização**. A classe **filha** (chamada de **subclasse**) pode acrescentar informação ou comportamento adicional àqueles **herdados** da classe **pai** (chamada de **superclasse**). A Figura 18 mostra esse tipo de relacionamento. A classe **filha** *Node* é uma **especialização** da classe **pai** *Elementos*.

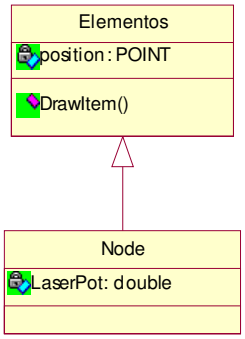

Figura 18 - Relacionamento generalização.

Podemos definir a visibilidade dos atributos e operações, ou seja, definir quem pode acessar a informação contida em uma classe. Visibilidade **privada** indica que os atributos ou operações só podem ser acessados de dentro da classe. Visibilidade **pública** permite que todas as classes possam acessar a informação. Visibilidade **protegida** permite que a informação seja acessada apenas de dentro da classe ou por uma classe filha.

Nas seções anteriores vimos duas técnicas de modelagem de requisitos: a *i** e a UML. Na seção seguinte veremos como podemos tranformar um modelo *i** em um diagrama de classes UML através de diretrizes de mapeamento.

3.3 Diretrizes de Mapeamento da Modelagem i para UML*

As diretrizes de mapeamento determinam as regras pelas quais a associação entre o modelo *i** e o modelo UML é feita. Elas foram propostas em [3, 4] e posteriormente modificadas para suportar mais elementos do modelo *i** de da UML em [11]. Faz-se necessário, então, a associação dessas duas técnicas. A técnica *i** é ideal para realizar a modelagem dos requisitos organizacionais e requisitos não funcionais e funcionais dos estágios iniciais do desenvolvimento, dando uma visão geral do sistema a ser desenvolvido de como ela irá afetar a organização na qual irá atuar. A UML é capaz de modelar os requisitos funcionais do sistema, sendo esses modelos capazes de se transformar em código fonte diretamente. A UML é, portanto, ideal para os estágios posteriores do desenvolvimento, onde se faz necessário um modelo mais concreto e que represente o sistema de forma mais detalhada. O objetivo das diretrizes é manter a consistência e o rastreamento entre o sistema a ser desenvolvido e o os objetivos da organização. Desta forma, aumentamos as chances do sistema que está sendo desenvolvido vir a atender os objetivos da organização de uma forma mais precisa.

Infelizmente os diagramas da UML sozinhos não são capazes de modelar todos os aspectos dos requisitos finais, pois não fornece especificações para as limitações dos sistemas (tais como invariantes, pré-condições, etc.). Para isto, adotamos a Object Constraint Language (OCL). A OCL [46] é uma linguagem textual formal utilizada para expressar restrições (constraints). Restrições são expressões semânticas representadas em forma de texto. Essas restrições, tipicamente, especificam condições invariantes que o sistema modelado deve satisfazer.

Um diagrama UML tipicamente não é suficiente para fornecer todos os aspectos relevantes de uma especificação. Existe a necessidade de descrever restrições adicionais sobre os objetos do modelo. A prática tem demonstrado que escreve estas restrições em linguagem natural acarreta ambigüidades. As linguagens formais foram desenvolvidas para definir as restrições de uma maneira não ambígua. Porém, estas linguagens formais são de difícil utilização para aqueles que não possuem uma base matemática forte. A OCL foi desenvolvida para preencher esta lacuna: apesar de ser uma linguagem formal, ainda é fácil de se ler e escrever.

A seguir, tem-se as diretrizes que realizam o mapeamento entre a técnica *i** e a UML. No total são sete diretrizes: D1 a D7. De forma a agilizar a aplicação das regras e diminuir a possibilidade de erro quando da aplicação dessas regras, propomos neste trabalho uma ferramenta capaz de automatizar a aplicação dessas regras, gerando os modelos UML diretamente dos modelos *i**, com o mínimo de esforço por parte do usuário: XGOOD (eXtended Goals into Object Oriented Development). A ferramenta de mapeamento foi construída de tal forma que é capaz de se integrar e interagir com outras ferramentas de apoio, utilizadas para realizar a modelagem *i** e a modelagem UML. São mostrados vários exemplos de mapeamento com a utilização da nova ferramenta proposta neste trabalho. Os exemplos são extraídos do modelo *i** do simulador proposto neste trabalho (mais detalhes no capítulo 7).

Diretriz D1: relacionada com o mapeamento de atores *i**.

- **Diretriz D1.1:** agentes, papéis ou posições no modelo *i** podem ser mapeados em classes UML.
- **Diretriz D1.2:** o relacionamento *i** ***IS-PART-OF*** entre posições, agentes e papéis podem ser mapeados como uma agregação de classes em UML . Por exemplo na Figura 19 os atores *Simulador, Logica, Visao* e Documento foram mapeado em classe UML. Os relacionamentos ***IS-PART-OF*** foram transformados em agregação de classes.

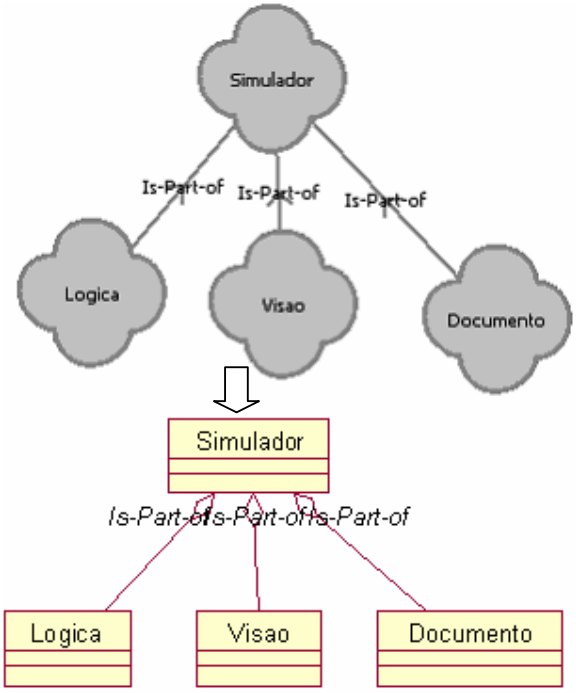

Figura 19 - Relacionamento *is-part-of* para diagrama de classes.

- **Diretriz D1.3**: o relacionamento *i** ***IS-A*** entre posições, agentes e papéis podem ser mapeados em generalização/especialização de classes, em UML. No exemplo da Figura 20 o ator *Administrador* é uma especialização do ator *Usuario*. Os atores são mapeados em classes UML com o relacionamento de generalização entre eles.

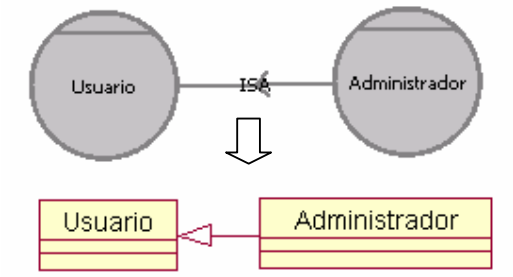

Figura 20 - Relacionamento *isa* para diagrama de classes.

- **Diretriz D1.4**: o relacionamento *i** ***OCCUPIES*** entre um agente e uma posição pode ser mapeado em UML como uma associação de classe denominada *OCCUPIES*.
- **Diretriz D1.5**: o relacionamento *i** ***COVERS*** entre uma posição e um papel pode ser mapeado em UML como uma associação de classe denominada *COVERS* (Figura 21). A posição *Adminstrador* cobre os papéis *Instalacao* e *Manutencao*. Eles foram transformados em classes UML, e relacionamentos de associação foram criados entre as classes.

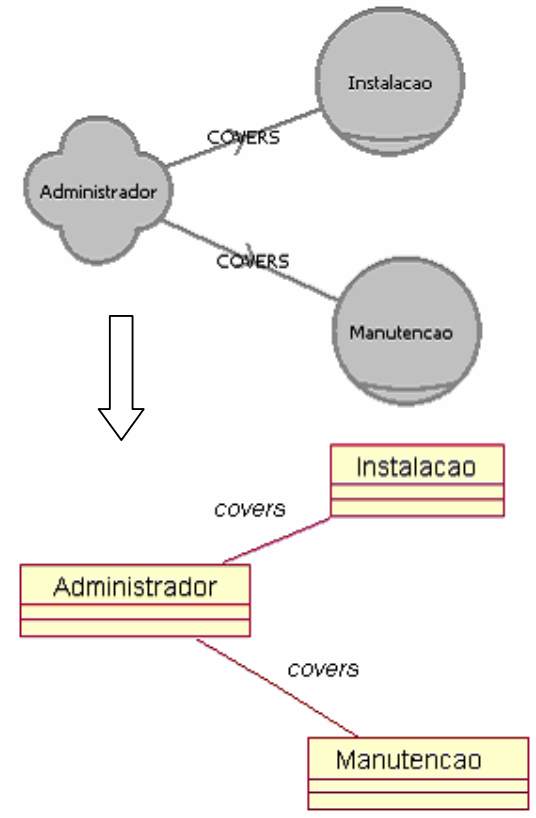

Figura 21 – Relacionamento *covers* para diagrama de classes.

- **Diretriz D1.6**: o relacionamento *i** ***PLAYS*** entre um agente e um papel pode ser mapeado em UML como uma associação de classe denominada *PLAYS*.

Diretriz D2: Relacionada com o mapeamento de tarefas *i**.

- **Diretriz D2.1**: uma tarefa definida no modelo SD (*Strategic Dependency*) pode ser mapeada como uma operação na interface que é realizada pela classe que representa o *dependee* (aquele que realiza a tarefa). As dependências de tarefas entre os mesmo *depender* (aquele que depende da tarefa) e *dependee* são agrupadas em uma mesma interface. Além disso, é criada uma dependência UML entre a classe (*depender*) e a interface. O nome da interface é constituído pelos nomes das classes que representam o *dependee* e o *depender*. Na Figura 22, a tarefa *Autenticar* foi transformada em um método da interface *SistemaUsuario,* nome formado pela junção dos nomes dos atores *Usuario* e *Sistema.* A classes *Sistema,* que representa o *dependee*, implementa o método *Autenticar* da *interface*.

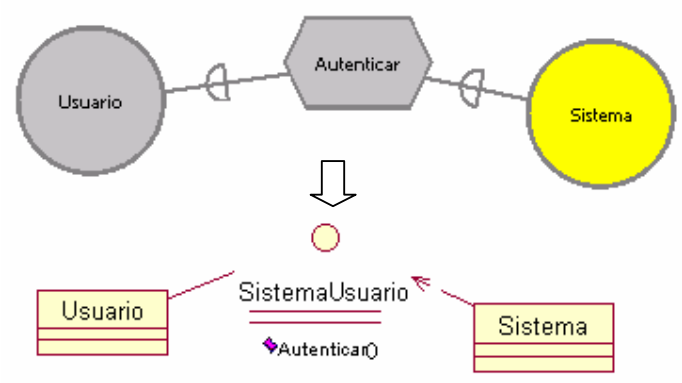

Figura 22 - Tarefas do modelo SD para o diagrama de classes.

- **Diretriz D2.2**: uma tarefa definida no modelo SR (*Strategic Rationale*) pode ser mapeada como uma operação com visibilidade privada na classe que representa o ator a qual a tarefa pertence, exemplo Figura 23. As tarefas *Armazenar Topologia, Salvar* e *Carregar*, são mapeados como métodos com visibilidade privada na classe *Documento*.

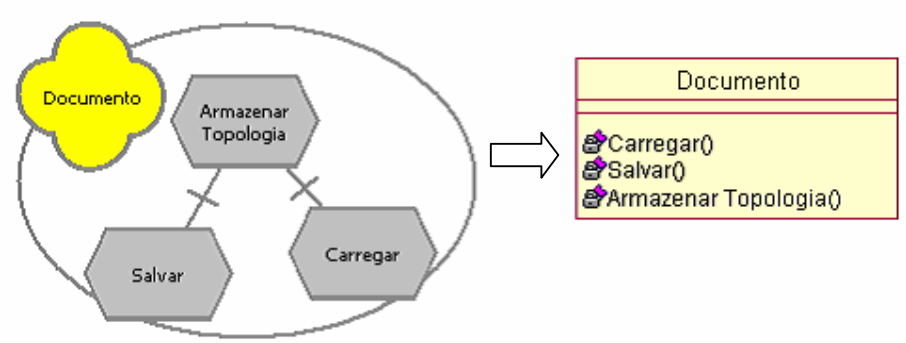

Figura 23 - Tarefas do modelo SR para o diagrama de classes.

Diretriz D3: Os recursos *i** podem ser mapeadas em classes UML.

- **Diretriz D3.1**: os recursos definidos no modelo SD podem ser mapeados de duas maneiras:
  - um recurso pode ser mapeado em uma classe UML se ele possui características de um objeto, tal qual definido pelo paradigma de orientação à objeto. Uma associação entre a classe que representa o ator *depender* (aquele que depende do recurso) e a classe que representa o recurso é criada. Da mesma forma, uma outra associação é criada entre a classe que representa o recurso e a classe que representa o ator *dependee* (aquele que fornece o recurso), exemplo Figura 24. O recurso *Elementos* foi transformado em uma classe UML. Os relacionamentos de

associação criados entre as classes *Logica, Documento* e *Elementos* possuem papéis que indicam quem é o *dependee, depender* e o *dependum* do modelo *i**.

- um recurso pode ser mapeado como um atributo com visibilidade pública na classe que representa o ator *dependee* (agente, posição ou papel) se esta dependência não está caracterizada como um objeto, tal qual definido no paradigma de orientação à objeto. Para representar a dependência entre atores é criada uma associação para indicar quem é o *dependee* e quem é o *depender;*

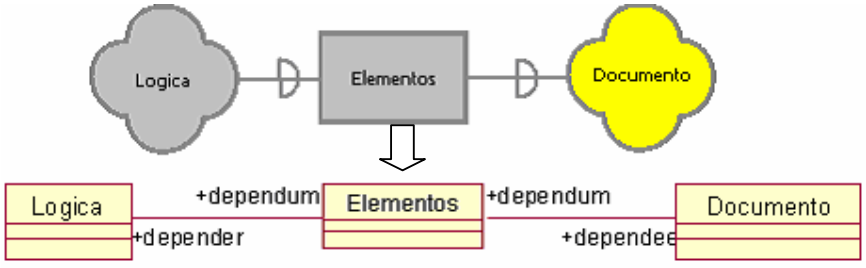

Figura 24 - Recurso do modelo SD no diagrama de classe.

- **Diretriz D3.2**: um recurso definido no modelo SR pode ser mapeado como um atributo com visibilidade privada na classe que representa o ator (agente, posição ou papel) do qual o sub recurso pertence (se este sub recurso não é caracterizado como um objeto). Caso contrário, este recurso será mapeado em UML como uma classe independente. É criado também um link de dependência entre o ator e o recurso.

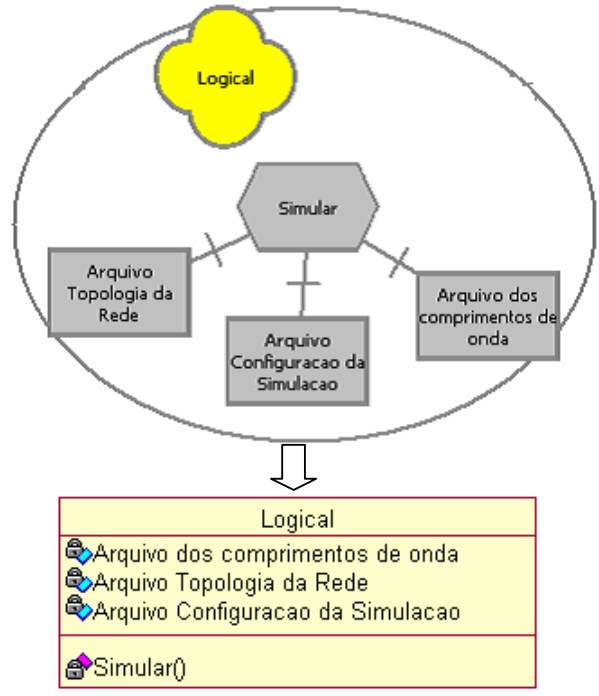

Figura 25- Recurso no modelo SR mapeado como atributo privado.

Diretriz D4: relacionada com o mapeamento de objetivos *i**.

- **Diretriz D4.1:** um objetivo definido no modelo SD pode ser mapeado como um atributo boleano com visibilidade pública na classe que representa o ator (agente, posição ou papel) *dependee* (aquele que fornece o objetivo). Para representar a dependência entre atores é criada uma associação para indicar quem é o *dependee* e quem é o *depender.* Na Figura 26, o objetivo *Salva/Carregar Topologia* deve ser satisfeito pelo ator *Documento.* Sendo assim, o objetivo é mapeado como atributo boleano na classe *Documento*.

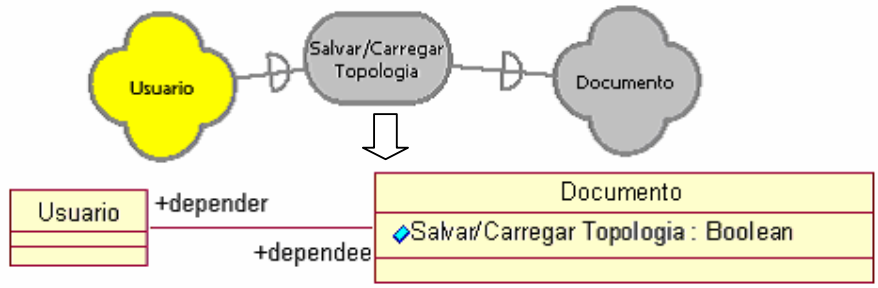

Figura 26 - Objetivos do modelo SD no diagrama de classe.

- **Diretriz D4.2:** um objetivo definido no modelo SR pode ser mapeado como um atributo boleano com visibilidade privada na classe que representa o ator (agente, posição ou papel) a qual o objetivo pertence.

Diretriz D5: relacionada com o mapeamento de objetivos *soft i**.

- **Diretriz D5.1:** um objetivo *soft* definido no modelo SD pode ser mapeado como um atributo do tipo númerico, representando os vários níveis de satisfação do objetivo, com visibilidade prública na classe que representa o ator (agente, posição ou papel) *dependee* (aquele que fornece o objetivo). Para representar a dependência entre atores é criada uma associação para indicar quem é o *dependee* e quem é o *depender.*
- **Diretriz D5.2:** um objetivo *soft* definido no modelo SR pode ser mapeado como um atributo do tipo númerico com visibilidade privada na classe que representa o ator (agente, posição ou papel) a qual o objetivo pertence.

Diretriz D6: Relacionada com o mapeamento de relacionamentos de decomposição de tarefa no modelo *i**.

- A decomposição de tarefa no modelo SR é representada através de pré e pós-condições (definidas em OCL) da operação correspondente em UML. A pré-condição é a conjunção (conector AND OCL) das pré-condições das tarefas. A pós-condição é a conjunção (conector AND OCL) de todas: (i) pós-condições das tarefas, (ii) atributos boleanos dos recursos; (iii) atributos boleanos dos objetivos; (iv) atributos enumerados do objetivo *soft*. Como exemplo, temos a tarefa *Salvar*() (Figura 27). A tarefa *Salvar* é decomposta em duas

tarefas: *Gerar Arquivo de Simulacao* e *Gerar Arquivo Topologia*. A tarefa *Salvar* é considerada cumprida somente quando essas duas forem executas:

```
Documento:Salvar:Boolean
Pre: pre-Salvar
Pos: result = GerarArquivoSimulacao and GerarArquivoTopologia
```

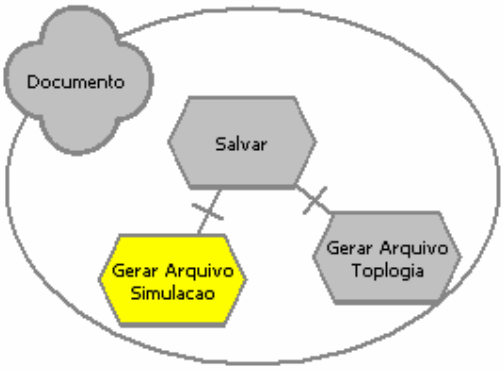

Figura 27 - Tarefa salvar do modelo SR.

Diretriz D7: relacionada com o mapeamento de relacionamentos de meio-fim no modelo *i**. Análise de meio-fim é representada através de disjunções de todos os possíveis meios de alcançar o fim.

- **Diretriz D7.1**: se o fim é uma tarefa e os meios são tarefas então a disjunção das pós-condições das tarefas meio implica a pós-condição da tarefa fim.
- **Diretriz D7.2**: se o meio é uma tarefa e o fim é um objetivo, um objetivo *soft* ou um recurso, o valor do fim é igual a disjunção das pós-condições das tarefas meio.
- **Diretriz D7.3**: se o fim é um objetivo (*soft*) e os meios são objetivos (*soft*) então a disjunção dos valores dos meios implica o valor fim.

3.6 Conclusões

Neste capítulo foi discutido a importância da modelagem visual para o desenvolvimento de *software* e duas técnicas de modelagem utilizadas para o desenvolvimento de *software*. Temos a técnica *i**, utilizada para a captura e modelagem dos requisitos organizacionais, sendo indicada para as fases iniciais do desenvolvimento e a UML para as fases posteriores do desenvolvimento, com ênfase no diagrama de classes. Além disso, vimos também um conjunto de diretrizes utilizadas para mapear um modelo *i** em um diagrama de classes UML.

Para a modelagem e utilização dessas técnicas, bem como a realização do mapeamento entre modelos, são necessários ferramentas de apoio, que automatizem o processo de modelagem e

mapeamento, de forma a agilizar o desenvolvimento. No caso da técnica *i**, temos a ferramenta OME (Organization Modeling Environment) [29]. Para a modelagem UML existem várias ferramentas disponíveis no mercado, como a Rational Rose [30], MagicDraw [31] e TelelogicTau [32]. Para automatizar o mapeamento entre os modelos, temos as ferramentas de apoio GOOD (Goals into Object Oriented Development) [10] e, a proposta neste trabalho, XGOOD (eXtended GOOD) [11]. Além de automatizar o processo de mapeamento entre os modelos, a ferramenta proposta neste trabalho foi acoplada diretamente dentro da ferramenta OME e os modelos gerados por ela podem ser importados por várias ferramentas comercias do mercado utilizadas para a modelagem UML. Examinaremos em detalhes essas ferramentas e como integrá-las nos capítulos seguintes.

4 Ferramentas de Apoio

As ferramentas CASE (*Computer Aided Software Engineering*) fornecem suporte automatizado para muitos dos métodos existentes de análise e projeto de sistemas [15]. Elas fornecem um meio ambiente que automatiza grande parte das tarefas repetitivas e que consomem bastante tempo do desenvolvedor, por exemplo: desenhar e redesenhar diagramas; manter a consistência dos modelos; geração de documentação, etc. As ferramentas CASE também podem incluir um gerador de códigos que origina o código-fonte a partir do modelo de sistema, de forma automática [15].

Um dos problemas do uso de ferramentas CASE é sua integração. Apesar dos benefícios do uso de ferramentas CASE individuais, elas são mais úteis quando trabalham juntas e de uma forma integrada. Entre os benefícios da integração de ferramentas CASE temos: transferência da informação (modelos, programas, documentos, dados) de uma ferramenta para outra; comunicação melhorada entre as pessoas que estão trabalhando em um mesmo projeto utilizando ferramentas diferentes; interfaces do usuário integradas que podem reduzem o tempo de aprendizado e as taxas de erro, etc.

A modelagem através da UML é suportada por uma grande variedades de ferramentas existentes no mercado: Rational Rose [30], MagicDraw [31] e TelelogicTau [32], ArgoUML [33], etc. Todas elas suportam os diversos diagramas da UML. Cada uma delas possuem seu próprio formato para armazenar seu modelos, porém todas conseguem importar modelos que estejam no padrão XMI. Para a modelagem utilizando a técnica *i** existe a ferramenta OME, que será descrita na próxima seção.

4.2 Ambiente de Modelagem Organizacional - OME

O OME (Organization Modeling Environment) [29] é uma ferramenta de análise e modelagem orientada a objetivos e/ou agentes. Fornece ao usuário uma interface gráfica para o desenvolvimento de modelos. Atualmente, a ferramenta suporta dois *frameworks* de modelagem: *i** e NFR (Non-Functional Requirement – requisitos não funcionais). Um terceiro *framework*, o GRL (*Goal-oriented Requirement Language* – linguagem de requisitos orienta a objetivos), esta sendo definido e estará disponível em breve. A ferramenta foi projetada de modo que possibilita aos usuários estende-la para suporta novos *frameworks*. Além disso, fornece uma API Java que permite a criação de *plugins* para adicionar novas funcionalidades a ferramenta.

O OME está sendo desenvolvido no *Knowledge Management Lab* na Universidade de Toronto como parte do projeto "*Agent-Oriented Approach to System Architecture: Models and*

Analysis Tools". Sua versão mais recente é o OME3 [29]. Nas seções seguintes apresentaremos o OME, procurando entender como os modelos *i** são salvos e como podemos ler um arquivo gerado pela OME e dele extrair as informações do modelo. Também veremos como é possível estendê-la para suportar a nova ferramenta XGOOD de forma integrada com o ambiente de desenvolvimento.

4.2.1 Arquitetura

O OME3 é basicamente composto de duas partes: o núcleo (*kernel*) e os *plugins* [34]. O *kernel* possui uma arquitetura em camadas, composta de três módulos principais, cada um composto de várias classes Java e/ou ferramentas externas. *Plugins* são classes codificadas em Java que implementam funções específicas do *framework*. Quando da execução da ferramenta, os *plugins* são carregados de acordo com o *framework*.

As camadas que fazem parte do núcleo podem ser visualizadas na Figura 28. Nessa tem-se: a camada Visão, a camada Modelo/*Framework* e a camada Base de Conhecimento (*Knowledge Base* - KB). Na camada **KB**, temos o modulo **KB**, que armazena os objetos em um modelo, seus relacionamentos e atributos. Os detalhes do modelo do *framework* também são armazenados. Além disso, o **KB** realizar uma checagem de semântica, para assegurar que o modelo sendo desenvolvido está de acordo com o *framework* armazenado. Existe um gerenciador **KB** responsável por criar a novo **KB** para cada modelo.

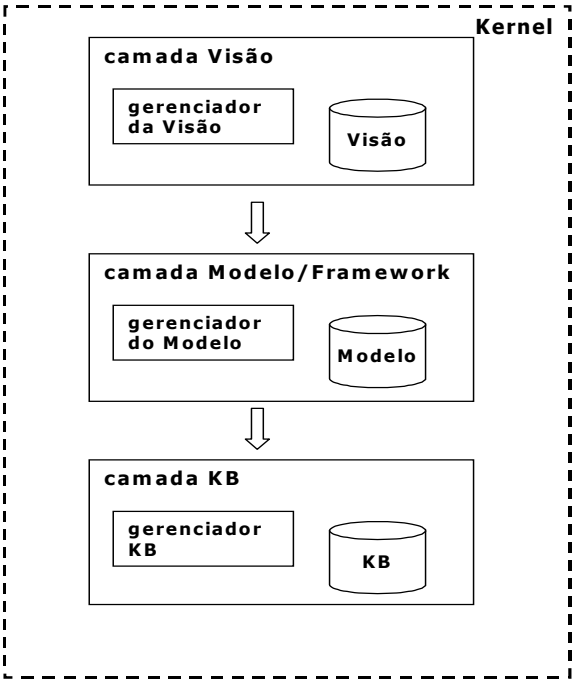

Figura 28 - Arquitetura em camadas do Kernel.

Em seguida temos a camada Modelo/*Framework*. Esta camada possui duas funções. A primeira é fornecer uma interface abstrata e simples para a informação armazenada no **KB**. A segunda, isolar o resto da ferramenta de uma possível reimplementação do **KB**. Esta camada também separa, conceitualmente, o modelo do seu *framework*. Quando um usuário deseja criar ou abrir um modelo previamente armazenado, os detalhes são passados para o **gerenciador do Modelo**, que realizar o trabalho. O **gerenciador do Modelo** instrui o **gerenciador KB** a criar um novo **KB.** O **gerenciador do Modelo** inicia então a construção dos componentes do módulo Modelo/*Framework*. O **gerenciador do Modelo** inicia a construção dos componentes do módulo Visão. Além disso, determina quais os *plugins* que devem ser carregados e fornece uma interfae para que esses *plugins* sejam acessados pelo módulo Visão. O **gerenciador do Modelo** também é responsável por salvar os modelos, que nada mais é do que instruir o **KB** para armazenar o seu conteúdo em um arquivo.

Finalmente, temos a camada Visão. Seu trabalho é apresentar uma representação do modelo ao usuário e fornecer ao usuário os meios de manipular esta representação. Cada modelo pode ter vários tipos de visões e para cada modelo existe o **gerenciador de Visão** que controla todas as visões. A camada Visão também é responsável por fornecer para os usuários o acesso às funcionalidades fornecidas pelos *plugins*, como também é o meio pelo quais os *plugins* recebem as entradas dos usuários.

Um *plugin* é constituído por um conjunto de classes que se encarregam da maioria das manipulações que podemos realizar no modelo e na visão. Fornecem um meio fácil de estender a funcionalidade da ferramenta. Quando um novo modelo é criado ou carregado, cada *plugin* pode examinar o modelo e o *framework*, determinando se as operações fornecidas pelo *plugin* devem se tornar disponíveis para o usuário.

Um *plugin* pode ser considerado um conjunto de métodos. Neste sentido, um método é um processo iniciado pelo usuário com o objetivo de modificar a visão e/ou o modelo e captura alguma entrada do usuário. Um dos modos de se captura as entradas do usuário é através da requisição de parâmetros. Parâmetros são entradas que os usuários devem fornecer para o método antes do mesmo ser invocado. O módulo Visão é o responsável por coletar estes parâmetros necessários pelo método. O *plugin* especifica ainda como os métodos estarão disponíveis ao usuário (por exemplo, como um botão na barra de tarefas, como um menu, uma tecla de atalho do teclado, etc.).

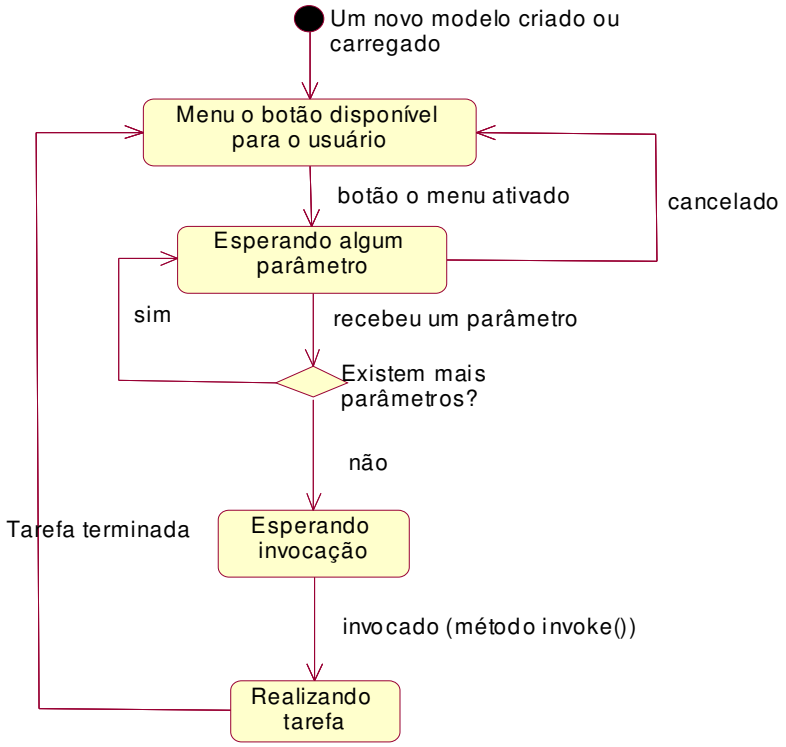

Figura 29 - Ciclo de vida de um método de um *plugin*.

Na Figura 29 vemos o ciclo de vida de um *plugin*. A figura mostra as várias estados que o *plugin* pode ocupar.

1. a Visão irá perguntar ao método do *plugin* a informação necessária para configurar a interface com o usuário. Por exemplo, determinar o nome do menu ou botão através do método *getName()* e/ou solicitar uma imagem para representar graficamente o *plugin;*
2. um *plugin* pode não estar sempre disponível. Através do método *isEnabled()* a Visão determina se um *plugin* esta disponível ou não;
3. o usuário ativa o método do *plugin* (i. e. clicar no botão ou menu correspondente) a Visão irá perguntar o método do *plugin* por parâmetros;
4. o método do *plugin* irá responder, se existirem parâmetros, com um objeto do tipo *Parameter,* descrevendo o tipo de parâmetro necessário;
5. a Visão captura o parâmetro e transmite para o método do *plugin*, perguntando se são necessários mais parâmetros.

6. o método do *invoke()* do *plugin* é chamado, quando todos os parâmetros forem capturados. Esse método realiza alguma tarefa (executar um aplicativo externo, por exemplo) e o clico se inicia novamente.

Cada *plugin* deve implementar a interface Java *OMEPlugin* e cada método do *plugin* deve implementar a interface Java *PluginMehtod.* Além disso, temos a classe Java *PluginParameter* que é usada para encapsular a informação necessária para que o método requisite um parâmetro do usuário. Para que um *plugin* seja carregado com o modelo, primeiramente é necessário que a classe compilada esteja no diretório "OME3\program\plugins\". Segundo, a classe principal do *plugin* deve ter um método com a assinatura:

```
public static boolean isCompatibleWith(OMEModel model)
```

Este método deve retorna *true*, caso o *plugin* deva ser carregado, ou *false*, caso contrário. Os modelos no OME são representados na linguagem de modelagem conceitual *Telos,* que será vista na próxima seção.

4.2.2 Linguagem Telos

Uma das camadas do *kernel* da ferramenta OME é o módulo Knowledge Base (**KB**). O **KB** armazena os objetos em um modelo, suas relações e seus atributos. O modulo principal no **KB** é um repositório *Telos* [35], implementado em C++. Ele fornece a funcionalidade de salvar ou carregar modelos de sistemas de arquivos. Os modelos salvos possuem uma extensão "*.tel".

Telos é uma linguagem desenvolvida com o intuito de apoiar o desenvolvimento de sistemas de informação, não é uma linguagem de programação. Ela é baseada na premissa de que o desenvolvimento de sistemas de informação é intensivamente dependente do conhecimento e que para suporta esta tarefa, a linguagem utilizada deve ser capaz de representar de maneira formal todo o conhecimento relevante. O conhecimento relevante inclui:

- conhecimento sobre o meio ambiente dentro do qual o sistema irá funcionar e de como o sistema irá trabalhar;
- conhecimento sobre o tipo de informação que o sistema irá armazenar o significado desta informação;
- conhecimento sobre as decisões de projeto, juntamente com e informação;
- conhecimento sobre o processo de desenvolvimento em si.

A linguagem *Telos* foi desenvolvida a partir do CML (Conceptual Modelling Language) [35], sendo o *Telos* uma versão mais simples da última. Fornece meios para construir, pesquisar e atualizar bases de conhecimento (Knowledge base - **KB**).

Através do uso da linguagem *Telos*, foi definido um *framework* orientado a objetos capaz de representar todos os elementos do modelo *i**. Uma KB *Telos* consiste de objetos estruturados provenientes de dois tipos de unidades primitivas: **individuais** (*individuals*) e **atributos** (*attributes*). **Individuais** representam entidades e **atributos** representam os relacionamentos entre as entidades ou outros relacionamentos. **Individuais** e **atributos** são chamados de proposições. Cada atributo consiste de uma fonte (from), um rótulo (label) e um destino (to). As proposições podem ser classificadas em quatro tipos: *Tokens, SimpleClass, MetaClass* e *MetaMetaClass*, sendo que *Tokens* representam o nível mais baixo de abstração e *MetaMetaClass* o nível mais alto.

Na definição das proposições em *Telos*, utilizamos a cláusula *IN* para indicar uma lista de classes no qual o objeto definido pode torna-se uma instância, a cláusula ISA para definir a herança (a classe pai), e a cláusula WITH para introduzir os atributos.

Na Figura 30 temos um exemplo gráfico dessa hierarquização. No exemplo não são considerados todos os relacionamentos e classes para não sobrecarregar a figura. Os níveis de abstração mais baixos são instâncias dos níveis de abstração mais altos.

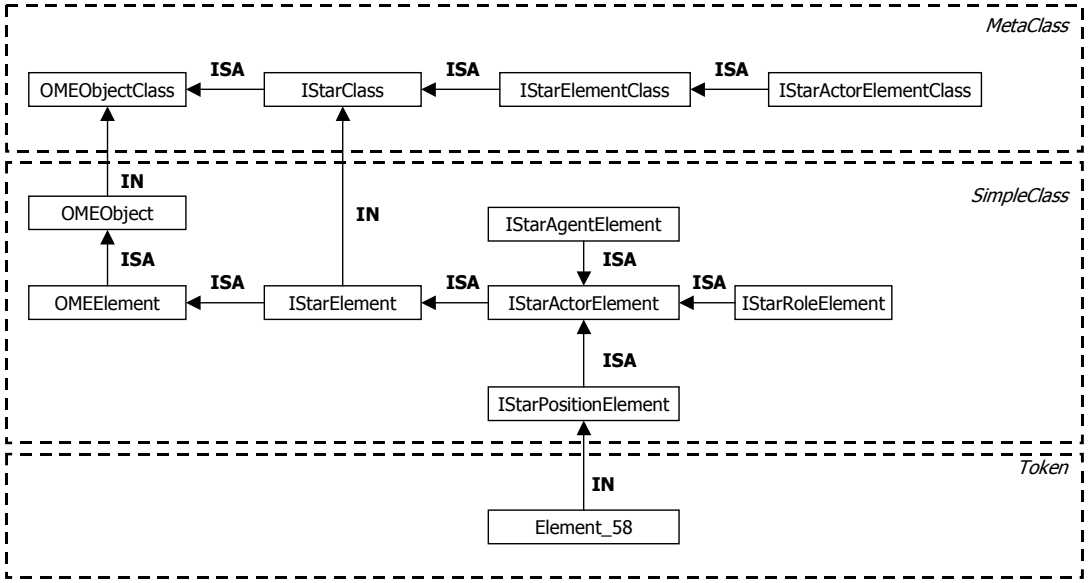

Figura 30 - Exemplo de níveis de abstração.

O exemplo da Figura 30 seria assim descrito em linguagem *Telos*:

- Nível *Token:* nesse nível temos instâncias de classes simples (*SimpleClass*). No caso, o elemento *Element_58* é uma instância de um elemento do tipo **posição** do modelo *i** (*IStarPositionElement*).

```
Token Element_58
    IN IStarPositionElement
    WITH
  attribute, name
```

```
        : "Documento"
END
```

- Nível *SimpleClass:* nesse nível temos instâncias de meta classes (*MetaClass*). Nesse caso temos o elemento *i** ator (*IstarActorElement*), o elemento posição (*IstarPosition Element* – que é uma especialização (*ISA)* do *IstarActorElement*), e assim por diante.

```
SimpleClass IStarPositionElement
    IN OMEInstantiableClass, IStarActorElementClass
    ISA IStarActorElement
    WITH
  attribute, attribute, name
        : "Position"
END
SimpleClass IStarActorElement
    IN OMEInstantiableClass, IStarActorElementClass, IStarClass
    ISA OMEGrowableElement, IStarElement
    WITH
        attribute, attribute, name
             : "Actor"
END
SimpleClass IStarElement
    IN IStarClass
    ISA OMEElement
    WITH
  attribute
      name : String
END
SimpleClass OMEElement
    IN OMEElementClass
    ISA OMEObject
    WITH
        attribute
            parent : OMEElement
END
SimpleClass OMEObject
    IN OMEObjectClass
    WITH
        attribute
            comment : String
        attribute
            links : OMELink
        attribute
            name : String
END
```

- Nível *MetaClass:* o nível mais alto de abstração.

```
MetaClass OMEObjectClass
    ISA OMEClass
    WITH
        attribute
            OMEValueAttributes : SClass
        attribute
            defaultname : String
        attribute
            OMEAttributes : OMEAttributeMetaClass
END
MetaClass IStarClass
    ISA OMEObjectClass
END
```

A Figura 31 contém um modelo *i** de um ator denominado *Documento*. Logo abaixo da figura temos a representação do elemento *i* Armazenar Topologia*, uma tarefa desse ator, na linguagem *Telos*. Este elemento possui dois relacionamentos de ligações e um elemento "pai", ao qual pertence.

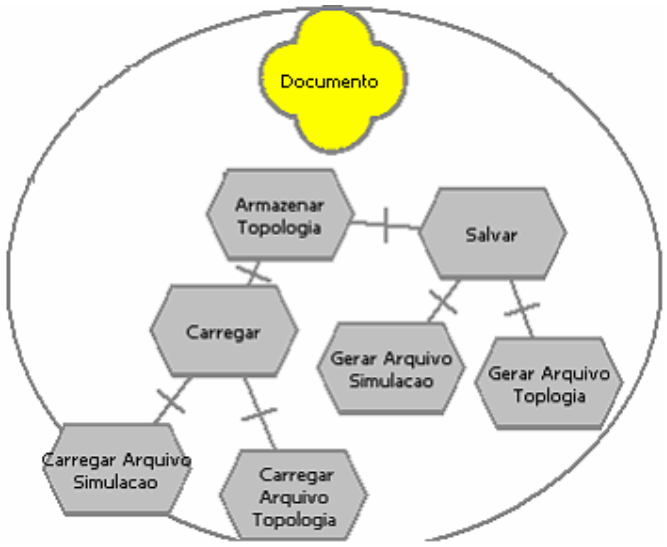

Figura 31- Modelo *i SR.**

```
Token Element_69
    IN IStarTaskElement
    WITH
        attribute, name
             : "Armazenar Topologia"
        attribute
            guide : 4
        attribute, parent
             : Element_58
        attribute, links
             : Link_54
        attribute, links
             : Link_55
END
```

Na representação da Figura 31 em linguagem *Telos* vê-se apenas um fragmento do arquivo que contém todo o modelo *i**. *Token* é uma palavra reservada da linguagem *Telos* que representa a instanciação de uma classe simples. *Element_69* é um é um identificador gerado automaticamente pela ferramenta OME. O indicador *IN* também é uma palavra reservada que especifica as classes as quais o elemento *Element_69* é uma instância. No exemplo, esta indicado que *Element_69* é um instância da classe simples *IstarTaskElement*, uma classe utilizada para representar tarefas do modelo *i**. A palavra reservada *WITH* indica o início dos atributos de *Element_69.* Os atributos *name, parent, links* indicam, respectivamente, o nome do elemento dado pelo usuário (o nome que aparece no diagrama visual), o elemento ao qual o mesmo pertence (presente apenas para os elementos do modelo SR, podendo ser um ator, agente, posição ou papel) e os relacionamentos que

fazem parte do elemento. Os *Tokens* contém as informações essenciais do modelo *i**. As classes simples utilizadas para representar os elementos do modelo são:

- *IstarActorElement* – ator;
- *IstarResourceElement* – recurso;
- *IstarGoalElement* – objetivos;
- *IstarSoftGoalElement* – objetivos *soft*;
- *IstarTaskElement* – tarefas;
- *IstarAgentElement* – agentes;
- *IstarPositionElement* – posição;
- *IstarRoleElement* – papel;
- *IstarDependencyLink* – relacionamento de dependência;
- *IstarMeansEndsLink* – relacionamento meio fim;
- *IstarDecompositionLink* – relacionamento de decomposição
- *IstarPlaysLink* – relacionamento *plays* entre agente e papel;
- *IstarCoversLink* – relacionamento *covers* entre posição e papel;
- *IstarOccupiesLink* – relacionamento *occupies* entre agente e posição;
- *IstarPartsLink* – relacionamento *is-part-of* entre atores;
- *IStarISALink* – relacionamento *isa* entre atores;
- *IStarINSLink.* - relacionamento *ins* entre atores;

A OME é utilizada para a modelagem i*. Para a modelagem existem outras ferramentas que veremos a seguir.

4.3 Ferramenta Para Modelagem UML.

O objetivo dessa seção é apenas proporciona uma visão geral de algumas ferramentas disponíveis no mercado capazes de realizar a modelagem UML. Isso porquê a utilização do XGOOD não estará atrelado a uma única ferramenta CASE de modelagem UML.

Para a modelagem dos requisitos através da técnica UML existem várias ferramentas no mercado. Essas ferramentas suportam a modelagem de todos os diagramas disponíveis na UML (diagrama de classes, diagrama casos de usos, diagrama de estados, etc.). Nesse trabalho, testamos três ferramentas comerciais: Rational Rose [30], MagicDraw [31] e TelelogicTau [32]. Apesar de cada ferramenta possuir seu formato proprietário para a persistência dos modelos, essas três ferramentas possuem em comum, além do suporte a UML, a capacidade de exportar e importar seus modelos para o formato XMI, que será visto com mais detalhes no capítulo seguinte. Isso facilita a troca de modelos entre as ferramentas.

O Rational Rose é capaz de gerar código fonte a partir dos diagramas nas linguagens C++, Visual Basic, Java, Oracle8, CORBA ou DDL (*Data Definition Language*). É capaz de realizar engenharia reversa, ou seja, podem construir um diagrama de classes a partir do código fonte. Também pode ser extendido através de uma interface denominada REI (*Rose Extensibility Interface*), que permite a personalização dos menus do Rose e a criação de scripts para automatizar suas funções e acessar seus elementos. Por padrão o Rational Rose não suporta a utilização do formato XMI, mas graças a interface REI é possível adicionar o suporte ao XMI através do pacote Unisys XMI Export/Import for Rational Rose [36], disponibilizado gratuitamente. Após a instalção desse pacote, o Rose consegue importar/exportar seus modelos para o formato XMI.

O MagicDraw também é capaz de realizar engenharia reversar a partir de código fonte escrive em Java ou C++. Além de importar modelos no formato XMI, o MagicDraw é capaz de ler os modelos no formato do Rose. O Telelogic Tau suporta a UML versão 2.0, com a facilidade de engenharia reversa do código fonte e capacidade de gerar código fonte C++ ou Java a partir dos diagramas.

Existem ainda outras ferramentas que não foram testadas, devido a indisponibilidade de versões gratuitas e/ou de demonstração. Mas tendo elas a capacidade de importar modelos no formato padrão XMI, não deverá haver maiores problemas de integração entre essas ferramenta e a ferramenta proposta nesse trabalho, a XGOOD.

4.4 Ferramenta GOOD - Goal Into Object Oriented Development

O GOOD (Goals into Object Oriented Developement) [10] é o protótipo de uma ferramenta que realiza o mapeamento das descrições dos requisitos organizacionais modelados em *i** pela ferramenta OME, em diagramas de classes em UML, suportados pela ferramenta *Rational Rose*.

O mapeamento é baseado nas diretrizes apresentadas detalhadamente em [3, 4] e resumidamente na seção 3.3. Essas regras foram atualizadas, o que deixou esta ferramenta desatualizada. Os componentes utilizados pelo mapeamento são: repositório de dados da ferramenta OME, a ferramenta GOOD e a ferramenta de modelagem UML *Rational Rose*.

A ferramenta lê o modelo armazenado no repositório da ferramenta OME (um arquivo *.tel* contendo o modelo *i**, representado em linguagem *Telos*) e, através da regras de mapeamento, constrói o digrama de classes UML. A ferramenta GOOD pode ser vista como uma extensão do Rose. Ela foi construída utilizando a linguagem *Rose Scripting* que permite o acesso a funções da ferramenta *Rational Rose* através da API definida no REI (*Rational Rose Extensibility Interface*). Através do REI, temos acesso aos pacotes, classes, propriedade e método da ferramenta, que definem e controlam todas as sua funções. Depois de instalada, a ferramenta pode ser invocada através de um menu localizado no próprio *Rational Rose*.

A ferramenta possui dois componentes: **Configuração do Mapeamento** e **Execução do Mapeamento**. O componente **Configuração do Mapeamento** permite ao usuário escolher o melhor conjunto de regras que irão ser utilizada para realizar o mapeamento (Figura 32), ou seja, qual regra, daquelas disponíveis, melhor de encaixa em cada elemento do modelo a ser mapeado.

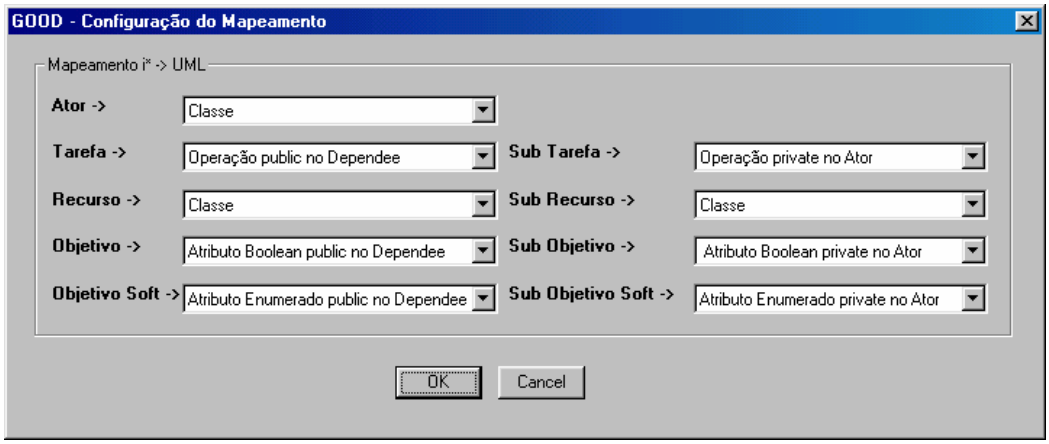

Figura 32 - Tela de Configuração do Mapeamento

Na Figura 32, temos a tela de configuração de mapeamento. Essa tela é utilizada para definir como os elementos *i** serão mapeados em elementos UML. Vemos os principais elementos do modelo SD e SR da técnica *i**: **ator**, **tarefa**, **recurso**, **objetivo**, **objetivo *soft***, **sub-tarefa**, **sub-recurso**, **sub-objetivo**, **sub-objetivo *soft***. Ao lado de cada elemento temos os elementos UML (classe, atributo boleano, etc.). As opções escolhidas pelo usuário são salvas em um arquivo *mapSetting.cfg* para posterior referência.

O componente de **Execução do Mapeamento** está disponível através de uma opção no menu denominada *Executar Mapeamento*. Quando ativado, o componente solicita ao usuário a localização do arquivo ".tel" contendo o modelo *i** a ser mapeado. A ferramenta então extrai deste arquivo os elementos do modelo *i** (atores, tarefas, recursos, objetivos e objetivo *soft*). O arquivo *mapSetting.cfg* é lido e dele é extraído as opções de mapeamento. Finalmente, a ferramenta realiza o mapeamento de cada tipo de elemento e depois dos tipos de relacionamentos, e os representa graficamente no Rational Rose.

Para utilizar o GOOD o usuário necessita instalá-lo. Para isso, deve criar o diretório *Good* em *C:\Arquivos de Programas*. Em seguida, copiar os *scripts GOOD_Configure.ebx* e *GOOD_Map.ebx* para este diretório recém criado. Para permitir o acesso a esses *scrpits* pela ferramenta Rational Rose, é necessário abrir o arquivo *Rose.mnu*, que está localizado no diretório raiz da instalação do Rose (normalmente *C:\Arquivos de Programas\Rational\Rose*) e adicionar as linhas que estão em destaque:

```
Menu Tools
{
  Menu "GOOD"
   {
       option "Configurar Mapeamento"
         {
              RoseScript C:\Arquivos de Programas\GOOD\GOOD_Configure.ebx
         }
       option "Executar Mapeamento"
         {
              RoseScript C:\Arquivos de Programas\GOOD\GOOD_Map.ebx
         }
   }
  ...
}
```

Feito isso, são criados dois novos menus na barra de menus do Rose: *Tool -> GOOD -> Configurar Mapeamento* e *Tool -> GOOD -> Executar Mapeamento*. O usuário deve acessar esses dois menus para realizar o mapeamento. O primeiro serve para definir como os elementos *i** serão mapeados para elementos UML. O segundo executa o mapeamento em si.

4.5 Conclusões

Neste capítulo apresentamos as ferramentas OME, utilizada para criar o modelo *i** e a linguagem *Telos*, utilizada para armazenar os elementos do modelo *i** em arquivo. Foram apresentadas algumas ferramentas para a criação dos modelos em UML. Vimos também a ferramenta GOOD, utilizada para realizar o mapeamento entre o modelo *i** e o diagrama de classes UML.

No capítulo 6 apresentaremos uma versão aprimorada da ferramenta GOOD capaz de mapear mais elementos do modelo *i** e com suporte ao padrão XMI: a ferramenta XGOOD. Para facilitar a utilização da nova ferramenta XGOOD, a mesma foi integrada a ferramenta OME através de um *plugin.* Os detalhes desta integração também foram mostrados nessa seção.

Uma das vantagens dessa nova ferramenta em relação a antiga é a adoção do padrão XMI. No capítulo seguinte iremos apresentar o XMI e justifica o porque da sua adoção.

5 O Padrão XMI

Como visto no capítulo 3, é possível através de diretrizes de mapeamento transformar um modelo *i** em um diagrama de classes UML. A ferramenta proposta nesse trabalho, XGOOD, realiza esta tarefa de forma automática. O diagrama UML é gerado pela ferramenta de tal forma que possa ser importado por quaisquer outras ferramentas disponíveis que suporte o formato XMI e a modelagem UML. A ferramenta proposta XGOOD utiliza justamente o padrão XMI para a representação dos modelos gerados por ela. Desta forma o usuário poderá utilizar a ferramenta de sua preferência para visualizar e refinar o diagrama UML. Isto só é possível através do uso do padrão XMI para realizar a persistência do diagrama UML. A seguir veremos o que é este padrão e como o diagrama de classes é representado por ele. Entendendo o padrão XMI, poderemos visualizar o conteúdo de uma arquivo no formato XMI e identificar, utilizando apenas um editor de texto simples, os elementos do modelo e modificá-los sem problemas.

XMI integra três padrões:

1. XML - *eXtensible Markup Language*, um padrão do W3C standard;
2. UML - *Unified Modeling Language*, um padrão da OMG [12] para modelagem orientada a objetos;
3. MOF - *Meta Object Facility*, um metamodelo e repositório de metadados da OMG;

O XML *Metadata Interchange* (XMI) é um padrão que permite expressarmos objetos através do uso da *eXtensible Markup Language* (XML) [37, 38, 39], o formato universal para a representação de dados na *World Wide Web*. Seu principal objetivo é permitir o fácil intercâmbio de metadados entre ferramentas de modelagem (baseadas na UML – *Unified Modeling Language*) e repositórios de metadados (baseados no MOF – *Meta Object Facility* [40]).

Utiliza o DTD (*Document Type Definition* [37, 38, 39]) e, mais recentemente na versão 2.0, o XSD (*XML Schema Definition* [37, 38, 39]) como meio de validar e manter a consistência dos modelos. Foi adotado em 1999 e atualmente se encontra na versão 2.0.

O padrão UML define uma linguagem de modelagem orienta à objetos, que é suportada por diversas ferramentas gráficas. O padrão MOF define um *framework* para definição de modelos para metadados, fornecendo as ferramentas os meios e as *interfaces* para armazenar e acessar metadados em um repositório (ou seja, MOF é uma linguagem usada para definir linguagens, como a UML). Em outras palavras, o XMI aproveita o uso da UML (que se tornou padrão para o desenvolvimento de sistemas orientados á objeto), da definição da UML em MOF e da XML (que também se tornou um padrão amplamente aceito), para produzir um meio padrão para salvar modelos UML em XML (de fato, não só modelos UML mas qualquer metamodelo baseado no MOF), produzindo um meio de se trocar modelos UML entre ferramentas diferentes.

O propósito do XMI é resolver certas questões que surgem com bastante freqüência no decorrer de um desenvolvimento de *software* [14]:

1. **combinar ferramentas em um ambiente heterogêneo:** em um ambiente de desenvolvimento, não existe uma única ferramenta que implemente a solução. Uma combinação de várias ferramentas de diferentes fabricantes é necessária: ferramentas para modelagem UML, ferramentas de ambientes integrados de desenvolvimento (IDE), ferramentas de Bancos de Dados, etc. Muitas vezes não é possível haver troca de dados de maneira fácil entre estas ferramentas. A única alternativa, então, seria a conversão e tradução das informações, o que pode levar a erros. XMI resolve este problema fornecendo um formato padrão flexível e facilmente interpretável para as informações. Basicamente, é necessário apenas que a ferramenta seja capaz de salvar e carregar os seus dados no formato XMI para interagir com outra ferramenta capaz de usar o XMI.
2. **trabalhar em um meio ambiente distribuído:** o uso do XMI facilita a troca de dados e modelos pela Internet, inclusive em conexões por telefone. Por usar o formato XML, os dados podem ser disponibilizados como simples páginas em um servidor Web, facilitando a transmissão através de *firewalls.* É possível ainda tirar proveito da XSLT (*eXtensible Stylesheet Language*) [41] para transformar um modelo XMI em uma página HTML [42].

Nas seções seguintes, veremos com mais detalhes os padrões relacionados ao XMI.

5.2 eXtensible Markup Language - XML

Com a popularização e o sucesso da WWW (Word Wide Web) e do HTML (Hiper Text Markup Language), os usuários passaram a demandar mais funcionalidades [14]. O HTML, porém, não é extensível. A adição de novas funcionalidades implica em uma nova versão da linguagem (atualmente na versão 4.0). Além disso, de modo a suportar estas novas funcionalidades, as aplicações para a interpretação da linguagem foram ficando cada vez mais complexas.

Para solucionar esses problemas foi criado o padrão XML (Extensible Markup Language). O XML é uma linguagem de marcação, similar ao HTML, projetada para descrever dados. Ela foi criada como um modo fácil, porém poderoso, de se salvar dados em arquivos. O XML é um subconjunto do SGML [14] (*Standard Generalized Markup Language* – uma linguagem poderosa, porém bastante complexa, para descrever dados e criar linguagens de marcação). Sendo um padrão, possibilita ao usuário salvar os dados em um arquivo que pode ser acessado por outros aplicativos

que não o usado para criar o documento. XML permite ainda o uso de DTDs (Document Type Definition) e XSD (XML Schema Definition) para validar os documentos produzido.

Ao contrário do HTML, um documento XML não inclui a apresentação da informação. Para termos uma representação visual de um documento XML, precisamos aplicar ao mesmo um *layout*, utilizando tecnologias como a XSL (Extensible Style Language).

O XML apresenta várias vantagens que justificam a sua adoção pela OMG [12] para representar metadados.

1. XML já é um padrão aberto, independente de plataforma e de fabricante;
2. XML é neutro em relação a representação de metamodelos, podendo, então, representa metamodelos criados com o MOF;
3. XML também é neutro com relação à linguagem de programação e API (Application Programamming *Interfaces*) utilizada. Existem várias APIs disponíveis (inclusives gratuitas) para criar, interpretar e integrar documentos XML através de linguagens de programação, como o Java;
4. Documentos XML podem ser criados utilizando-se desde de editores de texto simples até ferramentas mais sofisticadas próprias para a edição destes documentos com recursos mais avançados;
5. XML possui uma sintaxe clara e simples, possibilitando sua interpretração de forma fácil por programas de computador e também por pessoas;

5.2.1 Elementos do XML

XML é um conjunto de regras para a construção de linguagens de marcação. Uma **linguagem de marcação** (em inglês *markup language*) é um conjunto de símbolos que podem ser adicionados ao texto de um documento para demarca e nomear partes do mesmo. Em um documento XML, a marcação é utilizada para definir elementos.

Um documento XML é constituído de elementos, onde cada elemento representa algum tipo de dado. Os elementos são delimitados por um par *tags,* indicado o inicio (abertura) e o fim (fechamento) de cada elemento, e podem conter outros elementos aninhados e/ou dados. A *tags* que indicam o início e o fim dos elementos começam com o caractere “<“ (menor que) e terminam com o caractere “>” (maior que). Este par de *tags* possuem o mesmo nome, sendo que o nome da *tag* que indica o fim do elemento deve vir precedido do caractere “/” (barra). O XML é extensível (*extensible*), o que significa que podemos criar nossas próprias *tags*.

No topo de um documento XML temos o que chamamos prólogo do documento (*document prolog*). O prólogo do documento tem a função de informar que o arquivo é um documento XML e qual a versão utilizada:

```
<?xml version="1.0"?>
```

Como dito antes, os elementos podem incluir outros elementos aninhados, que por sua vez podem conter outros elementos e assim por diante. Quando um elemento estiver contido em outro, sua *tag* de fechamento deve vir antes do que a do elemento em que está contido. Isto proporciona ao documento XML uma estrutura de dados em forma de árvore.

Abaixo temos a forma geral de um elemento XML:

```
<nome atributos=”valor”>
      conteúdo
</nome>
```

O texto **<***nome attributos***>** representa a abertura ou inicio de um elemento, sendo *atributos* uma lista opcional de atributos com seus respectivos valores. O texto *nome* denota o nome do elemento, sendo que a distinção entre maiúsculas e minúsculas. O *conteúdo* pode incluir outros elementos e/ou algum texto. O texto **</***nome***>** indica o fechamento ou fim do elemento. Abaixo temos um resumo das regras de sintaxe da XML. As regras são bastante simples e fáceis de usar:

- todos os elementos do documento devem possuir uma *tag* de fechamento;
- maiúsculas e minúsculas faz diferença em relação ao nome dos elementos;
- a *tag* de fechamento de um elemento que está contido em outro deve vir antes do que a do elemento em que está contido;
- os valores dos atributos devem vir entre aspas (simples ou duplas);

5.2.2 Validação

Um documento XML que segue as regras acima pode estar com a sintaxe correta, porém ao mesmo tempo estar inválido. Para ilustrar, temos abaixo duais maneiras de representar os mesmos dados:

```
<?xml version="1.0"?>
<Referencia titulo=”O Iluminado” ano=”1999”>
      <Autor nome=”Stephen King” />
</Referencia>
<?xml version="1.0"?>
<Referencia>
      <titulo>
```

```
            O Iluminado
      </titulo>
      <ano>
            1999
      </ano>
      <Autor>
            <nome>Stephen King</nome>
      </Autor>
</Referencia>
```

As duas formas estão corretas e representam a mesma informação. Mas qual delas é válida? A validade vai depender do DTD associado ao documento XML. A primeira utiliza elementos e atributos, enquanto que a segunda utiliza apenas elementos. Existem ainda diversas possibilidades de representar os mesmos dados. Se tivermos diferentes ferramentas, cada uma representado seus dados em um documento XML de forma diferente, a troca de informações entre elas seria muito difícil.

A validação dos documentos é essencial para eliminar essa ambigüidade. A seguir veremos como validar os documentos XML.

5.2.3 Document Type Definitions - DTD

O XML permite aos usuários criar suas próprias linguagens de marcação, definindo quais os elementos e atributos que melhor se ajustam à informação que se quer representar. Para que os dados sejam validados e para eliminar a ambigüidade na representação da informação, faz-se necessário ainda definir a linguagem de um modo formal (modelar). Abaixo temos algumas razões para o porquê da necessidade de modelar os documentos XML:

- a restrição do padrão dos documentos torna mais fácil escrever *softwares* capazes de interpreta-los e reduz a possibilidade de erros;
- a existência, muitas vezes, de elementos obrigatórios em documento. Definindo o documento de modo formal, asseguramos que todos os elementos obrigatórios estejam presentes;
- o usuário pode solicitar documentos de outros usuários que não conheçam a linguagem. Para isto, o modelo pode ser disponibilizado em um repositório público para ser baixado por outras pessoas;

Para modelar um documento XML utilizamos um DTD (Document Type Definition). O objetivo de um DTD é definir quais os elementos e atributos válidos de um documento XML. Deste modo, o DTD é usado para validar um documento XML. Quando interpretado, o documento

XML é comparado com o seu DTD, verificando se todos elementos obrigatórios e atributos estão presentes e se o conteúdo de cada elemento está correto.

Um documento XML pode conter dois níveis de corretude:

1. um documento XML é bem formado quando o mesmo segue todas as regras e restrições descritas na seção anterior;
2. um documento XML é válido quando, além de bem formado, esta conforme com a estrutura definida por um DTD, ou seja, contém apenas elementos e atributos definidos no DTD;

A declaração de elementos em um DTD consiste do nome do elemento, o conteúdo do elemento e os atributos do mesmo. A forma geral da declaração de um elemento é vista abaixo:

```
<!ELEMENT nome conteúdo>
<!ATTLIST nome decl. dos atributos>
```

O texto *nome* identifica o nome do elemento que fará parte do documento (lembrando que existe diferença entre maiúsculas e minúsculas). O *conteúdo* especifica o que pode ser colocado dentro dos elementos. O ATTLIST é opcional. Ele contém as declarações dos atributos que fazem parte do elemento.

Podemos apontar certas desvantagens que a sintaxe utilizada no DTD possui:

- os documentos XML possuem uma sintaxe diferente da dos DTDs. Isto dificulta a criação dos documentos por parte dos usuários, pois é necessário que o mesmo conheça as duas sintaxes;
- as declarações da lista de atributos e dos conteúdos são difíceis de se compreender;
- os tipos de dados para os atributos e conteúdos não podem ser definidos, como, por exemplo, números inteiros, números de ponto flutuante, data, etc;

A seguir veremos uma alternativa ao DTD que veio sanar estes problemas.

5.2.4 XML Schema Definition - XSD

O propósito do XML Schema Definition, assim como o DTD, é definir quais os elementos que irão fazer parte do documento XML. O XSD é uma alternativa ao DTD, mas espera-se que futuramente venha a substituí-lo. Suas principais vantagens em relação ao DTD:

- é escrito utilizando a sintaxe do XML;
- suporta definição de tipos de dados;

Todos os documentos XSD possuem um elemento raiz chamado *schema.* Utilizamos um *namespace* para definir o contexto dos elementos XML. O XSD também é um documento XML. Sendo assim, deve incluir na primeira linha o prólogo do documento:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   ...
</xsd:schema>
```

No exemplo acima, criamos o *namespace xsd* para deixar claro quais elementos fazem parte da declaração do *schema.* Note ainda que foi utilizado como elemento obrigatório *schema* como elemento raiz. Os elementos XML são declarados como se fossem atributos. Cada elemento declarado em um XSD utiliza o elemento XML chamado de *element.* O atributo *name* especifica o nome do atributo:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<xsd:element name="A"/>
<xsd:element name="B"/>
</xsd:schema>
```

No exemplo acima, utilizando o elemento *element* e atributo *name,* foram declarados dois elementos: *A* e *B*. Como não foi declarado nenhum conteúdo para os elementos, eles devem vir vazios, ou seja, sem elementos ou atributos, nos documentos XML que serão validados por este XSD.

Os elementos em XSD podem ser do tipo simples ou complexos. Os elementos complexos são aqueles que podem conter outros elementos ou atributos. Elementos simples são aqueles que contém apenas texto, não podem conter outros elementos ou atributos. Podemos ainda restringir o conteúdo destes elementos simples definindo um tipo para o elemento através do atributo *type*. De forma geral:

Complexo:

```
<prefixo:comnplexType name="nome" />
```

Simples:

```
<prefixo:element name="nome" type="tipo"/>
```

O tipo pode ser um dos tipos padrões do XSD (*string, boolean, integer, data, time, etc.*) ou um definido pelo usuário. Os elementos *complextType* e *element* são utilizados para declarar elementos complexos e simples, respectivamente.

A declaração dos atributos segue a mesma sintaxe da declaração dos elementos. Utilizamos o elemento XML *attribute*:

```
<prefixo:attribute name="nome" type="tipo" use="uso" />
```

Onde *nome* é o identificador do atributo e *tipo* dever ser um tipo simples (não complexo). O atributo *use* pode assumir os valores *required* (indica que o atributo deve ser especificado no documento XML) e *optional* (o atributo não precisa ser especificado). Podemos ainda especificar os atributos *fixed* (indica que o atributo tem um valor fixo) e *default* (para indicar um valor padrão, caso o atributo não seja especificado no documento XML). A declaração dos atributos deve vir após a declaração do conteúdo de um elemento complexo.

5.3 Meta-Object Facility - MOF

O MOF (Meta-Object Facility) [40] é uma tecnologia adotada pela OMF para a definir metadados é representá-los como objetos CORBA (Common Object Request Broker Architecture). Metadados denota um tipo de informação utilizada para descrever outras informações (por exemplos, modelos UML). As informações descritas podem ser informações utilizadas em sistemas de computação (arquivos, banco de dados, etc.).

A especificação MOF define uma linguagem abstrata e um *framework* para especificação, construção e gerenciamento de tecnologias independentes de um metamodelo específico. Além disso, define também um *framework* para implementar repositórios para armazenar metadados descritos por metamodelos.

MOF possui a capacidade de representar a informação em diversos níveis de abstração (metaníveis). As informações contidas no metanível mais alto fornecem uma representação mais abstrata da informação contida no metanível inferior. Tipicamente, são utilizados quatro níveis de abstração. Os metaníveis estão descritos na
Tabela 1.

Tabela 1 - Os diferentes níveis de abstração da informação.

Nível	**Informação**	**Exemplo**
M3	Meta-metamodelo	Modelo MOF
M2	Metamodelo (Meta-metadado)	Metamodelo UML
M1	Modelo (Metadado)	Modelo UML
M0	Dado	Conta bancária

Começando no nível mais baixo M0, a cada nível que subimos o nível de abstração aumenta. Os níveis de abstração podem ser identificados pelo número de ocorrência do prefixo meta.

No nível mais baixo M0 temos uma informação ou um dado, que pode ser representado como uma simples instância de alguma coisa. Como exemplo, temos as informações relativas a um elemento de uma conta bancária. Como foi dito no parágrafo acima, no nível superior M1, a informação representada é mais abstrata que do nível M0, chamada de metadado ou modelo. Neste nível, temos a informação representada em um modelo UML, como, por exemplo, uma classe *Elemento*. No nível superior M3, a informação é chamada de metamodelo. O metamodelo é uma linguagem abstrata para representar diferentes tipos de dados. Um exemplo de um metamodelo e o metamodelo UML. O metamodelo UML contém os construtores que são necessários para representar um modelo UML. No nível mais alto, o M4, a informação é chamada de meta-metamodelo. Neste nível temos a descrição da estrutura e semântica do metamodelo. Em outras palavras, a linguagem apara descrever metamodelos. Deste modo, o metamodelo UML é considerado uma instância do meta-metamodelo MOF e um modelo UML pode ser descrito como uma instância do metamodelo UML.

Embora seja comum o uso destas quatro camadas de abstração, o número de níveis utilizados vai depender da arquitetura descrita. Além disso, uma mesma informação pode estar contida em diferentes níveis dependendo da aplicação.

O padrão MOF suporta qualquer tipo de informação que possa ser descrita utilizando-se técnicas de modelagem orientada a objetos. Como o XMI é baseado no MOF, ele é capaz de representar informação em qualquer dos níveis da hierarquia do MOF (M0, M1, M2, M3). Particularmente, no nosso trabalho, estamos interessados no nível M1.

5.4 XMI Regras de Produção.

As regras de produção de um documento XMI ditam de que maneira serão representados os modelos e metamodelos em um arquivo XML. Nesta seção veremos como os metamodelos são produzidos e um exemplo de como representar uma classe UML utilizando o XMI. O arquivo XML contendo o diagrama de classes UML pode ser lido em qualquer editor de texto ou navegador da internet, como por exemplo o bloco de notas. O conhecimento destas regras de produção pode capacitar o usuário interpretar o arquivo sem a necessidade de uma ferramenta CASE mais elaborada.

Sendo um documento XML, o XMI é composto de duas partes: os DTDs e o documento XML. O processo de produção deste documento XML é definido através uma série de regras de produção. Estas regras são aplicadas a um modelo para gerar o documento XML. O inverso destas regras pode ser aplicado ao documento XML para recuperar o modelo. O XMI possui dois conjuntos de regras: regras para a geração de DTDs e regras para a geração de documentos [14, 43, 44].

Esses conjuntos de regras são definidos na notação EBNF (Extended Backus-Naur Form) [14]. Cada regra é numerada para referência. Os nomes das regras devem vir contidos entre "<" e ">". Partes do texto que vierem entre aspas (simples ou duplas) são interpretados como valores literais. Partes do texto que vem entre barras duplas ("//") representam os lugares onde o texto deve ser substituído com algum valor externo. Os sufixos "*", "?", "+" e "|" e os parênteses tem o mesmo significado que os apresentados na seção sobre DTDs (seção 4.2.5).

Usando o XMI, os DTDs de um metamodelo são obtidos definindo o metamodelo em MOF e então aplicando as regras de produção para gerar o DTD. O uso do padrão MOF como intermediário entre o metamodelo e as regras de produção garante que um dado metamodelo será sempre mapeado para o mesmo DTD, independente de qual aplicação será usada para gerar o metamodelo ou o DTD [43, 44].

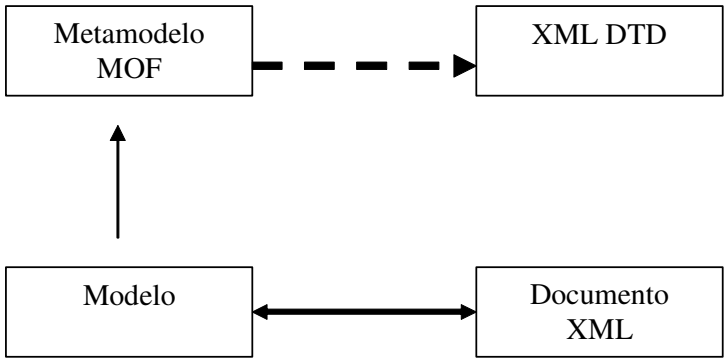

Figura 33 - Geração de DTD a partir de metamodelos MOF.

Na Figura 33, a seta tracejada indica um conjunto de regras definidas em ENBF para a geração de DTDs a partir de metamodelos MOF. A seta cheia indica um conjunto de regras para a geração de um documento XML a partir do modelo MOF. Como dito acima, aplicando o inverso destas regras é possível recuperar o modelo MOF.

Como exemplo de uma regra para produção de um DTD temos:

```
<DTD> ::= <1b:FixedContent>
          <1d:XMIAttList>?
             <2:PackageDTD>+
```

No exemplo acima, o DTD resultante será constituído de exatamente um elemento *FixedContent,* um ou zero elemento *XMIAttrList* e um ou mais elementos *PackageDTD*. Os significados de "?" e de "+" são, respectivamente, **um ou zero** e **um ou mais**. Um outro exemplo:

```
<XMI> ::= "<XMI" <2a:Namespaces>
            "version=" //XMI version//
            ("timestamp=" //timestamp//)?
            ("verified=" //verified//)? ">"
            ( <3:Header> )? ( <6:Content> )?
            ( <4:Differences> )? ( <5:Extensions> )?
          "</XMI>"
```

Neste exemplo temos a regra de geração de como o elemento raiz *XMI* deve ser definido. Como dito acima, os textos que vêm entre aspas são valores literais. O texto que vêm entre barras duplas deve ser substituído por alguma informação externa. Por exemplo, *//timestamp//* deve ser substituído pela data e hora atuais. O documento XMI assim ficaria :

```
<XMI xmi.version='1.1' xmlns:UML='href://org.omg/UML/1.3'
     timestamp="Sun Jul 11 11:31:26 2004">
    <XMI.header>
    ...
    </XMI.header>
    <XMI.content>
    ...
    </XMI.content>
</XMI>
```

Existem alguns elementos do DTD que são fixos e podem estar presentes em todos os DTD gerados a partir de modelos MOF. Estes elementos devem ser incluídos no começo dos arquivos DTD, embora isto não seja uma regra. Estes elementos são:

```
<!ELEMENT XMI (XMI.header?, XMI.content?, XMI.difference*,
XMI.extensions*) >
```

O *XMI* é o elemento raiz dos documentos XMI. *XMI.header* contém as declarações que identificam o modelo, metamodelo e o metametamodelo e, opcionalmente, documentação referente ao método de transferência do arquivo (como, por exemplo, nome e versão da ferramenta utilizada para gerar o arquivo XMI). *XMI.content* contém a informação a ser transferida. *XMI.difference* especifica a diferença entre dois documentos XMI que serão transferidos. Isto é útil quando queremos, por exemplo, realizar pequenas alterações em um grande conjunto de informações. Finalmente, *XMI.extensions* contém informação que será transferida e que não está de acordo com o metamodelo definido no cabeçalho (header). *XMI.extensions* são úteis para incluir informações privativas específicas da arquitetura ou da ferramenta utilizada para geração do documento.

Tipicamente, são utilizados para incluir informações sobre a representação gráfica do diagrama UML. Como exemplo da utilização do XMI, considere o diagrama de classes da Figura 34.

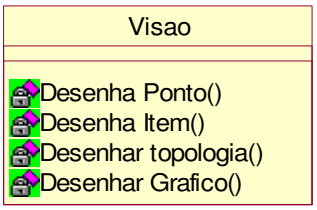

Figura 34 - Classe Visao.

Utilizando o DTD já descrito em [14], podemos gerar um arquivo XML contendo o diagrama de classes. Esse DTD foi produzido utilizando as regras de produção descritas nesta seção. O documento contém os elementos padrões do XM: *XMI.header* e *XMI.content*. Os elementos UML:Model, UML:Namespace.ownedElement, UML:Class, UML:Classifier.feature e UML:Attribute são específicos da UML e estão definidos no DTD. O documento XML pode ser visto abaixo. Esse documento foi gerado pela ferramenta XGOOD e apenas analizando o conteúdo do arquivo podemos identificar os elementos do modelo. Comentários estão em itálico.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
 <XMI xmi.version="1.1" xmlns:UML="href://org.omg/UML/1.3" >
  <XMI.header>
   <XMI.documentation>
    <XMI.exporter>XGOOD</XMI.exporter>
    <XMI.exporterVersion>1.0.0</XMI.exporterVersion>
   </XMI.documentation>
<!– O nome do metamodelo -->
   <XMI.metamodel xmi.name="UML" xmi.version="1.3" />
  </XMI.header>
<!– O modelo em si esta contido a seguir  -->
  <XMI.content>
    <UML:Model xmi.id="G.0" name="Simulador">
     <UML:Namespace.ownedElement>
<!– Classe Visao  -->
       <UML:Class xmi.id="Visao" name="Visao" namespace="G.0">
<!– Atributos/Metodos da classe -->
        <UML:Classifier.feature>
         <UML:Operation   xmi.id="AA.Element_110"   name="Desenha   Ponto"
       visibility="private" />
```

```
        <UML:Operation   xmi.id="AA.Element_108"   name="Desenha   Item"
      visibility="private" />
        <UML:Operation  xmi.id="AA.Element_76"  name="Desenhar  topologia"
      visibility="private" />
        <UML:Operation  xmi.id="AA.Element_74"  name="Desenhar  Grafico"
      visibility="private" />
        </UML:Classifier.feature>
        </UML:Class>
       </UML:Class>
     </UML:Namespace.ownedElement>
    </UML:Model>
   </XMI.content>
</XMI>
```

5.5 Conclusões

Vimos que o XMI utiliza a sintaxe XML para representar qualquer informação, em diversos níveis de abstração, modelada através da orientação a objetos. Em especial neste trabalho estamos interessados mais especificamente no diagrama de classes da UML. Vimos como um diagrama de classes pode ser representado utilizando o XMI. O XMI faz uso dos padrões XML, UML e MOF para oferecer uma forma padrão que possibilite a troca de informações e metamodelos entre ferramentas diversas. Essa padronização diminuí a diferença entre as ferramentas CASE, minimizando o esforço de integração entre elas. Entre as vantagens de sua adoção temos:

- intercâmbio aberto de meta-metamodelos MOF, UML e entre diversas ferramentas;
- infra-estrutura existente em XML/HTML para publicação de metadados na Web é melhor aproveitada;
- as diversas ferramentas CASE se tornam compatíveis;
- a troca de informação de metamodelos na Web para desenvolvedores que trabalham em ambientes remotos é facilitada;

Devido a essas vantagens, adotamos o XMI para representar os modelos gerados pela nova ferramenta proposta neste trabalho. Assim, esses modelos gerados poderão ser importados por diferentes ferramentas CASE, ou até mesmo publicado em página na internet e lido por qualquer navegador padrão.

6 Ferramenta XGOOD – eXtended Goal Into Object Oriented Development

No capítulos 3 vimos duas técnicas para a modelagem de requisitos: a técnica i*, ideal para realizar a modelagem dos requisitos organizacionais, requisitos não funcionais e requisitos funcionais dos estágios iniciais do desenvolvimento, dando uma visão geral do sistema a ser desenvolvido, de como ele irá afetar a organização na qual irá atuar; e, a UML, capaz de modelar os requisitos funcionais do sistema nos estágios posteriores do desenvolvimento, sendo os modelos gerados por essa técnica capazes de se transformar em código fonte diretamente; ideal para os estágios posteriores do desenvolvimento, onde se faz necessário um modelo mais concreto e que represente o sistema de forma mais detalhada.

Foram vistas também as diretrizes de mapeamento, cujo objetivo é manter a consistência e o rastreamento entre o sistema a ser desenvolvido e o os objetivos da organização. Dessa forma, aumentam-se as chances do sistema em desenvolvimento vir a atender os objetivos da organização de uma forma mais precisa.

O XGOOD (*eXtended Goal Into Object Oriented Development*) se trata de uma ferramenta para auxiliar o mapeamento do modelo *i** diretamente para o digrama de classes em UML. A versão anterior da ferramenta, o GOOD [11], não permitia selecionar de forma individual os elementos que iriam participar do mapeamento e seu funcionamento estava restrito à ferramenta Rational Rose [30], uma ferramenta utilizada para a modelagem UML. Além disso, não eram suportados os elementos do modelo i* como: atores, papéis e posições. A ferramenta XGOOD tenta sanar esses problemas, contendo algumas funcionalidades adicionais em relação à ferramenta GOOD anteriormente desenvolvida:

- adoção do formato XMI [13, 14] (suportando tanto as versões 1.0 e 1.1) para representar os modelos – que sendo um padrão adotado e reconhecido pela OMG facilita o compartilhamento dos modelos entre diversas ferramentas CASE (Rational Rose, ArgoUML, Poseidon, MagicDraw UML, etc), não se restringindo à uma única ferramenta comercial;
- opção para selecionar as diretrizes de mapeamento adequadas, ou seja, quais elementos serão mapeados - possibilitando ao usuário excluir certos elementos capturados segundo a técnica *i**, que não são computacionais no modelo UML.
- adaptação à inclusão de novas diretrizes de mapeamento, ou seja, o XGOOD suportará mais diretrizes relacionadas a novos elementos capturados pelos modelos *i** (ex.: papel, posição e agente).

Na Figura 35 vê-se o princípio de funcionamento da ferramenta. O arquivo "*.tel" gerado pela ferramenta OME [29] contendo o modelo *i** é lido pela ferramenta XGOOD. A partir do XGOOD é gerando então o arquivo XMI (contendo o modelo de classes UML). Temos o diagrama de classes completo, que pode ser lido por diversas ferramentas CASE: MagicDraw, Rational Rose, Telelogic Tau, etc.

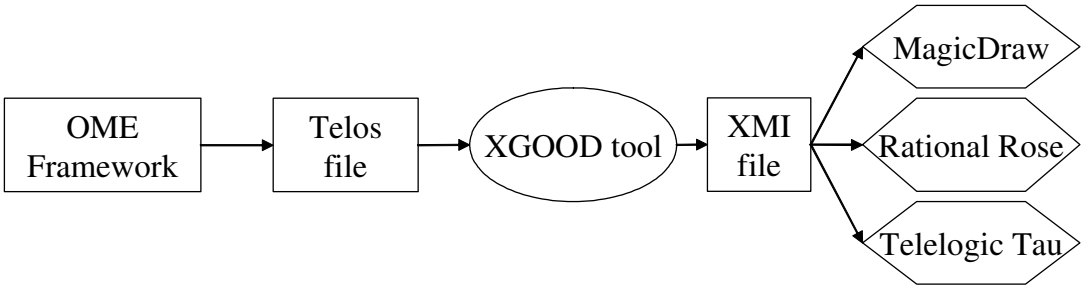

Figura 35 - Princípio de funcionamento da nova ferramenta XGOOD.

Para o desenvolvimento da ferramenta foi utilizada a linguagem C++ e o ambiente de desenvolvimento integrado Visual C++ versão 6.0, tendo-se um primeiro protótipo em teste. A linguagem C++ é amplamente difundida entre os profissionais de desenvolvimento de *software*. É uma linguagem bastante poderosa, sendo utilizada para o desenvolvimento de diversos aplicativos comerciais. Suas principais vantagens são a velociadade de execução e o baixo consumo de memória de seus aplicativos. Outro motivo da utilização do C++ é o nosso conhecimento nesssa linguagem em específico. Todavia, é possível a utlização de outras linguagens, por examplo Java [57]; essa é uma sugestão que será deixada como trabalho futuro, mesmo porque o próprio OME foi desenvolvido em Java.

O XGOOD funciona no sistema operacional Windows e, atualmente, implementa as diretrizes D1, D2, D3, D4 e D5. Infelizmente, o padrão XMI não possui mecanismos ainda para representar as restrições em OCL (*Object Constraint Language*) [46], assim como nem todas as ferramentas do mercado suportam a inclusão de restrições OCL. Porém, na nova versão do OCL (OCL 2.0 [47]) esta previsto a geracão de um DTD XMI para o metamodelo OCL. Essa nova versão deverá está disponível em breve. Por isso, as diretrizes D6 e D7 foram deixadas de fora nessa versão do XGOOD, mas poderão vir a ser extensãos deste trabalho.

Foram testadas as ferramentas CASE Rational Rose (testado nas versões 2002 e 2003) [30], Telelogic Tau (testado na versão 2.2.30.691) [31] e MagicDraw (testado nas versões 7.1, 7.5) [32]. Todas essas conseguiram importar o modelo gerado pelo XGOOD sem problemas e também possuem grande aceitação no mercado e versões de demonstração gratuitas. Foram utilizadas as *Microsoft Foundation Classes* (MFC) [48] para construir a interface gráfica com o usuário.

A interação com o usuário é feita através de menus ou de botões, localizados na barra de ferramentas da ferramenta XGOOD. Na Figura 36 vê-se a tela da ferramenta (em sua versão mais recente) mostrando como é feita a seleção da diretriz adequada, para um recurso (*Elementos*) modelado segundo a técnica *i** para o model do simulador que foi desenvolvido. A figura mostra como o modelo i* é visualizado, após ser aberto na ferramenta.

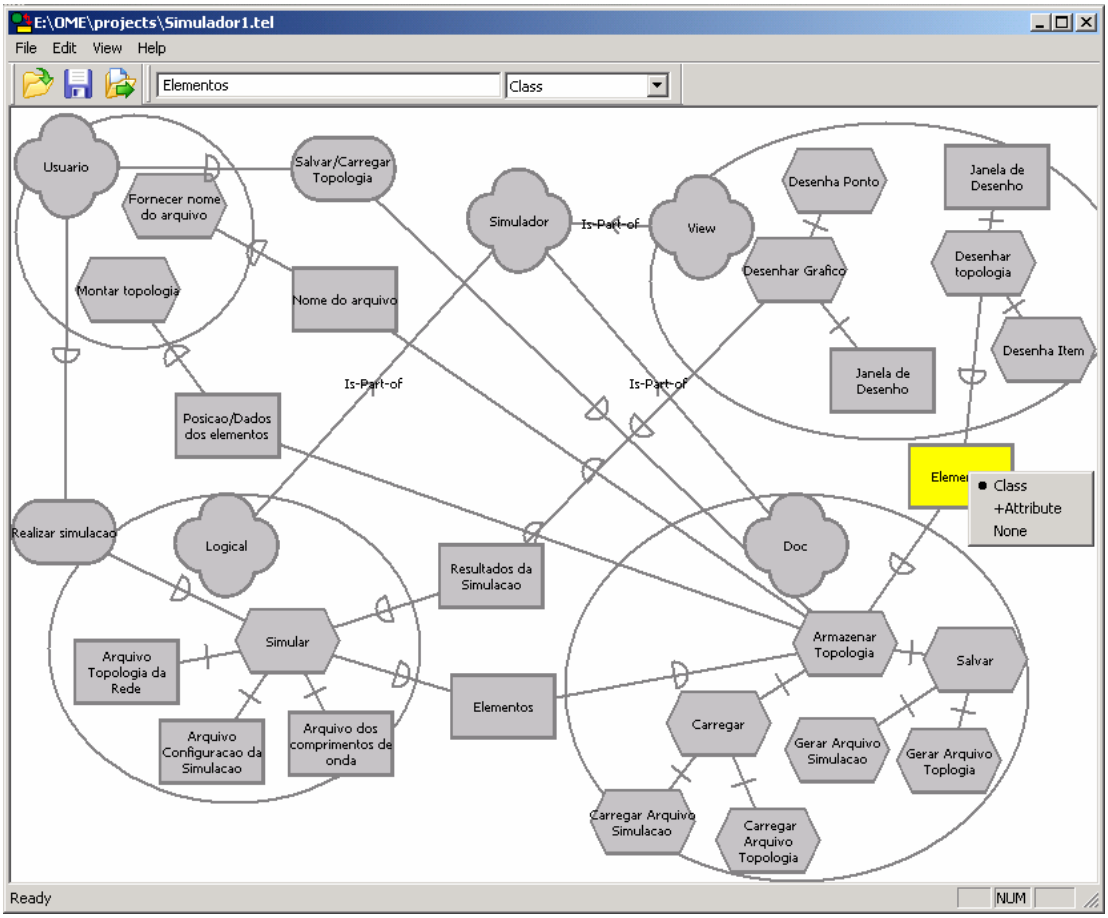

Figura 36 - Ferramenta XGOOD, versão atual.

Muitos elementos do modelo *i** possuem mais de uma opção de mapeamento. Na seção 3.3 vimos que um elemento recurso, por exemplo, do modelo i* pode ser mapeado como uma classe ou um atributo de uma classe no diagrama de classes UML. Além disso, alguns elementos não devem fazer parte do diagrama de classes. Por exemplo, atores que representam indivíduos ou papéis em uma organização (gerentes, usuários, etc.) externos ao sistema ou a organização que está sendo modelada e os elementos a eles ligados podem não fazer parte do sistema de *software* que está sendo desenvolvido.

Em vista disso, é dada ao usuário a opção de selecionar a diretriz de mapeamento adequada para cada elemento. No total, estão disponíveis oitos opções possíveis para a seleção dos mapeamentos: ***Class, Interface, +Attribute, +bool Attribute, -Method, -Attribute, -bool Attribute, None***. O sinal "**+**" indica que o elemento será público e o sinal "**–**" privado. A opção ***None*** indica que o elemento não deve ser mapeado. As opções não são todas visíveis ao mesmo tempo, isto vai depender do elemento e da diretriz de mapeamento. Por exemplo:

- diretriz D1 - agentes, papéis ou posições no modelo i* podem ser mapeados em classes em UML, logo, para esses elementos, as únicas opções possíveis são ***Class*** e ***None*****.**
- diretriz D2.1 – uma tarefa do modelo SD pode ser mapeada como uma operação de uma interface. Para esse elemento as opções disponíveis são ***Interface*** e ***None;***
- diretriz D2.2 - uma tarefa do modelo SR pode ser mapeada como uma operação de uma classe com visibilidade privada. Para esse elemento as opções disponíveis são ***-Method,*** e ***None;***
- diretriz D3.1 – um recurso no modelo SD pode ser mapeado como uma classe. As opções disponíveis são ***Class*** e ***None*****;**
- diretriz D3.2 – um recurso no modelo SR pode ser mapeado como uma classe ou um atributo de uma classe. Nesse elemento as opções ***Class, -Attribute*** e ***None*** estão disponíveis para o usuário.
- diretriz D4.1 – um objetivo no modelo SD pode ser mapeado como um atributo boleando com visibilidade pública em uma classe. Opções visíveis: ***+bool Attribute*** e ***None***;
- diretriz D4.2 – um objetivo no modelo SR pode ser mapeado como um atributo boleando com visibilidade privada em uma classe. Opções visíveis: ***-bool Attribute*** e ***None***;
- diretriz D5.1 – um objetivo *soft* no modelo SD pode ser mapeado como um atributo do tipo número com visibilidade pública em uma classe. Opções visíveis: ***+Attribute*** e ***None***;
- diretriz D5.2 – um objetivo *soft* no modelo SR pode ser mapeado como um atributo do tipo número com visibilidade privada em uma classe. Opções visíveis: ***-Attribute*** e ***None***;

Um recurso, por exemplo, só terá as opções ***Class, -Attribute*** e ***None*** visíveis, de acordo com a diretriz D3 (Figura 36 – recurso *Elementos*). Caso o usuário selecione um ator com a opção ***None*** (i.e. o ator não será mapeado), automaticamente seriam excluídos todos os elementos (se houver algum) do modelo SR correspondentes ao ator, assim como todos as dependências ligadas a ele.

6.1 Utilização da Ferramenta

O uso da ferramenta é bastante intuitivo e simples. Basicamente, existem três passos que o usuário deve executar:

- Passo1: O usuário deve selecionar o arquivo contento o modelo *i** e abrí-lo na ferramenta. Para isso ele deve acessar o menu *File* >> *Open* (Figura 37) e selecionar o arquivo "*.tel" desejado (Figura 38) e clicar em *Open*.

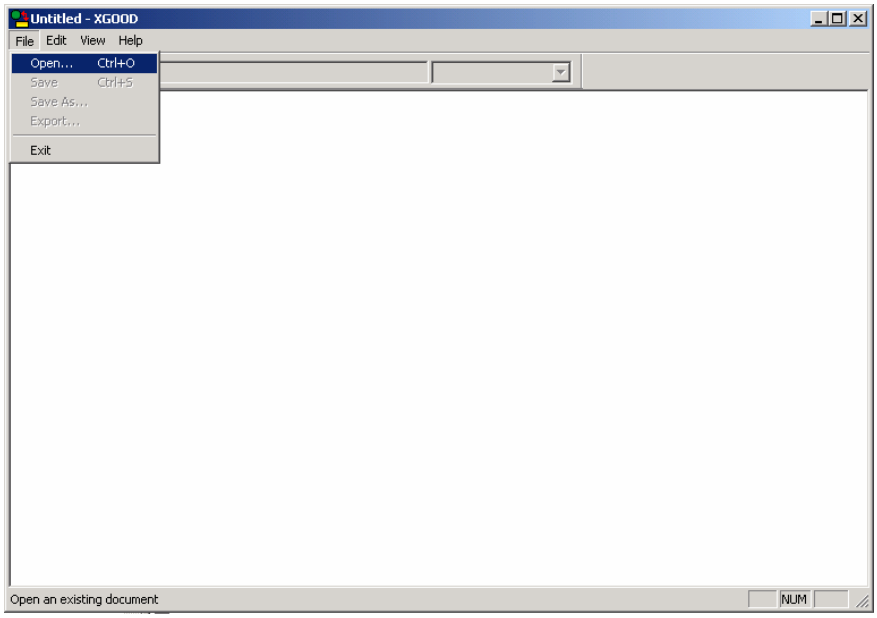

Figura 37 - Uso da ferramenta: abrindo um modelo *i.**

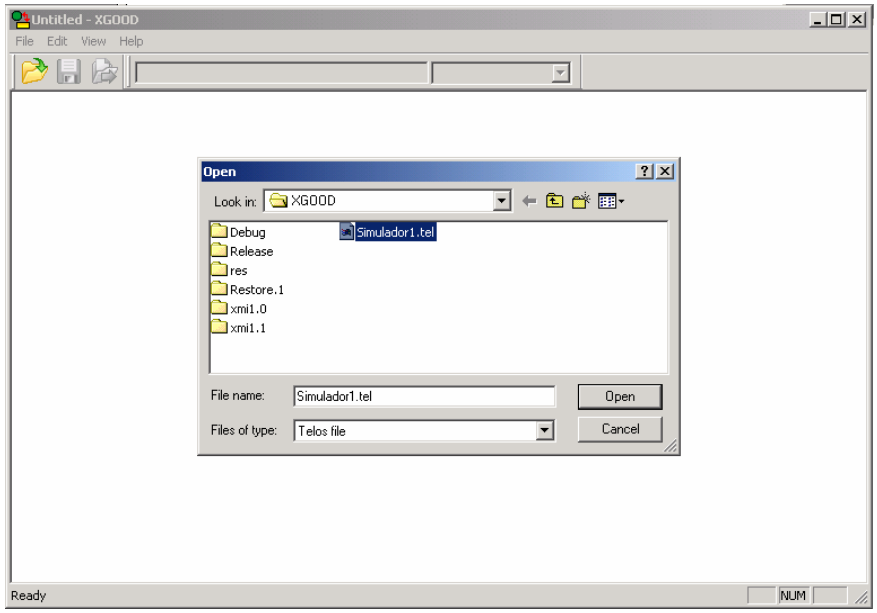

Figura 38 - Uso da ferramenta: selecionando o modelo i*

- Passo1: Após feito isso, o modelo i* selecionado será mostrado na ferramenta (Figura 39);

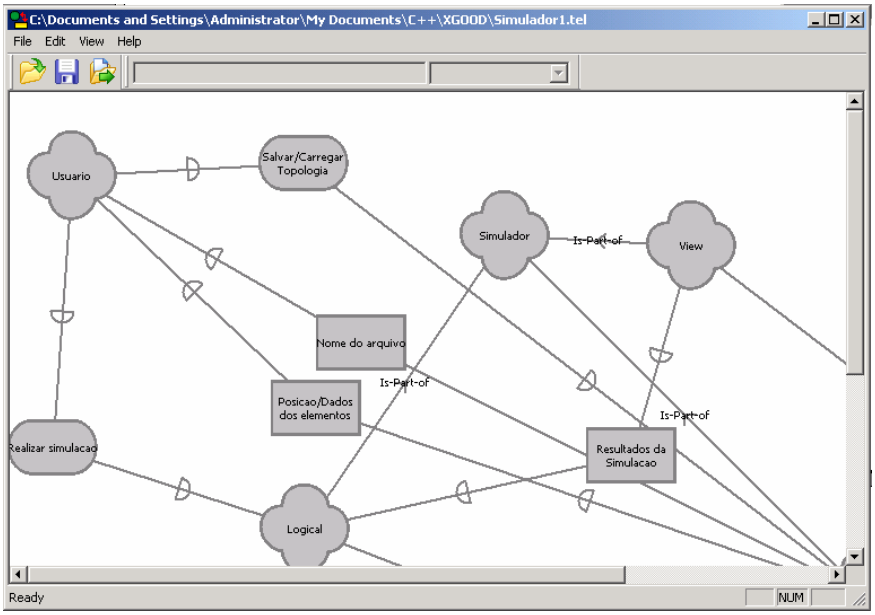

Figura 39 - Uso da ferramenta: modelo i* importado na ferramenta.

- Passo 2: O usuário tem a opção de selecionar diretriz de mapeamento que ele julgar mais adequada para cada elemento, selecionando quais elementos deverão fazer parte do diagrama de classes UML que será gerado. Para isso o usuário deve clicar com o botão direito do *mouse* sobre o elemento e selecionar uma das opções que aparecem (Figura 40). Neste caso em questão, o elemento *Usuario,* que é uma posição e que está submetido a regra D1, apresenta as opções ***Class*** e ***None***. O usuário também pode, caso queira, mudar o nome dos elementos do modelo i* (Figura 41 – a parte em destaque mostra o local onde o usuário pode alterar o nome dos elementos. No caso em questão o elemento *Visao*);

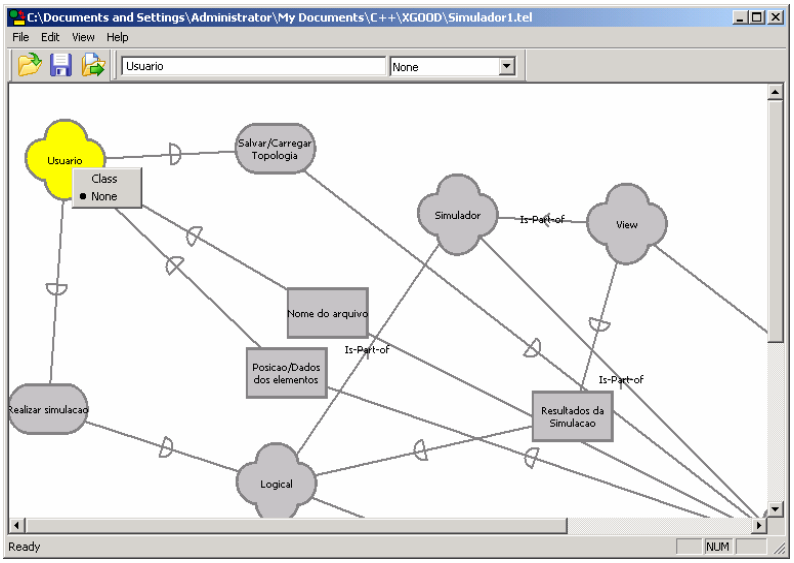

Figura 40 - Uso da ferramenta: seleção das diretrizes.

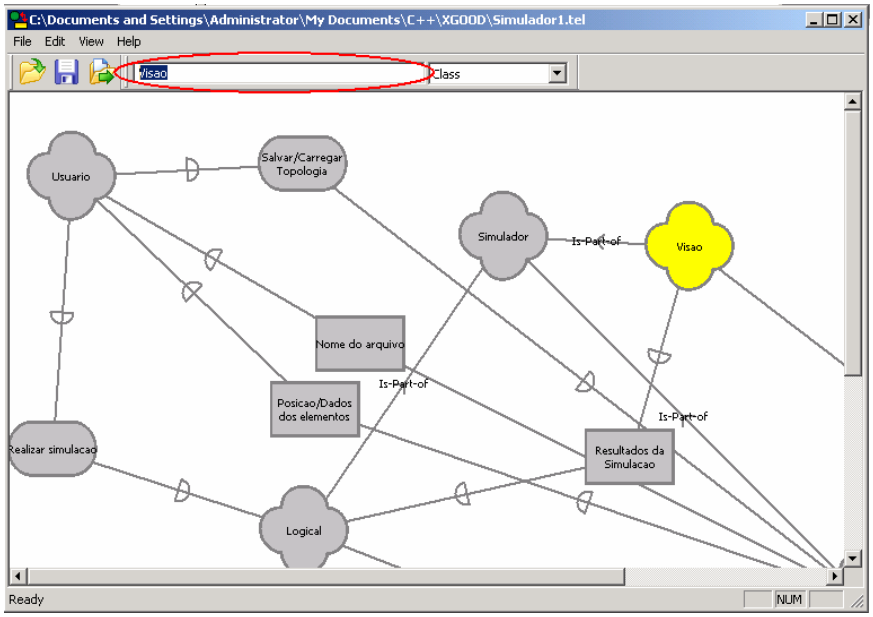

Figura 41 - Uso da ferramenta: alterando o nome dos elementos.

- Passo 3: O usuário pode gerar o diagrama de classes UML acessando o menu *File* >> *Export*... (Figura 42). No campo *File name* o usuário deve entrar o nome do arquivo. No campo *Save as type* devemos informa qual o formato do arquivo. É possível escolher entre os formatos "*.xmi" versão 1.0 ou "*.xmi" versão 1.1. A ferramenta irá então converter os elementos do modelo i* em elementos do diagrama de classes, de acordo com as diretrizes escolhidas na etapa anterior.

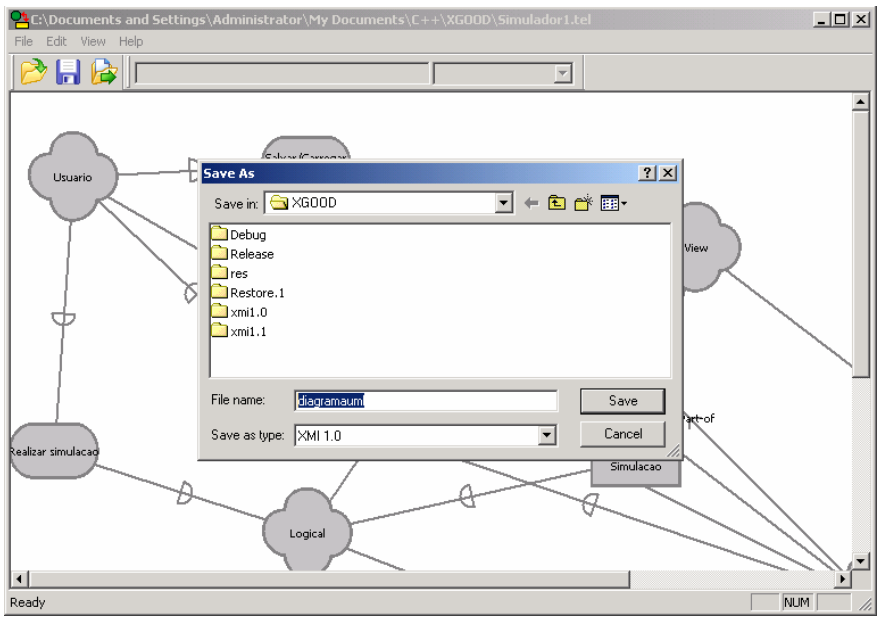

Figura 42 - Uso da ferramenta: gerando diagrama de classes UML.

- Passo 4: O arquivo gerado pela ferramenta pode, então, ser importado em outra ferramenta de modelagem UML. Para o caso do Rational Rose, por exemplo, basta acessar o menu *Tools->UML 1.3 XMI Addin->UML 1.3 XMI Import*, selecionar o arquivo gerado previamente e clicar em Open (Figura 43). O diagrama gerado pelo XGOOD irá aparecer no Rational Rose e poderá se trabalhado normalmente (Figura 44). O *layout* dos elementos irá seguir os dos elementos do modelo *i**. Se, por exemplo, um ator está localizado na parte superior à esquerda, a classe correspondente também irá se localizar no parte superior à esquerda.

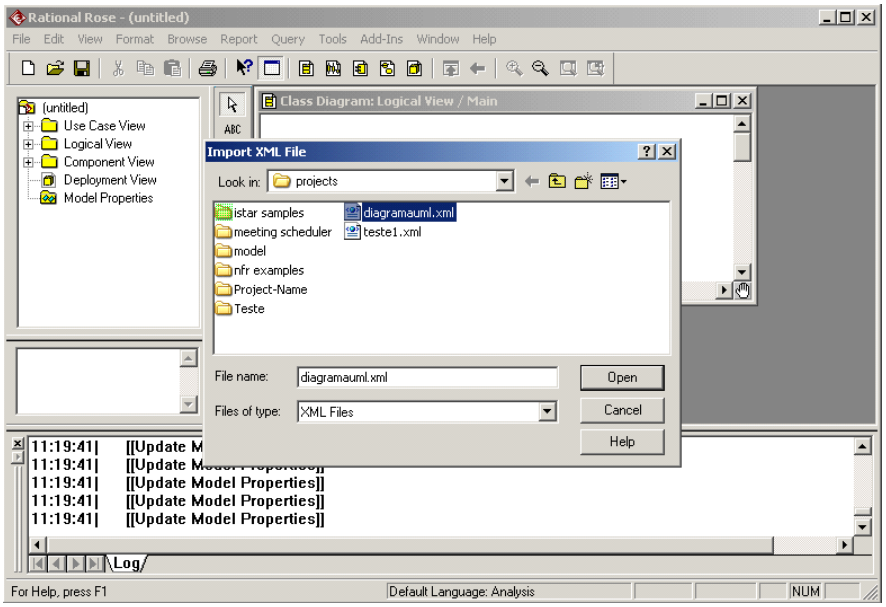

Figura 43 - Uso da ferramenta: importando diagrama no Rational Rose.

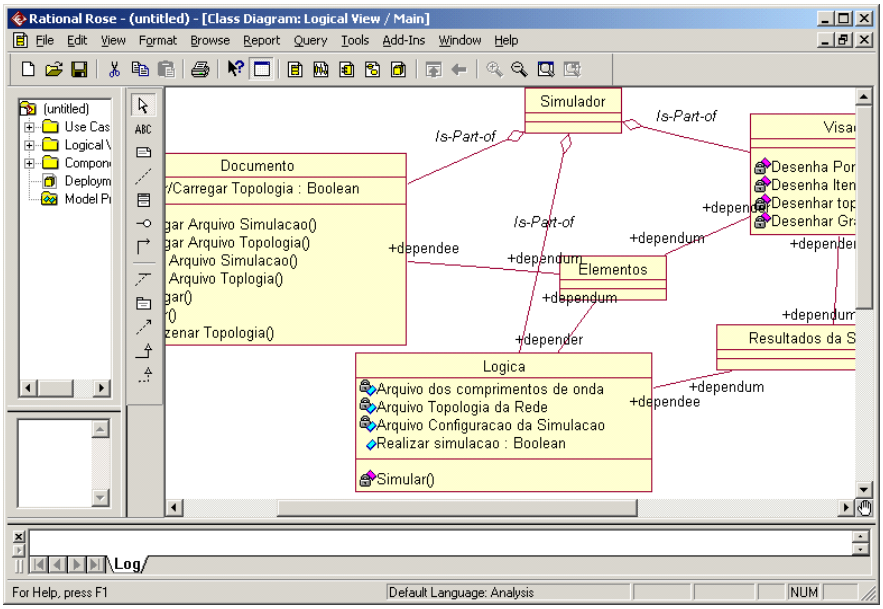

Figura 44 - Uso da ferramenta: diagrama importado no Rational Rose.

Também é possível salvar as diretrizes de mapeamento selecionadas para o modelo *i**, para que não seja necessário selecionar novamente todas as diretrizes para cada elemento. Para isso o usuário deve acessar o menu *File* >> *Save*. Ao salvar as opções de mapeamento, é criada uma cópia de segurança do arquivo original (extensão "*.~tel").

A ferramenta é compostas dos seguintes arquivos, que devem estar em diretórios específicos (mais detalhes na seção 6.3.1). Esses arquivos são distribuídos junto com a ferramenta XGOOD:

- *XGOOD.exe* – arquivo executável da ferramenta. O usuário executa este arquivo para iniciar o funcionamento da ferramenta;
- *xmi1.0* – subdiretório contendo os arquivos necessários para a geração do arquivo ".xml" no padrão XMI 1.0. Este subdiretório deve estar presente no mesmo diretório onde o arquivo *XGOOD.exe* se encontra;
- *xmi1.1* – diretório contendo os arquivos necessários para a geração do arquivo ".xml" no padrão XMI 1.1. Também deve estar presente no mesmo diretório onde o arquivo *XGOOD.exe* se encontra;
- *XGOODPlugin.class* – classe que implementa um *plugin* para a integração da ferramenta XGOOD com a ferramenta OME (maiores detalhes na seção 5.3);
- *XGOODMethod.class* - classe que implementa os métodos utilizados pelo *plugin* (maiores detalhes na seção 5.3);
- *SavePlugin.class* - classe que implementa um método para salvar o modelo *i**. Utilizada pela classe *XGOODMethod* (maiores detalhes na seção 5.3);

Veremos agora a arquitetura de *software* da ferramenta XGOOD, analisando o diagrama de classes da mesma, e, posteriormente, como é realizada a integração com a ferramenta OME.

6.2 Arquitetura

A ferramenta, desenvolvida em C++, possui basicamente duas camadas: uma camada de **lógica e armazenamento apresentação**; e uma camada de **apresentação** (Figura 45).

Figura 45 - Arquitetura da ferramenta XGOOD.

A camada de **lógica e armazenamento** é responsável por:

- lê o arquivo em sintaxe *Telos* contendo o modelo *i**;
- transforma os elementos *i** contidos no modelo em objetos e transformá-los em elementos do diagrama de classes UML (classes, atributos, interface, etc.);
- exporta o modelo para um arquivo ".xml" contendo o diagrama de classes no padrão XMI;
- salva as diretrizes de mapeamento selecionadas;

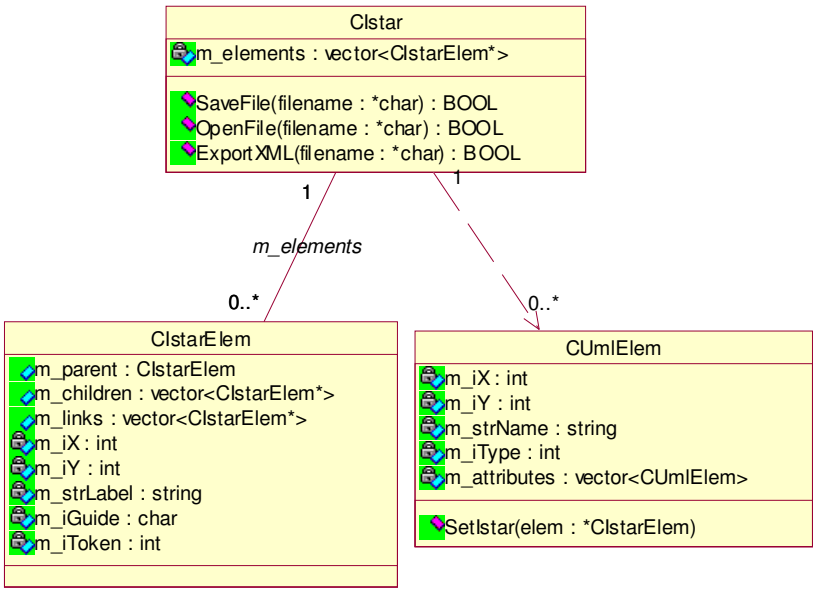

Figura 46 - Diagrama de classes: camada de lógica e armazenamento.

Na Figura 46 vê-se o diagrama de classes da camada de **lógica e armazenamento**. A classe *CIstar* é responsável por ler o modelo *i** (método *Cistar::OpenFile)* e transformar os elemento *i** em objetos do tipo *CIstarElem.* O método *CIstar::ExportXML* transforma os objetos *CIstarElem* em objetos do tipo *CUmlElem* através do método *CUmlElem::SetIstar* e salva o modelo de classes correspondente em um arquivo ".xml" no padrão XMI.

A classe *CistarElem* é utilizada para representar os elementos do modelos *i**. Esta classe possui um conjunto de atributos para representar as características de um elemento, como o seu tipo (*m_iToken*), a posição no diagrama (*m_iX, m_iY*), o seu nome (*m_strLabel*), o elemento ao qual pertence (*m_parent*), os elementos que estão contidos nele (*m_children*), os relacionamento com outros elementos (*m_links*) e qual a sua diretriz de mapeamento (*m_iGuide*). Esta classe está relacionada com a classe *CIstar* através de um relacionamento de 1 para muitos, ou seja, um objeto *CIstar* pode instanciar nenhum ou vários objetos do tipo *CistarElem.*

Para representar os elementos do diagrama de classes temos a classe *CUmlElem*, que armazena a posição no diagrama (*m_iX, m_iY*), o tipo de elemento (*m_iType*), o nome do elemento (*m_strName*) e os seus atributos (*m_attributes*). Também está relacionada com a classe *CIstar* através de um relacionamento de 1 para muitos, ou seja, um objeto *CIstar* pode instanciar nenhum ou vários objetos do tipo *CistaUml*.

A camada de **apresentação** é responsável pela criação da interface gráfica com o usuário (janelas, botões, barra de ferramentas, etc.). Esta camada realiza a apresentação visual do modelo, e recebe e processa as entradas do usuário. Foi desenvolvida utilizando-se as *Microsoft Foundation*

Classe (MFC). As *MFCs* são uma biblioteca de classes desenvolvidas pelas *Microsoft* que encapsulam grande parte das rotinas da API (*Application Program Interface*) do Windows, reduzindo o esforço por partes dos usuários no desenvolvimento de aplicativos para a plataforma Windows.

Na Figura 47 podemos visualizar o diagrama de classes da camada de **apresentação**. As classes seguem o padrão para o desenvolvimento de aplicativos utilizando as *MFC*. As classes padrões da *MFC* estão em cinza. São criadas especializações destas classes para o desenvolvimento da ferramenta.

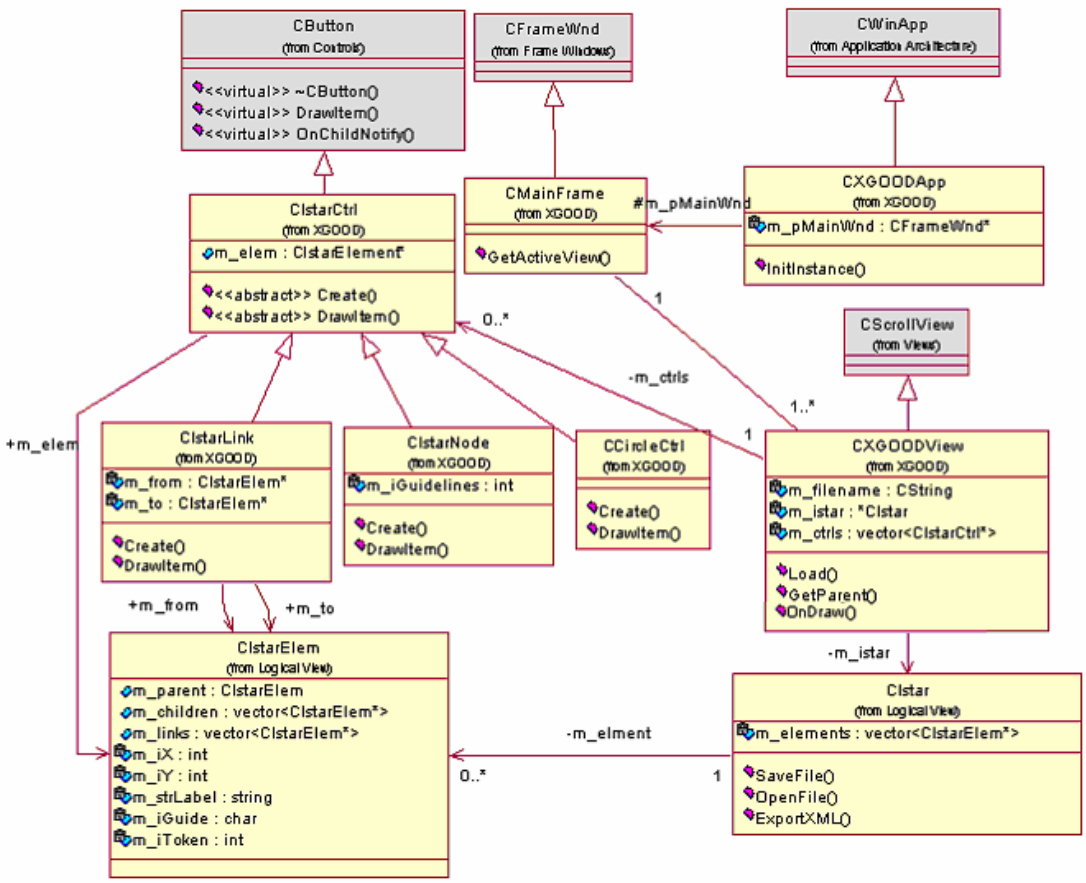

Figura 47 - Diagrama de classes: camada de apresentação.

Temos uma classe denominada *CXGOODApp* que cria uma instância do aplicativo e das classes *CMainFrame* e *CXGOODView*, através de uma chamada ao método *CXGOODApp:InitInstance*. A janela principal do programa é representada pela classe *CMainFrame.* A classe *CXGOODView* representa a **visão** do aplicativo. Esta classe é responsável pela visualização do modelo *i** que atualmente está sendo manipulado. Ela desenha todos os

elementos do modelo *i**, armazenados no atributo *m_ctrls*, que é uma coleção de objetos do *CIstarCtrl.* O método *CXGOODView::Load* é utilizado para carregar o modelo *i** a partir de um arquivo ".tel", contendo o modelo em sintaxe *Telos*. Este método realiza uma chamada ao método *CIstar::OpenFile* do atributo *m_istar*. O método *CXGOODView::OnDraw* é invocado toda as vezes que é necessário redesenhar os controles e janelas do aplicativo.

A classe *CistarCtrl* é utilizada para representar os elementos do modelo *i** como se fossem controles (botões) do Windows. O atributo *m_elem*, do tipo *CIstarElem,* aponta para o objeto do modelo *i** a ser representado (ligações, atores, recursos, etc.). As especializações dessa classe *CIstarLink, CIstarNode, CCircleCtrl* são utilizada para representar, respectivamente, um relacionamento de ligação qualquer (dependência, meio-fim, decomposição, etc.); um elemento do modelo *i** (ator, recurso, objetivo, etc.); um círculo. Essas classes implementam os métodos abstratos *CistarCtrl::Create* (invocado quando o controle e criado), e *CistarCtrl::DrawItem* (chamado toda vez que é necessário desenhar o controle). A classe *CistarLink* em particular possui dois atributos do tipo *CElemIstar: m_from* (que aponta para a origem da ligação) e *m_to* (que aponta para o destino da ligação).

A ferramenta XGOOD pode ainda ser integrada junto com a ferramenta OME, possibilitando ao usuário uma alternativa mais rápida para transformar seus modelos *i** em diagrama de classes UML.

6.3 Integração com a Ferramenta OME

Na seção **4.2** vimos que é possível estender a funcionalidade da ferramenta OME através dos *plugins.* Para facilitar o uso da ferramenta, foi desenvolvido um *plugin* que incorpora a ferramenta XGOOD à ferramenta OME. Deste modo, é possível gerar o arquivo ".xml" contendo o diagrama de classes UML diretamente a partir da ferramenta OME, sem a necessidade de se executar ou abrir outra ferramenta. O *plugin* é composto de vários arquivos.

Primeiramente, devemos ter um meio de salvar o modelo que esteja sendo construído. Isto é feito através da classe *SavePlugin*. Esta classe deve estar contida no pacote *OME*. Deste modo, podemos acessar o método protegido[1] *View::getModel.save*. Invocando o método *SavePlugin::Save*, o modelo *i** é salvo no arquivo ".tel".

```
package OME;

public class SavePlugin {

      static public void Save(View v) throws Exception {
            v.getModel().save(v.getSavePathname());
```

[1] Em java, métodos protegidos só podem ser acessados por classes do mesmo pacote.

```
        }
}
```

A Classe *XGOODPlugin*: implementa (*implements*) a interface *OMEPlugin*. Esta é a classe principal do *plugin*:

```
public class XGOODPlugin implements OMEPlugin {
   …
}
```

Nesta classes XGOODPlugin podemos destacar o método *getToolbarMethods*. Esse método é responsável por adicionar novos botões aos já existentes na barra de ferramentas do OME. Na implementação deste método, é adicionado um objeto do tipo *XGOODMethod* a barra de ferramentas do OME.

```
        …
        public Collection getToolbarMethods(View v) {
              popupmethods.add(new XGOODMethod(v));
              return popupmethods;
        }
        …
```

A classe *XGOODMethod.class* implementa (*implements*) a interface *PluginMethod*. Esta é a classe que implementa os métodos do *plugin*:

```
import OME.SavePlugin;

class XGOODMethod implements PluginMethod {
   …
}
```

Entre os métodos da classe *XGOODMethod.class* destacamos o *invoke()*. Este método é chamado todas as vezes que o botão ou menu correspondente for acessado na ferramenta OME. Este método salva o modelo *i** que atualmente está sendo editado e abre o mesmo na ferramenta XGOOD.

```
        …
        public void invoke() {
              try {
                  SavePlugin.Save(v);
                  // Executa a ferramenta
                  Process p = Runtime.getRuntime().exec(".\\XGOOD.exe \"" +
                  v.getSavePathname() + "\"");
                  // Espera que o usuário feche a ferramenta
                  p.waitFor();
              }
              catch (Exception ex) {
                System.out.println(ex);
              }
        }
        …
```

6.3.1 Instalando a ferramenta XGOOD

Para que a ferramenta XGOOD fique integrada ao ambiente da ferramenta OME, faz-se necessário copiar os arquivos da ferramenta em diretórios específicos dentro do diretório principal da ferramenta OME. Esses arquivos são distribuídos junto com a ferramenta. Considerando, por exemplo, que a ferramenta OME está localizada no diretório "OME", o usuário deve copiar os arquivos e classes que acompanham a ferramenta XGOOD nos diretórios descritos a seguir:

- *XGOOD.exe* – deve ser copiado para o diretório "OME\program\";
- *xmi1.0* – deve ser copiado para o diretório "OME\program\";
- *xmi1.1* – deve ser copiado para o diretório "OME\program\";
- *XGOODPlugin.class* – deve ser colocado no diretório "OME\program\plugins\";
- *XGOODMethod.class* - deve ser colocado no diretório "OME\program\plugins\";
- *SavePlugin.class* - deve ser colocado no diretório "OME\program\classes\OME\";

Pode-se visualizar na Figura 48 a barra de ferramentas do OME antes da instalação do XGOOD e na Figura 49 após a integração do XGOOD com a ferramenta OME. Comparando as duas figuras observamos que são criados dois novos botões na barra de ferramentas. O botão "XGOOD – Guidelines" executa a ferramenta mostrando o modelo *i** que está sendo construído atualmente. O usuário pode então seleciona as regras de mapeamento de cada elemento e gerar o arquivo ".xml" contendo o diagrama de classes. O botão "XGOOD – Export" exporta o modelo *i** diretamente para o diagrama de classes. Caso o usuário não tenha definido as regras de mapeamento anteriormente (através do botão "XGOOD – Guidelines"), serão utilizadas as regras de mapeamento *default* (todos os elementos são mapeados, atributos serão transformados em classes.).

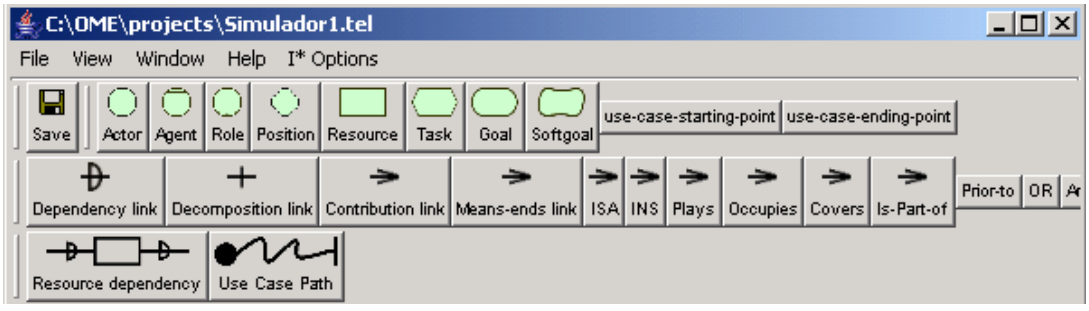

Figura 48 - Ferramenta OME antes da integração do XGOOD.

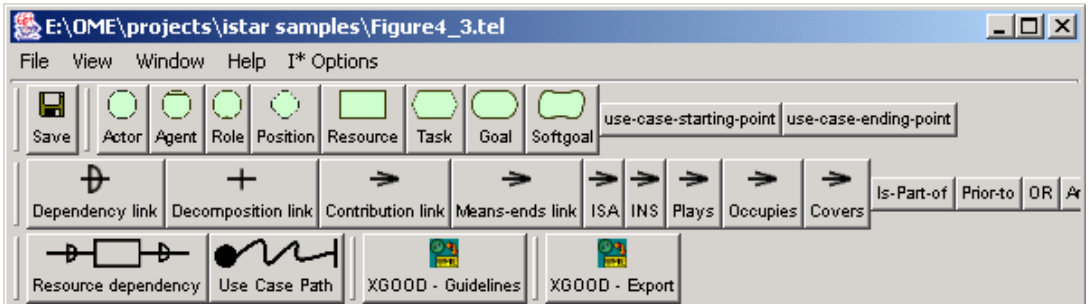

Figura 49 - Ferramenta XGOOD incorporada a ferramenta OME.

6.4 Conclusões

Foi apresentada neste capítulo a ferramenta XGOOD, que auxilia o mapeamento do modelo *i** diretamente para o digrama de classes em UML, através da automatização das diretrizes de mapeamento. Vimos também quais as vantagens dessa nova ferramenta em relação à ferramenta anterior GOOD: permitir a seleção dos elementos do modelo i* de forma individual; adotar o formato XMI para representação dos modelos, possibilitando ao usuário escolher qualquer ferramenta CASE para modelagem UML que suporte esse padrão; integração com a ferramenta OME, agilizando a utilização da ferramenta XGOOD. Explicamos também como o usuário deve instalar a ferramenta e o como utilizá-la, mostrando passa a passo como ler o modelo *i*;* realizar a a seleção de cada um de seus elementos;e gerar o diagrama de classes.

O desenvolvimento da ferramenta como um todo foi um grande desafio, mas um dos grandes problemas encontradas na implementação da ferramenta foi o de como dar ao usuário a opção de selecionar de forma individual os elementos. Isto implica em não só ler e interpretar o modelo *i** armazenado no arquivo ".tel", mas também mostrá-lo graficamente, de forma semelhante a ferramenta OME, para que o usuário da OME identificasse os mesmos elementos na ferramenta XGOOD. Além disso, o usuário deveria também interagir com esses elementos (selecioná-los, arrasta-los, renomeá-los, etc.). Foi necessário um estudo árduo da API (*Application Program Interface*) e da arquitetura utilizada pelas *Microsoft Foundation Classes*.

No estágio atual, a ferramenta só suporta o diagrama de classes UML. Em ALENCAR [49] já se tem a ferramenta com as diretrizes para transformar o modelo *i** em um diagrama de casos de uso UML. Como o padrão XMI suporta a representação de todos os diagramas da UML, seria possível então incorporar essas diretrizes para a geração do diagrama de casos de uso na ferramenta XGOOD. Além disso, quando a nova versão da linguagem OCL (2.0) for formalmente aprovada, teremos mecanismos para a representação do seu metamodelo através do padrão XMI. Com isso, as diretrizes de mapeamento D6 e D7 poderão ser incorporadas na ferramenta XGOOD.

Um dos objetivos desse trabalho como forma de validação do XGOOD é o desenvolvimento de um simulador para a especificação topológica e funcional de uma rede de comunicação que faz uso de meios ópticos. O diagrama de classes desse simulador será modelado segundo as regras de mapeamento e utilizando o XGOOD. Para o desenvolvimento do simulador de redes ópticas, primeiramente modelamos o sistema através da técnica *i** e posteriormente geramos o diagrama de classes com o uso da ferramenta XGOOD. Esse diagrama foi importando em uma ferramenta CASE comercial para realizarmos um refinamento do mesmo. O simulador também foi implementado utilizando-se a linguagem C++. Veremos como foi realizado o desenvolvimento do simulador no próximo capítulo.

7 Estudo de caso

O principal objetivo deste capítulo é mostrar o desenvolvimento de um simulador de redes ópticas através da utilização da ferramenta XGOOD para apoiar o processo de modelagem de requisitos. O desenvolvimento começará a partir do modelo *i** e será utilizada a ferramenta XGOOD para gerar o diagrama de classes UML. O diagrama será refinado para, posteriormente, ser gerado o código do sistema.

7.1 As redes óticas

As redes ópticas surgiram para solucionar o problema da crescente demanda por largura de banda cada vez maior nas comunicações de dados. As novas aplicações que estão surgindo no mercado, como videoconferência, computação distribuída, educação à distância, telemedicina, voz sobre **IP**, entre outras, têm exigido um volume de dados e banda passante cada vez maiores. A capacidade dos sistemas de transmissão ópticos superam em muito os dos sistemas tradicionais de transmissão por fio de cobre ou por cabo coaxial. A velocidade de transmissão das redes ópticas chegam a ordem de Terabits por segundo [50]. Além disso, os sistemas ópticos possuem uma taxa de erro por bit muito baixa em relação aos outros sistemas, chegando a um taxa de erro por bit da ordem de 10^{-9} [50].

As redes ópticas utilizam a fibra óptica como meio de transmissão. O princípio de funcionamento da fibra ótica é a transmissão de luz através de um meio constituído de vidro, o que proporciona o tráfego de mais informação através de grandes distâncias do que um cabo de cobre ou cabo coaxial. A perda de potência do sinal por quilômetro é muito menor do que os sistemas com cabos coaxiais, guias de onda ou transmissão pelo espaço livre. A fibra ótica possui baixas perdas, não é suscetível a interferência eletromagnética e possui uma largura de banda bastante elevada. Hoje em dia, o grau de pureza do vidro da fibra permite a transmissão de sinais digitais de luz através de distâncias maiores que 100km sem amplificação [51]. Isso significa uma quantidade menor de repetidores para cobertura total do enlace.

Os cabos de fibra óptica possuem no seu centro um núcleo de vidro, através do qual a luz propaga (Figura 50). O núcleo é coberto por uma casca de vidro com um índice de refração menor que o vidro do núcleo, confinando a luz no núcleo. A terceira é uma camada é um revestimento de plástico, para proteger a casca.

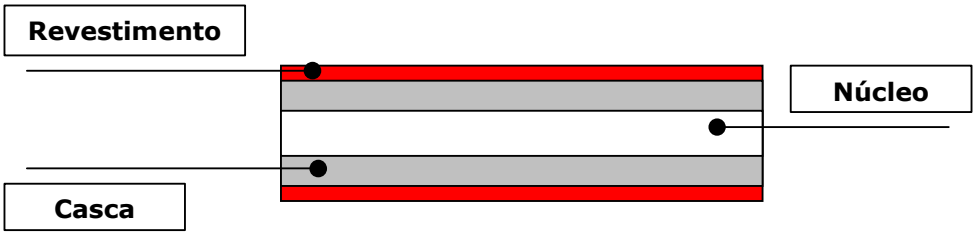

Figura 50 - Fibra óptica

Inicialmente, as companhias telefônicas utilizavam cada uma seu próprio sistema de multiplexação por divisão do tempo (TDM - *Time Division Multiplex*) em suas redes ópticas. Mas com a necessidade de padronização surgiram o **SONET** (*Synchronous Optical Network* – padrão norte-americano) e o **SDH** (*Syncronous Digital Hierarchy* – padrão europeu internacional). Esses dois padrões são bastante semelhantes. Com o aperfeiçoamento da tecnologia, surgiram as redes com multiplexação por divisão de comprimento de onda (WDM - *Wavelenght Division Multiplexing*) e, posteriormente, o DWDM (*Dense Wavelenght Division Multiplexing*). Essas duas últimas, ao contrário do **SONET** e do **SDH**, não utilizam uma taxa de transmissão ou uma estrutura de quadros específica. Isso que dizer que as redes WDM suportam vários tipos de padrões: **IP**, **ATM – (***Asynchronous Transfer Mode*- modo de transferência assíncrona), **SONET, SHD.**

Isso torna possível vários tipos de arquiteturas. A mais amplamente utilizada é a arquitetura de quatro camadas: **IP** sobre **ATM** sobre **SONET** sobre **DWDM**. Embora ainda vá existir por algum tempo, essa arquitetura será substituída por uma mais simples, de duas camadas: **IP** sobre **DWDM**. No futuro, o padrão **SONET/SDH** não será mais utilizado [51]. Veremos nas seções seguintes as redes DWDM.

7.1.1 Multiplexagem por Divisão de Comprimento de Onda Densa (DWDM)

O avanço da tecnologia, o aperfeiçoamento dos lasers e dos filtros ópticos e o surgimento de amplificadores óticos de banda larga tornou possível a transmissão de vários comprimentos de onda através de uma única fibra. Essa técnica de transmissão é denomidada Multiplexagem por Divisão de Comprimento de Onda (*Wavelength Division Multiplexing* - WDM). Através do método WDM os sinais que transportam a informação, em diferentes comprimentos de onda, são combinados em um multiplexador óptico e transportados através de uma única fibra, com o objetivo de aumentar a capacidade de transmissão e, conseqüentemente, usar a largura de banda da fibra óptica de uma maneira mais adequada. Os sistemas que utilizam esta tecnologia, em conjunto com amplificadores ópticos, podem aumentar significativamente a capacidade de transmissão de uma rota sem a necessidade de se aumentar o número de fibras.

O DWDM (*Dense Wavelenght Division Multiplexing*) nada mais é do que a tecnologia WDM com um número de comprimentos de onda transmitidos bem maior, pois o espaçamento entre eles é menor. A faixa de comprimentos de onda vária, tipicamente, de 1,530 nm a 1,565 nm. Futuramente, as fibras comportam atualmente mais de 300 canais simultaneamente. Hoje em dia são implementados, basicamente, dois tipos DWDM: unidirecional e bidirecional. Em um sistema unidirecional, todos os comprimentos de onda viajam em uma mesma direção na fibra, enquanto que um sistema bidirecional o sinal é divido em duas bandas separadas, cada uma viajando em uma direção diferente.

7.1.2 Redes WDM

Na Figura 51 temos um exemplo de um sistema WDM. Sinais ópticos com quatro comprimentos de ondas diferentes ($\lambda 1$, $\lambda 2$, $\lambda 3$, $\lambda 4$) são combinados através de um multiplexador, localizado a esquerda. As características chave de um bom multiplexador são baixa perda e um bom acoplamento para impedir que parte da luz seja refletida de volta para o transmissor. Os sinais viajam através da mesma fibra. No meio da fibra, o muitiplexador *add / drop* rotea um dos sinais ($\lambda 4$) para um terminal receptor, enquanto os outros sinais passam inalterados. Outro sinal diferente ($\lambda 4^*$), mas com o mesmo comprimento, de onda é inserido na fibra pelo terminal transmissor. O multiplexador add / drop tem a função de filtrar um ou mais comprimentos de onda de um sinal combinado e jogá-los (*drop*) em um terminal localizado ao longo do caminho da fibra. Existe também a possibilidade de adicionar (*add*) um ou mais sinais provenientes de um transmissor em canais vazios. No lado direito o sinal da fibra óptica passa através de um demultiplexador e os quatro sinais são roteados para receptores diferentes, um para cada comprimento de onda. Para o correto funcionamento do sistema, os multiplexadores devem separar de forma bastante seletiva os comprimentos de onda do sinal, impendindo que partes dos sinais de canais adjacentes interfiram uns com os outros. Os sistema WDM podem ainda incorporar outros elementos à medida que a rede se torna complexa, como por exemplo chaveadores ópticos e conversores de comprimento de onda.

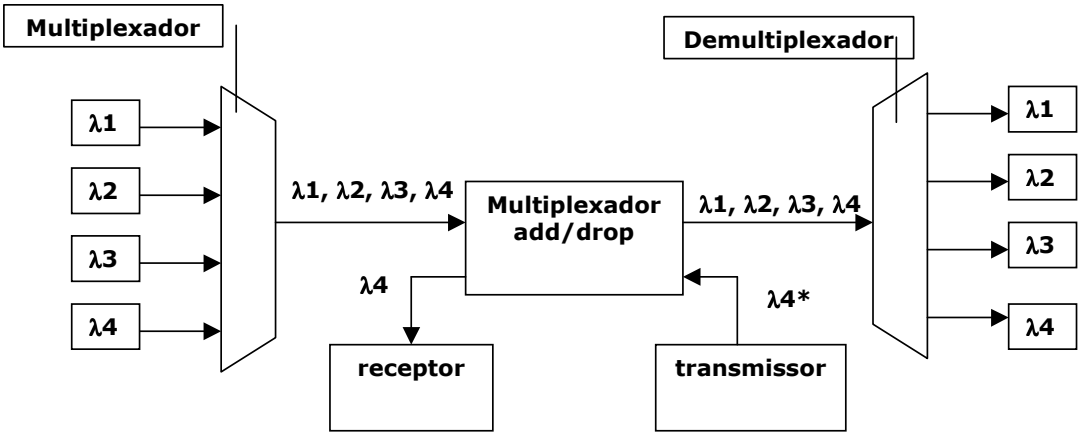

Figura 51 - Sistema WDM.

As redes DWDM são a resposta para a crescente necessidade de banda das aplicações. Elas podem aumentar a capacidade de transmissão dos enlaces existentes sem a necessidade de se modificar o que já foi implantado, utilizando de forma total a capacidade de transmissão das fibras. Uma das vantagens das redes WDM é sua capacidade de expansão e sua flexibilidade, permitindo o crescimento gradual da rede. Uma rede, por exemplo, pode ser planejada para operar com quatro canais e ir introduzindo canais quando necessário, bastando para isso adicionar novos equipamentos terminais. Dividindo e mantendo comprimentos de onda diferentes dedicados para clientes diferentes, por exemplo, os provedores de serviço podem alugar um comprimento de onda individual ao invés de colocar uma fibra inteira para um único cliente. Além disso, uma rede WDM é transparente quanto aos sinais transmitidos, pois por não haver envolvimento de processos elétricos, diferentes taxas de transmissão e sinais poderão ser multiplexados e transmitidos para o outro lado do sistema sem que seja necessária uma conversão óptico-elétrica. A mesma fibra pode transportar sinais SONET SDH e ATM de maneira transparente. Com isso a distâncias entre os elementos da rede pode ser aumentada em comparação com outras redes que utilizam o sistema TDM. Isso torna as redes DWDM ideais para redes MAN (*Mentropolitan Area Network*) e/ou WAN (*Wide Area Network*). Uma das desvantagens das redes DWDM é o alto custo dos equipamentos.

7.2 Especificação do Estudo de Caso

Os requisitos do sistema foram capturados com os alunos do Grupo de Fotônica, do Departamento de Eletrônica e Sistemas. Os requisitos descrevem a funcionalidade do simulador e quais as entradas que os usuários devem fornecer. O aplicativo tem por objetivo a simulação de uma rede de fibra óptica, com nós e fibras. Simulação tem se tornado uma ferramenta indispensável

nas pesquisas de redes ópticas ou não. Ela ajuda aos pesquisadores avaliar de forma rápida e barata a performance das redes. A simulação auxilia no estudo de novos protocolos, topologias e esquemas de chaveamento e roteamento em redes WDM. Uma rede ótica é constituída de nós e fibras, que interligam os nós (Figura 52). Entre os nós é possível ter uma ou mais fibras que os interliguem. Os nós são terminais da rede, onde os sinais transmitidos pela fibra são roteados para terminais ou outros nós da rede. Cada nó é constituído de um ou mais dos seguintes elementos, como pode ser visto na figura Figura 53: Tap, Amp, Demux, Mux, Switch.

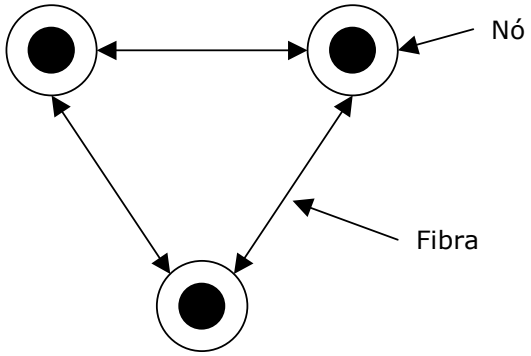

Figura 52 - Rede ótica.

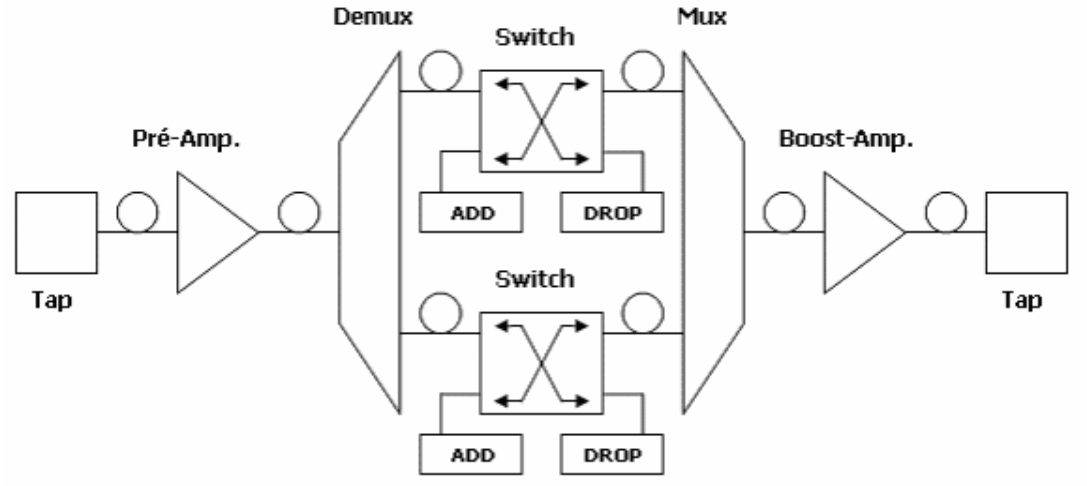

Figura 53 - Conteúdo de um nó em uma rede óptica.

O TAP é utilizado para conectar a fibra ao um PRÉ-AMPLIFIER, que amplifica o sinal da fibra. O sinal é inserido em um demultiplexador que separa os canais. Os canais são inseridos em SWITCHES que roteiam o sinal do canal para um terminal (DROP), enquanto outro sinal é inserido no canal vago (ADD). Os sinais são re-combinados no multiplexador e amplificados

novamente pelo BOOST-AMPLIFIER. Um outro TAP é utilizado para conectar o nó ao enlace e o sinal é encaminhado para o próximo nó.

Cada um dos elementos da rede ótica possui diversas características ou atributos:

- Fibra
    - Tipo de Fibra: podendo assumir os valores: *Partern* (Padrão, o tipo mais comum, utilizada com comprimentos de onda próximos de 1330nm), *DSF* (*Dispersion-Shifted Fiber* – fibras que possuem o ponto de dispersão zero movido para o comprimento de onda de valor 1550nm, mas possuem grande não linearidade quando utilizada com outros comprimentos de onda) ou *NZDF* (*Nonzero Dispersion Fiber* - possuem o ponto de dispersão zero movido para o comprimento de onda de valor 1550nm, mas podem operar em outras faixas de comprimentos de onda);
    - Comprimento (tamanho) da fibra, dado em quilômetros;
    - Número de comprimentos de onda e seus valores. Numa mesma fibra ótica é possível trafegar vários comprimentos de onda;
    - Atenuação da fibra em função do comprimento de onda, dado em DB por quilômetro;
- Nó
    - Potência do laser que incide sobre o nó, dado em DB;
- Tap
    - Perda de potência do elemento, dado em DB;
- Amp
    - Ganho do amplificador, dado em DB;
    - Potência de saturação do amplificador, dado em DB;
    - Figura de ruído do amplificador, dado em DB;
    - Largura de Banda (diferença entre a maior freqüência e a menor freqüência de operação) do amplificador, dado em nanometros.;
    - Comprimento de onda central, em nanometros;
- Demux
    - Perda de potência do elemento, dado em DB;
- Mux
    - Perda de potência do elemento, dado em DB;
- Switch
    - Tipo de Switch. Pode assumir os valores: SPANKE (arquitetura onde um *switch* NxN é construído combinando-se N *switches* do tipo 1xN com N *switches* do tipo

Nx1) ou MEMS (*Microelectromechanical System* – switches optomecâncicos que redirecionam a luz através de pequenas superfícies refletoras).

Todos esses parâmetros dos elementos serão fornecidos pelo usuário para a realização da simulação. Utilizaremos um modelo de simulação semelhante ao utilizado por FARBY-ASZTALOS [52]. A Figura 54 mostra o fluxograma do algoritmo de simulação. Nesse modelo as chamadas são geradas entre os nós de fonte e destino de forma aleatória. A duração de cada chamada também segue uma distribuição aleatória. Para cada chamada requisitada, uma rota é escolhida de acordo com um algoritmo pré-determinado, por exemplo, o algoritmo do menor caminho. Se não houver uma rota da fonte para o destino ou se não existirem mais comprimentos de onda livres ao longo da rota escolhida, a chamada é bloqueada.

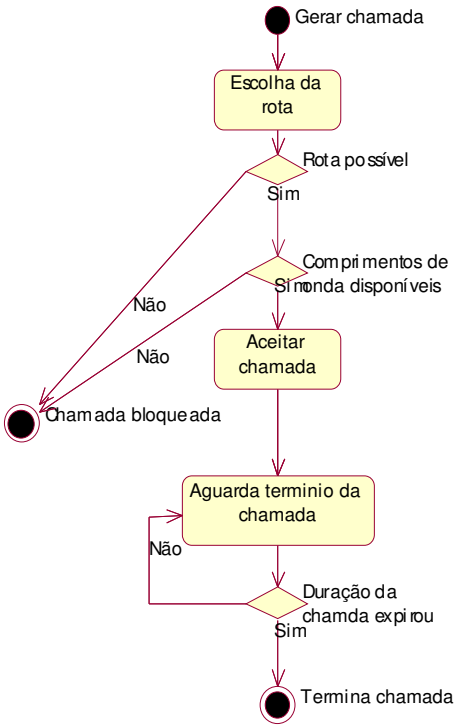

Figura 54 – Algoritmo de simulação.

Na seção subseqüente, veremos como será definida a arquitetura e a modelagem do sistema a ser desenvolvido.

7.3 Arquitetura do sistema.

De acordo com [53], um sistema de informação típico que apresente uma interface gráfica com o usuário e persistência de dados é geralmente projetado numa arquitetura clássica de três camadas (*three tie-architecture* - Figura 55):

- **apresentação –** interface gráfica , janelas, botões, etc;
- **lógica da aplicação –** tarefas e regras que governam o processo;
- **armazenamento –** mecanismo de persistência de dados;

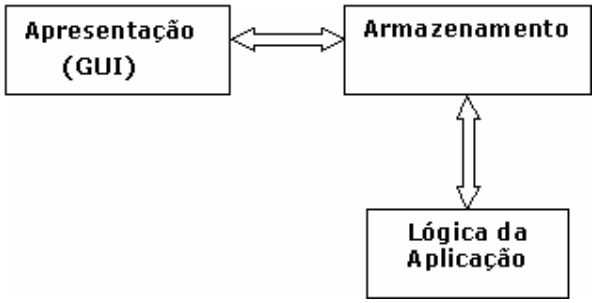

Figura 55 - Arquitetura em camadas do software de simulação.

Essa arquitetura de três camadas separa a lógica da aplicação em uma camada distinta. O software de simulação de redes óticas deve apresentar três camadas distintas. A camada de apresentação é relativamente independente da camada de aplicação. Utilizamos essa arquitetura clássica no desenvolvimento do simulador. Sendo assim, o sistema apresenta as seguintes camadas: **apresentação** (ou interface gráfico com o usuário - GUI), responsável pela interação do usuário com o software; **lógica da aplicação**, responsável pelo algoritmo de simulação e por efetuar os cálculos; **armazenamento**, responsável pela persistência dos dados.

Escolhemos essa arquitetura clássica em três camadas, pois pode-se adaptar mais facilmente o *software* a diversos algoritmos, bastando para isso mudar a implementação da camada de **lógica da aplicação.** Além disso, essa é a arquitetura utlizada mais comumente nos sistemas de *software* [53].

7.4 Modelagem dos Requisitos

Primeiramente iremos modelar os requisitos do sistema utilizando a técnica *i**. Através da arquitetura em camadas pode-se identificar os seguintes possíveis atores:

- *Usuario* – o que vai interagir com o software;

- *Visao* – irá acomodar a interface e irá interagir com o usuário;
- *Documento* – responsável por armazenar e posteriormente recuperar a topologia da rede ótica;
- *Logica* – responsável pelo algoritmo da rede e pelos cálculos;
- *Simulador* – o framework que contém o *Visao*, *Documento* e *Logica*;

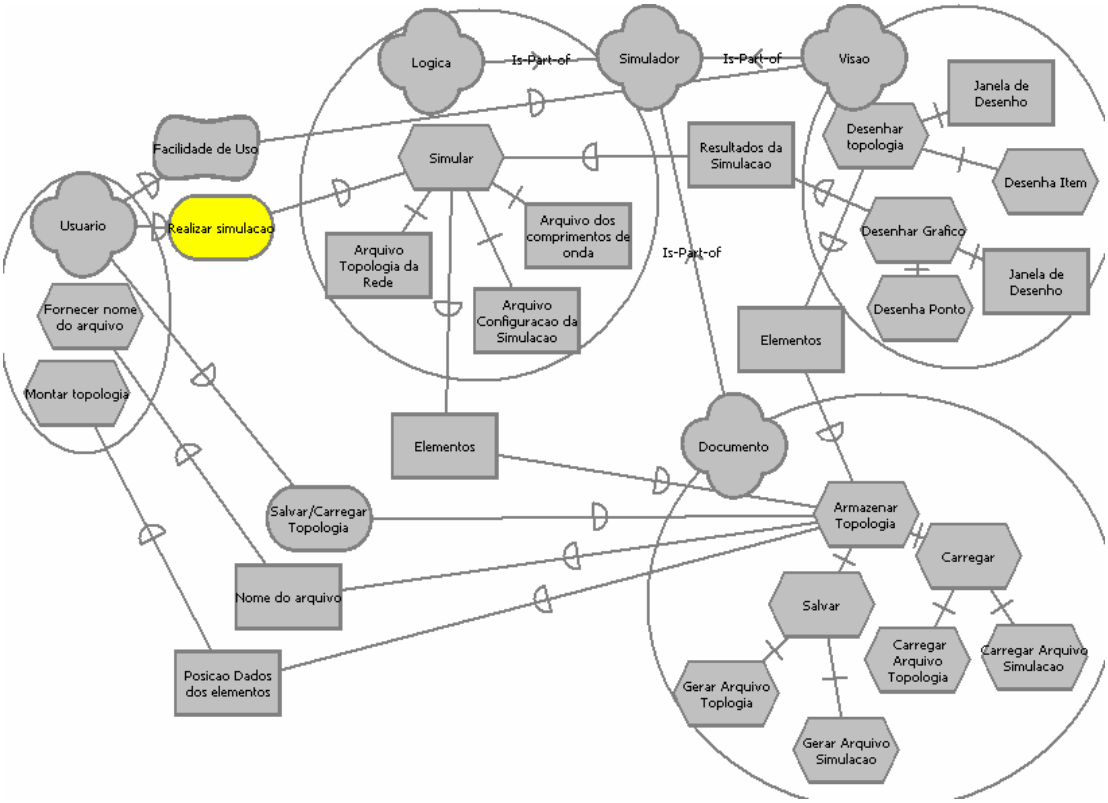

Figura 56 - Modelo *i do simulador de redes óticas.**

Na Figura 56 podemos visualizar o modelo SR do sistema a ser desenvolvido. Temos quatro atores (posições). O ator usuário representa aquele que vai interagir com o sistema. Este ator tipicamente não fará parte do sistema a ser desenvolvido. Está presente apenas para capturar as intenções e objetivos do usuário. Na captura inicial dos requisitos ficou definido que o sistema deveria, além de realizar a simulação, pode salvar e carregar a topologia da rede ótica para posterior referência. Isto está traduzido pelos objetivos *Realizar simulacao* e *Salvar/Carregar Topologia*.

O ator *Logica* é responsável por atender o objetivo *Realizar simulacao.* Para isto, ele deve realizar a tarefa *Simular*. Os parâmetros da simulação são armazenados em três arquivos diferentes: um arquivo para armazenar os comprimentos de onda, um para armazenar a topologia da rede e outro que armazenar todos os outros parâmetros de configuração e simulação. Estes arquivos são

representados, respectivamente, pelos recursos *Arquivo dos comprimentos de onda, Arquivo Topologia da Rede* e *Arquivo Configuracao da Simulacao.* Estão relacionados com a tarefa *Simular* através de ligações de decomposição.

Após realizada a simulação, o ator *Logica* fornece os resultados da simulação, representado pelo recurso *Resultados da Simulacao*, para o ator *Visao.* Este recurso é saída da tarefa *Simular* (do ator *Logica*) e entrada da tarefa *Desenhar Grafico* (do ator *Visao*). Esta tarefa é responsável por mostrar os resultados da simulação em forma gráfica, para uma melhor interpretação e visualização por parte do usuário dos resultados. Outra das atribuições deste ator é a representação gráfica da topologia da rede ótica, feita através da tarefa *Desenhar Topologia.* Esta tarefa necessita como entrada a topologia da rede (número de elementos, posição, ligações, etc.). Isto é fornecido pelo ator *Documento* (ator responsável por armazenar todos os dados da rede ótica) através do recurso *Elementos,* saída da tarefa *Armazenar Topologia* (do ator *Documento*).

Para atender o objetivo *Salvar/Carregar Topologia* temos o ator *Documento.* Isto é feito através da tarefa *Armazenar Topologia*. Para a realização desta tarefa, o ator *Documento* necessita que o *Uusário* informe o nome do arquivo e a forneça a posição e os dados dos elementos da rede ótica. Isto é feito, respectivamente, através das tarefas *Fornece nome do Arquivo* e *Montar Topologia.* A tarefa *Armazenar Topologia* esta decomposta em duas outras tarefas: *Carregar* (para recuperar uma topologia previamente salva em arquivo) e *Salvar* (para armazenar uma topologia em arquivo para posterior recuperação). Os atores *Documento, Visao* e *Logica* são partes do ator *Simulador*, que representa o sistema a ser desenvolvido.

Após a modelagem organizacional e a captura inicial dos requisitos ter sido finalizada, procedemos o mapeamento do modelo em um diagrama de classes UML. Primeiramente selecionamos as diretrizes de mapeamento adequadas, tal qual descrito na seção 6. (Figura 36). Em particular, devemos notar que o ator *Usuario* não fará parte do sistema a ser desenvolvido. Sua presença só e justificada na captura inicial dos requisitos do sistema. Devemos, portanto, selecionar a diretriz **None,** que significa que ele será ignorado, assim como todos os relacionamento ligados a ele. Porém, caso o simulador fosse desenvolvido com um sistema de autenticação, por exemplo, onde a sua utilização estaria ligado a um usuário, não poderíamos mais descartar o elemento *Usuario.* Teríamos a necessidade de representar as propriedades desse usuário (nome, senha, papel, etc.) através de objetos (instâncias de uma classe).

7.5 Mapeamento para Diagrama de Classes

O resultado do mapeamento pode ser visto na Figura 57. Esse diagrama foi gerado diretamente do modelo *i** do simulador utilizando a ferramenta XGOOD. Os elementos do modelo *i** foram transformados em elementos do diagrama de classes de acordo com as diretrizes de

mapeamento vistas no capítulo 3. As associações do diagrama de classes possuem papéis (*depender, dependum* e *dependee*), que explicitam quais os tipos de relacionamentos que os elementos possuíam no modelo *i**.

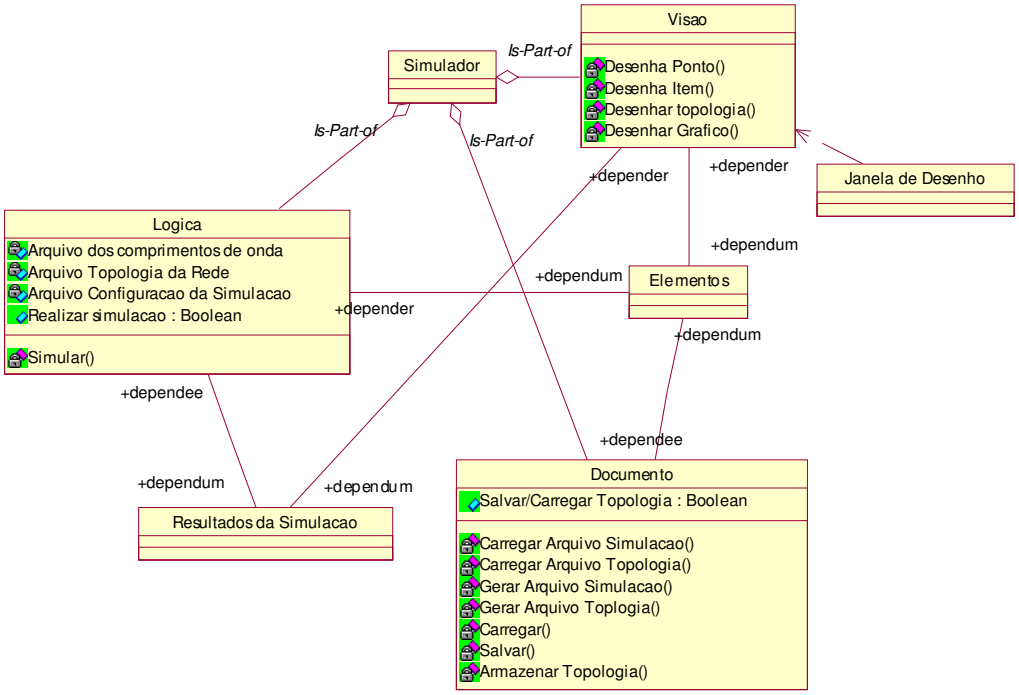

Figura 57 - Diagrama classes resultante do mapeamento realizado pela ferramenta.

Observamos na Figura 57 que os atores *Simulador, Logica, Visao* e *Documento* foram mapeados em classes no diagrama UML com os mesmos nomes, de acordo com a diretriz de mapeamento D1 (seção 2.4). Além disso, de acordo com a mesma diretriz, são criados relacionamentos de agregação entre as classes *Documento, Logica, Visao* e a classe *Simulador.* Ainda na mesma figura, vemos que os recursos *Resultados da Simulacao* e *Elementos* são mapeados em classes UML, como está descrito na diretriz D3.2. De acordo com a diretriz D3.1, os recursos *Arquivo dos comprimentos de onda, Arquivo Topologia da Rede* e *Arquivo Configuracao da Simulacao* são transformados em atributos com visibilidade privada na classe *Logica*. De acordo também com a diretriz D3.1, o recurso *Janela de Desenho* foi mapeado como uma classe UML, pois apresenta característica de um objeto.

O objetivo *Realizar Simulacao* foi mapeado como um atributo privado na classe *Logical*, de acordo com a diretriz D4.1. As tarefas *Desenhar Grafico, Desenhar Ponto, Desenhar Topologia, Desenhar Item, Armazenar Topologia, Carregar, Salva, Carregar Arquivo Simulacao, Carregar Arquivo Topologia, Gerar Arquivo Topologia, Gerar Arquivo Simulacao* e *Simular,* são mapeados em métodos com visibilidade privada em suas respectivas classes.

Devemos agora refinar o diagrama de classes gerado pela ferramenta XGOOD. Importamos o digrama da Figura 57 na ferramenta Rational Rose para posteriormente ser trabalhada. A linguagem escolhida para o desenvolvimento do simulador foi a mesma utilizada para o desenvolvimento da ferramenta XGOOD (C++, utilizando as MFC). Especializamos os tipos e classes padrões MFC *CWnd*, *CView*, *CWinApp*, *CDocument, CArray* e *CButton*, seguindo a arquitetura em camadas das aplicações *Windows* padrões. Estas classes já possuem vários atributos e funções específicas que auxiliam a produção de aplicativos para a plataforma *Windows*, não havendo razão para não aproveitá-las. Também foram definidas algumas cardinalidades dos relacionamentos entre os elementos, assim como foram adicionadas duas classes: *Node* e *Fiber*, que são especializações da classe *Elementos*.

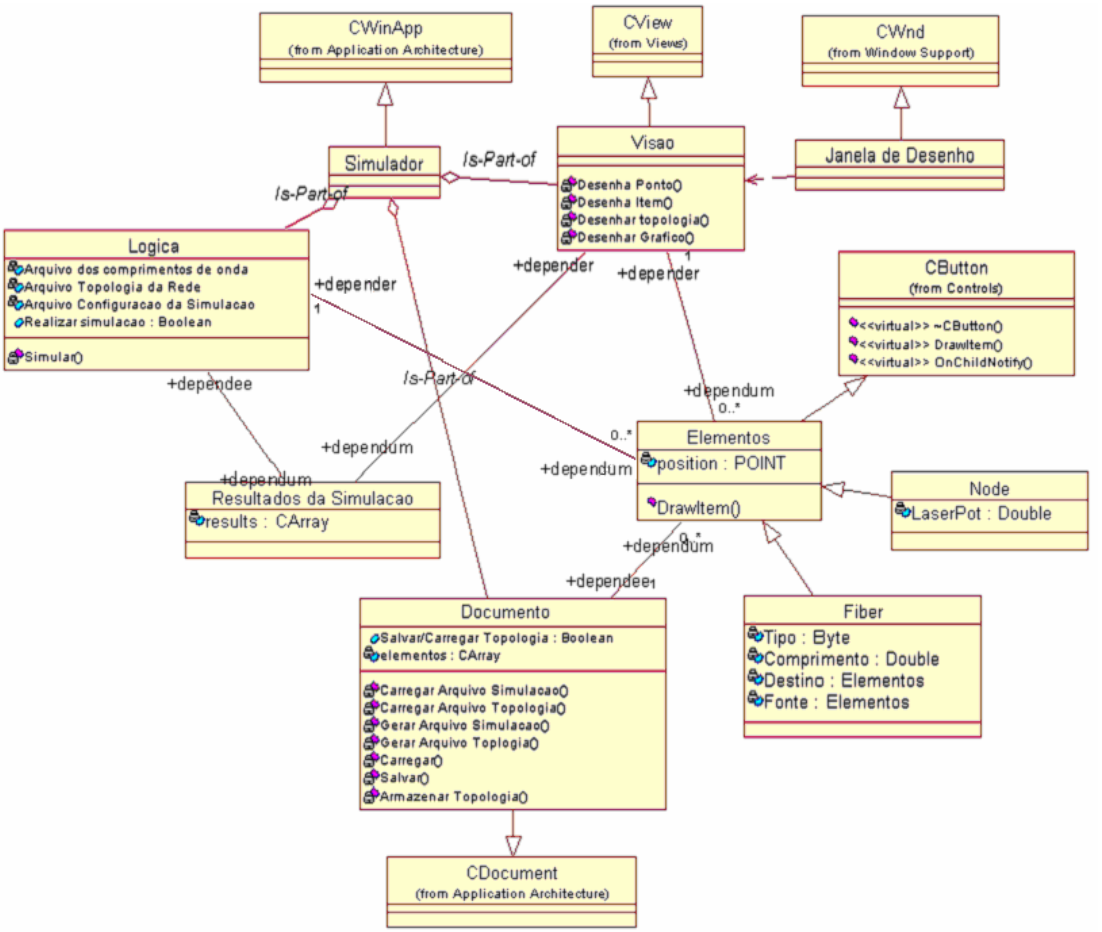

Figura 58 - Diagrama de classes do simulador refinado.

Com o diagrama de classes em mãos, parte-se para a codificação. O diagrama de classes nos fornece um esqueleto a partir do qual devemos implementar os métodos nele existentes. Utilizando a linguagem de desenvolvimento C++ implementamos todos os métodos presentes no diagrama de classes. A maior dificuldade esta na implementação do método *Logica::Simular()*.

Este método recebe como entrada um cojunto de objetos do tipo *Elementos*, fornecidos pelo objeto *Documento.* Nesse método devemos implementar o algoritmo de simulação, que é responsável por realizar os calculos e fornecer os resultados. Este método é o mais importante do sistema.

Devido à simplicidade do sistema, a modelagem através do diagrama de classes foi suficiente para desenvolvermos o sistema, por isso não foram utilizados os outros diagramas da UML.

Vimos que no nosso modelo de simulação é necessário um algoritimo para a escolha das rotas. Devido a sua simplicidade escolhemos o clássico algoritmo de Dijkstra [54], ou algoritmo de custo mínimo, para implementarmos na classe *Logica*. O algoritmo de custo mínimo calcula o menor caminho entre dois nós da rede ótica, tendo o comprimento das fibras óticas que interligam estes dois nós como peso. Este algoritmo pode servir como base para outros mais complexos, bastando para isso alterar o parâmetro utilizado como peso na decisão de escolha da rota. Por exemplo em [52], o autor utiliza além do algoritmo de menor caminho, uma estimativa da taxa erro por bit (TEB) na simulação. Como exemplo, o algoritmo de Dijkistra é suficiente para testar a ferramenta, mas enfatizamos que a arquitetura do sistema permite alterar o algoritmo de forma independente das outras classes e sem maiores problemas, bastanto para isso alterar a implementação do método *Logica::Simular()*.

7.6 Realizando as Simulações

A tela do *software do* simulador de redes óticas finalizado pode ser vista na Figura 59. Após o usuário montar a topologia da rede, o mesmo pode alterar os valores dos elementos (nós e fibras) clicando duas vezes sobre o elemento. Para salvar a topologia, o usuário deve clicar em *File->Save*.

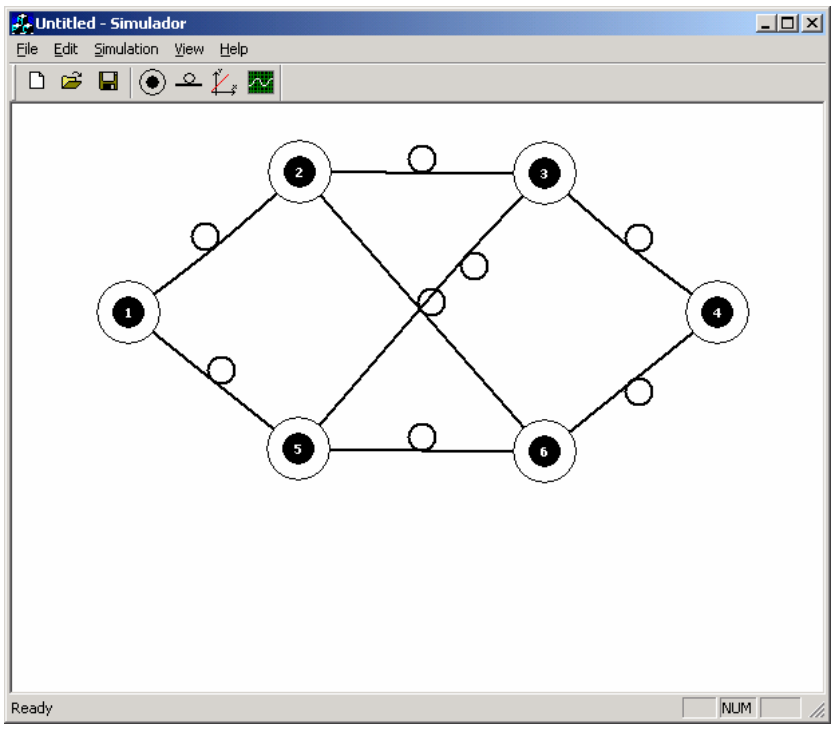

Figura 59 - Tela do simulador de redes óticas.

Para configurar os parâmetros da simulação, o usuário acessa o menu *Simulation->Config*. Na tela inicial, o usuário determina quais são os parâmetros que devem variar durante a simulação, como mostra a Figura 60. No decorrer na simulação, os parâmetros selecionados iram modificar o seu valor para cada iteração da simulação. Os valores iniciais, os valores finais e o número de pontos determinam como será esta variação.

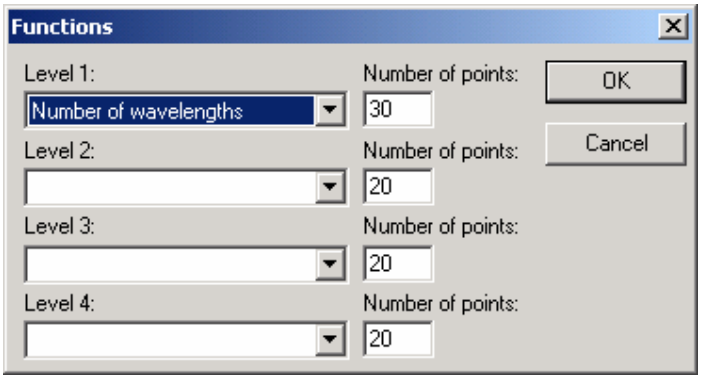

Figura 60 - Tela de configuração do simulador.

Na tela seguinte (Figura 61), o usuário deve informar os valores dos parâmetros da simulação. Para os parâmetros que irão variar (selecionados na tela anterior), o usuário deve informar, além do valor inicial, o valor final. As configurações são salvas no arquivo *config.txt*. Para executar a simulação, o usuário deve clicar em *Simularion->Plot Graphic.*

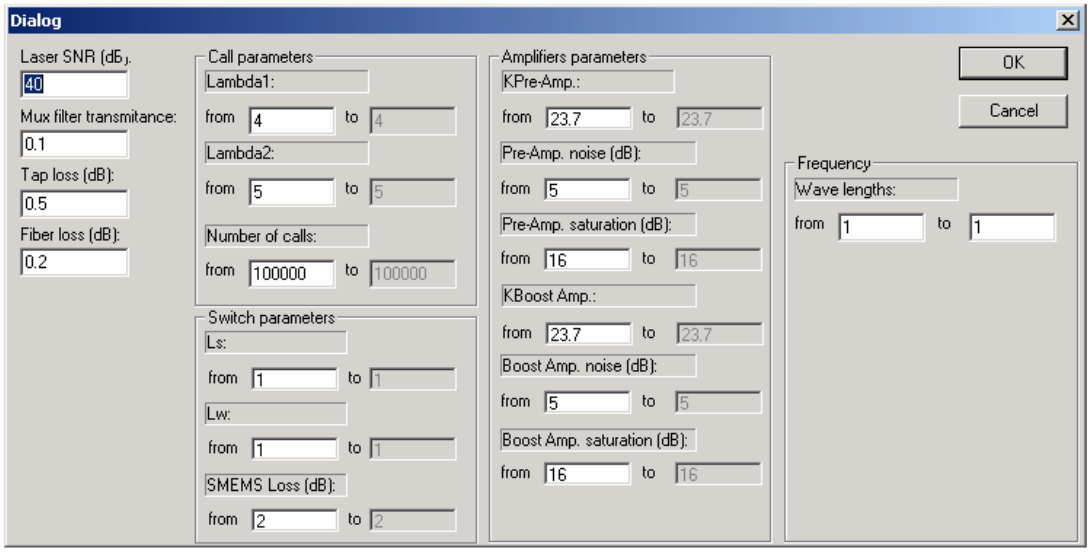

Figura 61 - Tela de configuração dos parâmetros da simulação.

Na execução da simulação, são geradas várias chamadas entre os elementos da rede. Quando uma chamada é realizada, o canal fica ocupado por um período de um até 9 chamadas. Se não houver mais canais disponíveis, o próximo caminho mais curto é utilizada pela chamada seguinte. Isto se repete até que não haja mais caminhos livres. Quando isto acontece, a chamada seguinte é bloqueada ou perdida. Ao fim da simulação, é mostrado um gráfico com o número de chamadas perdidas ao longo das iterações da simulação (Figura 62).

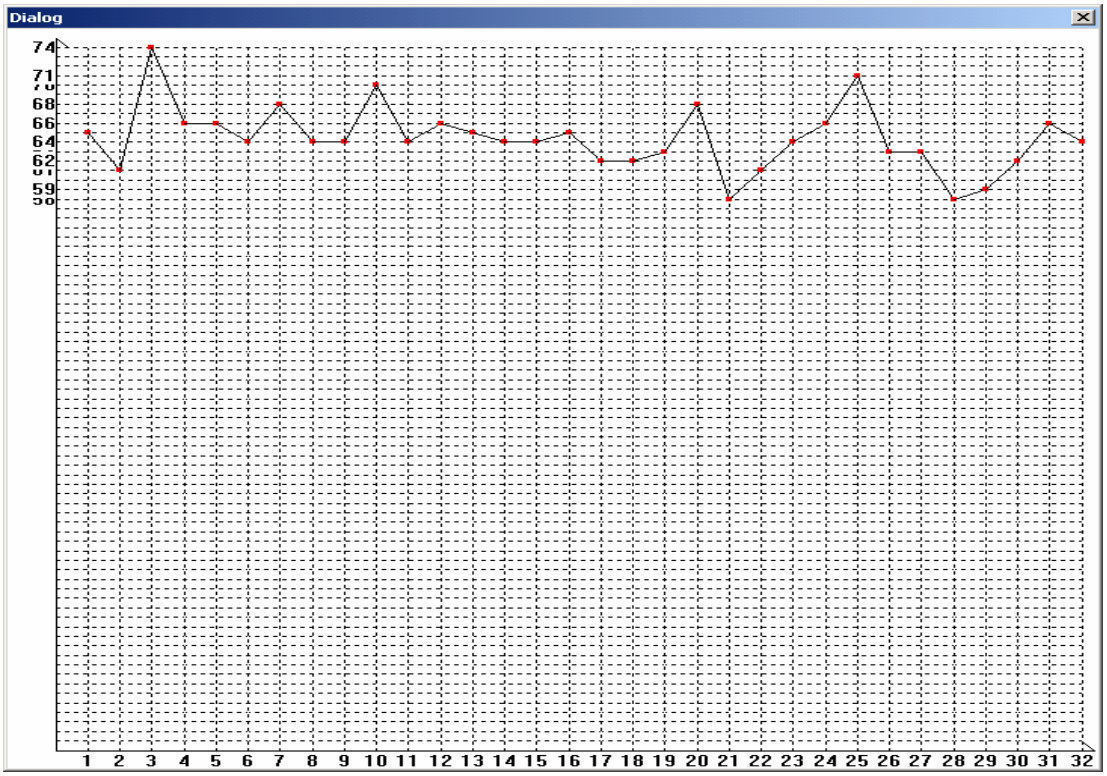

Figura 62 - Resultado simulação com um canal por fibra, 125 chamadas por iteração.

O resultado da simulação pode ser visto na Figura 62. Neste exemplo, foram utilizadas trinta iterações, com 125 chamadas cada uma. Foi utilizada a topologia da Figura 59. Cada fibra possui o mesmo comprimento e apenas um canal disponível. Se agora aumentarmos o número de canais de cada fibra para (vide Figura 63), como esperado, com mais canais por fibra o número de chamadas perdidas foi menor. Obviamente, se aumentarmos o tráfico da rede para 250 chamadas por iteração, no número de chamadas perdidas será maior (Figura 64).

Para determinar qual o melhor número de canais por fibra este tipo de tráfico (125 chamadas, com duração de um a nove chamadas), realizamos uma nova simulação, desta vez variando o número de canais por fibra de um até nove (Figura 65). A primeira interação possui apenas um canal e a iteração 32 possui 16 canais. Vemos no gráfico que a partir da iteração 25, o número de chamadas bloqueadas se reduz a, no máximo, uma. Isto corresponde a 13 canais por fibra.

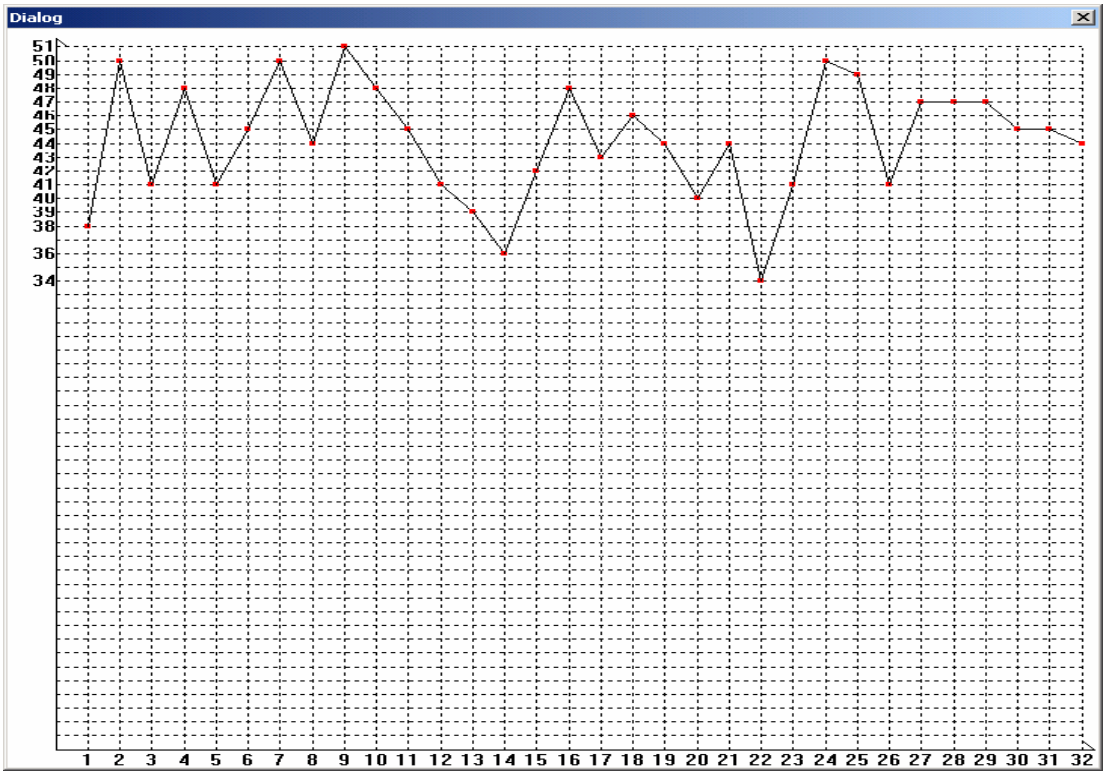

Figura 63 - Resultado da simulação com dois canais por fibra, 125 chamadas por iteração.

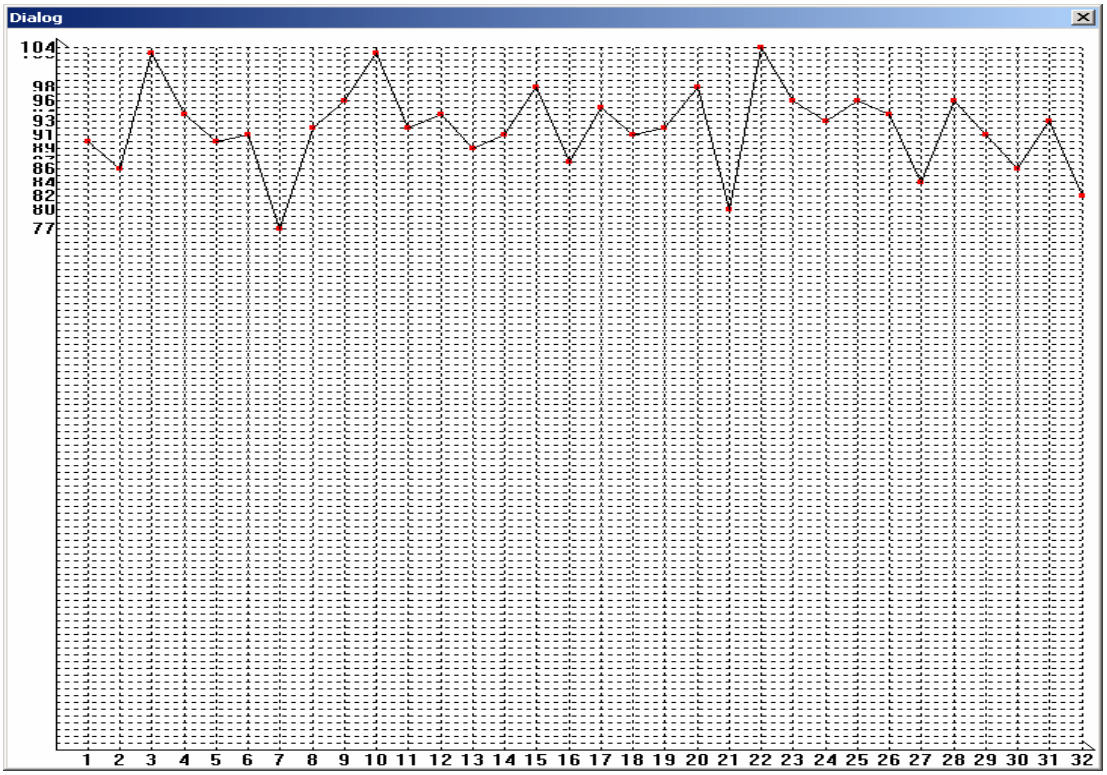

Figura 64 - Resultado da simulação com dois canais por fibra, 250 chamadas por iteração.

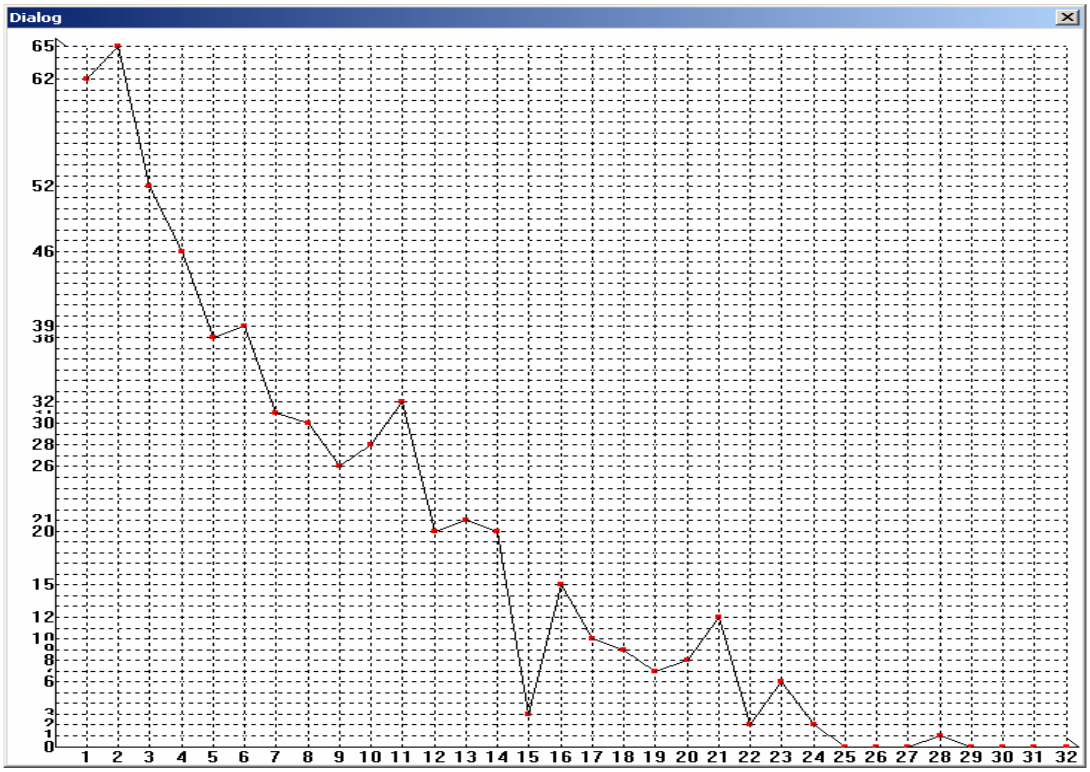

Figura 65 - Resultado da simulação com 1 até 16 canais por fibra, 125 chamadas por iteração.

Esse algoritmo é apenas para testes da arquitetura do *software*, numa simulação mais detalhada seria necessário levar em conta os outros parâmetros da rede ótica (perda da linha, potência dos nós, potência de saturação dos amplificadores, comprimentos de ondas, etc.). O algoritmo do menor caminho só considera o comprimento da fibra e o número de canais. Mas, como dito antes, a implementação da camada *Logica* é independente do resto do *software* sendo, portanto, facilmente alterável.

7.7 Conclusões

Os simuladores de redes auxiliam no estudo e avaliação, de forma barata e rápida, das redes ópticas. Eles permitem que se estude a eficácia e viabilidade de uma determinada topologia antes que a mesma seja implementada. Vimos nesse capítulo como a utilização da ferramenta XGOOD e as diretrizes de mapeamento podem auxiliar no desenvolvimento de sistemas e aplicativos de *software.* Partindo do conceito de redes ópticas, estabelecemos a arquitetura em três camadas e construímos um modelo *i** do sistema a ser desenvolvido. Em seguida, a ferramenta XGOOD foi utilizada para gerar o diagrama de classes do simulador. Apesar desse diagrama estar mais próximo do sistema real, ele ainda precisa ser refinado. No refinamento, foram adicionados novos atributos, cardinalidades e especializações de classes antes do sistema ser desenvolvido. Paritmos então para a implementação em si, com atenção especial para a implementação do método *Simular()* da classe *Logica*.

Devido a sua simplicidade escolhemos o algoritmo de Dijkistra, ou algoritmo de custo mínimo, para implementarmos na classe *Logica*. Os resultados da simulação mostrados neste capítulo servem para calcular, de forma grosseira, a probabilidade de bloqueio de uma chamada de um determinado sistema. Analisando a Figura 63, temos que o número de chamadas bloqueadas variou entre 36 e 51, com 125 chamadas para cada interação. A probabilidade de bloqueio estimada varia entre 36/125 (0,288) e 51/125 (0,408). Esses resultados, por exemplo, podem auxiliar ao projetista do enlace na determinação da necessidade ou não de mais canais. Esse algoritmo é apenas para testes da arquitetura do *software*, numa simulação mais detalhada e próxima do real seria necessário levar em conta os outros parâmetros da rede ótica. Mas, como dito antes, a implementação da camada *Logica* é independente do resto do *software* sendo, portanto, facilmente alterável, vide, por exemplo, os trabalhos de [52, 55, 56] que mostram como altera o algoritmo de menor caminho pode ser alterado para fornecer resultados mais práticos.

8 Conclusões e Trabalhos Futuros

A captura de requisitos é um das fases críticas da Engenharia de Software. É necessária para garantir que o sistema a ser desenvolvido possa atender, realmente, às necessidades dos clientes. Os requisitos do sistema definem que serviços o sistema deve prover e quais são as limitações do mesmo. O modelo dos requisitos iniciais deve ser independente de como o sistema irá ser implementado. Sugerimos, nesse trabalho, que a captura dos requisitos seja feita em diferentes níveis de abstração (indo da fase de captura de requisitos iniciais – relacionada com os requisitos organizacionais - até a fase final de captura – relacionada com os requisitos funcionais).

Para auxiliar na captura e documentação dos requisitos temos as técnicas de modelagem. O objetivo da modelagem é obter um documento formal contendo uma definição oficial do que seja necessário aos desenvolvedores do sistema, o **documento de requisitos**. Diversas são as técnicas para a modelagem de requisitos existentes, cada uma se adaptando melhor a um tipo de requisito (funcional, não funcional, organizacional) e/ou ao estágio de desenvolvimento do sistema (*early requirements / late requirements*).

Para a captura dos requisitos iniciais temos a técnica de modelagem *i** [5, 6], que foca no relacionamento entre os atores e suas intenções. Com ela é possível realizar a captura dos requisitos organizacionais que irão influenciar na determinação dos requisitos funcionais do sistema. Após a captura inicial, devemos refinar os requisitos para que os mesmos reflitam as funcionalidades do sistema que será implementado. Para isso temos a técnica de modelagem UML [7, 8, 9]. Argumentamos que a UML sozinha não é adequada para lidar com os diferentes tipos de análises e razões que são necessárias durante a fase de captura de requisitos. Logo, faz-se necessário o uso de técnicas complementares: a técnica *i**, para a captura dos requisitos iniciais e a modelagem do negócio; e a UML, para a captura dos requisitos finais, com o auxílio da OCL para a modelagem das limitações (pré(pós)-condições, alternativas, etc.).

Todavia, o uso de duas técnicas diferentes, sugere a necessidade de uma ferramenta para promover a integração dessas (através das diretrizes propostas), suportando o rastreamento e as mudanças entre os modelos. O XGOOD (*eXtended Goal Into Object Oriented Development*) trata-se de uma ferramenta para auxiliar no mapeamento do modelo *i** diretamente para o diagrama de classes em UML A ferramenta foi desenvolvida utlizando a linguagem C++ e integradada a ferramenta OME (ferramenta utilizada para realizar a modelagem *i**). Através da utilização da ferramenta XGOOD, o usuário é capaz de abrir e visualizar os modelos *i** gerados pela ferramenta OME. O usuário também pode selecionar de forma individual como e quais os elementos serão mapeados. O digramas de classes gerado é salvo no formato XMI, um formato padrão capaz de ser importado por uma variedade de ferramentas CASE disponíveis atualmente.

Um dos objetivos deste trabalho foi o desenvolvimento de um simulador para a especificação topológica e funcional de uma rede de comunicação que faz uso de meios ópticos para servir de validação para a ferramenta XGOOD. Assim, o desenvolvimento foi apoiado pela ferramenta XGOOD, utilizada para gerar o diagrama de classes UML a partir do modelo *i**. O simulador utiliza o algoritmo de Dijkstra [54], ou algoritmo de custo mínimo, para a escolha das rotas. Este algoritmo pode servir como base para outros mais complexos, e a arquitetura em camadas do simulador permite a alteração do algoritmo sem maiores problemas. O simulador realiza uma série de chamadas entre os nós de uma rede óptica (a topologia da rede é definida pelo usuário). Cada chamada possui uma duração aleatória. É realizado um determinado número de chamadas (definido pelo usuário) por iteração. Ao fim de cada iteração é computado o número de chamadas bloqueadas. Este resultado fornece uma estimativa da probabilidade de bloqueio de uma de uma rede WDM.

Através da utilização da ferramenta XGOOD foi possível passar do modelo *i** diretamente para o diagrama de classes UML. Isso agilizou o desenvolvimento da ferramenta, pois forneceu um diagrama de classes que mantém a consistência com o modelo organizacional de forma automática, diminuindo o esforço e o risco de erros.

8.1 Trabalhos Relacionados.

A ferramenta e a técnica para realizar a integração dos requisitos iniciais e dos requisitos mais próximos da implementação foram propostas em [3, 4, 10]. Posteriormente, as diretrizes foram aprimoradas e refinadas em [11]. Neste trabalho, buscamos dar suporte as novas diretrizes e suprir algumas deficiências da ferramenta anterior, de forma que:

- Permita a seleção dos elementos mapeados. Como nem todos os conceitos capturados na fase de requisitos iniciais irão corresponder aos modelos de sistema de software, a nova ferramenta permite que o usuário selecione quais elementos ele deseja que participe do mapeamento.
- Suporte a XMI. O XMI (XML Metadata Interchange) é um padrão proposto pela OMG que tem se tornado um padrão de fato para troca de dados entre ferramentas CASE. O objetivo principal do XMI é criar um mecanismo fácil de troca de metadados entre ferramentas de modelagem (baseadas no padrão UML). A adoção de um padrão já aceito e reconhecido pela OMG (o XMI), traz maior flexibilidade à ferramenta. O suporte a novas ferramentas de modelagem (como o Poseidon) poderá ser realizado sem maiores problemas, desde de que essas ferramentas

suportem a importação do formato XMI. Adoção do padrão XMI diminui o esforço de integração entre as ferramentas CASE.

- Tenha independência de ferramentas CASE. A nova ferramenta pode ser utilizada independente da ferramenta utilizada para modelar o diagrama de classes;
- Suporte a mais elementos do modelo *i**, como posições, papéis, agentes, relacionamentos *plays, is-part-of,* etc.;
- Seja integrada com a ferramenta de modelagem *i** OME.

A nova ferramenta, sem dúvida, representa uma evolução em relação a anterior.

8.2 Trabalhos futuros.

A ferramenta apresenta algumas restrições: suporta somente o diagrama de classes, não suportar as restrições em OCL, funciona somente no sistema operacional *Microsoft Windows*. Sendo assim, o próximo passo é tentar aprimorar as regras de mapeamento existentes e incorporar novas diretrizes para geração de outros tipos de diagrama da UML, como por exemplo o diagrama de diagrama de casos de uso [49]. Estas novas regras de mapeamento serão suportadas por uma nova versão da ferramenta, assim através de uma única ferramenta podemos gerar tanto o diagrama de classes quanto o diagrama de casos de uso a partir do modelo *i**.

Além disso, quando a nova versão da linguagem OCL (2.0) for formalmente aprovada, teremos mecanismos para a representação do seu metamodelo através do padrão XMI. Com isso, as diretrizes de mapeamento D6 e D7 poderão ser incorporadas na ferramenta XGOOD.

Como evolução natural da ferramenta, propomos uma versão multi plataforma, independente do sistema operacional, desenvolvida em Java, que será capaz de rodar em vários sistemas operacionais. Outra vantagem da utilização da linguagem Java e sua integração com a ferramenta OME se dará de forma mais natural e amigável, pois a própria ferramenta OME e seus plugins são desenvolvidos nesta linguagem.

REFERÊNCIAS

[1] ROCHA, Ana Regina Cavalcanti da, MALDONADO, José Carlos, WEBER, Kival Chaves. *Qualidade de Software – Teoria e Prática.* São Paulo: Prentice Hall, 2001.

[2] SOMMERVILEE, Ian, SAWYER, Pete. *Requirements Engineering – A good practice guide.* New York: John Wiley & Sons, 1997.

[3] ALENCAR, Fernanda Maria Ribeiro de. Mapeando a Modelgagem Organizacional em Especificação Precisa. 1999. Tese (Ciência da Computação) - Universidade Federal de Pernambuco, (Orientador) Jaelson Freire Brelaz de Castro.

[4] CASTRO Jaelson, ALENCAR Fernanda, CYSNEIROS Gilberto. *Closing the Gap between Organizational Requirements and Object Oriented Modeling*. Journal of the Brazilian Computer Society, v.7, n.1, 2000.

[5] YU, Eric. *Modelling Strategic Relationships for Process Reengineering*. Toronto, 1995. Phd Thesis - Department of Computer Science, University of Toronto.

[6] Yu, Eric. *Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering*. In: IEEE INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING – RE97. *Proceedings* (3. : Jan. 1997: Washington D.C). Washington D.C, 1997. pp.226-235.

[7] RUMBAUGH, James, JACOBSON, Ivar, BOOCH, Grady. *The Unified Modeling Language Reference Manual.* Reading: Addison-Wesley, 1999.

[8] BOGGS, Wendy, BOGGS, Michael. *Mastering UML with Rartional Rose 2002.* Alameda: SYBEX, 2002.

[9] *OMG Unified Modeling Language Specification*. v. 1.5. Mar. 2003. . Disponível em: http://www.omg.org/technology/documents/formal/uml.htm. Acessando em: 02 nov. 2004.

[10] CYSNEIROS, Gilberto. *Ferramenta para o Suporte do Mapeamento da Modelagem Organizacional em i* para UML*. Recife, 2001. Tese de mestrado – Centro de Informática, Universidade Federal de Pernambuco.

[11] ALENCAR, Fernanda M. R., PEDROZA, Flávio, CASTRO, Jaelson F. B. Et al. New Mechanisms *for the Integration of Organizational Requirements and Object Oriented Modeling*. WORKSHOP EM ENGENHARIA DE REQUISITOS. *Anais* (4.: Novembro, 2003: Piracicaba). Piracicaba, 2003. pp 109-123.

[12] Object Management Group (OMG). Disponível em: http://www.omg.org/. Acessando em: 18 out. 2004.

[13] GROSSE, Timothy J., DONEY, Garay C., BRODSKY, Stephen A. *Mastering XMI Java Programming with XMI, XML, and UML*. New York: John Wiley & Sons, 2002.

[14] *OMG XML Metadata Interchange (XMI) Specification* v. 1.2. Jan. 2002

[15] SOMMERVILEE, Ian. *Engenharia de Software*. 6 ed. São Paulo: Person Education do Brasil, 2003.

[16] IEEE Std. 830. *IEEE Guide to Software Requirements Specification*. The Institute of Electrical and Electronics Engineers. New York , 1984.

[17] SALVADOR, L. T. Camargo. *Conversão de Definição de Requisitos de Software de SADT para LOTOS.* Porto Alegre, 1996. Tese de mestrado - Curso de Mestrado em Informática, Pontifícia Universidade Católica do Rio Grande do Sul.

[18] Introduction: Overview of Human Engineering Analysis Techniques. Disponível em: http://www.manningaffordability.com/s&tweb/PUBS/Man_Mach/part1.html. Acessado em: 02 nov. 2004.

[19] ROSS, Douglas: *Structured Analysis (AS): A Language for Communicating Ideas*. IEEE Transactions on Software Engineering. 3,1. pp. 16-34. Jan. 1977.

[20] MULLERY, G.. *CORE : A Method for Controlled Requirements Expression.* Fourth International Conference on Software Engineering. IEEE Computer Society Press, 1979, pp. 126-135.

[21]FINKESTEIN, Anthony, KRAMER, Jeff, NUSEIBEH, Bashar. et al. *Viewpoints: A Framework for Integrating Multiple Perspectives in Systems Development*. International Journal of Software Engineering and Knowledge Engineering, vol 2, pp. 31-58. 1992.

[22] KOTONYA, Gerald, SOMMERVILLE Ian. *Requirements Engineering with Viewpoints*. Software Engineering, 1(11), pp. 5-18. 1996.

[23] RUMBAUGH, James, BLAHA Michael, PREMERLANI, William et al. *Object-Oriented Modeling and Design, Englewood Cliffs*, NJ: Prentice Hall, 1991.

[24] BOOCH, Grady. *Object-Oriented Analysis and Design with Applications*, 2nd edition. Redwood City, CA: The Benjamin/Cummings Publishing Company, 1993.

[25] JACOBSON, Ivar, CHRISTERSON, Magnus, JONSSON, Patrik et al. *Object-Oriented Software Engineering—A Use Case-Driven Approach*, Addison-Wesley, 1992.

[26] LAMSWEERDE, Axel van. *Requirements engineering in the year 00: a research perspective*, International Conference on Software Engineering. *Proceedings* (22.: Junho, 2000: Limerick). Limerick, 2000.

[27] LAMSWEERDE, Axel van. *Handling Obstacles in Goal-Oriented Requirements Engineering.*. Journal of Software Engineering, v. 26, n. 10, 2000

[28] KIRIKOVA, Marite., BUBENKO, Janis A., *Software Requirements Acquisition through Enterprise Modelling.* International Conference on Software Engineering and Knowledge Engineering. *Proceedings* (6.: Junho, 1994, Jurmala). Jurmala, 1994. pp 20-27.

[29] OME Organization Modeling Environment. Disponível em: http://www.cs.toronto.edu/km/ome/. Acessado em: 18 out. 2004.

[30] Rational Rose. Disponível em: http://www.ibm.com/rational/. Acessado em: 18 out. 2004.

[31] MagicDraw UML. Disponível em: http://www.magicdraw.com/. Acessado em: 18 out. 2004.

[32] Telelogic Tau. Disponível em: http://www.telelogic.com/. Acessado em: 18 out. 2004.

[33] ArgoUML: The Cognitive CASE Tool. Disponível em: http://argouml.tigris.org. Acessado em: 18 out. 2004.

[34] OME3 Architecture. Disponível em: http://www.cs.toronto.edu/km/ome/docs/architecture/architecture.html. Acesso em: 18 out. 2004.

[35] MYLOPOULOS, John, BORGIDA, Alex, JARKE, Marthias et al. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems.* October, 1990. ACM Trans. Information Systems, vol. 8, Out. 1990, pp. 325-362.

[36] Unisys. Unisys XMI Export/Import for Rational Rose. Disponível em: http://www-106.ibm.com/developerworks/rational/library/content/03July/2500/2834/Rose/rational_rose.html. Acessado em: 02 nov. 2004.

[37] Extensible Markup Language (XML) 1.0. Disponível em: http://www.w3.org/TR/2004/REC-xml-20040204/. Acessado em: 18 out. 2004.

[38] HAROLD, Elliotte R. *XML Bible.* 2 ed. New York: Hungry Minds, 2001.

[39] RAY, Erik T. *Learning XML.* 1 ed. O'Reilly, 2001

[40] *Meta Object Facility (MOF) Specification*. v. 1.4. Abr. 2002. Disponível em: http://www.omg.org/technology/documents/formal/mof.htm. Acessado em: 18. out. 2004.

[41] Extensible Stylesheet Language (XSL) Version 1.0. Disponível em: http://www.w3.org/TR/xsl. Acessado em: 18 out. 2004.

[42] Objects by Design.Transforming XMI to HTML. Disponível em: http://objectsbydesign.com/projects/xmi_to_html.html. Acessado em 18 out. 2004.

[43] JIANG, Juan-Juan, SYSTÄ, Tarja. *UML model exchange using XMI*. 2002. Dinponível em: http://www.cs.tut.fi/~xmlohj/linkit/XMIReport.pdf. Acessado em: 18 out. 2004.

[44] Brodsky, S., *XMI Opens Application Interchange*. Disponível em:http://www4. ibm.com/software/ad/standards/xmiwhite0399.pdf. Acessado em 18 out. 2004.

[45] Working XML: UML, XMI, and code generation, Part1. Disponível em: http://www-128.ibm.com/developerworks/xml/library/x-wxxm23/. Acessado em 18 out. 2004.

[46] Object Constraint Language Specification. In: OMG Unified Modeling Language Specification. v. 1.5. Mar. 2003. p. 6-1 – 6-48.

[47] UML 2.0 OCL Specification. Disponível em: http://www.omg.org/docs/ptc/03-10-14.pdf. Acessado em 20 jan. 2004.

[48] WILLIAMS, Al. *MFC Black Book.* Scottsdale: Coriolis Group, 1997.

[49] ALENCAR, Fernanda M. R, PEDROZA, Flávio P., CASTRO, Jaelson F. B. et al. *Ferramentas para Suporte do Mapeamento da Modelagem i* para a UML: eXtended GOOD – XGOOD e GOOSE*. Workshop on Requirements Engineering (4.: Dezembro, 2004: Buenos Aires). Buenros Aires, 2004.

[50] TANEBAUS, Andew S. *Computer Networks*. 3 ed. New Jersey: Prentice Hall, 1996.

[51] Dense Wavelength Division Multiplexing (DWDM)**.** Disponível em: http://www.iec.org/online/tutorials/dwdm/. Acessado em: 20/01/2005.

[52] FARBY-ASZTALOS, Tibor, BHIDE, Nilesh, SIVALINGAM, Krishna M. *AdaptiveWeight Functions for Shortest Path Routing Algorithms for Multi-Wavelength Optical WDM Networks* in Proc. of IEEE Intl. Conference on Communications, (New Orleans, LA), June 2000.

[53] LARMAN, Craig. *Utilizando UML e Padrões*. Porto Alegre: Bookman, 2000.

[54] Dijkstra's Algorithm. Disponível em http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/dijkstra.html. Acessado em 20 jan. 2005.

[55] RAMAMURTHY, Byrav, DATTA, Debasish, FENG, Helena et al. *SIMON: A Simulator for Optical Networks*. SPIE All Optical Networking 1999: Architecture, Control and Management Issues. *Proceedings* (Setembro, 1999). v. 3843, pp 130-135.

[56] BERMOND, Jean-Claude, GARGANO, Luisa, PERENNES, Stephan et al. *Efficient collective communication in optical networks*. Journal of Theoretical Computer Science, v. 233, n. 1-2, pp. 165-189, 2000.

[57] Java Technology. Disponível em http://java.sun.com. Acessado em 20 jan. 2005.