UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Um Metamodelo da Linguagem de Modelagem, Real Time UML, de Suporte à Criação de Dicionário de Dados para Ferramentas de Modelagem de Sistema Tempo Real, Visando a Verificação de Consistência dos Modelos.

por

ISABEL FERNANDES DE SOUZA

Dissertação submetida à avaliação,
como requisito parcial, para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Carlos Eduardo Pereira
Orientador
Prof. Dr. Roberto Tom Price
Co-Orientador

Porto Alegre, Janeiro de 2000.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Souza, Isabel Fernandes

Metamodelo para a Linguagem de Modelagem UML-RT / por Isabel Fernandes de Souza. – Porto Alegre: PPGC da UFRGS, 2000.
162f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2000. Orientador: Pereira, Carlos Eduardo; Co-Orientador: Price, RobertoTom.

1. Verificação de Consistência. 2. Metamodelagem. 3. Metodologias para Modelagem de Sistemas de Tempo Real. 4. Ferramentas para Modelagem e Simulação de Sistemas de Tempo Real. I. Pereira, Carlos Eduardo. II. Price, RobertoTom. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Prof[a]. Wrana Panizzi

Pró-Reitor de Pós-Graduação: Prof. Franz Rainer Semmelmann

Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenadora do PPGCC: Prof[a]. Carla Maria Dal Sasso Freitas

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Às duas pessoas mais importantes de minha vida: ***Tito Fernandes de Souza,*** *meu pai, e* ***Darci Biff de Souza,*** *minha mãe, dedico este trabalho.*

Sumário

Agradecimentos

Este trabalho é resultado de muita dedicação e de um longo esforço de pesquisa e auto-superação. São tantas as pessoas que me ajudaram, com idéias fantásticas, com apoio moral, científico e financeiro. Lugares que me inspiraram. Ambientes que proporcionaram o andamento e concretização da pesquisa. A tudo e a todos, posso apenas expressar minha eterna gratidão. Estou certa de que, sem esta união de forças, o fim não seria o mesmo.

Não citarei nomes para não ser injusta com todos que colaboraram para a conclusão satisfatória deste trabalho. Porém quero mencionar minha especial gratidão ao meu Orientador e Co-Orientador, pela paciência, pela clareza dos ensinamentos que me foram conduzidos nestes últimos três anos e, principamente, pela amizade firmada.

Agradeço a todos os professores que contribuiram para o meu crescimento pessoal e profissional, no período em que estive vinculada a esta instituição.

Não poderia deixar de mencionar a exímia competência de todos os profissionais que tornam o Instituto de Informática em um exemplo. A dedicação com que nos prestam favores e facilitam o nosso trabalho é de merecido reconhecimento.

Sou grata a minha casa de formacão, a UNISUL - Universidade do Sul de Santa Catarina, a todos os professores que apoiaram minha vinda a Porto Alegre e, também, pelas cartas de recomendação. Em especial, gostaria de manifestar o meu apreço à Prof[a]. Islândia e ao Prof. Wilson.

Aos meus amigos, "**Pessoas Fantásticas**", o meu muitíssimo obrigada, por tudo. Tenho certeza de que saberão quem são ao lerem esta frase.

Agradeço aos meus colegas, pelas dicas, pela paciência que tiveram comigo e pela convivência maravilhosa dos últimos três anos.

Sou profundamente grata a Paulo César Moraes, meu namorado, que sempre - com muito respeito, paciência e carinho - soube me compreender.

Agradecer às duas pessoas mais importantes de minha vida, meu pai Tito Fernandes de Souza e minha mãe Darci Biff de Souza, é pouco para o muito que recebo diariamente.

À minha família, sem esta estrutura maravilhosa eu não teria conseguido, disto estou certa.

Aos demais, que não citei explicitamente, quero expressar meu sentimento pleno de agradecimento, por tudo que me foi proporcionado.

Lista de Abreviaturas

ADOORATA	*A Distributed Object Oriented Approach for Real Time Systems Development*
AMADEUS	Ambiente de Metodologias Adaptáveis de Desenvolvimento Unificado de *Software*
ASCII	*American Standard Code for Information Interchange*
DD	Dicionário de Dados
DDL	*Data Definition Language*
ER	Entidade&Relacionamento
MOF	*Meta-Object Facility*
MVC	*Model View Controler*
OA&DF	*Object Analysis and Design Facility*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OMT	*Object Modeling Technique*
OO	Orientação a Objetos
OOA	*Object Oriented Analysis*
OOA&D	*Object Oriented Analysis and Design* (Análise e Projeto Orientado a Objeto)
OOAD	*Object Oriented Analysis and Design* (Metodologia do Booch)
OOD	*Object Oriented Design*
OOPSLA	*Object Oriented Programming, System and Language*
OOSE	*Object Oriented System Engineering*
ROOM	*Real Time Object Oriented Modeling*
SGBD	Sistema Gerenciador de Banco de Dados
SiMOO-RT	*Simulation Modeling Object-Oriented Real-Time*
UML	Unified Modeling Language (Linguagem de modelagem Unificada)
UML-RT	*Unified Modeling Language for Real Time* (Linguagem de Modelagem Unificada para Modelar Sistemas de Tempo Real)

Lista de Figuras

Lista de Tabelas

Resumo

A computação de tempo real é uma das áreas mais desafiadoras e de maior demanda tecnológica da atualidade. Está diretamente ligada a aplicações que envolvem índices críticos de confiabilidade e segurança. Estas características, inerentes a esta área da computação, vêm contribuindo para o aumento da complexidade dos sistemas tempo real e seu conseqüente desenvolvimento. Isto fez com que mecanismos para facilitar especificação, delimitação e solução de problemas passem a ser itens importantes para tais aplicações.

Este trabalho propõe mecanismos para atuarem no desenvolvimento de sistemas de tempo real, com o objetivo de serem empregados como ferramenta de apoio no problema da verificação de presença de inconsistências, que podem vir a ocorrer nos vários modelos gerados partir da notação da linguagem de modelagem gráfica para sistemas de tempo real - UML-RT(*Unified Modeling Language for Real Time*).

Estes mecanismos foram projetados através da construção de um metamodelo dos conceitos presentes nos diagramas de classe, de objetos, de seqüência, de colaboração e de estados. Para construir o metamodelo, utiliza-se a notação do diagrama de classes da UML (*Unified Modeling Language*). Contudo, por intermédio das representações gráficas do diagrama de classes não é possível descrever toda a semântica presente em tais diagramas. Assim, regras descritas em linguagem de modelagem OCL (*Object Constraint Language*) são utilizadas como um formalismo adicional ao metamodelo. Com estas descrições em OCL será possível a diminuição das possíveis ambigüidades e inconsistências, além de complementar as limitações impostas pelo caráter gráfico da UML.

O metamodelo projetado é mapeado para um modelo Entidade&Relacionamento. A partir deste modelo, são gerados os *scripts* DDL (*Data Definition Language*) que serão usados na criação do dicionário de dados, no banco de dados Oracle. As descrições semânticas escritas através de regras em OCL são mapeadas para *triggers*, que disparam no momento em que o dicionário de dados é manipulado.

O MET Editor do SiMOO-RT é a ferramenta diagramática que faz o povoamento dos dados no dicionário de dados. SiMOO-RT é uma ferramenta orientada a objetos para a modelagem, simulação e geração automática de código para sistemas de tempo real.

Palavras-chave: Sistema de Tempo Real, Orientação a Objetos, Verificação de Consistência, Metamodelagem, Metodologias para Desenvolvimento de Sistemas de Tempo Real.

Abstract

Real time computing is one of the most challenging and growing areas in computer science nowadays. This area is strong related with high levels of reliability and safety, specially when deadling in critical real time applications. These demands on dependability increases the complexity envolved in the development of real time systems. In order to tackle this complexity, mechanisms to facilitate the specification and domain problem delimitation are required.

This work proposes some mechanisms to help designers during the development process of real time systems. A tool support and concepts for performing consistency checking during the real time systems design is presented. These inconsistencies usually occur when designers are working in different model views, such as those proposed by the Unified Modeling Language (UML) and its Real Time extension (UML-RT).

Consistency checking is performed through the use of a metamodel and a set of semantic rules to define constraints in the metamodel. The metamodel, described in UML class diagram notation, depicts the concepts used in different UML diagram (class diagram, object diagram, sequence diagram, collaboration diagram and statechart diagram). The approach suggests the use of semantic rules described in modeling language OCL (Object Constraint Language) as an additional formalism to the metamodel. These rules should overcome some of the drawbacks of the graphic notation, reducing the ambiguities and inconsistencies in the model.

The metamodel is translated to Entity&Relationship(ER) model and then is translated to scripts in DDL(Data Definition Language). These scripts are used to create a data dictionary in a comercial database system (Oracle). The OCL semantic rules are translated to database triggers, that are dispatched when the data dictionary is populated.

As front-end interface to system designers the MET editor available in the SiMOO-RT environment is adopted. SiMOO-RT is an object oriented tool for the modeling, simulation and automatic generation of code for real time systems.

Keywords: Real Time System, Object Oriented, Consistency Checking, Metamodeling, Methodologies for Real Time Modeling, Tools for Modeling and Simulation of Real Time Systems.

1 Introdução

A tecnologia de controle de componentes eletrônicos e microeletrônicos, e a própria informática vêm sendo desenvolvidas exaustivamente nas últimas décadas. Este avanço tecnológico vem exigindo, cada vez mais, a utilização de sistemas computacionais, cujo bom funcionamento não dependa somente do processamento dos sinais de entrada em sinais de saída mas que também sejam processados levando-se em consideração rígidas restrições temporais.

Sistemas de automação industrial, sistemas de controle de tráfego aéreo, marca-passos implantados, eletrodomésticos, entre outros, são alguns exemplos de aplicações que necessitam que restrições temporais sejam implementadas em seus sistemas de controle. Eventuais falhas nestes sistemas podem trazer conseqüências catastróficas. Equipamentos como os marca-passos, que são implantados no peito de um ser humano, necessitam ser gerenciados por um sistema capaz de enviar os estímulos ao coração no tempo e freqüência necessários para o bom funcionamento deste músculo. Os sistemas de controle de tráfego aéreo dos aeroportos devem impedir que duas aeronaves aterrizem simultaneamente em uma mesma pista.

Além de situações deste tipo, tendências indicam que os sistemas de tempo real do futuro deverão ser ainda maiores e mais complexos, exigirão, cada vez mais, um comportamento dinâmico e com alta capacidade de adaptação a diversas condições de operação. Isto torna o projeto de sistemas de tempo real mais complexo do que o projeto de sistemas convencionais, nos quais restrições temporais não são necessárias.

Assim, fica evidente que mecanismos para minimizar os erros, que possivelmente ocorrerão, durante a análise, o projeto e a implementação de sistemas de tempo real devem ser desenvolvidos, objetivando a diminuição, ao máximo, do impacto e conseqüências de tais erros sobre o produto final, o software.

Este trabalho tem como objetivo propor um mecanismo de acompanhamento no desenvolvimento de sistemas de tempo real, para ser utilizado pelos profissionais que os desenvolvem. Para tanto, foi utilizada a linguagem de modelagem visual UML (*Unified Modeling Language*), oriunda da tecnologia de orientação a objetos. Esta tecnologia adota conceitos que descrevem a realidade a ser trabalhada de forma muito similar ao ambiente real. Isto torna o desenvolvimento de sistemas de tempo real mais simples e fácil de ser compreendido.

1.1 Motivação

A exigência de embutir alto nível de qualidade e produtividade aos sistemas de tempo real, induziu à necessidade de adotar metodologias para análise e projeto destes sistemas.

Amparados por este tipo de tecnologia, alcançar níveis de qualidade condizentes com os exigidos pelas aplicações da atualidade, é satisfatório. Porém, em uma análise do estado da arte atual das metodologias para o desenvolvimento de aplicações de sistemas de tempo real, feita durante a primeira etapa desta pesquisa, foram evidenciadas deficiências na descrição de requisitos temporais.

Além dos requisitos temporais, as atuais metodologias nem sempre apresentam mecanismos que permitem identificar possíveis inconsistências, cometidas pelos

desenvolvedores durante o processo de modelagem de sistemas de tempo real, em sua totalidade.

As metodologias de desenvolvimento que apresentam estes mecanismos elucidam apenas quais procedimentos devem ser executados, eximindo-se de indicar como estes devem ser aplicados.

Concomitantemente, estas metodologias também são dotadas de um alto grau de expressividade gráfica. Esta "grande liberdade" de ação suscita o surgimento de ambigüidades na interpretação de conceitos fundamentais.

As limitações das metodologias citadas motivaram o desenvolvimento desta dissertação, que tem como principal objetivo propor um mecanismo de validação das descrições que são realizadas durante a modelagem de sistemas de tempo real.

1.2 Objetivo do Trabalho

O objetivo principal desta dissertação é a obtenção de um metamodelo que permita efetuar a verificação de consistência das descrições, que devem ser realizadas durante a modelagem de sistemas de tempo real.

Um metamodelo é, basicamente, um modelo usado para descrever os vários tipos de modelos que se deseje construir [ODE 95]. A descrição de um sistema de tempo real, utilizando uma metodologia, realiza-se através de vários modelos. Neles, podem ocorrer dois tipos básicos de inconsistências: inconsistências encontradas dentro de um modelo e inconsistências entre mais de um modelo. O mecanismo proposto, através da metamodelagem juntamente com um conjunto de regras, permite analisar estes dois tipos de inconsistências - que podem ser geradas quando a linguagem de modelagem utilizada seja UML-RT (*Unified Modeling Language for Real Time*) [DOU 98].

Neste trabalho serão utilizados os diagrama de classes, diagrama de objetos, diagrama de estados, diagrama de seqüência e diagrama de colaboração, sendo, estes dois últimos, também chamados de diagramas de interação. Estas técnicas, propostas na linguagem de modelagem UML-RT, serão descritas através do uso dos conceitos de metamodelagem, com a utilização da notação gráfica do diagrama de classes da UML.

Em contrapartida, através das representações gráficas do diagrama de classes, não é possível descrever toda a semântica presente em um modelo, pois também poderão contribuir para o surgimento de ambigüidades de interpretação. Para poder complementar esta descrição, será utilizada a linguagem de modelagem OCL (*Object Constraint Language*). Com ela, será possível diminuir as prováveis ambigüidades, além de complementar as limitações impostas pelo caráter gráfico da UML.

Outro objetivo deste trabalho consiste em avaliar a possibilidade de incorporar ao metamodelo UML-RT conceitos que permitam uma descrição mais adequada dos requisitos temporais, tais como: a periodicidade na execução de operações, o tempo de execução, a determinação de limites máximos no tempo de execução, descrever se a execução pode ou não ser interrompida e o tratamento de exceção.

1.3 Principais Contribuições

Dentre as contribuições do presente trabalho podem-se destacar:

- A obtenção de um metamodelo para os conceito presentes na notação da UML-RT.
- A obtenção de um conjunto de regras formalizadas em OCL, melhorando a precisão do modelo. Ou seja, diminuindo a presença de inconsistências e ambigüidades.
- A permissão das descrições dos requisitos temporais, dos sistemas de tempo real, através do mapeamento dos conceitos de tempo de execução das operações, do limite máximo de execução das operações (*Deadlines soft, hard* e *firm*), dos períodos em operações periódicas, dos intervalos máximo e mínimo ente as execuções nas operações aperiódicas, das operações atômicas e operações não atômicas.
- O mapeamento do metamodelo orientado a objetos para um modelo ER-Entidade&Relacionamento.
- A implementação de um dicionário de dados, a partir do modelo de ER, na ferramenta de banco de dados Oracle. Este dicionário de dados irá fornecer suporte, como repositório de dados, à ferramenta SiMOO-RT.

1.4 Estrutura da Dissertação

As demais partes, desta dissertação, estão organizadas em cinco capítulos e dois anexos.

O Capítulo 2 apresenta a contextualização da pesquisa, expondo de forma detalhada os conceitos que estão inseridos neste trabalho.

O Capítulo 3 descreve como foi feita a modelagem dos conceitos da notação da linguagem de modelagem UML-RT. Observações ao metamodelo proposto, descrição das modelagens estática e comportamental, e os componentes da modelagem são abordados neste capítulo. Em essência, este descreve a parte do metamodelo que trata dos conceitos da notação UML-RT e as restrições em OCL, formuladas sobre estes conceitos. Adotou-se uma forma de organização do texto de maneira que a leitura seja acessível à compreensão e o entendimento dos conceitos, que foram mapeados para o metamodelo, seja facilitado.

Primeiramente, é mostrado a figura com os conceitos modelados na notação do diagrama de classe da UML. Logo em seguida, é feito uma breve descrição, expondo o que a figura está reproduzindo.

Concluída esta descrição, segue a exposição de cada classe de conceito representada no metamodelo. São descritas em negrito com a primeira letra em maiúsculo. Seus atributos são descritos em letra minúscula e em itálico. A semântica presente na classe e em seus relacionamentos é descrita, inicialmente, em linguagem natural (português) estruturada. Esta descrição é feita em forma de frases curtas, numeradas com as letras do alfabeto em minúsculo, por exemplo: a), b), c),.... Em seguida, na seção cujo título se chama "Descrição das Restrições em OCL", esta semântica é formalizada através de expressões na linguagem OCL.

Nesta seção, que descreve as restrições OCL, é exposto o nome da classe do conceito em negrito com a primeira letra em maiúsculo, seguido pelas frases em linguagem natural numeradas, como: a), b), c), ..., e sua correspondência em OCL.

Este procedimento é repetido para todos os diagramas resultantes do mapeamento dos conceitos e da representação gráfica destes conceitos. Sendo que esta última é exposta no anexo I.

No Capítulo 4, a exposição do protótipo do trabalho. Aqui, são descritas as regras utilizadas para mapear o metamodelo, orientado a objetos, para um modelo ER. Como procedeu-se a criação do dicionário de dados a partir deste modelo ER, também está à mostra. Seguidamente, é feito uma descrição do ambiente SiMOO-RT e das formas como este irá interagir com o dicionário de dados.

No Capítulo 5 são feitas as considerações finais, conclusões e propostas sobre trabalhos que poderão ser desenvolvidos por futuras pesquisas.

O Anexo I descreve a parte do metamodelo que trata das representações gráficas de cada conceito presente na notação UML-RT. Em seguida, no Anexo II, são descritas as DDL que gerarão o dicionário de dados no banco de dados Oracle.

2 Contextualização da Pesquisa

2.1 Sistemas de Tempo Real

A aplicabilidade dos sistemas de tempo real, FIGURA 2.1, vem aumentando e proporcionando conforto em vários setores da vida moderna. Estes sistemas variam em tamanho e escopo [DOU 97]. São utilizados em ambientes domésticos, hospitalares, industriais e até em objetos de uso pessoal. Os fornos de microondas, a automação de processos industriais, sistemas de eletrônica embarcada de veículo e marca-passos cardíacos artificiais são alguns exemplos de sistemas que utilizam a tecnologia de tempo real.

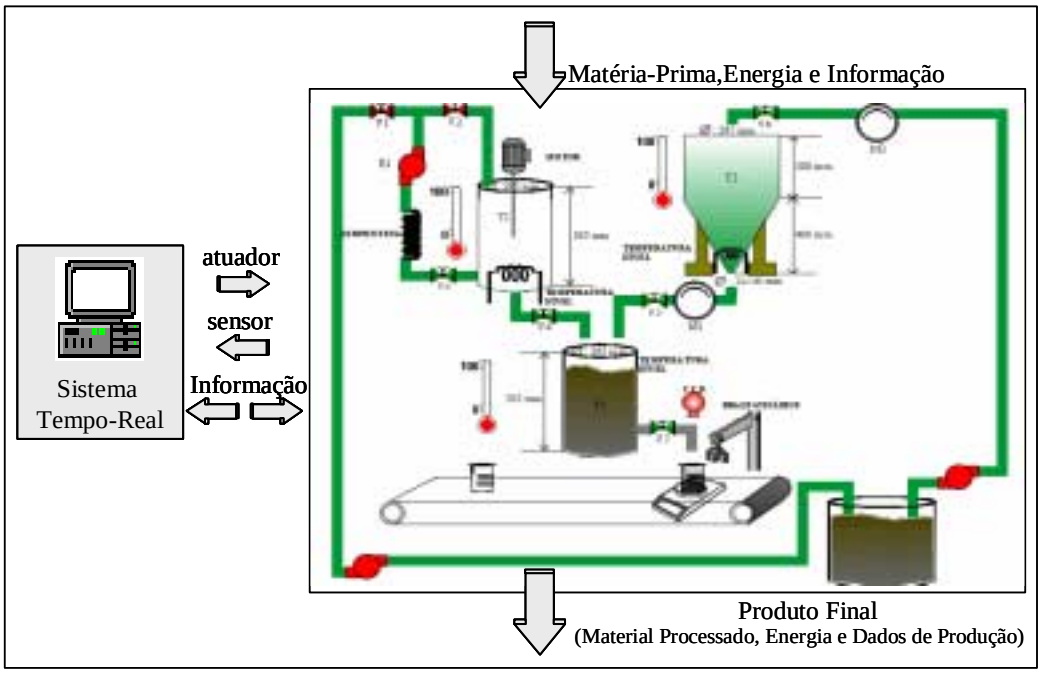


FIGURA 2.1 – Sistema de Tempo Real.

Existem várias definições para explicar o que são sistemas de tempo real. Destas, três elucidam, de forma bem sucinta, tais sistemas. Pereira em [PER 96], caracteriza sistema de tempo real como um sistema computacional, cujo correto funcionamento não depende apenas do processamento de sinais de entrada ou estímulos e sua transformação em sinais ou variáveis de saída, mas especialmente da satisfação de requisitos temporais. Bennett em [BEN 94] conceitua estes sistemas de maneira semelhante a Pereira [PER 96], expõe que os sistemas de tempo real são aqueles que devem produzir respostas corretas, dentro de um tempo limite bem definido. Caso tais respostas excedam o tempo determinado são denominadas de respostas não válidas. Laplante em [LAP 97] conceitua sistemas de tempo real de forma mais abrangente. Explicita as conseqüências do não cumprimento dos requisitos temporais especificados. Explana que um sistema de tempo real é um sistema que

deve satisfazer restrições explícitas de tempo de resposta, caso contrário, há riscos de sérias conseqüências, inclusive falha[1].

Uma característica importante dos sistemas de tempo real é seu forte acoplamento com o ambiente de funcionamento. Esta interação com o meio, geralmente, é propiciada através de sensores e atuadores, que respondem a eventos dentro de limites de tempo especificado [AWA 96]. De acordo como os requisitos temporais impostos pelo ambiente de funcionamento, os sistemas de tempo real são classificados como *hard real time* e *soft real time* [BEN 94, PER 96, AWA 96].

Nos sistemas classificados como *hard real time*, os requisitos temporais devem ser cumpridos, pois as respostas, após o tempo limite determinado, são consideradas falhas. Estas falhas podem acarretar conseqüências sérias [PER 96], criando situações em que há riscos para a segurança/vida humana. Como exemplos, podem ser citados: o não cumprimento dos requisitos temporais por sistemas que controlam usinas nucleares, equipamentos médicos e freios automáticos de carros ou de aeronaves [PER 96].

O determinismo temporal em sistemas *hard real time* é uma propriedade essencial [BOO 94]. Um sistema é classificado como determinístico se para cada estado possível e para cada conjunto de entradas, um único conjunto de saídas e um próximo estado possam ser determinados. Portanto, entende-se por determinismo temporal que o tempo de resposta de cada conjunto de saída seja conhecido [LAP 97].

Nos sistemas *soft real time* as respostas fora do tempo determinado, normalmente, são aceitáveis. Como exemplos destas situações, podem-se evidenciar sistemas de telefonia digital ou um sistema de base de dados *online* de reservas de passagens aéreas [DOU 98]. Nestas situações, respostas obtidas fora do intervalo de tempo determinado não costumam oferecer riscos à vida humana.

Outra classificação de sistemas de tempo real, de acordo com os requisitos temporais, é dada por Laplante e Douglass [LAP 97, DOU 98]. Estes autores classificam os sistemas, além de *soft* e *hard*, como *firm real time*. *Firm* é uma combinação de ambos, do *hard* e do *soft*. Respostas fora do tempo determinado são válidas, porém a não obediência aos requisitos temporais, por um longo intervalo de tempo, torna as respostas inadmissíveis. O sistema de monitoração da respiração em pacientes com problemas respiratórios é um exemplo, típico, de sistema *firm real time*. O oxigênio pode chegar ao pulmão do paciente com alguns segundos de atraso, sem afetar a sua saúde. Porém, se este atraso for de alguns minutos, é inaceitável e prejudicial à saúde do mesmo.

Os sistemas de tempo real também podem ser classificados como: reativos ou embutidos. No primeiro caso, são sistemas que estão continuamente interagindo com meio, como, por exemplo, um sistema de controle de tanques. No segundo caso, os embutidos, são completamente encapsulados pelo *hardware* controlado. Um exemplo destes sistemas é o microprocessador usado para controlar a mistura de combustível e ar em carburadores eletrônicos de muitos automóveis, [LAP 97].

---

[1] Por falha, entende-se o não cumprimento de um ou mais requisitos determinados na especificação do sistema.

2.2 Orientação a Objetos

Até pouco tempo atrás, os sistemas de tempo real eram desenvolvidos desprovidos de qualquer procedimento metodológico (*ad-hoc*). As conseqüências oriundas deste tipo de desenvolvimento eram traduzidas em cronogramas de projetos freqüentemente transpostos, em custos adicionais agregados aos orçamentos originais, em dificuldades no cumprimento das especificações do cliente e na manutenção dispendiosa dos sistemas [PER 96]. Além destas conseqüências, os rápidos avanços tecnológicos das últimas décadas aumentou, consideravelmente, a complexidade dos sistemas de tempo real. Estes fatos motivaram os desenvolvedores de sistemas a buscar metodologias que permitissem aprimorar todo o processo de desenvolvimento.

A popularidade da tecnologia de Orientação a Objetos (OO), nas últimas duas décadas, trouxe inovações para auxiliar o desenvolvimento de sistemas de tempo real. Não sob a forma de uma solução "mágica" que, repentinamente, torna simples a construção de um sistema mas como uma tecnologia que dispõe de mecanismos que facilitam a construção de sistemas mais complexos, em menos tempo e com menos erros [DOU 98].

Segundo [SIL 96, FRE 96, DOU 98, TKA 94, BEC 99], a utilização da OO, no desenvolvimento de sistemas de tempo real, propicia a obtenção de vantagens. Estas vantagens podem ser destacadas na forma dos seguintes tópicos:

a) **Maior consistência nas visões dos modelos:** Na OO, o mesmo conjunto de visões dos modelos é usado em todas as fases do desenvolvimento do projeto. Estas visões dos modelos vão agregando detalhamentos até chegarem a implementação propriamente dita. Por exemplo, os objetos e classes identificados na análise têm representação direta no código. É quase trivial mostrar os relacionamentos entre definição do problema e a sua solução [DOU 98].

b) **Maior abstração do domínio do problema:** Na OO, os conceitos de classe e de objeto mantêm forte acoplamento com os dados e as operações que manipulam estes dados, do mesmo modo que ocorre com os elementos do domínio real. Isto torna a tecnologia intuitiva, permitindo que os desenvolvedores e pessoas envolvidas no domínio do problema tenham facilidades para compreender e validar as descrições feitas [SIL 96].

c) **Manutenção facilitada e maior estabilidade frente às mudanças:** Conceitos como abstração, encapsulamento, herança e polimorfismo fazem com que as alterações ou as extensões da aplicação sejam estáveis, sem agregar maiores complicações. Por exemplo, o conceito de encapsulamento permite que a classe sofra mudanças internas, sem alterar a sua interação com as demais classes do sistema. Outro tópico que vem ao encontro de facilitar a manutenção é a boa coesão dos sistemas orientados a objetos. Esta boa coesão dos aspectos dos sistemas ocorre, novamente, através do conceito de encapsulamento, inerente às classes e aos objetos, uma vez que seus dados mantêm-se fortemente acoplados com o comportamento [TKA 94].

d) **Reuso facilitado:** O reuso poderá ocorrer nas situações em que há semelhança nas descrições de domínios de problema [SIL 96]. Esta possibilidade de reuso pode ser facilitada através do conceito de herança. As classes genéricas, da estrutura de herança, facilitam o reuso através da adição de novas classes ou

extensão das já existentes. Isto pode ocorrer sem grandes alterações no código fonte. São as diferenças que deverão ser codificadas.

Através do conceito de herança, é permitido a especificação incompleta dos objetos, sendo que esta pode ser refinada, satisfazendo as necessidades do domínio do problema. No entanto, cabe uma ressalva, estas duas formas de utilizar o conceito de herança não garantem que o reuso irá ocorrer automaticamente.

e) **Melhor suporte ao conceito de confiabilidade[2]:** Através dos conceitos de abstração e encapsulamento, com o auxílio de pré e pós-condições[3], é possível projetar interfaces, bem definidas, de interação entre os objetos. Isso resulta em um controle maior da forma como os objetos interagem, melhorando a confiabilidade e diminuindo a ocorrência de erros no sistema [DOU 98].

f) **Suporte a concorrência:** O conceito de concorrência, importante aos desenvolvedores de sistemas de tempo real, é naturalmente modelado pelos objetos na OO [PER 96].

2.3 Comparação das Metodologias

Como foi comentado anteriormente, a evolução tecnológica e a sofisticação dos requisitos funcionais elevaram a complexidade dos sistemas computacionais, levando a problemas no cumprimento dos cronogramas, extrapolação de orçamentos e dificuldades em realizar o que havia sido especificado. Isto evidenciou a necessidade de mecanismos que auxiliassem a equipe de desenvolvimento a gerenciar os problemas. Esta necessidade intensificou as investigações, cujo resultado foi um produtivo esforço de pesquisa na busca de soluções para estes problemas.

Muitas propostas de metodologias, para desenvolver sistemas, surgiram no final da década de oitenta. Algumas delas foram originadas da evolução de metodologias de desenvolvimento (análise e projeto) estruturado. Outras, inovaram, radicalmente, os conceitos de desenvolvimento de sistema [FIC 92].

Este grande número de ferramentas metodológicas, colocadas à disposição das equipes de desenvolvimento, evidenciou a necessidade de conhecer o potencial de cada uma a ser utilizada. Avaliações prévias, mostrando as características das metodologias, denotando vantagens e deficiências, se tornaram imprescindíveis. Estas avaliações devem ser aplicadas sob muitos aspectos, dos quais alguns são destacados:

a) Em relação ao suporte dado a todo o ciclo de vida do sistema - ciclo de vida de um sistema consiste no conjunto de etapas que são divididas e nomeadas, de formas distintas, por diferentes autores. Em [SIL 96], esta etapas são expostas na forma de análise, projeto, implementação e manutenção.

---

[2]Confiabilidade é a probabilidade que um sistema opere sem falhas, por um período de tempo especificado. Em outras palavras, um sistema é dito confiável quando este é capaz de reabilitar-se dos erros, de forma natural, não prejudicando o funcionamento.

[3] As pré-condições e pós-condições são codificações que fazem a verificação de integridade dos métodos. São executadas antes e depois de cada método da classe e fazem as consistências baseando-se nos argumentos de entrada e saída.

b) Quanto aos mecanismos que a metodologia disponibiliza para fazer as descrições: funcional, estática e comportamental do sistema.

c) Com relação ao grau de suporte aos conceitos da OO, tais como: abstração, herança, objetos, classes, encapsulamento e polimorfismo.

d) Quanto aos mecanismos para modelagem e gerenciamento da complexidade em sistemas de grande porte.

e) Quanto à clareza e à expressividade dos modelos gerados, ou seja, quanto à notação gráfica intuitiva.

Como foi mencionado, é imprescindível a ponderação dos aspectos citados acima ao se analisar uma metodologia. No entanto, para esta pesquisa, como abordado no item 1.3, é essencial o apreço a dois tópicos: os mecanismos para descrever os requisitos temporais e as formas, propostas pela metodologia, para consistir as descrições destes requisitos. Isso se deve ao fato de que a base para o bom funcionamento dos sistemas de tempo real começa pela correta especificação dos mesmos.

São inúmeros os trabalhos que apresentam análises comparativas de metodologias, [BEN 94, LAP 97, PAS 94, PAS 96, PER 96, SIL 96, BEC 97, FRE 96, FIC 92, SOU 97]. Algumas delas, analisadas nos trabalhos citados, tais como: ROOM [SEL 94], HRT-HOOD [BUR 94], Shlaer-Mellor [LEE 98], MASCOT e PAISley [BEN 94], foram concebidas, exclusivamente, para modelar sistemas de tempo real. Outras, como as apresentadas em OOSE [JAC 92], OOAD [BOO 94] e OMT [RUM 97], também são empregadas para desenvolver sistema de tempo real, porém foram criadas para modelar sistemas convencionais.

Uma análise minuciosa de cada metodologia, disponível para modelar sistemas de tempo real, foge ao âmbito desta dissertação. Entretanto, algumas destas técnicas e metodologias mais utilizadas são relatadas a seguir, na TABELA 1, acompanhadas da forma como disponibilizam ou não mecanismos para descrever e dar consistência aos requisitos temporais.

TABELA 1 – Resumo das Metodologias e Técnicas Aplicadas no Desenvolvimento de Sistemas de Tempo Real.

Metodologia/ Técnica	Descreve os Requisitos Temporais?	Implementa Consistência?
Diagrama de Fluxo de Dados de Tom DeMarco	Bastante utilizada, permite expressar concorrência, é complexo mostrar a sincronização, dificuldade em mapear a funcionalidade descrita para as classes, possui deficiências no que tange a representação de informações temporais.	Induz a criação de DD que permite ambigüidades.
Petri Nets [LAP 97]	Natureza matemática, ideais para modelar operações em ambientes multitarefa ou multiprocessado, não indicadas para modelar sistemas muito simples ou muito complexos.	Petri Nets podem ser empregadas para analisar restrições temporais.
Especificação Matemática [LAP 97]	Aplicável através de métodos formais, permite expressar os requisitos temporais, permite otimizar o código, o uso é muito complexo e pouco difundido, propensa a erro.	Formalismo matemático, pode ser empregado na verificação de consistência.

Shlaer - Mellor [LEE 98]	Descreve os requisitos temporais através da classe Timer [PER 96].	Propõe um conjunto de regras descritivas que explanam o que uma válida análise deve ter, mas não diz como produzir este resultado [LAN 93].
OMT [RUM 97]	Descreve os requisitos temporais: no diagrama de estados através dos atributos *time, start* e *end* [PER 96] e no diagrama de eventos expressando tempo no eixo vertical [PAS 96].	Não implementa.
OOSE [JAC 92]	Descreve os requisitos temporais: no diagrama de interação expressando o tempo no eixo vertical [JAC 92].	Não implementa.
OOAD [BOO 94]	Descreve os requisitos temporais: no diagrama de estados através de 'condição de transição' que expressa um *timeout*[4] para a permanência do objeto no estado [BOO94]; no diagrama de temporização através de um gráfico, sendo que no eixo x são expostos os objetos e no y, o tempo de execução das operações [PAS 96]; e no diagrama de objetos implementa sincronização através de adornos nas mensagens.	Não implementa.
ROOM [SEL 94]	De maneira semelhante ao método Shlaer-Mellor, ROOM descreve os requisitos temporais através da classe Timer [PER 96].	Não implementa.

Além das dificuldades para descrever os requisitos temporais e a ausência de mecanismos eficientes para fazer a verificação de consistência destes, o grande número de metodologias disponíveis traz outros inconvenientes para a equipe de desenvolvimento. As diferentes abordagens, aos conceitos básicos, por diferentes metodologias podem acarretar ambigüidades de terminologia, incorrendo nas mais diferentes interpretações [PAS 96]. Em outras situações, há a necessidade de combinar as técnicas presentes nas várias metodologias para obter uma descrição favorável do sistema [SOU 97].

Estas dificuldades conduziram as primeiras discussões que ocorreram na conferência OOPSLA[5], em 1993, com relação à necessidade de unificar os conceitos, melhores trabalhados pelas metodologias [PER 96]. Os debates prosseguiram. Em 1994, foi proposta a versão 0.8 da *UML - Unified Modeling Language* [RAT 97, OMG 99], unificando as metodologias de Grady Booch [BOO 94] e James Rumbaugh [RUM 97], incluindo a contribuição de vários outros metodologistas. Em 1997, foi lançada a versão 1.1 padronizada pela OMG e que agregou a metodologia de Ivar Jacobson [JAC 92]. Em janeiro de 1999, foi lançada a UML, versão 1.3, revisada [OMG 99].

[4] Timeout pode ser conceituado como um limite de tempo máximo, especificado para que algo aconteça/ocorra. Por exemplo: uma transição de estados ou uma execução completa de um método.

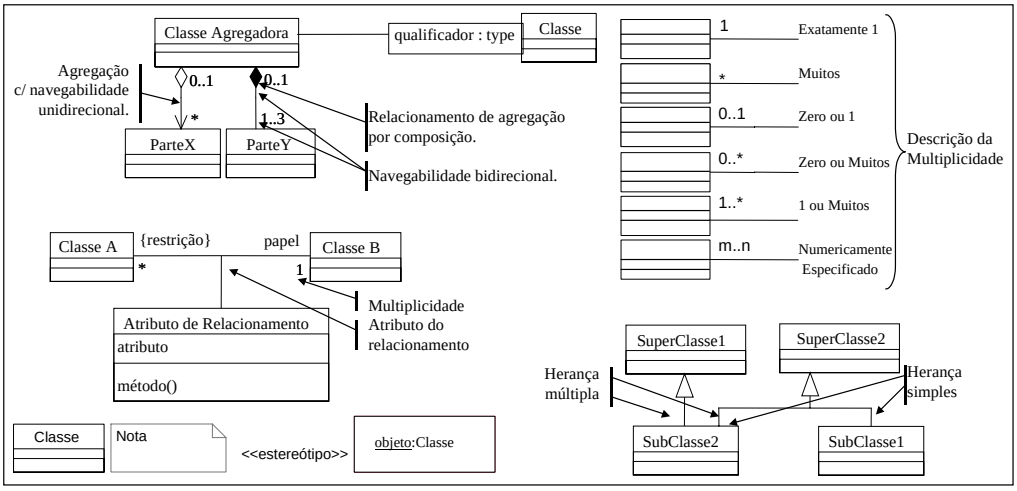
[5] OOPSLA - *Object Oriented Programming, System and Languages* – um dos maiores congressos da área de informática, que aborda o tema orientação a objetos.

2.4 UML e UML-RT

A *UML* (*Unified Modeling Language*) é conceituada como uma linguagem de modelagem visual, orientada a objetos, que foi concebida com o intuito de especificar, construir e documentar *software*, [RAT 97]. Esta linguagem é composta por um conjunto de diagramas que podem ser utilizados na modelagem, independentemente da tecnologia que será empregada na implementação do projeto.

Atualmente, a UML é considerada um passo significativo em termos de notação para descrição de sistemas OO, pois reúne relevantes experiências neste campo. Outra vantagem da utilização desta linguagem diagramática é o fato de que suas notações são aplicáveis a quaisquer processos de desenvolvimento. Além disto, propõe mecanismos de extensão, tais como: *stereotype*, *tagged value* e comentários [OMG 99]. Isto torna a notação da UML flexível e adaptável para a descrição de sistemas, em diferentes áreas de aplicação.

Douglass em [DOU 98], explana como utilizar a UML para modelar sistemas de tempo real. Na UML para tempo real, UML-RT, são propostos mecanismos para modelar as particularidades dos sistemas de tempo real. Basicamente, é empregado o mecanismo de extensão, *stereotype*, para instanciar novas subclasses de conceitos que irão, por exemplo, modelar os objetos ativos, as mensagens e os eventos. Esta extensão propõe uma notação gráfica para expressar marcas de tempo[6], marcas de estado[7], mensagem *broadcast* e mensagem concorrente no diagrama de seqüência. Propõe, também, que mensagens estereotipadas sejam empregadas no diagrama de colaboração. Sugere a utilização do diagrama de temporização, presente na metodologia de [BOO 94], para auxiliar nas descrições dos requisitos temporais. Outra sugestão relevante está no uso do diagrama de contexto da análise estruturada. O objetivo do uso do diagrama de contexto é mostrar os objetos do sistema, interagindo com os objetos externos do ambiente, de forma que possa ser visualizado a troca de mensagens e os eventos que ocorrem entre o 'sistema' e o 'meio'.



[6] Marcas de tempo são os adornos ou restrições expostas junto às mensagens, indicando um tempo máximo para a execução de um conjunto delas. Podendo, também, indicar um tempo máximo para que a funcionalidade, descrita através do diagrama de seqüência, seja executada.

[7] As marcas de estados, no diagrama de seqüência, indicam em que ‘estado’ o objeto se encontra no instante em que o método, disparado pela mensagem, está executando.

FIGURA 2.2 – Notação do Diagrama de Classes.

Neste primeiro momento, abordaremos os diagramas de classes e de objetos (mostrados na FIGURA 2.2), diagramas de interação: seqüência e colaboração (expostos na FIGURA 2.3) e o diagrama de estados (cuja a notação é mostrada na FIGURA 2.4). Através destes diagramas é possível descrever um sistema, fazer sua simulação e gerar o seu código.

No diagrama de classe é modelada a estrutura estática das entidades que compõem o sistema. Sendo que o diagrama de objetos mostra as instâncias das classes - útil nas descrições da simulação do modelo e na modelagem de características e comportamentos inerentes ao objeto.

O diagrama de estados descreve os estados possíveis que um objeto poderá assumir, num dado instante de tempo. Descreve, também, os eventos, as condições ou as ações que disparam as transições de um estado, do objeto, para outro. Este diagrama apresenta as ações que são executadas enquanto o objeto permanece no estado. O diagrama de estados é caracterizado por Douglass em [DOU 98] como construtivo porque com as descrições feitas no mesmo é possível gerar um código executável para o objeto.

Infelizmente, os diagramas de estados não mostram a execução de uma determinada funcionalidade do sistema. Esta visualização é proporcionada pelos diagramas de interação. No diagrama de seqüência (FIGURA 2.3 B), pode ser visualizado como o sistema se comporta ao realizar uma atividade. O diagrama de colaboração e o diagrama de seqüência mostram informações semelhantes, porém com representações gráficas distintas. Desta maneira, o uso do diagrama de colaboração é importante quando há a necessidade de evidenciar os objetos e seus vínculos, facilitando a compreensão da interação, como é mostrado na FIGURA 2.3 A.

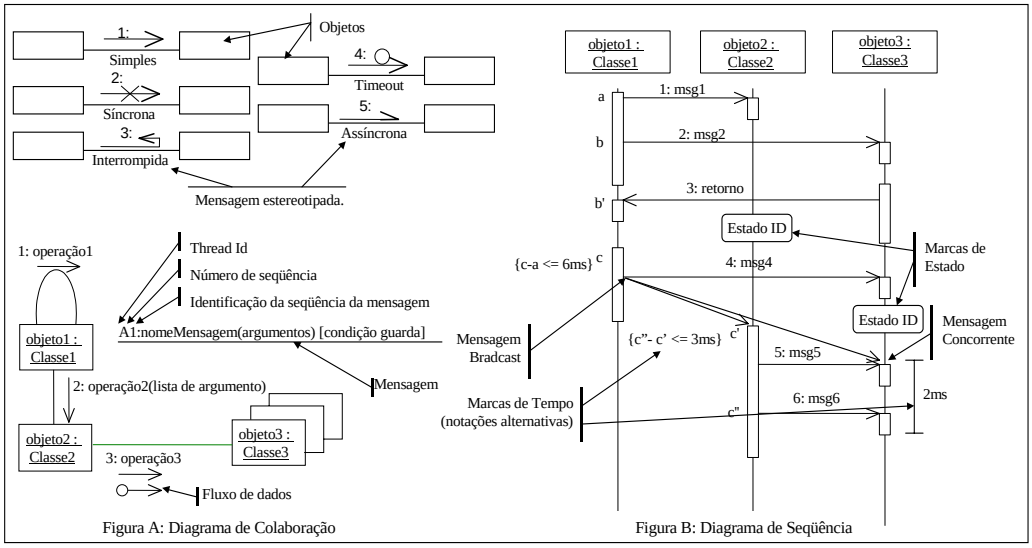


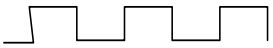
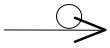
FIGURA 2.3 – Notação dos Diagramas de Interação.

Como mencionado anteriormente, foi proposto um padrão de estereótipos para representar as mensagens periódicas e aperiódicas e um padrão de sincronização das mesmas, para a notação dos diagramas de interação da UML-RT. Mensagens

periódicas são aquelas caracterizadas por um período, dentro do qual devem ocorrer, e pelo *jitter*, que é a variação em torno do período, o ocorrido *versus* o tempo em que deveriam ocorrer. Um mensagem é dita aperiódica quando o intervalo de chegada ao objeto não pode ser prognosticado. Estas mensagens possuem um intervalo mínimo de chegada. Caso este tempo seja ultrapassado, expressa que há possibilidade de chegada em grupo. São independentes e, de certa forma, randômicas. Isto significa que a chegada de uma mensagem não afeta a probabilidade de chegada da próxima, dentro do intervalo de tempo mínimo estabelecido.

A indicação dos estereótipos, para ambos os tipos de mensagens, pode ser feita tanto com a descrição textual dentro dos indicadores <<>> como através de adornos. A TABELA 2 ilustra os adornos para as mensagens.

TABELA 2 – Estereótipo para a Modelagem das Mensagens.

Estereótipo Mensagem	Estereótipo Sincronização das Mensagens		
Mensagem Periódica 	 Simples	 Síncrona	 Timeout
Mensagem Aperiódica 	 Interrompida	Assíncrona	 Esperando

O diagrama de estados é uma técnica bem conhecida dentro da orientação a objetos. É baseado nos *statechart* de David Harel [HAR 88]. Foi adotada por Rumbaugh no método OMT [RUM 97] e por Booch [BOO 94]. Esta técnica também incorpora características das máquinas de *Moore* e máquinas de *Mealy* [LAP 97]. A aplicabilidade deste diagrama é na descrição do comportamento dos objetos. Cada diagrama descreve os estados possíveis que um objeto pode assumir, num dado instante, dentro da delimitação do problema modelado. A notação para descrever os diagramas de estados é mostrada na FIGURA 2.4.

As visões estáticas e dinâmicas dos sistemas modelados com a notação (dos diagramas de classes, de objetos, de interação e de estados) da UML-RT utilizam conceitos que são descritos por mais de um diagrama. Isto dificulta a tarefa da equipe de projeto em manter a consistência destes conceitos sem o auxílio de mecanismos que propiciem, de forma automática, esta facilidade. Além disso, na modelagem de um sistema de tempo real, é imprescindível que cada diagrama apresente a informação descrita de forma consistente e complementar com os demais.

Na próxima seção serão abordados os mecanismos propostos pela UML, para verificar a consistência de seus modelos. Posteriormente, serão discutidas as formas de se obter a verificação de consistência, vantagens e desvantagens destas formas, como também a abordagem utilizada por esta pesquisa.

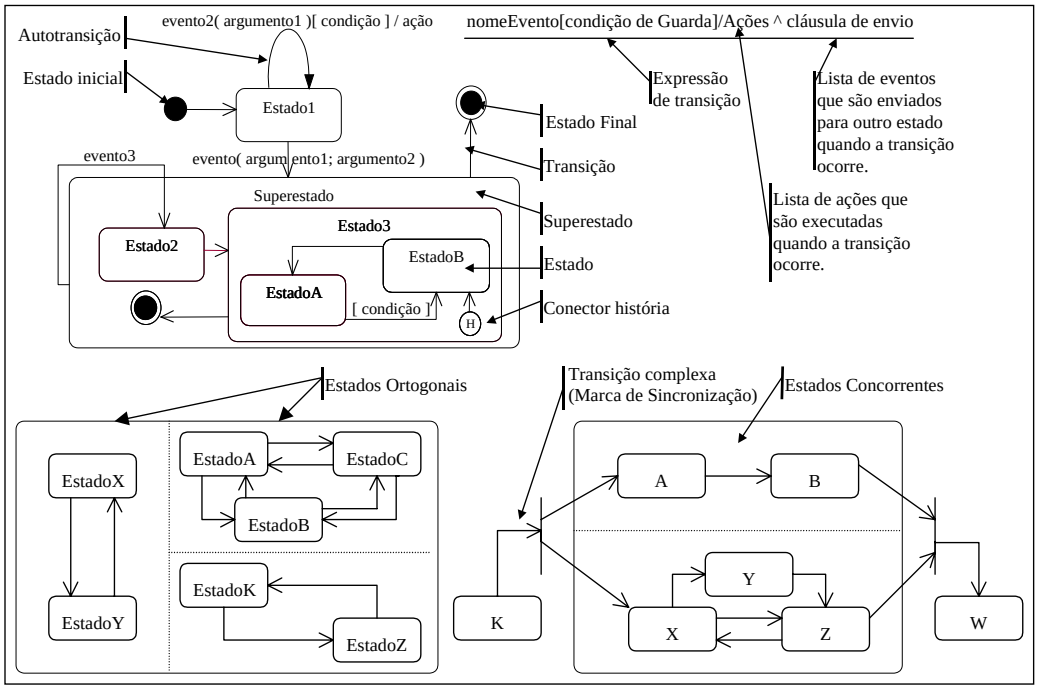


FIGURA 2.4 – Notação dos Diagramas de Estados.

2.5 Verificação de Consistência

Um modelo é definido como consistente quando não há presença de erros. De acordo com esta definição, a verificação de consistência pode ser descrita como a tarefa de detectar possíveis inconsistências, cometidas pelo desenvolvedor, durante o processo de desenvolvimento de sistemas (neste caso, sistemas de tempo real). Um exemplo que caracteriza uma inconsistência pode ser mostrado através de dois modelos que descrevem a mesma informação de maneira diferente. Na prática, isto pode ocorrer quando o tempo de execução de um método tenha sido descrito como 30ms por um modelo e o mesmo método, em um outro modelo, tenha sua execução especificada como 40ms. Isto é considerado uma inconsistência.

Na modelagem através de diagramas, como a notação da UML, as descrições obtidas não são, suficientemente, precisas. Esta falta de precisão pode levar a equipe de desenvolvimento a interpretações ambíguas das descrições dos sistemas. Em muitos casos, é necessário a utilização de descrições, em linguagem natural, para melhor expressar restrições sobre os elementos dos modelos do sistema. Estas descrições livres são feitas em determinado idioma, inglês, por exemplo. Infelizmente, este tipo de formalismo também não é efetivo. Descrições em linguagem natural são passíveis de interpretações ambíguas.

Para suprir esta deficiência descritiva dos diagramas, a UML propõe que se descrevam as restrições usando a linguagem OCL - *Object Constraint Language*, [OCL 98, WAR 99].

OCL é uma linguagem que tem sua origem no método Syntropy de Cook e Daniels [COO 94]. Dispensa detalhes de implementação, pois é uma linguagem de modelagem. Sua funcionalidade está em descrever expressões atômicas sobre

classes, relacionamentos, descrever pré e pós-condições relacionadas com a existência dos métodos, descrições de condições de guarda e outras mais.

A OCL, por ser uma linguagem semiformal, tem a vantagem de não agregar a complexidade dos métodos formais nas suas descrições. Conseqüentemente, a leitura e a compressão das restrições, descrita pela mesma, é facilitada. Isto contribui para a legibilidade do modelo.

Entretanto, o fato de ser uma linguagem semiformal passa a não ser vantajoso nas descrições de sistema altamente críticos. Ou seja, sistemas onde falhas podem levar a grandes perdas financeiras ou até mesmo pôr em risco vidas humanas. Esta desvantagem, segundo [EVA 99a,b,c], está diretamente relacionada com o fato da OCL apresentar carências de mecanismos que possibilitem:

- Uma maior precisão nas descrições.
- A verificação formal das propriedades dos modelos da fase de análise.
- A verificação, quanto à isenção de erros, das descrições na fase de projeto, não sendo possível fazê-las através da OCL.

Pesquisas realizadas pelo *Precise UML Group* (pUML), como as publicadas em [EVA 99a,b,c], [KEN 97], [KEN 99] e [BRU 99], propõem a transferência dos benefícios das linguagens de descrição formal, como Z e Spectrum [EVA 99], para a UML. Estes estudos apontam os benefícios dos formalismos matemáticos, tais como: tornar precisa a interpretação dos modelos; verificação de consistência e a solução do problema da ambigüidade. Em contrapartida, os formalismos matemáticos também têm desvantagens: o seu uso exige uma boa base matemática por parte do desenvolvedor; não tem treinamento difundido; sua aplicação é complexa e a legibilidade e compreensão dos modelos ficam comprometidas, principalmente, por parte dos componentes da equipe com menos conhecimento. A complexidade deste tipo de formalismo o torna propenso a erros.

Além da OCL e das pesquisas realizadas pelo *Precise UML Group* (pUML), ambas comentadas anteriormente, a UML propõe um modelo lógico que descreve os conceitos da sua notação.

Esta descrição é efetuada através de um metamodelo. Por metamodelo entende-se um mecanismo que reúne vários modelos em um único. Ou seja, um metamodelo é a descrição dos modelos e conceitos, trabalhados por uma ou mais metodologias, em um único modelo. Este pode ser construído utilizando uma notação qualquer. Sendo que todos os conceitos presentes no metamodelo poderão ser visualizados de várias formas diferentes. Por exemplo, o conceito classe pode ser visualizado segundo a metodologia de Booch [BOO 94], OMT de Rumbaugh [RUM 97], entre outras. Desta forma, várias representações diagramáticas, obedecendo uma notação, poderão ser geradas a partir dos conceitos presentes no metamodelo.

A UML é constituída por duas partes interdependentes: os conceitos e a notação gráfica, representativa destes conceitos. No documento, UML Semantics [RAS 97], estes conceitos são descritos através de um metamodelo. Este documento define toda a sintaxe através da notação do diagrama de classe da própria UML e a semântica é descrita em OCL e Linguagem Natural. Para melhor compreender sintaxe e semântica é necessário, primeiramente, verificar como estes conceitos são aplicados em engenharia de software. A sintaxe trata da simbologia e a seqüência em que estes

símbolos são aplicados. Ou seja, a sintaxe descreve "como fazer". A semântica trabalha com o que representa cada símbolo empregado, isto é, descreve significado do que está sendo feito. Para ilustrar estas definições pode ser utilizado um mapa das ruas de uma cidade. A sintaxe do mapa são as linhas, pontos e demais notações gráficas. A semântica descreve o significado destas linhas, pontos e demais símbolos.

O metamodelo da UML foi construído com base em MOF - *Meta Objects Facilities* [OMG 97], proposto pela OMG.

O MOF define um padrão para gerenciar metainformações em ambientes heterogêneos e distribuídos. Define, também, um padrão de interfaces para popular e consultar repositórios de modelos, que foram definidos usando várias linguagens de modelagem (por exemplo, a UML). MOF define o nível fundamental de uma arquitetura de quatro níveis de metamodelagem. Também é conceituado como um meta-metamodelo. Na TABELA 1 a seguir, são mostradas as definições de cada um destes quatro níveis da arquitetura MOF.

TABELA 3 – Apresentação dos Quatro Níveis da Arquitetura de Metamodelagem, Segundo OMG-MOF.

Nível	**Descrição**	**Exemplo**
Meta-Metamodel Nível MOF	É a infra-estrutura para uma arquitetura de metamodelagem, ou seja, a definição de uma linguagem para descrição de metamodelos.	MetaClasse, MetaAtributo, etc...
Metamodelo Nível OA&DF ou UML	Uma instância do meta-metamodelo, ou seja, a definição de uma linguagem para descrição de modelos.	Classe, atributo, etc...
Modelo Nível de Definições do Usuário	Uma instância do metamodelo, ou seja, a definição de uma linguagem para descrição das informações de um domínio.	Sensor, tipoSensor, leSensor(), Atuador, etc....
Objetos Nível de Dados do Usuário	Uma instância do modelo, ou seja, a definição das informações propriamente ditas.	<Sensor AX>, <sensorNível>, etc..

O metamodelo da UML é descrito em três partes distintas:

- Parte 1: na notação gráfica do Diagrama de Classes da própria UML.
- Parte 2: em linguagem natural (descrições em Inglês).
- Parte 3: em forma de regras OCL, que é uma linguagem semiformal [EVA 99a,b].

Os diagramas mostram as metaclasses conceituais e seus relacionamentos. As regras descrevem as restrições sobre as metaclasses, sobre os atributos das metaclasses e sobre os relacionamentos que foram especificados no metamodelo. Finalmente, as descrições em linguagem natural expõem a funcionalidade dos diagramas do metamodelo.

A descrição do metamodelo da UML - através de três formalismos: notação diagramática; textual e notação semiformal - evidenciou alguns benefícios. As ambigüidades e inconsistências foram reduzidas. A possibilidade de ocorrências de erros foi diminuída, pelo uso de técnicas complementares. A legibilidade das descrições foi mantida, constituindo, também, um benefício do metamodelo proposto.

Em contrapartida, o fato do metamodelo ser um modelo lógico, que pode ser implementado de formas diferentes, não se pode garantir que o repositório a ser construído com base nele agregará tais benefícios. Em ambientes distribuídos é complexo manter a consistência das informações, manipuladas pelas diversas ferramentas, que irão interoperar sobre este repositório.

2.5.1 Técnicas para Verificação de Consistência

Existem várias abordagens que podem ser aplicadas na verificação de consistência. Algumas destas apresentam um grau maior de formalismo. Outras, maior grau de expressividade. Quanto maior for sua expressividade maior as chances de incorrer em interpretações ambíguas. Porém quanto mais formal for a abordagem mais complexo é seu uso e compreensão. Para melhor entender a relação entre formalismo e expressividade são descritos a seguir alguns exemplos de abordagens para verificação de consistência:

- A abordagem baseada em métodos formais não permite ambigüidades nas interpretações, sendo eficiente na descrição do comportamento. Porém sua aplicação é complexa, exige uma base matemática por parte do desenvolvedor. Seu uso não é muito difundido e é propensa a erros [LAP 97].

- A abordagem baseada em regras consiste de um conjunto de regras que podem ser descritas em um formalismo qualquer. Seja através de linguagem natural ou uma linguagem formal [EVAa 99]. O problema na utilização do mecanismo de regras é a exigência de uma preocupação, no momento da construção das mesmas, de forma que estas não se contradigam.

  Na descrição das regras, se for utilizado linguagem natural, se terá as vantagens de que estas poderão ser descritas de forma mais clara e fácil. Porém implicará na desvantagem de que a linguagem natural é passível de interpretações ambíguas [LAP 97]. Se for utilizado formalismo matemático, para descrever estas regras, se terá as vantagens e inconvenientes, deste tipo de formalismo, os quais já foram citados anteriormente.

- A abordagem baseada nas técnicas da inteligência artificial para representação do conhecimento tem conotação formal. Pastor em [PAS 96], cita o *modelo ontológico* de Mário Bunge que foi trabalhado por Yair Wand e Ron Weber com o propósito de definição de um modelo formal para desenvolvimento de sistemas.

  Segundo Pastor, a Ontologia identifica e estuda os elementos de um domínio de maneira que um modelo ontológico pode ser descrito como um modelo formal para a descrição de modelos, ou seja, um metamodelo. Então, o modelo ontológico poderia ser usado para oferecer uma base formal para os conceitos envolvidos nas metodologias usadas no desenvolvimento de sistemas [PAS 96].

  A implicação do uso de modelos baseados na representação do conhecimento é complexidade da aplicação de formalismos matemáticos.

- Na abordagem baseada no conceito de metamodelo as vantagens e desvantagens vão estar atreladas ao formalismo que será utilizado para descrevê-lo.

Nas situações em que for utilizado um formalismo matemático incorre nas vantagens e desvantagens já mencionadas anteriormente.

Um metamodelo pode ser descrito utilizando a notação gráfica de uma metodologia qualquer, como, por exemplo, a UML ou OMT de Rumbaugh. Nesta situação, o metamodelo terá como vantagens a expressividade das representações gráficas e a facilidade na compreensão e utilização. Em contrapartida, as representações gráficas permitem interpretações ambíguas e a ocorrência de possíveis inconsistências.

Uma solução para o problema da ambigüidade e inconsistências seria a utilização de um formalismo adicional. Desse modo, além das representações gráficas, outro mecanismo poderia ser utilizado para descrever restrições sobre as mesmas.

Uma vez concluídas tais observações aos formalismos, o conceito de Metamodelo foi o escolhido para ser utilizado nesta dissertação. Utilizando o mecanismo de regras para descrever possíveis restrições aos elementos do modelo.

3 Modelagem dos Conceitos da Notação UML-RT

A UML é muito utilizada tanto no ambiente acadêmico quanto nos ambientes empresariais e industriais. É uma linguagem de modelagem gráfica, apta a modelar qualquer domíno de problema. Em setembro de 1997, foi adotada como um padrão, pela *OMG* (*Object Modeling Group's*), para especificar, construir e documentar *software*, sendo considerada como uma contribuição significativa nesta área. Neste trabalho optou-se por utilizar a linguagem de modelagem gráfica para tempo real UML-RT, em virtude destas características, listadas acima, e das facilidades que serão descritas a seguir.

O primeiro critério para escolher esta linguagem é o fato desta se tratar de uma proposta de utilização da UML para modelar sistemas de tempo real. Basicamente, são propostos elementos para modelar as particularidades dos sistemas de tempo real, empregando, para isso, o conceito de estereótipo que foi comentado na seção 2.4.

Os mecanismos de extensão, que são disponibilizados pela UML-RT, também influenciaram para que a mesma fosse escolhida. Tais mecanismos são conceitualmente aplicados através dos *tagged values*, *stereotypes* e comentários. Estes mecanismos possibilitam que todos os conceitos utilizados na modelagem possam ser estendidos, permitindo o emprego desta linguagem de modelagem na descrição de diferentes domínios.

Rememorando o conceito de metamodelo, também chamado de modelo de conceitos ou modelo conceitual (nomenclaturas que serão usadas no decorrer do trabalho), descrito anteriormente, é correto dizer que: um metamodelo é um modelo usado para descrever vários outros modelos os quais deseja-se construir. Ao ser projetado este modelo de conceitos, deve-se, primeiramente, adotar a notação na qual o mesmo será construído. A notação para descrever o metamodelo fica a critério do seu projetor, ou seja, de quem o contrói. Um exemplo para esta afirmação são os modelos gerados a partir dos diagramas de interação. Se fosse construído um metamodelo dos mesmos, este deveria descrever conceitos, tais como: Objeto, Mensagem, Interação e Operação.

Uma vez decido a notação na qual será feita a construção do metamodelo, é possível definir a maneira com que os conceitos presentes nos vários modelos serão estruturados no mesmo.

Neste trabalho optou-se pela notação do diagrama de classe da própria UML, descrito na seção 2.4. Este será utilizado para descrever os conceitos presentes nos seguintes diagramas da UML-RT: diagrama de classe, diagrama de objetos, diagrama de interação (seqüência e colaboração) e diagrama de estados.

Esta tarefa consistirá da modelagem destes diagramas através da representação dos conceitos presentes nos mesmos. Com os conceitos, serão modeladas as suas propriedades, a sua semântica e a sintaxe para construção destes diagramas.

Também foram mapeados para metamodelo a descrição dos elementos que modelam os requisitos temporais, tais como: tempo de execução, métodos periódicos, entre outros. Entende-se por mapeamento o ato de modelar tais elementos, ou seja, conceitos.

Existe alguns metamodelos que foram propostos em resposta ao OMG-AO&D-RFP-1 (*Object Management Group-Object Analysis and Design-Request for Proposal*). O metamodelo da UML proposto Booch, Jacobson e Rumbaugh juntamente com a OMG [RAT 97] e o metamodelo OOram Meta-Model [TAS 98] são alguns exemplos de submissão ao OMG-AO&D-RFP-1. Apesar destes metamodelos terem sido estudados e tidos como exemplo, não foram utilizados porque seu propósito é adverso ao desta pesquisa.

Estes metamodelos foram propostos com o objetivo de estabelecer uma arquitetura e um conjunto de especificações, baseado na tecnologia de objetos. Esta chamada de trabalhos, a OMG-AO&D-RFP-1 teve o intuito de proporcionar a integração de aplicações distribuídas de maneira que metas como reusabilidade, portabilidade e interoperabilidade de componentes de *software* em ambientes heterogêneos distribuídos pudessem ser alcançadas [TAS 98].

À contextualização da pesquisa feita no capítulo 2 juntamente com a verificação do estado-da-arte dos metamodelos foi possível projetar um modelo conceitual da linguagem de modelagem UML-RT como suporte à criação de um dicionário para ferramentas de modelagem de sistemas de tempo real, cujo objetivo é a verificação de consistência das descrições feitas por estas ferramentas.

Através do metamodelo proposto, é possível obter algumas constatações interessantes quanto à verificação de consistência. Tais como:

a) Um único conceito correspondente a diversas notações (gráficas e/ou textuais) em UML-RT terá uma única representação no metamodelo. Conseqüentemente, os diagramas que constituem as visões deste ou de mais conceitos serão consistentes pois estarão acessando o mesmo modelo de dados. Uma maneira de exemplificar esta afirmação é que uma atividade no diagrama de estados deve corresponder a uma definição de método, como modelado na seção 3.1.4.5. O fato dos conceitos terem uma única representação no metamodelo é uma forma de evitar inconsistências. Outros exemplos podem ser utilizados para ilustrar este tópico: um evento que dispara uma transição de estados pode estar associado à execução de um método. Conseqüentemente este método deverá estar definido na classe para a qual está sendo construído o diagrama de estados. Ou ainda, um mensagem enviada no diagrama de seqüência deve corresponder a um método na classe do objeto receptor da mensagem.

b) A geração automática completa ou de partes de um diagrama a partir de informações já modeladas em outros diagramas. Um exemplo de geração completa de diagrama são os diagramas de interação. Com o diagrama de seqüência modelado, o diagrama de colaboração pode ser gerado automaticamente. No caso de geração parcial de um diagrama, um exemplo que pode ser citado são os objetos modelados nos diagramas de colaboração e seqüência. Estes terão suas classes correspondentes no diagrama de classes. A mesma situação é observada com os métodos que são disparados pelas mensagens trocadas pelos objetos, nos diagramas de colaboração e seqüência. Estes também serão métodos que estarão modelados naquelas classes. Pode-se dizer ainda que se dois objetos trocam mensagens nos diagramas de interação isto significa que obrigatoriamente suas classes deverão estar relacionadas no diagrama de classes. Estas constatações são úteis para trabalhar com as preferências do usuário por notações diversas. Uma vez que os

dados já estejam modelados, a forma de visualizá-los irá depender das notações que o editor diagramático do usuário oferecer.

c) O modelo conceitual da notação UML-RT poderá ser utilizado para criar um Dicionário de Dados (DD) em um Sistema Gerenciador de Banco de Dados (SGBD). Este dicionário poderá funcionar como um repositório para os dados de ferramentas de modelagem de sistemas, independentes de serem tempo real ou não, FIGURA 3.1.

Nestas ferramentas estarão implementados mecanismos de manipulação da informações do DD, como os de inserir, remover, alterar e consultar dados. Dependendo da notação disponível no editor diagramático, estes poderão manipular quaisquer visões para dados e não somente as que foram mencionadas na FIGURA 3.2 (notações do diagrama de seqüência, de colaboração, de classes, de objetos e de estados).

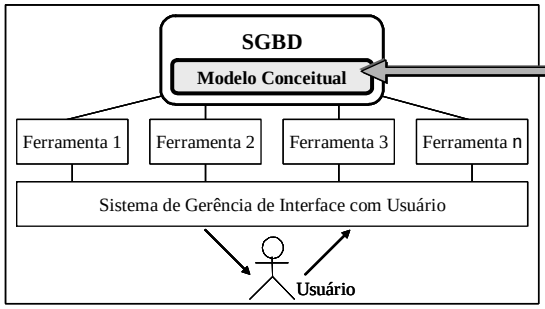


Figura a

FIGURA 3.1 – Ambiente de Desenvolvimento

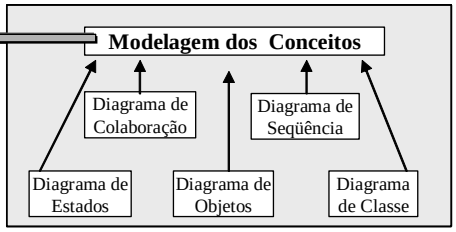


Figura b

FIGURA 3.2 – Diagramas que integram o modelo conceitual

Após os dados estarem presentes no banco de dados, a forma de visualizá-los fica a critério da equipe de desenvolvimento, que poderá inclusive escolher a notação que lhe favoreça a compreensão. Além do SiMOO-RT, o EDI[8] - Editor Diagramático na Internet [POM 98], também é um exemplo de ferramenta que pode utilizar o modelo conceitual proposto.

O modelo, que foi projetado na notação do diagrama de classes da UML e com a descrição semântica em OCL, será exposto detalhadamente no próximo tópico deste trabalho.

3.1 Descrição do Metamodelo

Muitos dos conceitos presentes na notação UML-RT foram modelados. A quantidade de conceitos tornou bastante extenso o metamodelo resultante. A fim de facilitar a compreensão da descrição do metamodelo, este foi estruturado utilizando a técnica de pacotes ('packages'). Pacotes, como abordados por [FOW 97, RAT 97],

[8] EDI é um editor de documentos visuais distribuídos de suporte ao projeto de sistemas de informação na Internet. Esse sistema é baseado em navegadores WWW utilizando sistema de banco de dados, tecnologia e funcionalidade de hiperdocumentos, e a programação é realizada utilizando a linguagem distribuída orientada a objetos Java, utilizando *frameworks* de aplicação e biblioteca de componentes [POM 98].

são mecanismos para a organização dos modelos em grupos, obedecendo uma lógica funcional.

Obedecendo esta notação, o metamodelo UML-RT foi estruturado em três pacotes (FIGURA 3.3): o pacote **Gerenciamento das Representações Gráficas**, o pacote **Gerenciamento da Implementação** e o pacote **Gerenciamento dos Modelos**.

O escopo deste trabalho tem o seu enfoque voltado para os conceitos presentes no pacote **Gerenciamento dos Modelos**. Apesar disto, o fato dos conceitos serem coesos, cria uma dependência lógica natural entre estes pacotes de maneira que haverá situações em que se fará necessário fazer abordagens aos demais.

Outro recurso utilizado para facilitar o entendimento do trabalho foi a subdivisão do pacote **Gerenciamento dos Modelos**. Ele foi expandido em três outros pacotes, FIGURA 3.3:

- **Modelagem Estática,** que descreve a notação para modelagem da estrutura estática dos sistemas.
- **Modelagem Comportamental,** que descreve as notações para a modelagem dinâmica,
- **Componentes da Modelagem,** que descreve conceitos comuns à estrutura estática e dinâmica.

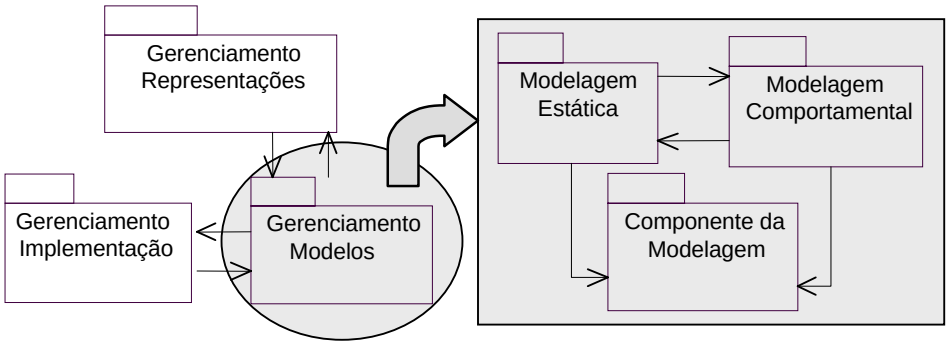


FIGURA 3.3 – Organização em pacotes do metamodelo UML-RT

Cada um destes três pacotes do **Gerenciamento dos Modelos,** acompanhados dos conceitos que foram utilizados para mapeá-los, serão descritos no próximo tópico do trabalho.

3.1.1 Observações ao Metamodelo Proposto

Conceitos semanticamente relacionados mapeados para um pacote podem ser representados por um ou mais diagramas de classe [RAT 97]. A opção por dividir os conceitos em diagramas separados é uma simples conveniência gráfica, não implicando em uma divisão ou particionamento do modelo em si.

O relacionamento de agregação por composição, abordado na seção 2.4, segundo a notação da UML em [RAT 97], pode ser representado através de duas formas:

- **Por atributo:** a classe composta especifica um atributo, cujo tipo é um componente. Como exemplo, vamos tomar Janelas (*Windows*) de um ambiente gráfico. A classe Janela especifica um atributo *título* que é do tipo Cabeçalho. Cabeçalho é um classe que especifica o cabeçalho da janela. Neste exemplo seria

título:Cabeçalho. Se o componente tivesse cardinalidade maior que 1, poderia ser especificado da seguinte forma: *título*[2]: Cabeçalho ou *título*[1..n]: Cabeçalho

Ao ser descrito o modelo conceitual da notação UML-RT, utilizou-se muito desta forma de definição de agregação. Isso melhorou a legibilidade do modelo, deixando-o mais conciso.

- **Por Relacionamento:** a classe composta estabelece o relacionamento com seus componentes através do adorno mostrado na FIGURA 3.4. A multiplicidade do Relacionamento deve ser sempre especificada com cardinalidade 1 para classe Composta. Para os seus componentes, a cardinalidade mínima deve ser 1 e máxima n (1..n ou numericamente especificada).

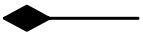


FIGURA 3.4 – Adorno Utilizado para Representar a Associação por Agregação

Foi convencionado para este trabalho a utilização do modelo e do diagrama com semânticas diferentes. Ao ser mencionada a palavra 'modelo', a referência está sendo feita aos conceitos. Já a palavra 'diagrama' faz menção às visões dos conceitos. Por exemplo: o Modelo de Classe agrega o conceito de classe, atributo, método, relacionamento, e outros, como mostra a FIGURA 3.8. Entretanto o Diagrama de Classe é uma visão para o Modelo de Classes. Assim, um Modelo de Classes pode ser visualizado como Diagrama de Classe na notação da metodologia de *Booch*, na notação *Shlaer&Mellor*, na notação *OMT* ou em qualquer notação que o editor diagramático utilizado disponibilizar.

Durante a descrição de como operar os modelos propostos, surgirão situações em que o relacionamento entre as classes não foi nomeado. Seguindo o padrão da OCL, [WAR 99], será utilizado o próprio nome da classe em minúsculo para representar o papel desempenhado na classe relacionada.

3.1.2 Modelos

A modelagem dos conceitos presentes da notação UML-RT foi mapeada para vários modelos. Estes modelos foram construídos com base na notação do diagrama de classe da UML. Cada classe do modelo representa um conceito. Por exemplo, o Modelo de Classe contém os conceitos da notação do Diagrama de Classe, FIGURA 3.5. Então, este modelo contém a classe Classe que modela o conceito de classe do diagrama de classe. Por sua vez, a classe Atributo modela o conceito de atributo, a classe Definição Método modela conceito de método, como mostra a FIGURA 3.8.

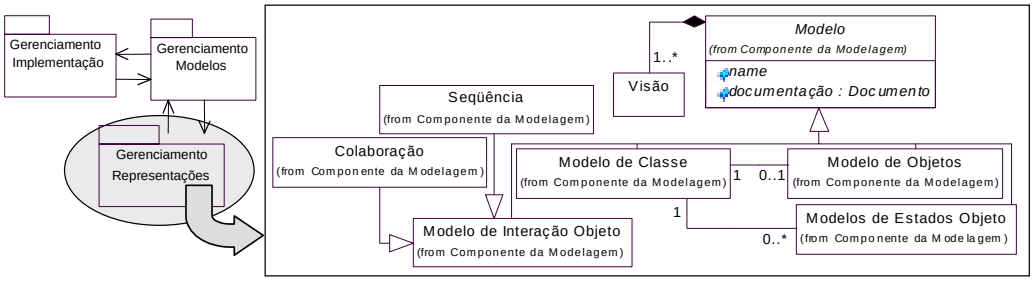


FIGURA 3.5 – Modelos dos Conceitos da Notação UML-RT

Cada um dos modelos especializados da classe Modelo poderão ser visualizados por várias notações. Por exemplo, os dados do Modelo de Classe poderão ser visualizados pelo Diagrama de Classe da notação da metodologia de BOOCH, Shlaer&Mellor, OMT e outras. Uma vez que as informações estejam disponíveis, a forma de visualizá-las dependerá da preferência do usuário e de seu editor.

Outros pontos que também devem ser observados estão relacionados ao dicionário de dados e ao editor diagramático. O primeiro tem que está preparado para armazenar as informações referentes à notação gráfica. Já o editor tem que estar preparado para controlar estas visões. Uma forma de implementar isso é através do padrão MVC - *Model View Controler*. Infelizmente esta abordagem foge do âmbito desta pesquisa, mas poderá ser consultado na bibliografia do Gamma [GAM 95].

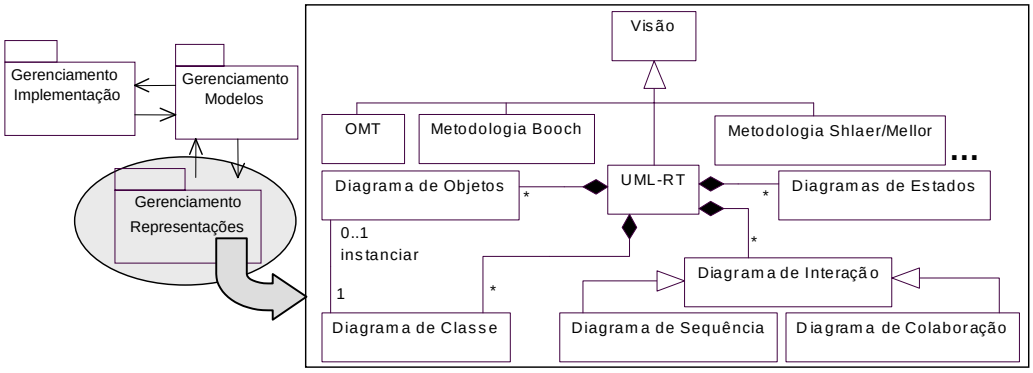


FIGURA 3.6 – Modelagem das Visões dos Conceitos da Notação UML-RT

As visões para os dados do dicionário de dados são gerenciadas pelo pacote **Gerenciamento das Representações Gráficas**. Embora os dados possam ser visualizados através de várias notações, será abordada a notação para linguagem de modelagem UML-RT. Na FIGURA 3.6 pode-se observar que a UML-RT agrega os diagramas de Classe, Objetos, Estados e Interação. Outros diagramas são propostos, mas não serão utilizados nesta pesquisa conforme está descrito na seção 2.4.

A modelagem da representação gráfica de cada um dos diagramas da UML-RT mostrados na FIGURA 3.6 é mostrada e descrita na seção Anexo1 .

3.1.2.1 Descrição dos Modelos dos Conceitos da Notação UML-RT

Diagramas que descrevem um sistema, seja de tempo real ou convencional, agregam um conjunto de conceitos e notações gráficas para os mesmos. No entanto, serão abordados somente os conceitos neste capítulo. Lembrando que foi adotado nesta pesquisa a referência ao diagrama como sendo uma visão para os conceitos e a referência ao modelo como sendo o conjunto de conceitos a ser visualizados por um diagrama.

Um modelo possui um nome e uma documentação, independente se é um modelo de classes, de objetos, de interação ou de estados. Cada um destes acrescentam descrições do sistema sob uma ótica diferenciada.

Na FIGURA 3.5 é mostrado o resultado da modelagem dos conceitos que são pertinentes a estes diagramas. Cada conceito é mapeado para uma classe. Cada característica destes conceitos é modelada como atributos destas classes. A semântica

destes conceitos é descrita primeiramente em linguagem natural e posteriormente formalizada em OCL.

A seguir, será exposto e detalhado cada conceito resultante do mapeamento dos diagramas da notação UML-RT presentes no modelo apresentado na FIGURA 3.5.

Modelo: Um modelo pode ser descrito como um conjunto de conceitos para descrever uma parte de um sistema.

Atributos:

nome: Nome para o modelo.

documentação: Descrição do modelo. Ver Documentação.

Semântica:

a) Um Modelo tem pelo menos uma visão associada.

b) Uma instância Modelo é um Modelo de Classe, um Modelo de Objetos, um Modelo de Estados Objeto ou é um Modelo de Interação Objeto.

Visão: Representação gráfica para os dados do Modelo.

Modelo de Classe: Descreve todas as classes com suas propriedades, comportamentos e relacionamentos pertencentes a um dado domínio sendo descrito.

Semântica:

a) Um Modelo de Classe é uma especialização de Modelo.

b) Um Modelo de Classe pode ou não estar associado a um Modelo de Objetos.

Modelo de Objetos: É uma instância do Modelo de Classe.

Semântica:

a) Um Modelo de Objetos é uma especialização de Modelo.

Modelo de Estados Objeto: Descreve os estados possíveis para uma classe de objetos num dado instante de tempo.

Semântica:

a) Um Modelo de Estados Objeto é uma especialização de Modelo.

b) Um Modelo de Classe pode ou não estar associado a um ou mais Modelo de Estados Objeto.

Modelo de Interação Objeto: Descreve a interação dos objetos na realização de uma determinada funcionalidade (execução) do sistema.

Semântica:

a) Um Modelo de Interação Objeto é uma especialização de Modelo.

b) Colaboração e Seqüência são especializações do Modelo de Interação Objeto.

3.1.2.2 Descrição das Restrições em OCL

Modelo:

a) Um Modelo tem pelo menos uma visão associada:

Modelo

```
self.visão -> size >=1.
```

b) Uma instância Modelo é um Modelo de Classe, um Modelo de Objetos, um Modelo de Estados Objeto ou um Modelo de Interação Objeto:

Modelo

```
self.allInstances -> select (oclType = Modelo) ->isEmpty.
```

Modelo de Classe:

a) Um Modelo de Classes é uma especialização de Modelo:

Modelo

```
self -> select(oclType = Modelo de Classe).
```

b) Um Modelo de Classe pode ou não estar associado a um Modelo de Objetos:

Modelo de Classe

```
self.modelo de objetos -> size <= 1.
```

Modelo de Objetos:

a) Um Modelo de Objetos é uma especialização de Modelo:

Modelo

```
self -> select(oclType = Modelo de Objetos).
```

Modelo de Estados Objeto:

a) Um Modelo de Estados Objeto é uma especialização de Modelo:

Modelo

```
self -> select(oclType = Modelo de Estados Objeto).
```

b) Um Modelo de Classe pode ou não estar associado a um ou mais Modelo de Estados Objeto:

Modelo de Classe

```
self.modelo de estados objetos -> size >= 0.
```

Modelo de Interação Objeto:

a) Um Modelo de Interação Objeto é uma especialização de Modelo:

Modelo

```
self -> select(oclType = Modelo de Interação Objeto).
```

b) Colaboração e Seqüência são especializações do Modelo de Interação Objeto:

Modelo de Interação Objeto

```
self -> select((oclType = Colaboração) or (oclType =
Seqüência)).
```

3.1.3 Modelagem Estática

No pacote denominado Modelagem Estática, FIGURA 3.7, foram mapeados os conceitos que fazem parte do diagrama de classes e do diagrama de objetos conforme as descrições da notação expostas no item 2.4.

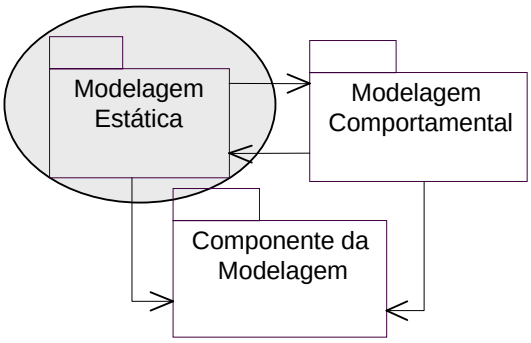


FIGURA 3.7 – Pacote Modelagem Estática.

No pacote Modelagem Estática estão modeladas as classes, os objetos, os métodos, os atributos e os relacionamentos, que foram mapeados para o diagrama classes **Modelagem do Diagrama de Classe e de Objetos**. O resultado deste mapeamento pode ser visualizado na FIGURA 3.8.

Estes conceitos agregam características, ou seja, atributos na notação de agregação que foi descrita na seção 3.1.1. Por exemplo, O conceito Terminador agrega o atributo *tipo* que é do tipo Tipo Relacionamento. Os conceitos Definição Métodos e Classe agregam o atributo *concorrência* que é do tipo Concorrência. Sendo que o conceito Definição Método agrega Execução Método por relacionamento e não por atributo, como é o caso das situações anteriores.

Para facilitar a compreensão e proporcionar maior clareza do modelo, estes conceitos foram mapeados em diagramas diferentes. O diagrama **Modelagem dos Métodos**, FIGURA 3.9, modela a execução e os requisitos temporais associados aos métodos. O diagrama **Modelagem dos Relacionamentos**, FIGURA 3.11, modela os tipos de relacionamento. Sendo que o diagrama **Modelagem da Concorrência**, FIGURA 3.10, modela concorrência expressada pelos métodos e pelas classes. Isto é possível, pois os conceitos modelados em um pacote podem ser representados por um ou mais diagramas de classe [RAT 97]. Vale a pena notar que a divisão proposta não implica no particionamento do modelo.

3.1.3.1 Descrição da Modelagem dos Conceitos do Diagrama de Classe e de Objetos

O diagrama de classes é, sem dúvida, um dos diagramas mais importantes descrito durante o desenvolvimento do sistema. É utilizado para descrever os elementos que compõem o domínio de uma aplicação. Este objetos ou elementos são mapeados para classes que agregam características, descritas através dos atributos, e

comportamentos, representados através dos métodos. As classes mantêm relacionamentos entre si e estes podem ser de tipos diferentes.

O diagrama de objetos é semelhante ao diagrama de classes, mostra os objetos e seus relacionamentos. Pode ser mencionado como uma instância do diagrama de classes. A notação dos diagramas de classe e de objetos é mostrada na FIGURA 2.2 da seção 2.4.

A seguir, estarão descritas todas as classes que compõem o metamodelo, FIGURA 3.8, que constituem o resultado do mapeamento destes diagramas. A semântica destes conceitos é descrita primeiramente em linguagem natural e posteriormente formalizada em OCL.

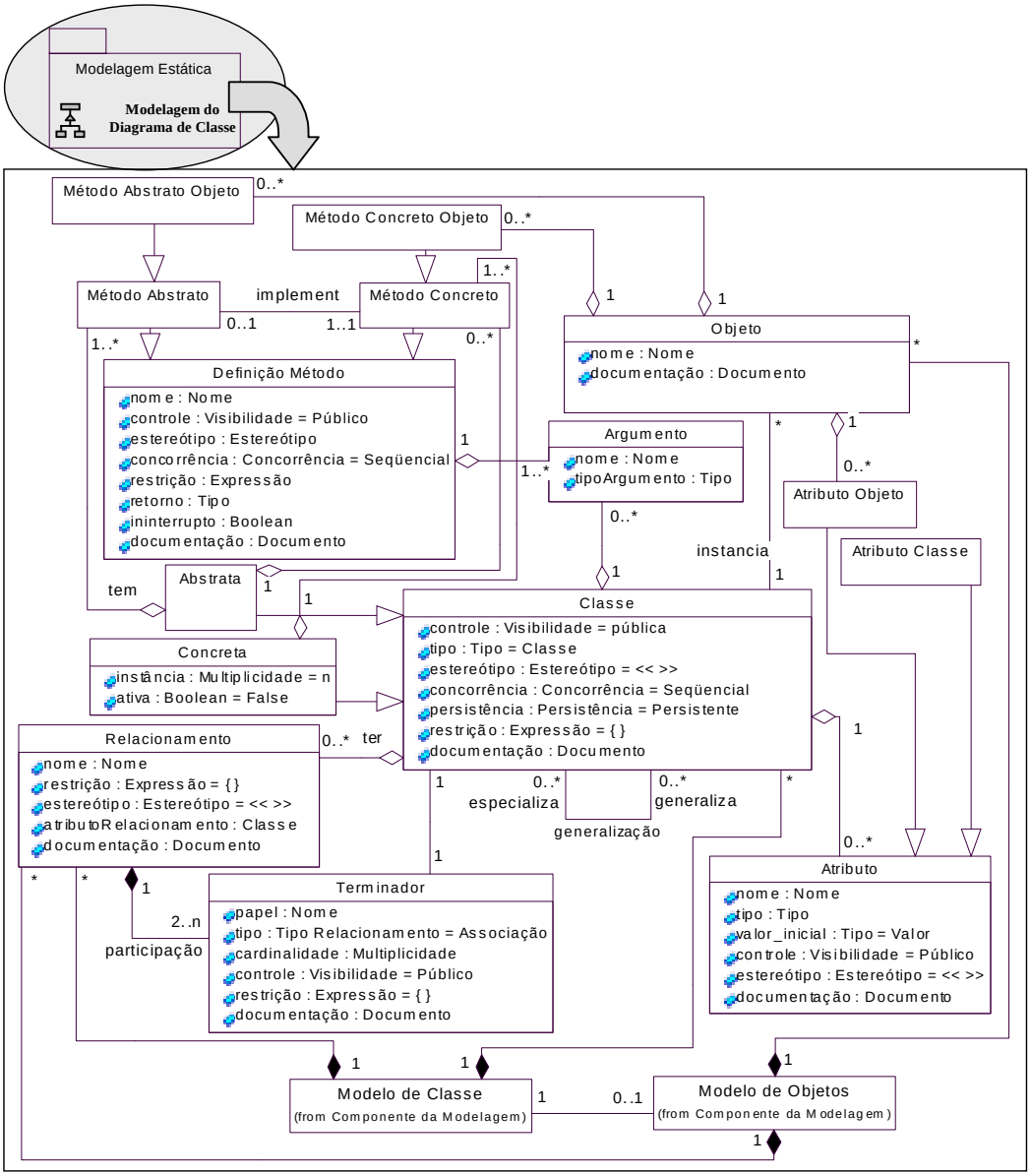


FIGURA 3.8 – Modelagem dos Conceitos do Diagrama de Classe e de Objetos.

Classe: É a representação de um conjunto de entidades do domínio do problema.

Atributos:

controle: Especifica a visibilidade da classe. Ver Visibilidade;

tipo: Tipo da Classe. Ver Tipo;

estereótipo: Mecanismo de extensão da classe. Ver Estereótipo;

concorrência: Especifica a semântica das chamadas concorrentes para as classes passivas (o atributo ativa=*false*). Ver Concorrência.

restrição: Possibilita a descrição semântica de restrições para a classe, obedecendo um formalismo qualquer. Ver Expressão;

documentação: Descrição da classe. Ver Documentação.

Semântica:

a) Uma Classe tem nome único dentro de um modelo.

b) Uma Classe não tem atributos repetidos.

c) Uma Classe não tem argumentos repetidos.

d) Uma Classe não tem relacionamentos iguais.

e) Somente as Classes Concretas podem ser instanciadas

f) Número de instâncias de uma classe é especificado pela multiplicidade.

g) Uma Classe pode generalizar e/ou especializar classes de objetos do domínio do problema. A forma ideal é permitir até quatro níveis de herança. Um maior número de níveis torna o modelo complexo demais.

Abstrata: É a representação de um conjunto de entidades abstratas do domínio do problema.

Semântica:

a) Uma classe Abstrata é uma Classe.

b) Uma classe Abstrata tem pelo menos um Método Abstrato definido.

c) Uma classe Abstrata pode ou não ter Métodos Concretos definidos.

Concreta: É uma Classe. Esta é uma representação de um conjunto de entidades do domínio do problema.

Atributos:

instância: Especifica o número de instâncias ou objetos da classe. Ver Multiplicidade;

ativa: Quando a classe for ativa (ativa=True), isto significa que a classe tem sua própria *thread* de controle.

Semântica:

a) Uma classe Concreta é uma Classe.

b) Uma classe Concreta tem pelo menos um Método Concreto definido.

Relacionamento: É a representação dos vínculos entre o conjunto de entidades do domínio do problema.

Atributos:

nome: Especifica o nome do relacionamento. Ver Nome;

restrição: Possibilita a descrição semântica de restrições para o relacionamento, obedecendo um formalismo qualquer. Ver Expressão;

estereótipo: Mecanismo de extensão do relacionamento. Ver Estereótipo;

atributoRelacionamento: Descreve os relacionamentos com atributos. Ver Classe;

documentação: Descrição do relacionamento. Ver Documentação.

Semântica:

a) Um Relacionamento tem no mínimo dois Terminadores.

Terminador: Um Terminador especifica cada classe conectada ao relacionamento, juntamente com um conjunto de propriedades que o torna válido. Geralmente, dois terminadores são especificados, pois os relacionamentos binários são ocorrência padrão nos modelos.

Atributos:

papel: Especifica o papel representado pela classe anexada ao terminador. O papel é opcional e, quando não especificado, é utilizado o nome da classe em minúsculo para representá-lo. Ver Nome;

tipo: São adornos que tem agregado uma representação semântica, ou seja, caracterizam o relacionamento. Ver Tipo Relacionamento;

cardinalidade: Especifica tamanho mínimo e máximo do conjunto de relações entre duas classes, ou seja, o conjunto de instâncias da classe que estão relacionadas. Ver Multiplicidade;

controle: Especifica a visibilidade do terminador. Ver Visibilidade;

restrição: Possibilita a descrição semântica de restrições para o terminador, obedecendo a um formalismo qualquer. Ver Expressão;

documentação: Descrição do terminador. Ver Documentação.

Semântica:

a) Um Terminador está associado a uma única classe.

b) Dois Terminadores de um mesmo relacionamento (Associação) poderão estar associados a uma mesma classe: Auto-Associação.

Atributo: Os atributos descrevem as características das classes do domínio do problema.

Atributos:

nome: Especifica o nome do atributo. Ver Nome;

tipo: Tipo do Atributo. Ver Tipo;

valorInicial: Valor inicial para o atributo. Ver Tipo;

controle: Especifica a visibilidade do atributo. Ver Visibilidade;

estereótipo: Mecanismo de extensão do relacionamento. Ver Estereótipo;

documentação: Descrição do terminador. Ver Documentação.

Semântica:

a) O valor inicial do atributo deve corresponder ao Tipo especificado pelo atributo *tipo*.

Atributo Classe: Especifica as característica da classe, ou seja, atributos da Classe.

Semântica:

a) Um Atributo Classe é uma especialização de Atributo.

Atributo Objeto: Especifica as característica das instâncias da classe, ou seja, atributos do Objeto.

Semântica:

a) Um Atributo Objeto é uma especialização de Atributo.

Argumento: Especifica uma lista de valores separados por vírgula que são utilizados por classes (Classe) e o método (Definição Método) como parâmetro. Cada argumento é uma variável com nome e tipo que é alterada com a passagem por valor/referência.

Atributos:

nome: Especifica o nome do Argumento. Ver Nome;

tipo: Tipo do Argumento. Ver Tipo;.

Objeto: Especifica as instâncias de uma Classe.

Atributos:

nome: Especifica o nome do Objeto. Ver Nome;

documentação: Descrição do Objeto. Ver Documentação.

Semântica:

a) Um Objeto pode ter Métodos Abstratos, Métodos Concretos ou ambos. Se forem definidos, estes métodos serão únicos para o objeto.

b) Um Objeto pode ou não ter atributos. Se forem definidos, estes atributos serão únicos para o objeto.

Definição Método: Definição Método e suas especializações estão descritas a seguir, na seção 3.1.3.3.

3.1.3.2 Descrição das Restrições em OCL

Classe:

a) Classes têm nome único dentro de um modelo;

```
Classe
Componentes da Modelagem::Modelo.allInstances implies
(self.allInstances -> forAll(p1, p2 | p1 <> p2 implies
```

p1.nome <> p2.nome)) -> asSet.

b) Uma Classe não tem atributos repetidos;

Classe

self.allInstaces -> exists(((self.atributo -> size >=0) -> asSet)).

c) Uma Classe não tem argumentos repetidos;

Classe

self.allInstaces -> exists(((self.argumento -> size >=0) -> asSet)).

d) Uma Classe não tem relacionamentos iguais;

Classe

self.allInstaces -> exists(((self.terminador -> size >=0) -> asSet)).

e) Somente a(s) Classe(s) Concreta(s) pode ser instanciada;

Abstrata

if self.oclIsKindOf(Abstrata) = true

then self.allInstances -> select (self. instancia) -> isEmpty

endif.

f) O número de instâncias de uma classe é especificado pela multiplicidade;

Classe

instancia -> size = Concreta.instância.

g) Uma Classe pode generalizar e/ou especializar classes de objetos do domínio do problema. Para não tornar o modelo muito complexo, o ideal é permitir até quatro níveis de herança; passando este limite, o modelo se torna complexo demais.

Classe

self.especializa -> size <= 4

Abstrata:

a) Um classe Abstrata é uma Classe;

Classe

self -> select(oclType = Abstrata).

b) Um classe Abstrata tem pelo menos um Método Abstrato definido;

Abstrata

self.allInstances -> select (self.tem) ->notEmpty.

h) Um classe Abstrata pode ou não ter Método(s) Concreto(s) definido(s).

Abstrata

self.allInstances -> select (self.método concreto) ->size >=0.

Concreta

a) Um classe Concreta é uma Classe;

Classe

self -> select(oclType = Concreta).

b) Um classe Concreta tem pelo menos um Método Concreto definido.

Concreta

self.allInstances -> select (self.método concreto) -> notEmpty.

Relacionamento:

a) Um Relacionamento tem no mínimo dois Terminadores.

Relacionamento

self.allInstances -> select (self.participação) -> size >= 2.

Terminador:

a) Um Terminador está associado a uma única classe;

Terminador

self.allInstances -> select (self.classe) -> size = 1.

b) Dois Terminadores de um mesmo relacionamento (Associação) que poderão estar associados a uma mesma classe: Auto-Associação.

Terminador

self -> select (self.classe-> exists(p1, p2 | p1 = p2 implies

p1.nome = p2.nome)) -> size = 2.

Atributo:

a) O valor inicial do atributo deve corresponder ao Tipo especificado pelo atributo *tipo*.

Atributo

self.tipo and self.valorInicial -> select (Componentes da Modelagem::Tipo) -> size = 1.

Atributo Classe:

a) Um Atributo Classe é uma especialização de Atributo.

Atributo

self -> select(oclType = Atributo Classe).

Atributo Objeto:

a) Um Atributo Objeto é uma especialização de Atributo.

Atributo

self -> select(oclType = Atributo Objeto).

Objeto:

a) Um Objeto pode ter Métodos Abstratos, Métodos Concretos ou ambos. Se forem definidos, estes métodos serão únicos para o objeto;

Objeto

self.allInstaces -> exists(((self.método abstrato objeto -> size >=0)

or (self.método concreto objeto -> size >=0)

or ((self.método abstrato objeto -> size >=0) and

(self.método concreto objeto -> size >=0))) -> asSet).

b) Um Objeto pode ou não ter atributos. Se forem definidos, estes atributos serão únicos para o objeto.

Objeto

self.allInstaces -> exists(((self.atributo objeto -> size >=0) -> asSet)).

3.1.3.3 Descrição da Modelagem da Execução dos Métodos

No desenvolvimento de sistemas de tempo real, a descrição dos requisitos temporais é um tópico de relevante importância. Conseqüentemente a maneira como este requisitos devem ser mapeados também é muito significativa.

Ao ser verificada qual a melhor forma de mapeamento dos requisitos temporais para o projeto deste metamodelo, chegou-se à conclusão que modelar o método e sua execução como conceitos separados constituía uma alternativa muito interessante. Isto porque a vantagem de se ter modelada a execução do método como conceito separado é que este poderá agregar notação gráfica e ser visualizado de forma distinta por diagramas diferentes - permite dar maior clareza às descrições e facilitar o desenvolvedor a validar os tempos estabelecidos. No anexo 1.1.2 é mostrada a modelagem da representação gráfica dos requisitos temporais.

Um método não necessariamente deve agregar uma descrição de execução. Há métodos em que o tempo de execução não tem relevância. Um exemplo que ilustra esta situação é um método que calcula salários.

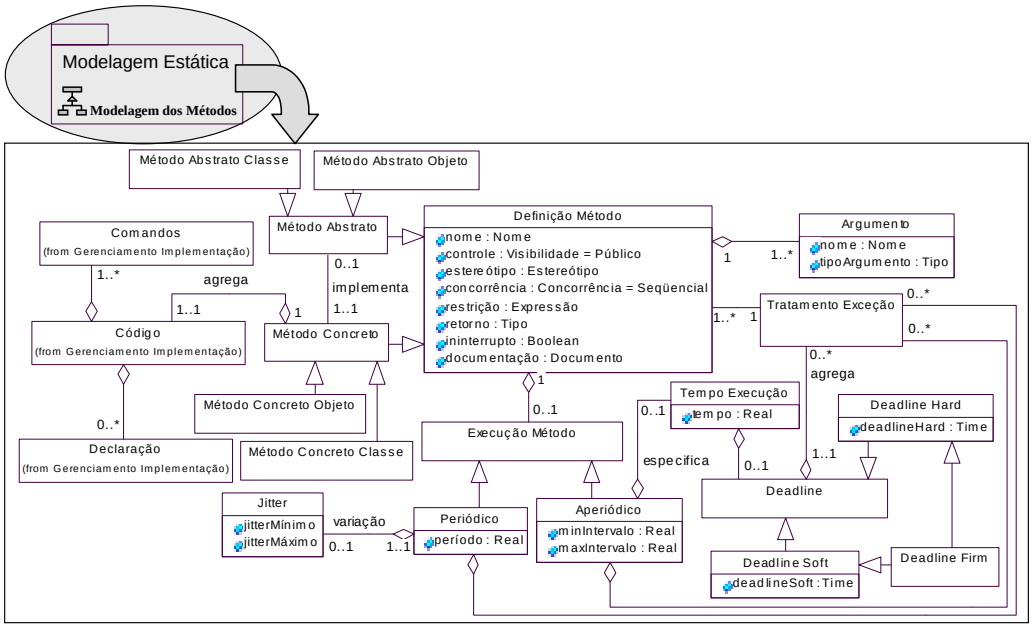


FIGURA 3.9 – Modelagem da Execução dos Métodos.

As classes de conceitos que compõem o modelo resultante deste mapeamento estarão todas descritas a seguir, como mostrado na FIGURA 3.9. Novamente a semântica dos conceitos é descrita primeiramente em linguagem natural e posteriormente formalizada em OCL

Definição Método: Especifica o comportamento, a funcionalidade de cada classe do domínio do problema.

Atributos:

nome: Especifica o nome do método. Ver Nome;

controle: Especifica a visibilidade do método. Ver Visibilidade;

estereótipo: Mecanismo de extensão do método. Ver Estereótipo;

concorrência: Especifica a semântica das chamadas concorrentes para as classes passivas (o atributo ativa=false). A concorrência de um método deve ser consistente com sua classe. Ver Concorrência;

restrição: Possibilita a descrição semântica de restrições para o método, obedecendo a um formalismo qualquer. Ver Expressão;

retorno: Descreve o tipo de retorno do método. Ver Tipo;

ininterrupto: É um atributo booleano que indica se um método é atômico ou não. Se o atributo for igual a *true*, indica que o método não pode ser interrompido. Neste caso, é um método atômico;

documentação: Descrição do método. Ver Documentação.

Semântica:

a) Um método pode ou não ter um conjunto de Argumento(s) único(s) especificado.

b) Um método pode ou não agregar uma especificação da execução do método.

Método Abstrato: Um método abstrato é a declaração de um método. Um método abstrato pode ser um método da Classe ou do Objeto.

Semântica:

a) Um Método Abstrato é uma especialização da Definição Método.

Método Abstrato Classe: Um Método Abstrato Classe especifica a declaração de um método de Classe.

Semântica:

a) Um Método Abstrato Classe é uma especialização de Método Abstrato.

Método Abstrato Objeto: Um Método Abstrato Objeto especifica a declaração de um método do objeto (ou instância da classe).

Semântica:

a) Um Método Abstrato Objeto é uma especialização de Método Abstrato.

Método Concreto: Um método concreto é a própria implementação de um método. Agrega código com comandos e declarações. Pode ser a implementação de um método abstrato.

Semântica:

a) Um Método Concreto é uma especialização da Definição Método.

b) Um Método Concreto pode ser a implementação de um Método Abstrato.

c) Um Método Concreto obrigatoriamente agrega código.

Método Concreto Classe: Um Método Concreto Classe especifica a implementação de um método de Classe.

Semântica:

a) Um Método Concreto Classe é uma especialização de Método Concreto.

Método Concreto Objeto: Um Método Concreto Objeto especifica a implementação de um método do objeto ou instância da classe.

Semântica:

a) Um Método Concreto Objeto é uma especialização de Método Concreto.

Código, Comando e Declaração: A especificação da implementação de um método é realizada através de um conjunto de declarações e comandos, obedecendo à sintaxe de uma determinada linguagem.

Semântica:

a) O Código de um método é um conjunto de Comandos (pelo menos um), que pode ou não conter Declarações.

Execução Método: Especifica a execução de um método. A execução do método agrega um adorno. Esta representação gráfica, ou adorno, para execução do método está modelada na FIGURA 1.5 – Modelagem da Representação Gráfica do Diagrama de Classes no anexo 1.1.2.

Atributos:

adorno: Figura geométrica, TABELA 2, que caracteriza a execução do método. Ver Figura Geométrica.

Semântica:

a) Os métodos que declararem uma execução do tipo Periódica ou Aperiódica tem um adorno agregado.

Aperiódico: Especifica uma forma de execução de um método. Aperiódico é a execução do método, cujo intervalo entre as execuções não pode ser prognosticado. Entretanto, um tempo mínimo e um tempo máximo pode ser apontado. Caso estes intervalos sejam ultrapassados, significa que há possibilidade de um grupo de requisição (mensagem) estar solicitando a execução do método. Nestas situações, um tratamento de exceção deve ser aplicado .

Atributos:

minIntervalo: É a especificação de um tempo mínimo para a chegada de uma requisição de execução do método;

maxIntervalo: É a especificação de um tempo máximo para a chegada de uma requisição de execução do método.

Semântica:

a) A execução aperiódica pode ou não ter um Tempo Execução especificado.

b) O atributo *minIntervalo* deve ser menor que o *maxIntervalo*.

c) Tratamentos de exceção deverão ser especificados para as situações em que os intervalos de tempo mínimo e tempo máximo entre as requisições de execução do método não forem cumpridos.

Periódico: Também especifica uma forma de execução de um método. O método que agrega uma execução periódica é caracterizado pela definição de um período, dentro do qual o método deve ocorrer, e pelo *jitter*, que é a variação em torno do período.

Atributos:

período: É a especificação de um período dentro do qual as execuções do método devem ocorrer.

Semântica:

a) A execução periódica pode ou não ter uma especificação de variação do *jitter*.

b) Tratamentos de exceção deverão ser especificados para as situações em que a especificação dos limites do *jitter* não forem cumpridos.

Jitter: Especifica a variação em torno do período estabelecido para execução do método. É a diferença entre o tempo em que foi executado e o tempo em que deveria ser executado.

Atributos:

jitterMínimo: É a especificação de um limite mínimo para a variação;

jitterMáximo: É a especificação de um limite máximo para a variação.

Semântica:

a) O atributo *jitterMínimo* deve ser menor que o atributo *jitterMáximo*.

Tempo Execução: Especifica o tempo em que a execução de um método deve ser completada.

Atributos:

tempo: É a especificação do tempo de execução do método.

Semântica:

a) O tempo de execução pode ou não agregar um *deadline*.

Deadline: Especifica um limite de tempo máximo dentro do qual a execução do método deve ser completada.

Semântica:

a) Se o *deadline* estabelecido não for cumprido, poderá ser requisitada ou não a execução de procedimentos de tratamento de exceção.

Deadline Soft: Especifica um limite de tempo máximo dentro do qual a execução do método deve ser completada. Porém, para *deadlines soft*, seção 2.1, as respostas das execuções fora do tempo determinado costumam ser aceitáveis.

Atributos:

deadlineSoft: É a especificação do limite de tempo máximo para a execução completa do método.

Semântica:

a) *Deadline Soft* é um especialização do *deadline*.

Deadline Hard: Especifica um limite de tempo máximo dentro do qual a execução do método deve ser completada. Entretanto, para *deadlines hard*, seção 2.1, as respostas das execuções fora do tempo determinado não são aceitáveis, e são consideradas falha do sistema.

Atributos:

deadlineHard: É a especificação do limite de tempo máximo para a execução completa do método.

Semântica:

a) *Deadline Hard* é um especialização do *deadline*.

b) Sempre que o *deadline* especificado for *Hard*, fica explicitada a obrigatoriedade de tratamento de exceção.

Deadline Firm: . Para *deadlines firm*, seção 2.1, tem-se uma combinação dos *deadline soft* e *hard*. Isto representa que as respostas das execuções fora tempo determinado, apenas por um curto período, são aceitáveis.

Semântica:

a) Sempre que o *deadline* especificado for *Firm*, fica explicitada a obrigatoriedade de tratamento de exceção;

b) *Deadline Firm* é um especialização de *Deadline Soft* e de *Deadline Hard*.

Tratamento Exceção: É a especificação de um ou mais métodos que manterão a integridade do sistema diante das situações em que o *deadline* estabelecido não for cumprido.

Semântica:

a) Para cada instância da classe Tratamento Exceção pelo menos uma instância da classe Definição Método deve estar associada.

3.1.3.4 Descrição das Restrições em OCL

Definição Método:

a) Um método pode ou não ter um conjunto de Argumentos únicos especificados;

Definição Método

```
self.allInstances –> existes (((self.argumento –> size >=0) –> asSet)).
```

b) Um método pode ou não agregar uma especificação da execução do método.

Definição Método

```
(self.execução método –> size =0) or (self.execução método –> size = 1)
```

Método Abstrato:

a) Um Método Abstrato é uma especialização da Definição Método.

Definição Método

```
self -> select(oclType = Método Abstrato).
```

Método Abstrato Classe:

a) Um Método Abstrato Classe é uma especialização de Método Abstrato.

Método Abstrato

```
self -> select(oclType = Método Abstrato Classe).
```

Método Abstrato Objeto:

a) Um Método Abstrato Objeto é uma especialização de Método Abstrato.

Método Abstrato

```
self -> select(oclType = Método Abstrato Objeto).
```

Método Concreto:

a) Um Método Concreto é uma especialização da Definição Método;

Definição Método

```
self -> select(oclType = Método Concreto ).
```

b) Um Método Concreto pode ser a implementação de um Método Abstrato;

Método Concreto

self.implementa -> size = 1 imples self.nome = Método Abstrato.nome.

c) Um Método Concreto obrigatoriamente agrega código.

Método Concreto

self.allInstances -> exists(self.agrega-> size = 1).

Método Concreto Classe:

a) Um Método Concreto Classe é uma especialização de Método Concreto.

Método Concreto

self -> select(oclType = Método Concreto Classe).

Método Concreto Objeto:

a) Um Método Concreto Objeto é uma especialização de Método Concreto.

Método Concreto

self -> select(oclType = Método Concreto Objeto).

Código, Comando, Declaração:

a) Código de um método é um conjunto de, pelo menos um, Comando, podendo ou não conter Declarações.

Código

self -> select((self.comando-> size >= 1) and (self.declaração -> size >= 0)) -> notEmpty.

Execução Método:

a) Os métodos que declararem uma execução do tipo Periódica ou Aperiódica tem um adorno agregado.

Definição Método

(self.execução método –> forAll(oclIsKindOf(Periódico)) –> exists (Gerenciamento das Representações Gráficas:: F_Execução Método –> size = 1)) or

(self.execução método –> forAll(oclIsKindOf(Aperiódico))) exists (Gerenciamento das Representações Gráficas:: F_Execução Método –> size = 1))

Aperiódico:

a) A execução aperiódica pode ou não ter um Tempo Execução especificado;

Aperiódico

(self.especifica –> size = 1) or (self.especifica –> size = 0).

b) O atributo *minIntervalo* deve ser menor que o *maxIntervalo*;

Aperiódico

```
self.minIntervalo < self.maxIntervalo.
```

c) Tratamentos de exceção deverão ser especificados para as situações em que os intervalos de tempo mínimo e máximo entre as requisições de execução do método não forem cumpridos.

Aperiódico

```
self.minIntervalo < (r:Real): Boolean
post: if self.minIntervalo < r then
        self.tratamento exceção –> size >= 1
      endif
or
self.maxIntervalo <(r:Real): Boolean
post: if self.maxIntervalo < r then
        self.tratamento exceção –> size >= 1
      endif.
```

Periódico:

a) A execução periódica pode ou não ter uma especificação de variação do *jitter*.

Periódico

```
(self.variação –> size = 1) or (self.variação –> size = 0).
```

b) Tratamentos de exceção deverão ser especificados para as situações em que a especificação dos limites do *jitter* não forem cumpridos.

Periódico

```
if self.variação –> size = 1 then
  self.variação.jitterMínimo < (r:Real): Boolean
  post: if self.variação.jitterMínimo < r then
          self.tratamento exceção –> size >= 1
        endif
  or
  self.variação.jitterMáximo < (r:Real): Boolean
  post: if self.variação.jitterMáximo < r then
          self.tratamento exceção –> size >= 1
        endif
endif.
```

Jitter:

a) atributo *jitterMínimo* deve ser menor que o atributo *jitterMáximo*.

Jitter

```
self.jitterMínimo < self.jitterMáximo.
```

Tempo Execução:

a) O tempo de execução pode ou não agregar um limite máximo (*deadline*) para que a execução do método se complete.

Tempo Execução

```
(self.deadline –> size = 1) or (self.deadline –> size = 0).
```

Deadline:

a) Se o *deadline* estabelecido não for cumprido, poderá ser ou não requisitada a execução de procedimentos de tratamento de exceção.

Deadline

```
  self.Deadline Soft.deadlinesoft < (t:Time): Boolean
  post: if self.Deadline Soft.deadlinesoft < t then
          self.agrega –> size >= 1
        endif
  or
  self.Deadline Hard.deadlinehard <= (t:Time): Boolean
  post: if self.Deadline hard.deadlinehard <= t then
          self.agrega –> size >= 1
        endif.
```

Deadline Soft:

a) *Deadline Soft* é um especialização do *deadline.*

Deadline

```
self -> select(oclType = Deadline Soft).
```

Deadline Hard:

a) *Deadline Hard* é um especialização do *deadline.*

Deadline

```
self -> select(oclType = Deadline Hard).
```

b) Sempre que o *deadline* especificado for *Hard,* fica explicitada a obrigatoriedade de tratamento de exceção.

Deadline Hard

```
if self.oclIsKindOf(Deadline Hard) = true then
  self.Deadline.agrega –> size >= 1
endif.
```

Deadline Firm:

a) Sempre que o *deadline* especificado for *Firm*, fica explicitada a obrigatoriedade de tratamento de exceção.

```
Deadline Firm
if self.oclIsKindOf(Deadline Firm) = true then
  self.Deadline.agrega –> size >= 1
endif.
```

b) *Deadline Firm* é um especialização de *Deadline Soft* e de *Deadline Hard*.

```
Deadline Firm
self.allInstances -> select((oclType = Deadline Soft) and
(oclType = Deadline Hard)) -> isEmpty.
```

Tratamento Exceção:

a) Para cada instância da classe Tratamento Exceção pelo menos uma instância da classe Definição Método deve estar associada.

```
Tratamento Exceção
self.allInstaces -> exists(self.definição método -> size >=1).
```

3.1.3.5 Descrição da Modelagem da Concorrência

O conceito controle da concorrência, modelado pelas classes Definição Método e Classe, é mostrado na FIGURA 3.10. Estas classes conceito, vistas na FIGURA 3.8, implementam um atributo chamado *concorrência* que também é do tipo Concorrência.

Concorrência é um conceito cujo resultado do seu mapeamento é mostrado na FIGURA 3.10 e é descrito a seguir. Logo em seguida, a semântica deste conceito é descrita primeiramente em linguagem natural e posteriormente é formalizada na linguagem OCL

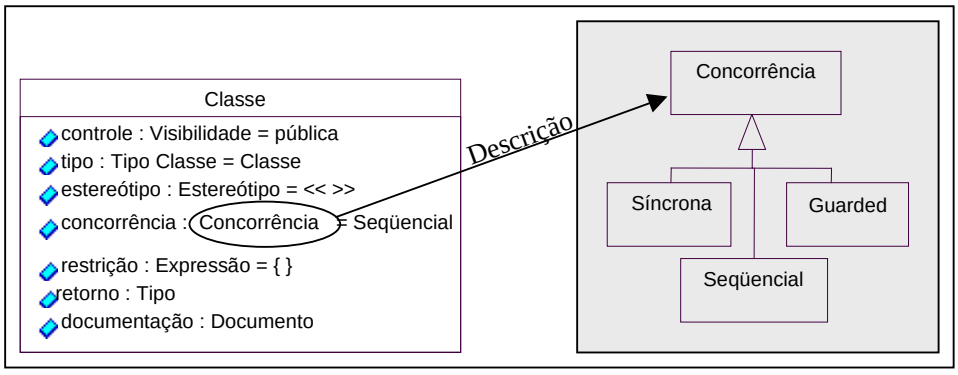


FIGURA 3.10 – Modelagem da Concorrência.

Concorrência: A concorrência é o campo que especifica a semântica na presença de múltiplas *threads* de controle. Sua modelagem mostra a concorrência para os elementos de uma classe; conseqüentemente, as operações devem ser consistentes

com a concorrência da sua classe. Quando a classe for ativa (ativa=*True*), significa que a mesma tem sua própria *thread* de controle.

Semântica:

a) Uma instância de Concorrência é Síncrona, ou é Seqüencial, ou é *Guarded*.

Síncrona: A semântica da classe é garantida na presença de múltiplas *threads* de controle. A exclusão mútua é fornecida pela própria classe.

Semântica:

a) Concorrência Síncrona é uma especialização de Concorrência.

Seqüencial: É a especificação padrão ou *defaut*. A semântica da classe é garantida na presença de uma simples *thread* de controle. Isto significa que somente uma thread de controle pode estar sendo executada a qualquer momento.

Semântica:

a) Concorrência Seqüencial é uma especialização de Concorrência.

Guarded: A semântica da classe é garantida na presença de múltiplas *threads* de controle. Uma classe com esta especificação de concorrência requer colaboração entre *threads* clientes para garantir a exclusão mútua.

Semântica:

a) Concorrência *Guarded* é uma especialização de Concorrência.

3.1.3.6 Descrição das Restrições em OCL

Concorrência:

a) Uma instância de Concorrência é Síncrona, Seqüencial ou *Guarded*.

Concorrência

```
self.allInstances -> select (oclType = Concorrência) ->isEmpty.
```

Síncrona:

a) Concorrência Síncrona é uma especialização de Concorrência.

Concorrência

```
self -> select(oclType = Síncrona).
```

Seqüencial:

a) Concorrência Seqüencial é uma especialização de Concorrência.

Concorrência

```
self -> select(oclType = Seqüencial).
```

Guarded:

a) Concorrência *Guarded* é uma especialização de Concorrência.

Concorrência

```
self -> select(oclType = Guarded ).
```

3.1.3.7 Descrição da Modelagem dos Tipos de Relacionamentos

O conceito que caracteriza a forma que duas classes estão relacionadas é descrito pela classe Terminador, FIGURA 3.8. Esta classe implementa um atributo chamado *tipo* que é do tipo Tipo Relacionamento.

O conceito Tipo Relacionamento foi modelado e o resultado desta modelagem é mostrado na FIGURA 3.11 e descrito a seguir. Sua semântica é mostrada em seguida em linguagem natural e posteriormente formalizada em OCL.

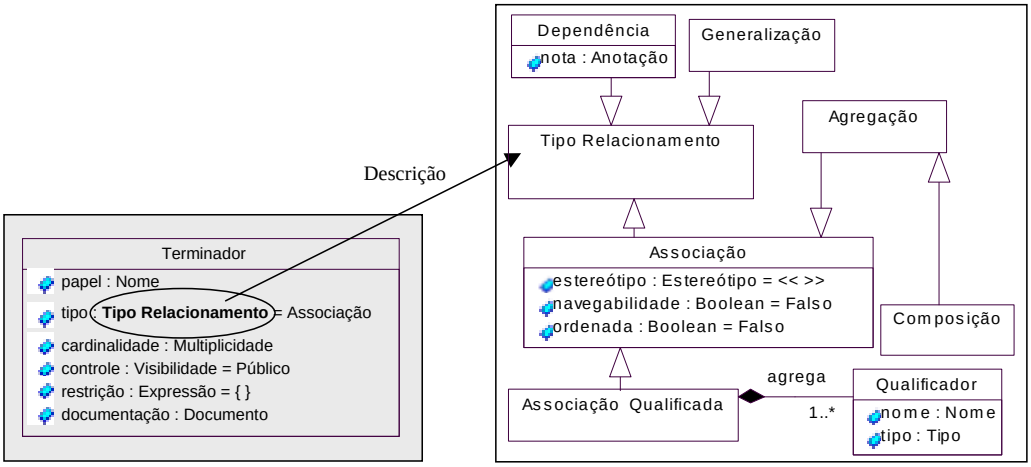


FIGURA 3.11 – Modelagem dos Tipos de Relacionamentos.

Tipo Relacionamento: Especifica os tipos de relacionamentos possíveis entre as classes de um modelo.

Semântica:

a) Um relacionamento pode ser de Associação, Dependência ou Generalização.

Generalização: É a especificação do relacionamento entre classes abstratas e concretas. É indicada a modelagem deste relacionamento quando uma classe consegue abstrair características e comportamentos semelhantes a um conjunto de outras classes. Este tipo de relacionamento tem um papel importante para os modelos pois, além proporcionar maior clareza, pode facilitar o reuso.

A Generalização Apresenta algumas restrições que agregam semântica diferenciada ao relacionamento:

- {completo}: Especifica os relacionamentos de generalização cujas classes concretas foram todas descritas. Nenhuma outra especificação de classe concreta é esperada. Para que um relacionamento de generalização possa ser considerado 'completo' pelo menos duas classes concretas devem ser especificadas.
- {incompleto}: Especifica os relacionamentos de generalização cujas classes concretas não foram todas descritas, por serem conjuntos desconhecidos.
- {disjunção}: A instanciação de uma classe concreta não implica na instanciação das demais que participam do relacionamento de generalização.

- {sobreposição}: Observando a FIGURA 3.12, na instanciação de K, as classes A, B e C são instanciadas simultaneamente, caracterizando sobreposição.

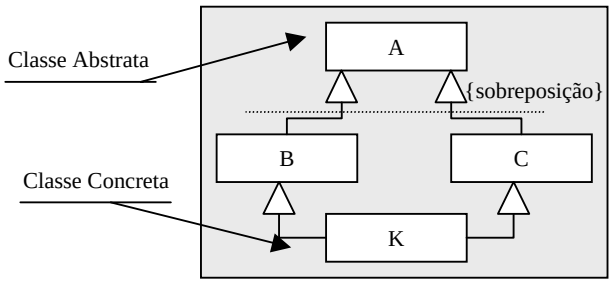


FIGURA 3.12 – Exemplo de Restrição Pré-Definida na Generalização.

Estas restrições são representadas pelo atributo *restrição* do Terminador.

Semântica:

a) Um relacionamento pode ser de Generalização.

b) Se especificada a restrição {completo}, o relacionamento de Generalização deverá ter no mínimo duas subclasses.

c) Uma classe concreta, ao ser instanciada, herda todos os atributos e métodos da classe abstrata.

d) O conjunto de atributos herdados mais os atributos da classe concreta instanciada devem ser únicos (uso ou não do conceito de polimorfismo).

e) O conjunto de método herdados mais os métodos da classe concreta instanciada devem ser únicos (uso ou não do conceito de polimorfismo).

Dependência: Especifica os relacionamentos entre as classes que mantêm dependência semântica entre elas. Normalmente utilizada para representar a dependência de um classe com sua interface. Um relacionamento de Dependência entre duas classes A (cliente) e B (fornece) caracteriza as seguintes situações:

- a classe A acessa um valor, podendo ser uma constante ou variável, da classe B;
- métodos presentes na classe A invocam métodos da classe B;
- a classe A tem definições de métodos cujo retorno ou argumentos são instâncias da classe B.

Outros mecanismos adicionais podem ser utilizados para explicitar as dependências, como os estereótipos e uma descrição em forma de 'nota' anexada ao relacionamento de dependência.

Atributos:

nota: É a especificação de uma anotação anexada ao relacionamento de Dependência. Esta nota expressa uma descrição sobre a forma de dependência entre as classes.

Semântica:

a) Um relacionamento pode ser de Dependência.

Associação: Especifica o relacionamento entre duas ou mais classes. Duas classes, A e B, mantêm um relacionamento quando uma instância de A associa-se a uma ou mais instâncias de B.

Atributos:

estereótipo: Mecanismo de extensão do relacionamento por Associação. Ver Estereótipo;

navegabilidade: Navegabilidade é um atributo booleano que caracteriza o sentido do conhecimento no relacionamento, através da presença do adorno, como mostra a FIGURA 3.13, no terminador. Quando o atributo for especificado como *false*, indicará que o conhecimento será feito em ambas as direções (bidirecional). Conseqüentemente, se for especificado como *true*, será unidirecional. Por exemplo, em um modelo contendo as classe A (origem) e B (destino) com a especificação de uma associação navegável de A para B, então A tem a responsabilidade de conhecer as instâncias de B. O contrário não é possível. Normalmente utiliza-se na análise a associação bidirecional. A unidirecional é mais comum na descrição do projeto.

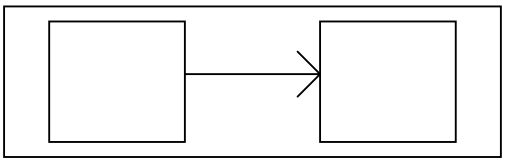


FIGURA 3.13 – Adorno do Relacionamento de Navegabilidade.

ordenada: É um atributo booleano que caracteriza se o conjunto instâncias relacionadas são ordenadas ou não. A este atributo pode ser conferido o valor *true* somente quando a multiplicidade do relacionamento for maior que um. A indicação de Associação Ordenada é dada pela palavra ordenado entre chaves, {ordenado}.

Semântica:

a) Associação é uma especialização de Tipo Relacionamento.

b) Uma Associação poderá ser ordenada somente se a multiplicidade do relacionamento for maior que um.

Agregação: A associação por agregação é também chamada todo/parte. A especificação deste tipo de relacionamento se dá quando se está descrevendo a classe todo e suas partes. Um exemplo clássico para elucidar agregação são as classes Pedido e Item Pedido. Nesta situação, um Pedido é um agregado de Item(s) Pedido(s). Na agregação subentende-se que as instâncias agregadas só são instanciadas após a instância agregadora ter sido criada.

Semântica:

a) Agregação é uma especialização de Associação.

b) Em um relacionamento de Associação por Agregação, somente um Terminador poderá ser do tipo Agregação.

c) Em um relacionamento de Associação por Agregação, a Multiplicidade Terminador da Agregação deve ser igual a um.

Composição: É um tipo de agregação onde a persistência da instância 'parte' é coincidente com a persistência da instância 'todo', FIGURA 3.14. Uma vez estabelecido, o relacionamento da instância agregadora com as instâncias agregadas não poderá mais ser alterado. As duas formas de apresentar composição apresentadas na FIGURA 3.14 são semanticamente equivalentes.

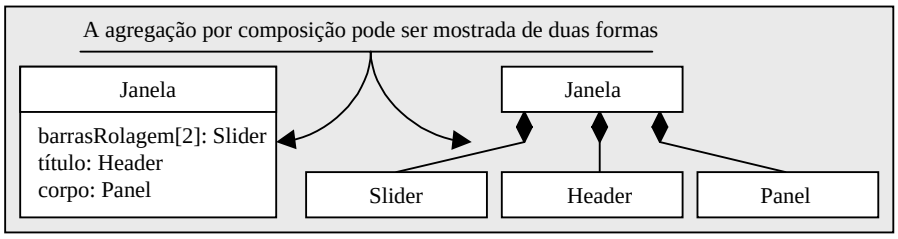


FIGURA 3.14 – Formas Diferentes de Modelar Agregação por Composição.

Semântica:

a) Composição é uma especialização de Agregação.

b) Em um relacionamento de Associação por Composição, somente um Terminador poderá ser do tipo Composição.

c) Em um relacionamento de Associação por Composição, a Multiplicidade Terminador da Composição deve ser igual a um.

Associação Qualificada: A especificação de um qualificador funciona de modo semelhante a uma chave que facilita o gerenciamento no acesso de determinadas instâncias. Supondo as classes A (qualificada) e B, a multiplicidade da associação qualificada é quem vai ditar o número de instâncias da classe B que estão relacionadas a uma instância de A acrescido do valor do qualificador.

Semântica:

a) Associação Qualificada é uma especialização de Associação.

b) Para ser uma Associação Qualificada deverá ser especificado pelo menos um qualificador.

c) Uma Associação Qualificada poderá ser ordenada somente se a multiplicidade do relacionamento for maior que um.

d) Se a multiplicidade de uma Associação Qualificada for '0..1', o valor do qualificador mais as instâncias poderão ou não acessar uma única instância da classe associada.

e) Se a multiplicidade de uma Associação Qualificada for '1', o valor do qualificador mais as instâncias acessarão uma única instância da classe associada.

f) Se a multiplicidade de uma Associação Qualificada for '*', o valor do qualificador mais as instâncias acessarão uma conjunto de instâncias da classe associada.

Qualificador: Especifica o atributo de qualificação nos relacionamentos por Associação Qualificada. O atributo de qualificação funciona como uma chave para acessar as instâncias da classe associada. Segundo descrito no *Notation Guide* [RAT 97], a cardinalidade anexada na classe associada pode ser '0..1', '1' ou '*'. Se a

multiplicidade de uma Associação Qualificada for '0..1', o valor do qualificador seleciona uma instância da classe associada. Os possíveis valores do qualificador poderão ou não acessar uma única instância da classe associada. No caso da multiplicidade de uma Associação Qualificada na classe destino for '1', o valor do qualificador identifica uma única instância nesta classe destino. Porém o conjunto dos valores para o qualificador deve ser finito. E se a multiplicidade de uma Associação Qualificada na classe destino for '*', o valor do qualificador é um índice que particiona as instâncias da classe associada em subconjuntos.

Por exemplo, em uma classe Pedido que implementa um qualificador, o *produto*, está associado à classe Item Pedido. A cardinalidade em Item Pedido é '0..1'. Então, para acessar um objeto Item Pedido é necessário passar o produto como argumento. O relacionamento também indica que não há dois Item Pedido para um mesmo produto.

Atributos:

nome: Especifica o nome do Qualificador. Ver Nome;

tipo: Tipo do Qualificador. Ver Tipo;

3.1.3.8 Descrição das Restrições em OCL

Tipo Relacionamento:

a) Um relacionamento pode ser de Associação, Dependência ou Generalização.

Tipo Relacionamento

```
self.allInstances -> select (oclType = Tipo Relacionamento) ->
isEmpty.
```

Generalização:

a) Um relacionamento pode ser de Generalização.

Tipo Relacionamento

```
self -> select(oclType = Generalização)
```

b) Se especificada a restrição {completo}, o relacionamento de Generalização deverá ter no mínimo duas subclasses.

Terminador

```
If self.tipo = Generalização and self.restrição = {completo}
then
  self.classe.especializa –> size >= 2
endif
```

c) Ao instanciar uma classe Concreta, esta herda todos os atributos e métodos da classe abstrata.

Classe

```
Set(Classe) conforms to Set(self.generaliza)
```

d) O conjunto de atributos herdados mais os atributos da classe concreta instanciada devem ser únicos (uso ou não do conceito de polimorfismo).

Classe

```
self.allInstances -> exists
(((self.generaliza.atributo -> size >=0) and
(self.especializa.atributo -> size >=0)) -> asSet).
```

e) O conjunto de métodos herdados mais os métodos da classe concreta instanciada devem ser únicos (uso ou não do conceito de polimorfismo).

Concreta

```
self.allInstances -> exists
(((self.generaliza.definição método -> size >=0) and
(self.especializa.definição método -> size >=0)) -> asSet).
```

Dependência:

a) Um relacionamento pode ser de Dependência.

Tipo Relacionamento

```
self -> select(oclType = Dependência).
```

Associação:

a) Associação é uma especialização de Tipo Relacionamento.

Tipo Relacionamento

```
self -> select(oclType = Associação).
```

b) Uma Associação poderá ser ordenada somente se a multiplicidade do relacionamento for maior que um.

Terminador

```
if self.tipo = Associação and self.cardinalidade >1 then
  Associação.ordenda = true
endif.
```

Agregação:

a) Agregação é uma especialização de Associação.

Associação

```
self -> select(oclType = Agregação).
```

b) Em um relacionamento de Associação por Agregação, somente um Terminador poderá ser do tipo Agregação.

Relacionamento

```
self.participação -> select (self.participação.tipo = Agregação)
-> size = 1.
```

c) Em um relacionamento de Associação por Agregação, a Multiplicidade Terminador da Agregação deve ser um.

Terminador

```
self.tipo = Agregação implies self.cardinalidade = 1.
```

Composição:

a) Composição é uma especialização de Agregação.

Agregação

```
self -> select(oclType = Composição).
```

b) Em um relacionamento de Associação por Composição, somente um Terminador poderá ser do tipo Composição.

Relacionamento

```
self.participação -> select (self.participação.tipo =
Composição) -> size = 1.
```

c) Em um relacionamento de Associação por Composição, a Multiplicidade Terminador da Composição deve ser um.

Terminador

```
self.tipo = Composição implies self.cardinalidade = 1.
```

Associação Qualificada:

a) Associação Qualificada é uma especialização de Associação.

Tipo Relacionamento

```
self -> select(oclType = Associação).
```

b) Para ser uma Associação Qualificada deverá ser especificado pelo menos um qualificador.

Associação Qualificada

```
self.agrega -> size >=1 .
```

c) Uma Associação Qualificada poderá ser ordenada somente se a multiplicidade do relacionamento for maior que um.

Terminador

```
if self.tipo = Associação Qualificada and self.cardinalidade >1
then
  Associação.ordenda = true
endif.
```

3.1.4 Modelagem Comportamental

O pacote **Modelagem Comportamental** modela os diagramas que descrevem a estrutura dinâmica dos sistemas. Este pacote efetua o mapeamento dos conceitos manipulados pelos diagramas de Interação (diagramas de colaboração e de seqüência) e pelos diagramas de estados, conforme a notação abordada na seção 2.4.

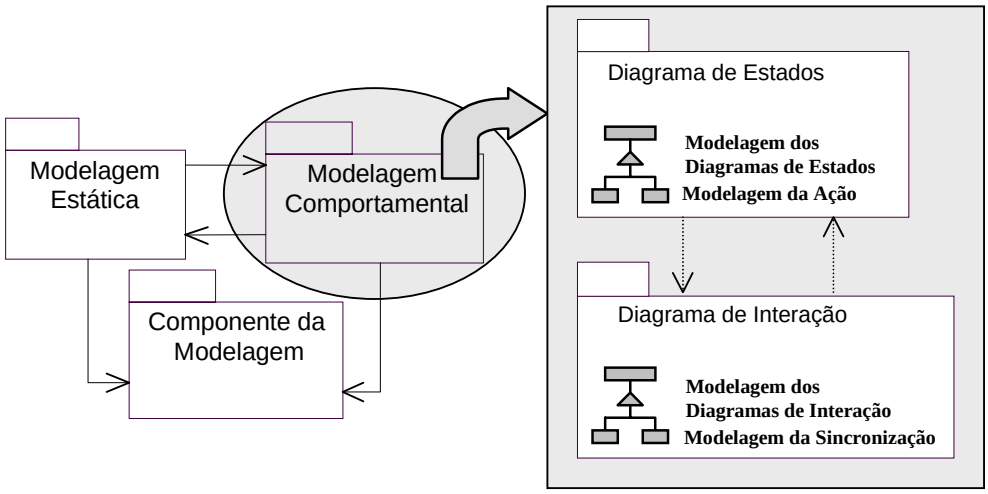


FIGURA 3.15 – Pacote Modelagem Comportametal.

Os conceitos mapeados para o pacote **Modelagem Comportamental,** FIGURA 3.15, foram estruturados em dois outros pacotes:

- **Diagrama de Estados:** Neste pacote foi modelada a notação da UML-RT para trabalhar com diagramas de estados. Os conceitos presentes na notação foram mapeados para os diagramas de classes **Modelagem dos Diagramas de Estado,** FIGURA 3.18**,** e para o diagrama **Modelagem da Ação**, FIGURA 3.20.

- **Diagrama de Interação:** Este pacote modela a notação empregada pelos diagramas de interação, que são os diagramas de seqüência e de colaboração. Os conceitos apresentados por esta notação foram modelados no diagrama de classe **Modelagem dos Diagramas de Interação,** FIGURA 3.16. Sendo que o mecanismo para designar a sincronização da troca de mensagens entre objetos ativos estão modelados no diagrama **Modelagem da Sincronização,** FIGURA 3.17.

A notação para descrever os requisitos temporais nos diagramas de interação também foi modelada, FIGURA 3.16.

3.1.4.1 Descrição da Modelagem dos Diagramas de Interação

Os diagramas de interação descrevem a estrutura dinâmica do sistema que está sendo modelado. Basicamente estes diagrama mostram a forma como os objetos colaboram para execução de uma funcionalidade ou um comportamento do sistema.

UML-RT propõe dois tipos de diagramas para descrever as interações entre objetos: os diagramas de colaboração e os diagramas de seqüência.

A notação para estes diagramas é mostrada através da FIGURA 2.3 na seção 2.4 deste trabalho. Foi estendida para modelar a mensagem *broadcasting*, a mensagem concorrente, as marcas de tempo e marcas de estado. Ambos os diagramas implementam um padrão de estereótipos para representar a chegada das mensagens no objeto *receiver* e para representar a forma de sincronização utilizada pelo objeto *sender*, mostrada na TABELA 2.

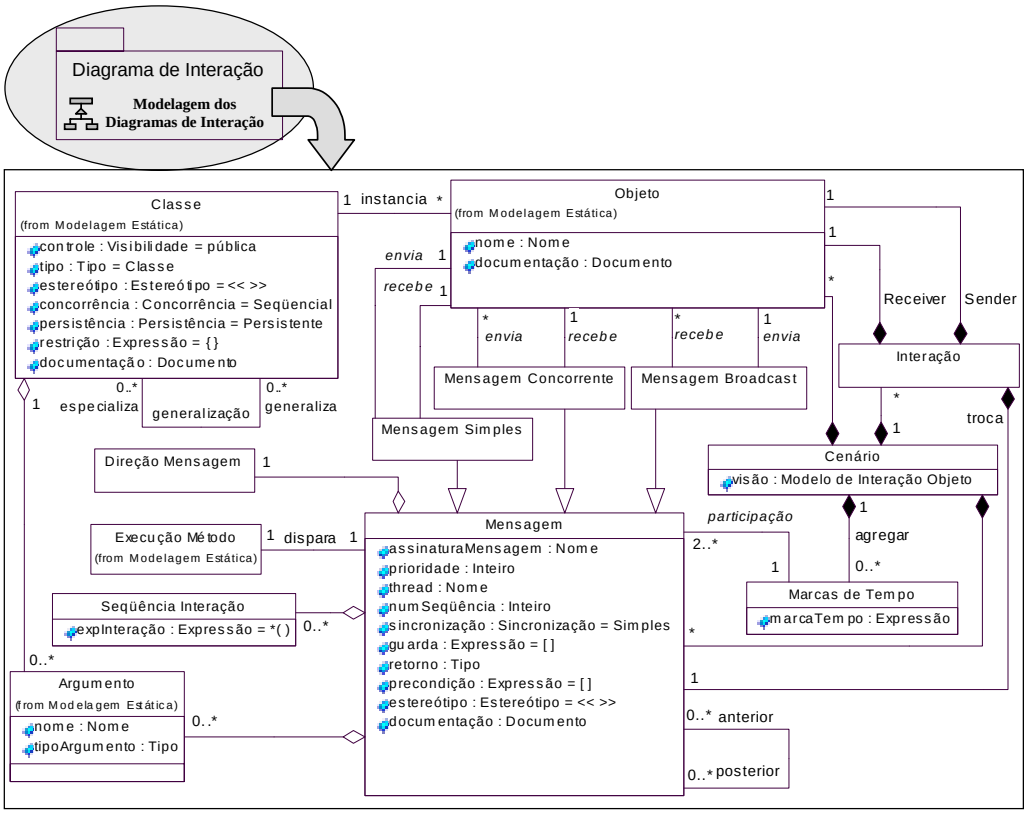


FIGURA 3.16 – Modelagem dos Diagramas de Interação.

Os conceitos presentes na notação dos diagramas de seqüência e de colaboração, introduzidos anteriormente e descritos na FIGURA 2.3 da seção 2.4 deste trabalho, foram mapeados para o diagrama Modelagem dos Diagramas de Interação, mostrado na FIGURA 3.16. Neste diagrama, a classe Cenário faz a identificação, através da implementação do atributo *visão*, de qual dos modelos de interação dos objetos está sendo descrito: o de seqüência ou o de colaboração.

A classe Interação permite mapear as trocas de mensagens entre os objetos. Para que uma interação ocorra, os conceitos de Mensagem, objeto *Sender* e objeto *Receiver* são manipulados.

Todos os conceitos e restrições semânticas relativas aos diagramas de interação serão descritos em seguida, e se encontram presentes no metamodelo resultante do mapeamento do diagrama de seqüência e de colaboração, FIGURA 3.16.

Cenário: É a especificação do diagrama que está sendo modelado.

Atributos:

visão: Especifica a visão para o diagrama de interação (seqüência ou colaboração). Ver Modelo de Interação Objeto.

Semântica:

a) Um Cenário agrega um conjunto que contenha dois ou mais objetos não repetidos.

b) Um Cenário agrega um conjunto que contenha pelo menos duas ou mais Interações.

c) Um Cenário pode ou não conter um conjunto de Marcas de Tempo especificada.

d) Um Cenário agrega um conjunto não vazio de Mensagens.

Interação: É a especificação da troca de uma mensagem entre dois objetos.

Semântica:

a) Uma Interação é composta por um objeto *Sender* (de envio), por um objeto *Receiver* (receptor) e pela mensagem.

Marcas de Tempo: É a especificação das marcas de tempo no Modelo de Interação Objeto, na visão Seqüência. As marcas de tempo agregam um adorno representativo deste conceito no diagrama de seqüência, modelado na FIGURA 1.7 – Modelagem da Representação Gráfica do Diagrama de Interação na seção 1.1.2 dos anexos.

As marcas de tempo são utilizadas no diagrama de seqüência seguindo o padrão de modelagem da UML-RT. Porém nada impede que estas marcas sejam utilizadas no diagrama de colaboração. Esta possibilidade irá depender somente da capacidade do editor diagramático, pois o conceito estará disponível no dicionário de dados.

Atributos:

marcaTempo: Especifica o tempo através da descrição de uma Restrição Temporal. Ver Expressão.

Semântica:

a) Uma Marca de Tempo está associada a no mínimo duas ou mais instâncias de Mensagem.

b) Uma Mensagem dispara uma Execução Método. Esta tem definido o tempo para que a execução seja concluída. Uma vez que a mensagem não tem tempo agregado, então, o tempo estabelecido pela *marcaTempo* deverá ser igual à soma dos Tempo(s) Execução de cada método.

Mensagem: É o mecanismo de comunicação, solicitação de serviços entre os objetos. A especificação de uma mensagem está associada ao disparo de um método.

Atributos:

assinaturaMensagem: Especifica o nome da mensagem. Ver Nome;

prioridade: Descreve a prioridade da mensagem.

thread: O *numSeqüência* identifica a ordem na qual as mensagens ocorrem. Este número seqüencial descreve um cenário para uma simples *thread* ativa de controle. Em modelos de sistemas concorrentes, o *numSeqüência* deverá ser precedido pelo nome ou um indicador da *thread* de controle. Isso possibilitará a inclusão de pré-condição e condição de guarda, mostrando, de forma explicita, a sincronização entre as *threads* de controle. Por exemplo [A1, B4]C1: [X>Y], neste caso, a *thread* A e a mensagem 1 juntamente com a *thread* B e a mensagem 4 devem preceder a *thread* C e mensagem 1. Além

disso, a condição de guarda deve ser verdadeira, ou seja, X maior que Y. Ver Nome;

numSeqüência: Especifica a ordem na qual a mensagem ocorre;

sincronização: Especifica a semântica da sincronização das mensagens trocadas entre objetos ativos em processamento concorrente. Ver Sincronização;

guarda: Expressão booleana que deve ser *true* para que a mensagem possa ser disparada. Possibilita a descrição de uma restrição ou condição de guarda. Ver Expressão;

retorno: Descreve o tipo de retorno da mensagem. Ver Tipo;

precondição: Permite a especificação de pré-condição no modelo. Em cenários *multithread,* a ativação de um conjunto de mensagens pode formar uma pré-condição - semelhante a uma condição de guarda. Representa um conjunto de mensagens que devem ser disparadas (estas mensagens são identificadas pela sua *thread* e seu *numSeqüência*, dispostas entre colchetes e separadas por vírgula) antes de uma determinada mensagem.

estereótipo: Mecanismo de extensão da mensagem. Ver Estereótipo;

documentação: Descrição da mensagem. Ver Documentação.

Semântica:

a) Mensagem é uma classe abstrata que especializa as mensagens Simples, Concorrente e Broadcast.

b) Se a visão especificada para o Modelo de Interação Objeto for Colaboração, a mensagem, então, agrega uma Direção Mensagem.

c) Uma mensagem sempre vai estar associada a uma única Execução Método (disparo).

d) Se o método o qual a mensagem dispara a execução tem argumentos definidos, a mensagem obrigatoriamente deverá passar estes argumentos como parâmetro.

e) Uma mensagem poderá ou não ter definidas mensagens anteriores ou posteriores a ela.

Mensagem Simples: É a especificação de uma mensagem em que um objeto é o remetente e o outro objeto, o receptor. Não é necessário que sejam objetos diferentes. É possível especificar um objeto que envia uma mensagem para si mesmo.

Semântica:

a) Mensagem Simples é uma especialização de Mensagem.

Mensagem Concorrente: É a especificação de várias mensagens enviadas por vários objetos remetentes a um único objeto receptor.

Semântica:

a) Mensagem Concorrente é uma especialização de Mensagem.

b) O conjunto de mensagens enviadas pelos objetos remetentes deve ser maior que um.

Mensagem Broadcast: É a especificação de uma mensagem enviada por um objeto remetente a vários objetos receptores.

Semântica:

a) Mensagem *Broadcast* é uma especialização de Mensagem.

b) O conjunto de objetos receptores da mensagem deve ser maior que um.

Direção Mensagem: É a especificação de indicadores de direção da mensagem no Modelo de Interação Objeto visão Colaboração. Quando modela-se a visão Colaboração, as mensagens expressam sua direção através de uma seta. Os adornos são diferenciados para mensagens e mensagens com fluxo de dados.

A modelagem da representação gráfica do adorno, que indica a direção ou se a mensagem tem argumentos (fluxo de dados), está modelada pela classe F_Direção Mensagem, na FIGURA 1.7 – Modelagem da Representação Gráfica do Diagrama de Interação no anexo 1.1.2.

Seqüência Interação: É a especificação de uma expressão onde são descritos os níveis de aninhamento da mensagem. Cada nível é separado por ":".

O conceito Seqüência Interação especifica um adorno que caracteriza a forma como serão executadas as mensagens na expressão de interação. Há possibilidade de modelá-la como execução consecutiva (*) ou concorrente (para expressar o paralelismo utiliza-se || - pipeline duplo). Este adorno é modelado pela classe F_Seqüência Interação, da FIGURA 1.7 – Modelagem da Representação Gráfica do Diagrama de Interação abordada no anexo 1.1.2.

Atributos:

expInteração: Define um conjunto de mensagens similares que são enviadas ao longo do link. Uma vez que as mensagens participativas da interação são parte de um mesmo fluxo de mensagem, possuem as mesmas descrições.

Ex.: *(j=1..n) ou seu equivalente *(1..n)

3.1.4.2 Descrição das Restrições em OCL

Cenário:

a) Um Cenário agrega um conjunto que contenha dois ou mais objetos não repetidos.

Objeto

```
Cenário –> select (Componentes da Modelagem::
Modelo.allInstances implies (self.allInstances –>
forAll(p1,p2 | p1 <> p2 implies p1.nome <> p2.nome)) –>
asSet) –> size >=2.
```

b) Um Cenário agrega um conjunto que contenha pelo menos dois ou mais objetos Interação.

Cenário

```
self.interação –> size >=2.
```

c) Um Cenário pode ou não conter um conjunto de Marcas de Tempo especificadas.

Cenário

```
self.agrega –> size >= 0.
```

d) Um Cenário agrega um conjunto não vazio de Mensagens.

Cenário

```
self.mensagem –> size >= 1.
```

Interação:

a) Uma Interação é composta por um objeto *Sender* (de envio), por um objeto *Receiver* (receptor) e pela mensagem.

Interação

```
(self.Sender –> size = 1) and
(self.Receiver –> size = 1) and
(self.troca –> size = 1).
```

Marcas de Tempo:

a) Uma Marca de Tempo está associada a no mínimo dois ou mais objetos Mensagem.

Marca de Tempo

```
self.participação –> size >= 2.
```

b) Uma Mensagem dispara uma Execução Método. Esta tem definido o tempo para que a execução seja concluída. Uma vez que a mensagem não tempo agregado, o tempo estabelecido pela *marcaTempo* deverá ser igual à soma dos Tempo(s) Execução de cada método.

Marcas de Tempo

```
self.participação.dispara –> iterate(
  em: Execução Método;
  result: real = 0 |
  if em.oclType = Aperiódico then
    result + em.especifica.tempo
  endif
) =
self.marcaTempo.
```

Mensagem:

a) Mensagem é uma classe abstrata que especializa as mensagens Simples, Concorrente e Broadcast.

Mensagem

self.allInstances –> select (oclType = Mensagem) –> isEmpty

b) Se a visão especificada para o Modelo de Interação Objeto for Colaboração, a mensagem, então, agrega uma Direção Mensagem.

Cenário

if self.visão = Colaboração::Modelo de Interaçao Objeto then

self.mensagem.direçao mensagem –> size = 1

endif

c) Uma mensagem sempre vai estar associada a uma única Execução Método (disparo).

Mensagem

self.dispara –> size = 1

d) Se o método no qual a mensagem dispara a execução tem argumentos definidos, a mensagem obrigatoriamente deverá passar estes argumentos como parâmetro.

Argumento

Mensagem.argumento and Definição Método –> select

((self.allInstances -> forAll(p1, p2 | p1 <> p2 implies

p1.nome <> p2.nome)) -> asSet)

e) Uma mensagem poderá ou não ter mensagens definidas anteriores ou posteriores a ela.

Mensagem

(self.anterior –> size >= 0) and (self.posterior –> size >= 0)

Mensagem Simples:

a) Mensagem Simples é uma especialização de Mensagem.

Mensagem

self -> select(oclType = Mensagem Simples).

Mensagem Concorrente:

a) Mensagem Concorrente é uma especialização de Mensagem.

Mensagem

self -> select(oclType = Mensagem Concorrente).

b) O conjunto de mensagens enviadas pelos objetos remetentes deve ser maior que um.

Mensagem Concorrente

self.envia –> size >= 1.

Mensagem Broadcast:

a) Mensagem *Broadcast* é uma especialização de Mensagem.

Mensagem

self -> select(oclType = Mensagem *Broadcast*).

b) O conjunto de objetos receptores da mensagem deve ser maior que um.

Mensagem Broadcast

self.recebe –> size >= 1

3.1.4.3 Descrição da Modelagem da Sincronização

Sincronização é um conceito modelado pela classe Mensagem. Esta modelagem é feita através do atributo chamado *sincronização* que também é do tipo Sincronização. Esta é uma classe genérica do diagrama Modelagem da Sincronização, resultado do mapeamento do conceito, como mostra a FIGURA 3.17. Este conceito é detalhado a seguir, e sua semântica é exposta em linguagem natural e formalizada na linguagem OCL.

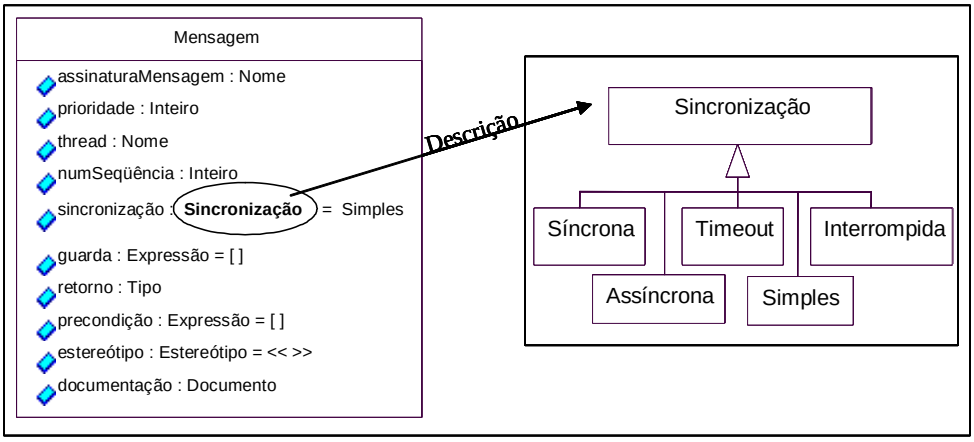


FIGURA 3.17 – Modelagem da Sincronização.

Sincronização: Especifica mecanismo para designar a sincronização da troca de mensagens entre objetos. O conceito de sincronização agrega uma representação gráfica que caracteriza visualmente a forma de sincronização da mensagem. Este adorno gráfico é modelado pela classe F_Sincronização, da FIGURA 1.7 – Modelagem da Representação Gráfica do Diagrama de Interação na seção 1.1.2 dos anexos.

Síncrona: Na especificação síncrona, observada em termos de sincronização, a execução do método procede somente quando o objeto cliente envia a mensagem para o objeto fornecedor e este aceita a mensagem. O objeto cliente executa até que envie a mensagem. Aguarda, então, o objeto fornecedor aceitá-la. O objeto cliente continua esperando até que a mensagem seja aceita.

Semântica:

a) Uma mensagem Síncrona é uma especialização de Sincronização.

Assíncrona: Na especificação assíncrona, observada em termos de sincronização, o objeto cliente envia uma mensagem para o objeto fornecedor processar e continua

executando seu código sem esperar por um retorno, sobre o recebimento ou não da mensagem, feito pelo objeto fornecedor.

Semântica:

a) Uma mensagem Assíncrona é uma especialização de Sincronização.

Timeout: Na especificação de mensagem *timeout*, observada em termos de sincronização, o objeto cliente abandona a mensagem se o objeto fornecedor não manuseá-la dentro de um período específico de tempo.

Semântica:

a) Uma mensagem *Timeout* é uma especialização de Sincronização.

Simples: A especificação de mensagem simples é a mais utilizada. Mensagens simples são mensagens com uma única *thread* de controle, ou seja, um objeto cliente envia uma mensagem a um objeto passivo.

Semântica:

a) Uma mensagem Simples é uma especialização de Sincronização.

Interrompida: Na especificação de mensagem interrompida, o objeto cliente passa uma mensagem somente se o objeto fornecedor estiver pronto para aceitá-la. O objeto cliente abandona a mensagem se o fornecedor não estiver pronto para recebê-la.

Semântica:

a) Uma mensagem Interrompida é uma especialização de Sincronização.

3.1.4.4 Descrição das Restrições em OCL

Síncrona:

a) Uma mensagem Síncrona é uma especialização de Sincronização.

Sincronização

```
self -> select(oclType = Síncrona).
```

Assíncrona:

a) Uma mensagem Assíncrona é uma especialização de Sincronização.

Sincronização

```
self -> select(oclType = Assíncrona).
```

Timeout:

a) Uma mensagem *Timeout* é uma especialização de Sincronização.

Sincronização

```
self -> select(oclType = Timeout).
```

Simples:

a) Uma mensagem Simples é uma especialização de Sincronização.

Sincronização

```
self -> select(oclType = Simples).
```

Interrompida:

a) Uma mensagem Interrompida é uma especialização de Sincronização.

Sincronização

```
self -> select(oclType = Interrompida).
```

3.1.4.5 Descrição da Modelagem do Diagrama de Estados

O diagrama de estados é uma técnica bem conhecida e muito utilizada pelas metodologias orientadas a objetos. Incorpora características das máquinas de *Moore* e máquinas de *Mealy* [LAP 97], e foi adotado pela UML-RT para descrever o comportamento dos objetos.

Cada diagrama de estados descreve os estados possíveis que uma classe de objetos pode assumir, dado um período de tempo. Basicamente, estes diagramas trabalham com os conceitos de estado e transição de estado. A notação empregada para descrevê-los pode ser consultada na FIGURA 2.4, da seção 2.4.

Os conceitos manipulados ao construir um diagrama de estados foram mapeados para um metamodelo. O resultado desta descrição pode ser visto no diagrama Modelagem do Diagrama de Estados, na FIGURA 3.18.

Estes conceitos mostrados na FIGURA 3.18 serão expostos e detalhados a seguir. Novamente, a semântica é descrita em linguagem natural e posteriormente formalizada em OCL.

Modelos de Estados Objeto: Especifica um conjunto de estados possíveis para uma classe de objeto num dado instante de tempo.

Semântica:

a) Um Modelo de Estados é composto por um conjunto de pelo menos três objetos Estado e pelo menos duas respectivas transições.

Conjunto Concorrente: É um Modelo de Objetos. Especifica um conjunto de estados e suas transições.

Semântica:

a) Um Conjunto Concorrente é uma especialização de Modelo de Estados.

Estado: Especifica um estado possível para uma classe de objeto num dado instante de tempo.

Atributos:

nome: Especifica o nome do estado. Ver Nome;

documentação: Descrição do estado. Ver Documentação.

Semântica:

a) Um estado agrega um conjunto que contém pelo menos uma Ação/Atividade a ser executada, enquanto o objeto permanece no mesmo.

b) Um estado é uma classe abstrata e especializa Estado Inicial, Estado Final, Estado Intermediário e SuperEstado.

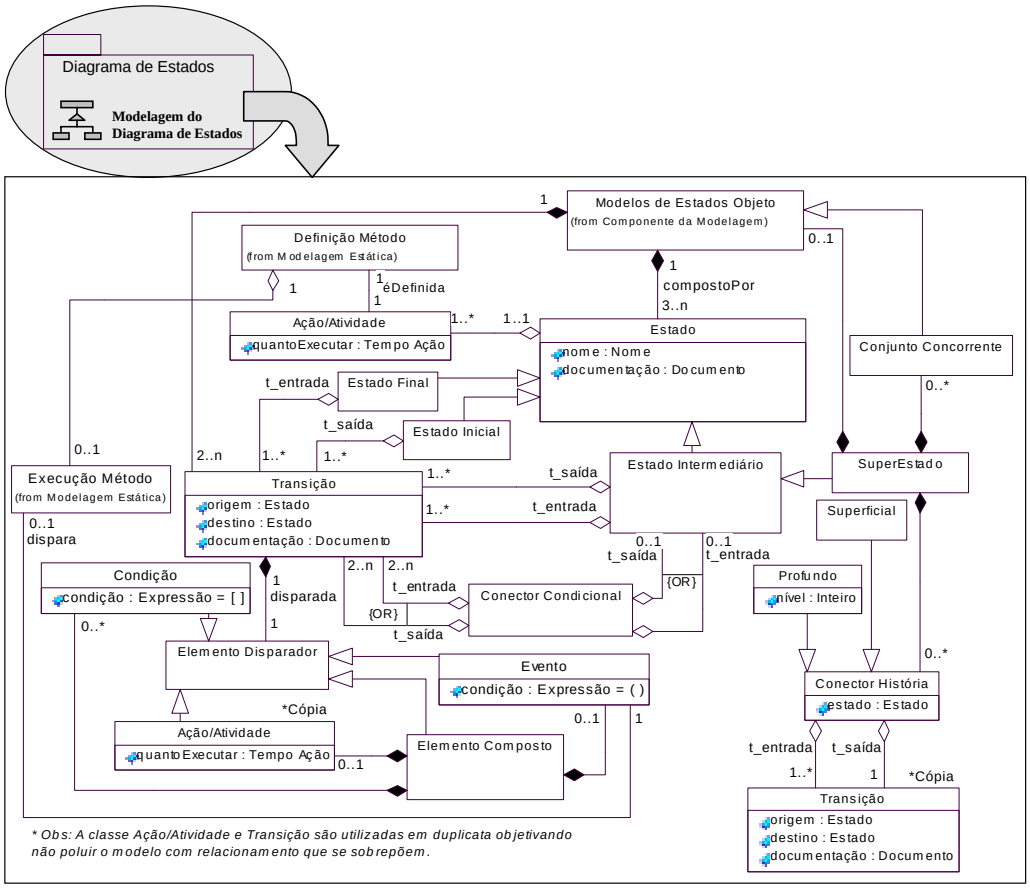


FIGURA 3.18 – Modelagem do Diagrama de Estados.

Estado Inicial: Especifica a criação do objeto.

Semântica:

a) Um estado inicial agrega pelo menos uma transição de saída.

Estado Final: Especifica a destruição do objeto.

Semântica:

a) Um estado final agrega pelo menos uma transição de entrada.

Estado Intermediário: Especifica os estados possíveis num dado instante de tempo para o objeto.

Semântica:

a) Um estado intermediário agrega pelo menos uma transição de entrada.

b) Um estado intermediário agrega pelo menos uma transição de saída.

c) Um estado intermediário pode ou não ter transições de entrada ou de saída para um Conector Condicional.

d) Um estado intermediário pode ou não ser um superestado.

SuperEstado: Especifica o aninhamento de estados. Estes estados aninhados são os que ocorrem simultaneamente com um superestado agregador. Um superestado caracteriza um conjunto de comportamentos independentes aninhado a si próprio.

Semântica:

a) Um superestado pode ser composto por um Modelo de Objetos executado seqüencialmente ou pode ser composto por objetos Conjunto Concorrente de estados, ou seja, Modelos de Objetos executando concorrentemente dentro de um superestado.

b) Um superestado poderá ou não ter especificado Conector História .

Conector História: Permite especificar mecanismos para identificar o estado aninhado no qual o objeto se encontrava no momento da transição (superestado para outro estado qualquer) de forma que torna possível a volta (transição de um estado qualquer para o superestado) ao estado seguinte, aninhado no superestado. Um Conector História pode ser um conector Superficial ou Profundo. Conector Superficial memoriza um estado presente na superfície do superestado. Já o conector Profundo memoriza o estado a qualquer nível de profundidade uma vez que for permitido mais de um nível de aninhamento.

Atributos:

estado: Especifica o estado em que o objeto estava no momento da transição. Ver Estado.

Semântica:

a) Um Conector História poderá ser um conector Profundo ou Superficial.

b) Um Conector História poderá agregar uma ou mais transições de entrada, mas somente uma de saída.

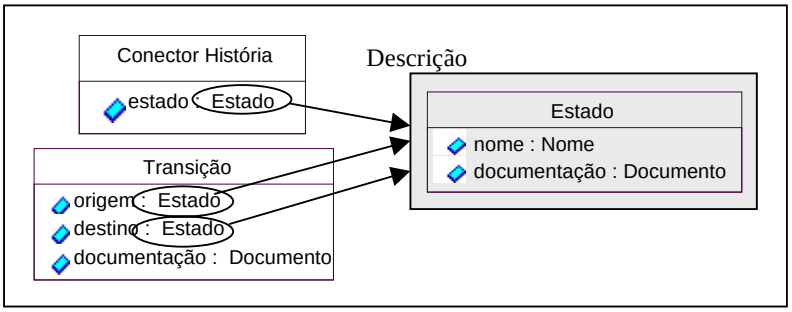


FIGURA 3.19 – Classes que Agregam Estado na Modelagem do Diagrama de Estados.

As classes Transição e Conector História agregam a classe Estado, FIGURA 3.19. Transição guarda o Estado origem e o destino, e Conector História memoriza o Estado, do conjunto de estados aninhados, no momento da transição.

Superficial: Conector Superficial permite especificar um estado presente na superfície do superestado, de onde ocorreu a transição.

Profundo: Conector Profundo permite especificar o estado a qualquer nível de profundidade em que ocorreu a transição.

Atributos:

nível: É um tipo inteiro que especifica o nível de profundidade do estado memorizado.

Transição: Permite especificar a mudança de estado do objeto. Dado dois estados E1 e E2, uma transição pode ser descrita como a ligação destes dois estados. Para que objeto no E1 passe para o E2, deve obedecer as restrições especificadas pelo Elemento Disparador da transição.

Atributos:

origem: Especifica o estado atual do objeto. Ver Estado;

destino: Especifica o estado destino, para qual o objeto está transitando. Ver Estado;

documentação: Descrição da transição. Ver Documentação.

Semântica:

a) Uma Transição é disparada por um Elemento Disparador.

Elemento Disparador: Permite especificar os mecanismos responsáveis pela transição de estado de um objeto. O Elemento disparador pode ser um Evento, Condição, Ação ou um Elemento Composto por quaisquer dos três mecanismos mencionados.

Semântica:

a) Um Elemento Disparador pode ser uma Condição, um Evento, uma Ação ou um Elemento Composto.

Evento: Permite especificar os eventos que disparam uma transição. Um evento pode ser:

- Um sinal explícito de um outro objeto (*SignalEvent*).
- Uma chamada para a execução de um método feita pelo próprio objeto ou por outro objeto (*CallEvent*).
- Um condição que se tornou verdadeira (*ChangeEvent*), por exemplo, **when(x>0)**. Este conceito difere de uma condição que é testada na transição. Se *true,* o objeto muda seu estado; se *false,* não.
- Uma condição temporal, ou seja, a especificação de um requisito temporal que se tornou verdadeira (*TimeEvent*). Dados os estados E1 e E1 e especificação **after (40 milisegundos)**, então, após 40ms é gerado o evento de transição.

Atributos:

condição: Especifica uma condição (*ChangeEvent* ou *TimeEvent*). Quando verdadeira, dispara a transição de Estado. Ver Expressão.

Semântica:

a) Um Evento poderá ou não disparar a Execução de um método.

Condição: Permite especificar uma condição. Um objeto só fará a transição de estados se a condição for satisfeita.

Atributos:

condição: Especifica uma condição para que a transição de Estado possa ocorrer. Ver Expressão.

Ação/Atividade: Uma ação de saída de estado poderá disparar uma transição.

Atributos:

quandoExecutar: Especifica o momento em que uma ação deve ser executada, ou seja, na entrada de um estado durante a permanência do objeto no estado ou na saída do estado. Ver Tempo Ação.

Semântica:

a) Somente uma ação de saída de estado poderá disparar uma transição.

Elemento Composto: Permite especificar um Elemento Disparador composto por mais de um mecanismo que fará a transição de estado. Um Elemento Composto pode ser um {Evento, Condição, Ação}, {Evento, Condição}, {Evento, Ação} ou {Condição, Ação}.

Conector Condicional: Permite especificar transições complexas em um modelo. Um conector condicional pode agregar várias transições de entrada e uma única transição de saída, também pode agregar muitas transições de saída mas um única de entrada.

Semântica:

a) Um conector condicional pode agregar duas ou mais transições de entrada mas uma única de saída.

b) Um conector condicional pode agregar duas ou mais transições de saída mas uma única de entrada.

c) Em um conector condicional as regras a) e b) não podem ser simultâneas.

3.1.4.6 Descrição das Restrições em OCL

Modelos de Estados Objeto:

a) Um Modelo de Estados é composto por um conjunto de pelo menos três objetos Estado e pelo menos duas respectivas transições.

Modelo de Estados

```
self.allInstances -> forAll(self.compostoPor –> size >= 3 and
self.transição –> size >= 2).
```

Conjunto Concorrente:

a) Um Conjunto Concorrente é uma especialização de Modelo de Estados.

Modelo de Estados

```
self -> select(oclType = Conjunto Concorrente).
```

Estado:

a) Um estado agrega um conjunto que contém pelo menos uma Ação/Atividade a ser executada enquanto o objeto permanece no mesmo.

Estado

self.acao/atividade –> size >= 1

b) Um estado é uma classe abstrata e especializa Estado Inicial, Estado Final, Estado Intermediário e SuperEstado.

Estado

self.allInstance –> select(ocltype = Estado) –> isEmpty.

Estado Inicial:

a) Um estado inicial agrega pelo menos uma transição de saída.

Estado Inicial

self.t_saída –> size >= 1.

Estado Final:

a) Um estado final agrega pelo menos uma transição de entrada.

Estado Final

self.t_entrada –> size >= 1

Estado Intermediário:

a) Um estado intermediário agrega pelo menos uma transição de entrada.

Estado Intermediário

self.t_entrada –> size >= 1.

b) Um estado intermediário agrega pelo menos uma transição de saída.

Estado Intermediário

self.t_saída –> size >= 1.

c) Um estado intermediário pode ou não ter transições de entrada ou de saída para um Conector Condicional.

Conector Condicional

(self.t_entrada –> size >= 2 implies self.t_saída –> size = 1) or

(self.t_saída –> size >= 2 implies self.t_entrada –> size = 1).

d) Um estado intermediário pode ou não ser um superestado.

Estado Intermediário

self –> select(oclType = SuperEstado).

SuperEstado:

a) Um SuperEstado pode ser composto por um Modelo de Objetos executado seqüencialmente ou por Conjunto(s) Concorrente(s) de estados, ou seja, Modelo de Objetos executando concorrentemente dentro de um superestado.

SuperEstado

(self.conjunto concorrente –> size >= 1) or

(self.modelo de estado –> size = 1).

b) Um superestado pode ou não ter especificado um Conector História.

SuperEstado

self.conector história –> size >= 0

Conector História:

a) Um Conector História pode ser um conector Profundo ou Superficial;

Conector História

self.allInstances –> select (oclType = Conector História) –> isEmpty.

b) Um Conector História pode agregar uma ou mais transições de entrada mas somente uma de saída.

Conector História

(self.t_entrada –> size >= 1) and (self.t_saída –> size = 1)

Transição:

a) Uma Transição é disparada por um Elemento Disparador.

Transição

self.disparada –> size = 1

Elemento Disparador:

a) Um Elemento Disparador pode ser uma Condição, um Evento, uma Ação ou um Elemento Composto.

Elemento Disparador

self.allInstances –> select (oclType = Elemento Disparador) –> isEmpty.

Evento:

a) Um Evento poderá ou não disparar a Execução de um método.

Evento

(self.dispara –> size = 1) or

(self.dispara –> size = 0).

Ação/Atividade:

a) Somente uma ação de saída de estado poderá disparar uma transição.

Ação/Atividade

if self.quandoExecutar = Saída then

  self.disparada –> size = 1

endif.

Conector Condicional:

a) Um conector condicional pode agregar duas ou mais transições de entrada mas uma única de saída.

Conector Condicional

self.t_entrada –> size >= 2 implies self.t_saída –> size = 1.

b) Um conector condicional pode agregar duas ou mais transições de saída mas uma única de entrada.

Conector Condicional

self.t_saída –> size >= 2 implies self.t_entrada –> size = 1.

c) Em um conector condicional as regras a) e b) não podem ser simultâneas.

Conector Condicional

(self.t_entrada –> size >= 2 implies self.t_saída –> size = 1) or

(self.t_saída –> size >= 2 implies self.t_entrada –> size = 1).

3.1.4.7 Descrição da Modelagem da Ação

Ao modelar um diagrama de estados, deve ser descrito o momento em que as atividades, destes estados, são executadas. Por exemplo: atividades de entrada são as operações executadas ao entrar no estado, atividades intermediárias são aquelas executadas durante a permanência do objeto no estado, e as de saída devem ser executadas ao sair do estado, antes da transição ser concretizada. O diagrama Modelagem da Ação mostrado na FIGURA 3.20 é resultado do mapeamento destes conceitos.

Em seguida, serão expostos detalhadamente os conceitos mostrados na FIGURA 3.20, juntos com suas semânticas tanto na linguagem natural como em OCL.

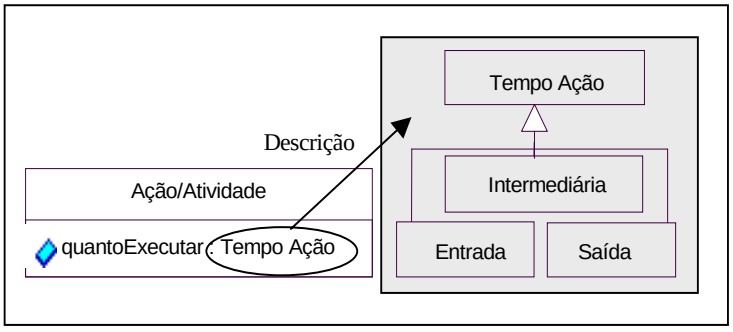


FIGURA 3.20 – Modelagem da Ação.

Tempo Ação: Especifica o momento em que a ação será executada. Uma ação pode ser executada no momento em que o objeto entra no estado. Estas são chamadas de 'ações de entrada'. As ações executadas durante a permanência do objeto no estado são as chamadas 'ações intermediárias' e as ações executadas no momento da transição são 'ações de saída'.

Semântica:

a) Tempo Ação é classe abstrata de Entrada, Intermediária e Saída.

Entrada: Especifica as ações que devem ser executadas no momento em que o objeto está entrando no estado.

Semântica:

a) Uma ação de Entrada é uma especialização de Tempo Ação.

Intermediária: Especifica as ações que devem ser executadas durante a permanência do objeto no estado.

Semântica:

a) Uma ação Intermediária é uma especialização de Tempo Ação.

Saída: Especifica as ações que devem ser executadas momentos antes da transição ocorrer. As ações de saída são as últimas executadas pelo objeto antes da transição para o próximo estado concretizar-se.

Semântica:

a) Uma ação de Saída é uma especialização de Tempo Ação.

3.1.4.8 Descrição das Restrições em OCL

Tempo Ação:

a) Tempo Ação é classe abstrata de Entrada, Intermediária e Saída.

Tempo Ação

```
self.allInstances -> select(oclType = Tempo Ação) -> isEmpty.
```

Entrada

a) Uma ação de Entrada é uma especialização de Tempo Ação.

Tempo Ação

```
self -> select(oclType = Entrada).
```

Intermediária:

a) Uma ação Intermediária é uma especialização de Tempo Ação.

Tempo Ação

```
self -> select(oclType = Intermediária).
```

Saída:

a) Uma ação de Saída é uma especialização de Tempo Ação.

Tempo Ação

```
self -> select(oclType = Saída).
```

3.1.5 Componentes da Modelagem

O pacote **Componente da Modelagem**, FIGURA 3.21, foi criado para modelar os conceitos que são comuns aos diagramas da UML-RT. O objetivo deste pacote é estruturar de forma mais clara e compreensível estas definições.

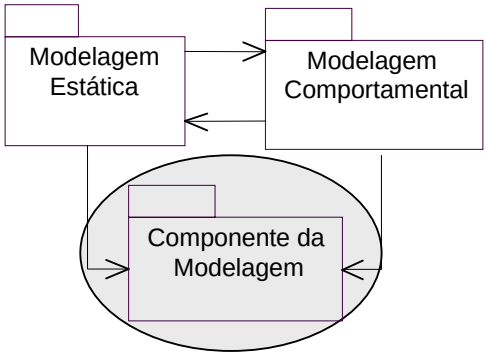


FIGURA 3.21 – Pacote Componentes da Modelagem.

O pacote **Componentes da Modelagem** efetua o mapeamento de conceitos como o de controle da visibilidade. Visibilidade é um conceito usado por classes, atributos, métodos, e outros. Esta especifica como estes conceitos são vistos fora do espaço onde estão definidos. Todos os conceitos que trabalham com visibilidade implementam um atributo chamado *controle*, cujo do tipo é Visibilidade - classe genérica do diagrama **Modelagem do Controle da Visibilidade**, FIGURA 3.22a, que mostra o resultado da sua modelagem.

Muitos conceitos da Notação da UML-RT usam expressões para especificar restrições. Por exemplo, um relacionamento entre duas classes pode ter restrições, formalizadas por intermédio de uma expressão. Os conceitos que trabalham com restrições implementam um atributo chamado *restrição.* Este atributo é do tipo Expressão, classe genérica do diagrama **Modelagem das Expressões**, FIGURA 3.22d, que mostra o resultado da modelagem deste conceito. Todas as restrições estabelecidas em um modelo são descritas através de expressões.

Todas as classes de conceitos que usam a definição de tipo implementam um atributo chamado *tipo e/ou retorno*. Este atributo é do tipo Tipo, classe genérica que descreve este conceito no diagrama **Modelagem dos Tipos**, FIGURA 3.22b.

Os conceitos que podem ser estereotipados implementam um atributo chamado *estereótipo.* Este atributo é do tipo Estereótipo, cujo resultado da sua modelagem é mostrado no diagrama **Modelagem do Estereótipo**, FIGURA 3.22c.

3.1.5.1 Descrição da Modelagem da Visibilidade, dos Tipos, do Estereótipo e das Expressões

A visibilidade, os tipos, o estereótipo e as expressões são conceitos comuns a vários elementos presentes nos diagramas. A classe, por exemplo, pode ser estendida e, a partir desta, criar um novo tipo através do conceito estereótipo. Na modelagem do atributo será necessário descrever seu tipo, um atributo *string,* por exemplo. Entretanto, os métodos podem definir o tipo de retorno, sua visibilidade, restrições através de uma expressão e assim por diante. Nas seções 3.1.3 e 3.1.4 podem ser vistos os conceitos que modelam visibilidade, estereótipo, tipo e outros.

Estes conceitos comuns foram mapeados para diagramas que são mostrados na FIGURA 3.22. Na FIGURA 3.22 **b** é mostrado o diagrama Modelagem de Tipos, resultado do mapeamento feito do conceito Tipo. As classes de conceitos, que modelam esta definição, implementam atributos chamados de *tipo* e/ou *retorno*.

O diagrama Modelagem do Controle da Visibilidade mostrado na FIGURA 3.22 **a** é o resultado do mapeamento do conceito Visibilidade. A visiblidade é um conceito que determina como será o acesso ao código do elemento que a define. Por exemplo, ao modelar o conceito Método com visibilidade privada, significa dizer que este método só poderá ser acessado por outros métodos da mesma classe que o definiu.

Cabe salientar que todas as classes de conceitos que trabalham com visibilidade implementam um atributo chamado *controle.*

O diagrama Modelagem das Expressões, mostrado na FIGURA 3.22 **d**, é o resultado do mapeamento do conceito Expressão. Este é empregado pelas classes de conceitos que modelam restrições sobre si, descritas em um formalismo qualquer. Todas estas classes de conceitos, que modelam restrições, implementam um atributo chamado *restrição.*

A maior parte dos conceitos que foram descritos nas seções 3.1.3 e 3.1.4 pode ser estendida de forma a gerar novos conceitos. Este mecanismo de extensão torna a UML-RT mais flexível para modelar as particularidades dos sistemas a serem desenvolvidos. Todas as classes de conceitos que usam tais mecanismos, ou seja, estereótipos, implementam um atributo chamado *estereótipo.* Na FIGURA 3.22 **c** é mostrado o resultado do mapeamento do conceito Estereótipo.

A seguir, são mostrados detalhadamente todos os conceitos introduzidos até aqui e mapeados na FIGURA 3.20. Mais uma vez a semântica dos conceitos é descrita em linguagem natural e posteriormente formalizada em OCL.

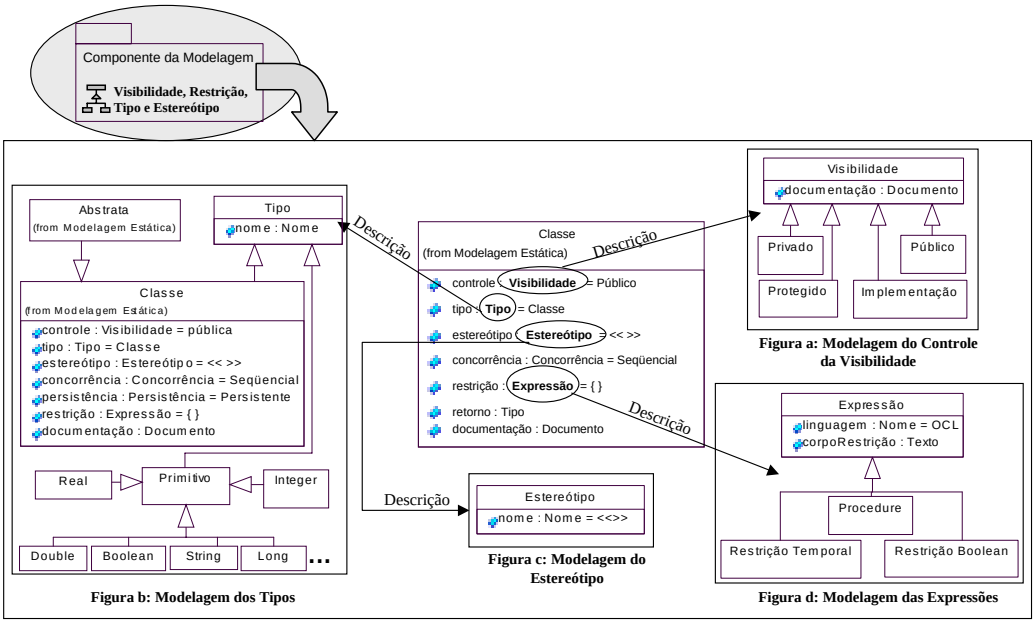


FIGURA 3.22 – Modelagem da Visibilidade, dos Tipos, do Estereótipo e das Expressões.

Visibilidade: Especifica através de um indicador de visibilidade como o conceito é visualizado na implementação. Este indicador pode ser um adorno ou uma restrição OCL, por exemplo: {Pública}, que especifica uma visibilidade pública para um conceito. Deve-se ter cuidado ao modelar a visibilidade pois esta pode representar diferentes semânticas em diferentes linguagens. Estas diferenças, embora pequenas, podem tornar a modelagem confusa. A visibilidade da UML está baseada na visibilidade implementada da linguagem C++.

Semântica:

a) Visibilidade é uma classe genérica e não pode ser instanciada.

b) Visibilidade é uma classe abstrata que generaliza as classes Público, Privado, Protegido e Implementação.

Público: O conceito com a visibilidade caracterizada como Pública especifica que o acesso é permitido em qualquer lugar no código e pode ser chamado por qualquer objeto do sistema.

Protegido: Visibilidade Protegida é semelhante à implementação. Os conceitos modelados como protegidos podem ser acessados pela classe que os contém e também por subclasses especializadas desta classe.

Privado: Os conceitos com visibilidade especificada como Privada são visíveis somente na porção do ambiente que os modelou. Isto significa que será desconhecido e inacessível fora desta porção.

Implementação: Visibilidade adicional que é implementada pela linguagem C++. Sua especificação é semelhante a da visibilidade Protegido.

Tipo: Especifica os tipos utilizados pela notação. A classe Tipo especializa as Classes e os Primitivos como mostra a FIGURA 3.22b. O Tipo Classe, por exemplo, modela as Metaclasse, classe Utility, entre outras. O Tipo Primitivo modela as *strings*, os *integer*s, os *booleans* e demais tipos.

Atributos:

nome: Especifica o nome do Tipo. Ver Nome.

Semântica:

a) Tipo é uma classe abstrata que especifica os tipos da notação.

Estereótipo: É um mecanismo que permite a extensão da notação. Os conceitos que podem ser estendidos agregam o atributo *estereótipo*. Este receberá uma *string*, escrita entre os símbolos <<>>, que descreve o novo conceito criado. Por exemplo, na modelagem de um sistema, a especificação das interfaces pode ser feita através de uma extensão de classe - classe interface. Na identificação destas classes pode ser usado um adorno (Figura Geométrica) ou a palavra <<*interface*>> na parte superior do compartimento do nome da classe, como é mostrado na FIGURA 3.23.

FIGURA 3.23 – Ilustração do Adorno que Permite a Visualização do Estereótipo da Classe.

Atributos:

nome: Especifica a *string* que caracteriza a extensão. Ver Nome.

Expressão: É uma classe genérica que especializa **Restrição Temporal**, **Restrição Boolean** e **Procedure** (código de um método). Especificações de expressão são empregadas para definir restrições ou alguma semântica em qualquer conceito presente nos modelos.

Atributos:

linguagem: Especifica qual o padrão adotado para descrever a restrição ou a semântica. As bibliografias que descrevem notação UML-RT utilizam a linguagem OCL. Ver Nome.

corpoRestrição: Especifica o corpo da restrição. Ver Texto.

Semântica:

a) Expressão é uma classe genérica e não pode ser instanciada.

b) Uma Expressão é uma Restrição Temporal, uma Restrição *Boolean* ou uma *Procedure*.

3.1.5.2 Descrição das Restrições em OCL

Visibilidade:

a) Visibilidade é uma classe genérica e não pode ser instanciada;

Visibilidade

```
self.allInstances -> select(oclType = Visibilidade) -> isEmpty.
```

b) Visibilidade é uma classe abstrata que generaliza as classes Público, Privado, Protegido e Implementação.

Visibilidade

```
(self.oclIsKindOf(Publico)= true) or
(self.oclIsKindOf(Privado)= true) or
(self.oclIsKindOf(Protegido)= true) or
(self.oclIsKindOf(Implementação)= true).
```

Tipo:

a) Tipo é uma classe abstrata que especifica os tipos da notação.

Tipo

```
self.allInstances -> select(oclType = Tipo) -> isEmpty.
```

Expressão:

a) Expressão é uma classe genérica e não pode ser instanciada;

Expressão

```
self.allInstances -> select(oclType = Expressão) -> isEmpty.
```

b) Uma Expressão é uma Restrição Temporal, uma Restrição *Boolean* ou uma *Procedure*.

Expressão

(self.oclIsKindOf(Restrição Temporal)= true) or

(self.oclIsKindOf(Restrição Boolean)= true) or

(self.oclIsKindOf(Procedure)= true).

3.1.5.3 Descrição da Modelagem do Nome e da Multiplicidade

Multiplicidade é uma definição implementada pelo conceito de Classe e pelo conceito de Terminador. Estes conceitos implementam os atributos chamados de, respectivamente, *instância* na Classe e *cardinalidade* no Terminador.

As semânticas da multiplicidade para os conceitos de Classe e de Terminador são diferentes. O atributo *cardinalidade* do conceito Terminador mostra quantas instâncias da classe Sensor estão associadas a um único objeto Monitor da classe Monitor, por exemplo. Em outras palavras, mostra o peso do relacionamento.

O atributo *instância* do conceito Classe indica quantos objetos serão instanciados a partir desta classe. Por exemplo: uma classe chamada Sensor especifica *instância=3*. Isto significa que a classe Sensor tem três objetos instanciados a partir dela, que poderiam ser: objeto sensor_temperatura, objeto sensor_umidade e objeto sensor_pressão.

O conceito Nome define as especificações de um conjunto de caracteres que farão a descrição dos nomes de todas as classes de conceitos que implementam um atributo chamado *nome.* O tipo Nome foi definido prevendo-se a necessidade de haver mudanças nos nomes dos conceitos descritos nas seções 3.1.3 e 3.1.4. Isto deverá ser feito em um único local: na definição Nome.

Todos os conceitos comentados até aqui e mapeados na FIGURA 3.24 são expostos na seqüência. A semântica dos mesmos é descrita em linguagem natural e posteriormente formalizada em OCL.

Nome: Especifica os nomes de cada conceito presente no modelo.

Atributos:

nome: Especifica a própria *string* nome.

Multiplicidade: Especifica tanto o número de instâncias, que estão interligadas através de um relacionamento, quanto o número de instâncias de uma classe. Deste modo, os conceitos de Classe e Terminador implementam respectivamente os atributos chamados *instância e cardinalidade.*

Semântica:

a) A Multiplicidade é composta por um Limite.

b) A Multiplicidade é composta por dois ou mais Limite Multiplicidade se o conceito trabalhado for relacionamento (terminadores).

Limite Multiplicidade: Especifica um limite máximo e mínimo para a variação da multiplicidade.

Atributos:

mínimo: uma *string* que especifica o limite mínimo;

máximo: uma *string* que especifica o limite máximo.

Semântica:

a) A multiplicidade máxima de um relacionamento (*cardinalidade* do Terminador) não pode ser maior que a multiplicidade máxima da classe (*instância*).

b) A multiplicidade mínima nunca pode ser menor que a multiplicidade máxima.

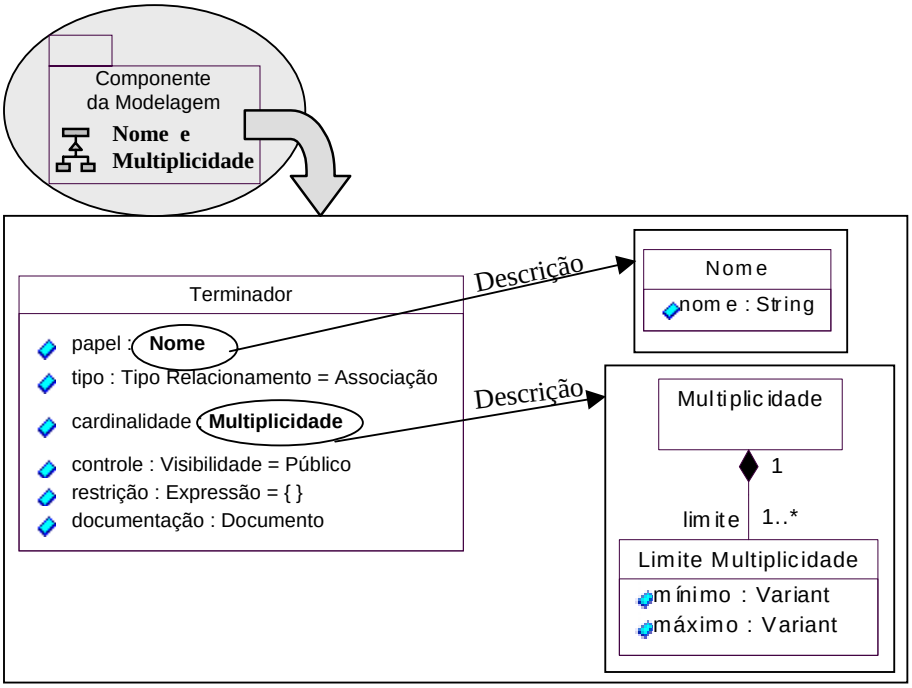


FIGURA 3.24 – Modelagem do Nome e da Multiplicidade.

3.1.5.4 Descrição das Restrições em OCL

Multiplicidade:

a) A Multiplicidade é composta por um Limite Multiplicidade.

Multiplicidade

```
self.limite –> size =1
```

b) A Multiplicidade é composta por dois ou mais objetos Limite Multiplicidade se o conceito trabalhado for um relacionamento (terminadores).

Multiplicidade

```
if Relacionamento.oclIsKindOf(Relacionamento) = true then
  self.limite –> size >=1
endif.
```

Limite Multiplicidade:

a) A multiplicidade máxima de um relacionamento (*cardinalidade* do Terminador) não pode ser maior que a multiplicidade máxima da classe (*instância*).

Modelagem Estática::Terminador

self.cardinalidade <= Modelagem Estática::Concreta.Instancia

b) A multiplicidade mínima nunca pode ser menor que a multiplicidade máxima.

Limite Multiplicidade

self.mínimo <= self.máximo.

3.1.5.5 Descrição da Modelagem da Persistência

Ao modelar um diagrama de classes, de objetos ou de interação, os conceitos de Classe ou de Objeto devem descrever sua persistência. A Persistência pode ser conceituada como o período de existência de um objeto. Este pode ser transiente quando seu período de existência é no máximo o mesmo período de execução do programa que o criou, ou persistente quando seu período de existência é maior que o período de execução. Neste casos, a persistência é feita em disco e gerenciada por ferramentas especializadas em persistência dos objetos, como os bancos de dados orientados a objetos ODBMS (*Object Data Base Management System*).

Para especificar Persistência, os conceitos de Classe e Objeto, modelados na FIGURA 3.8, implementam um atributo chamado *persistência*. Este atributo é do tipo Persistência, classe genérica do diagrama **Modelagem da Persistência**.

Na FIGURA 3.25 é mostrado o modelo resultante do mapeamento do conceito Persistência. Posteriormente à figura, serão expostos, de forma detalhada, estes conceitos. Sua semântica é descrita em linguagem natural e posteriormente formalizada em OCL.

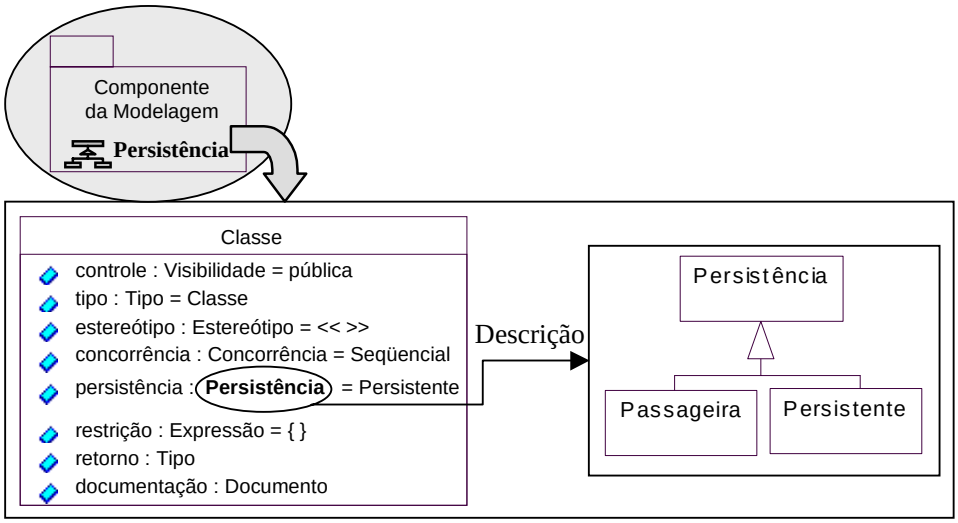


FIGURA 3.25 – Modelagem da Persistência.

Persistência: define o período de vida das instâncias de uma classe. É esperado que um elemento persistente tenha um tempo de vida além daquele do sistema que o criou. Um exemplo de instância persistente pode ser dado através do objeto Empresa KXY da classe Pessoa Jurídica. Nem todos os dados manipulados pelos sistemas são persistentes. Alguns podem ser caracterizados como transientes ou passageiros.

Dados transientes são aqueles cuja existência está associada ao tempo em que o sistema está sendo executado. Geralmente são informações obtidas através de cálculos.

A persistência de um Objeto deve ser compatível com a persistência especificada para sua Classe. Se a persistência da Classe é definida como Persistente; então, a persistência do Objeto deve ser Persistente ou Passageira. Porém, se a persistência da Classe é definida como Passageira; logo, a persistência do Objeto é Passageira.

Semântica:

a) Persistência é uma classe genérica e não pode ser instanciada.

b) Persistência é uma classe abstrata que generaliza as classes Passageira e Persistente.

Persistente: É a especificação *default*. O período de vida do objeto supera o da execução do sistema que o criou.

Passageiro: O período de vida do objeto é no máximo o mesmo da execução do sistema que o criou.

3.1.5.6 Descrição das Restrições em OCL

Persistência:

a) Persistência é uma classe genérica e não pode ser instanciada.

```
Persistência
self.allInstances -> select(oclType = Persistência) -> isEmpty.
```

b) Persistência é uma classe abstrata que generaliza as classes Passageira e Persistente.

```
Persistência
(self.oclIsKindOf(Passageira)= true) or
(self.oclIsKindOf(Persistente)= true).
```

3.1.5.7 Descrição da Modelagem da Documentação dos Conceitos

A documentação é um tópico de relevante importância ao modelar um sistema, seja ele convencional ou de tempo real. E imprescindível que cada descrição feita fique devidamente documentada. Isto facilita o entendimento do que está sendo feito pelos demais componentes da equipe de desenvolvimento.

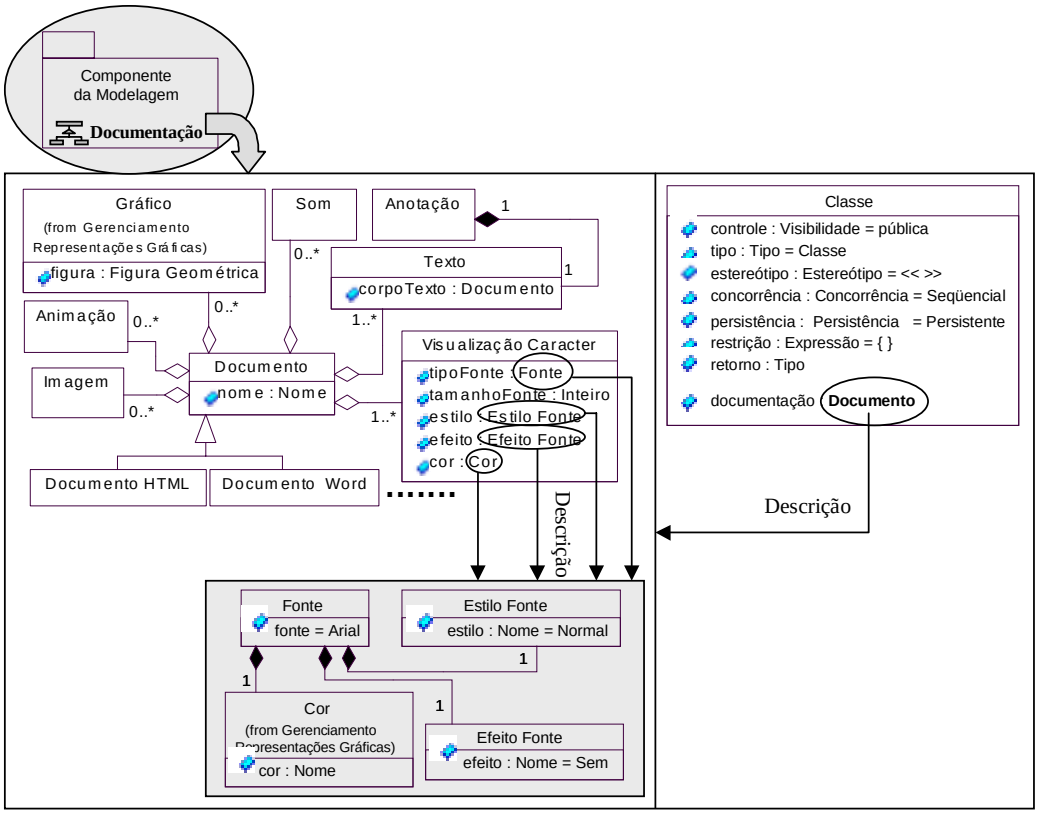


FIGURA 3.26 – Modelagem da Documentação dos Conceitos.

Uma vez que a utilidade da documentação é valida e importante durante todo o ciclo de vida de um sistema, buscou-se inserir nos conceitos descritos nas seções 3.1.3 e 3.1.4 uma forma de documentação.

A descrição da documentação se dá através da implementação de um atributo chamado *documentação.* Este atributo é do tipo Documento, mapeado para um metamodelo do mesmo. O modelo de conceitos resultante deste mapeamento pode ser visualizado na FIGURA 3.26. Após a figura, encontram-se expostos estes conceitos. Sua semântica é descrita em linguagem natural e posteriormente formalizada na linguagem OCL.

Documento: Permite especificar uma documentação para os conceitos do modelo. Documentação é um item importante do modelo pois facilita o seu entendimento pela equipe do projeto. A documentação é feita através do atributo *documentação* que é do tipo Documento. Documento é uma das classes presentes no diagrama **Modelagem da Documentação dos Conceitos**, FIGURA 3.26.

Um objeto Documento pode ser um documento descritivo qualquer. Seja feito em HTML ou pelo processador de textos Microsoft Word, tem especificado uma ou mais formas de visualização do seu conjunto de caracteres, como tipo de fonte, tamanho da fonte, cor, e outros. Podem ser agregadas imagens, animação, figuras e sons.

Atributos:

nome: Especifica o nome do Documento. Ver Nome.

Semântica:

a) Documento é uma classe genérica e não pode ser instanciada.

b) Documento é uma classe genérica que especializa documentos como os Documentos HTML e os Documentos criados pelo *Word*.

c) Documento pode ou não agregar objetos Imagem, Animação, Gráfico e Som.

d) Um Documento é composto de pelo menos um corpo do texto (Texto).

e) Um Documento pode ter diferentes tipos de fontes (Visualização Caracter).

Imagem: Permite especificar uma Imagem que compõe o documento.

Animação: Permite especificar diversas animações no documento.

Gráfico: Permite especificar figuras como parte do documento. Ver Figura Geométrica.

Som: Permite especificar sons no documento.

Texto: É a especificação do texto do documento.

Atributos:

corpoTexto: É o corpo do documento. Ver Documento.

Visualização Caracter: É a especificação visual do texto do documento.

Atributos:

tipoFonte: Especifica a fonte dos caracteres do texto, por exemplo, a fonte "Arial". Ver Fonte;

tamanhoFonte: Especifica o tamanho da fonte, como, por exemplo, "tamanhoFonte = 12";

estilo: Permite especificar um estilo para fonte, como, por exemplo, "Negrito". Ver Estilo Fonte;

efeito: Permite especificar um efeito para fonte, como sublinhado, riscado, e outros. Ver Efeito Fonte;

cor: Especifica a cor da fonte. Ver Cor.

Fonte: É a especificação das fontes nas quais o texto do documento foi escrito.

Atributos:

fonte: Especifica a fonte do texto, como, por exemplo, a fonte "Arial".

Semântica:

a) A especificação de uma fonte é composta por um Estilo Fonte, um Efeito Fonte e uma Cor.

Estilo Fonte: É a especificação do estilo da fonte.

Atributos:

estilo: Especifica o estilo utilizado, como, por exemplo, normal, negrito, itálico ou outros.

Efeito Fonte: O efeito é aplicado ao desejar uma fonte com características de sublinhado, riscado, riscado duplo, sobrescrito ou outro qualquer.

Atributos:

efeito: Especifica o efeito da fonte.

Cor: É a especificação da cor da fonte.

Atributos:

cor: Especifica a cor da fonte.

3.1.5.8 Descrição das Restrições em OCL

Documento:

a) Documento é uma classe genérica e não pode ser instanciada.

Documento

self.allInstances -> select(oclType = Documento) -> isEmpty

b) Documento é uma classe genérica que especializa documentos, como, Documentos HTML ou até Documentos no formato Microsoft Word.

Documento

(self.oclIsKindOf(Documentos HTML)= true) or

(self.oclIsKindOf(Documentos Word)= true).

c) Documento pode ou não agregar objetos Imagem, Animação, Gráfico e Som.

Documento

(self.imagem –> size >=0) and

(self.animação –> size >=0) and

(self.gráfico –> size >=0) and

(self.som –> size >=0).

d) Um Documento é composto de pelo menos um corpo do texto (Texto).

Documento

self.texto –> size >=1.

e) Um Documento pode ter diferentes tipos de fontes (Visualização Caracter).

Documento

self.visualização caracter –> size >=1.

Fonte:

a) A especificação de uma fonte é composta por um Estilo Fonte, um Efeito Fonte e uma Cor.

Fonte

self.allInstances –> forAll ((self.cor –> size =1) and

```
(self.efeito fonte –> size =1) and
(self.estilo fonte –> size =1)).
```

4 Prototipação

Neste capítulo serão descritas as etapas que foram seguidas para a prototipação do modelo proposto na seção 3.1. Primeiramente serão vistas as regras utilizadas para fazer o mapeamento do modelo de classes para o modelo ER. Sendo que a notação utilizada na construção deste modelo foi a notação disponível na ferramenta *Oracle Designer*. Também foram utilizadas instruções de como construir um modelo ER presentes nos manuais para desenvolvimento de software estruturado proposto pela NTS. Estas instruções se encontram disponíveis nos volumes I e II de [NTS 90]. Finalmente, será feita uma descrição do SiMOO-RT e explanado como este funcionará integrado ao dicionário de dados proposto.

Na FIGURA 4.1 são mostrados, de forma resumida, os passos que foram sendo seguidos nesta pesquisa. O modelo conceitual projetado e as descrições semânticas em OCL vistos na seção 3.1. O mapeamento do metamodelo para um modelo ER e deste ER para *scripts* DDL, mostrados no Anexo2 . Estes *scripts* serão usados na criação do dicionário de dados no banco de dados Oracle. As regras OCL são mapeadas para *triggers*, que disparam no momento em que o dicionário de dados é manipulado.

O MET Editor do SiMOO-RT, será descrito na seção 4.1.3, é a ferramenta diagramática que faz o povoamento dos dados no dicionário de dados, com base na arquitetura em dois níveis cliente/servidor.

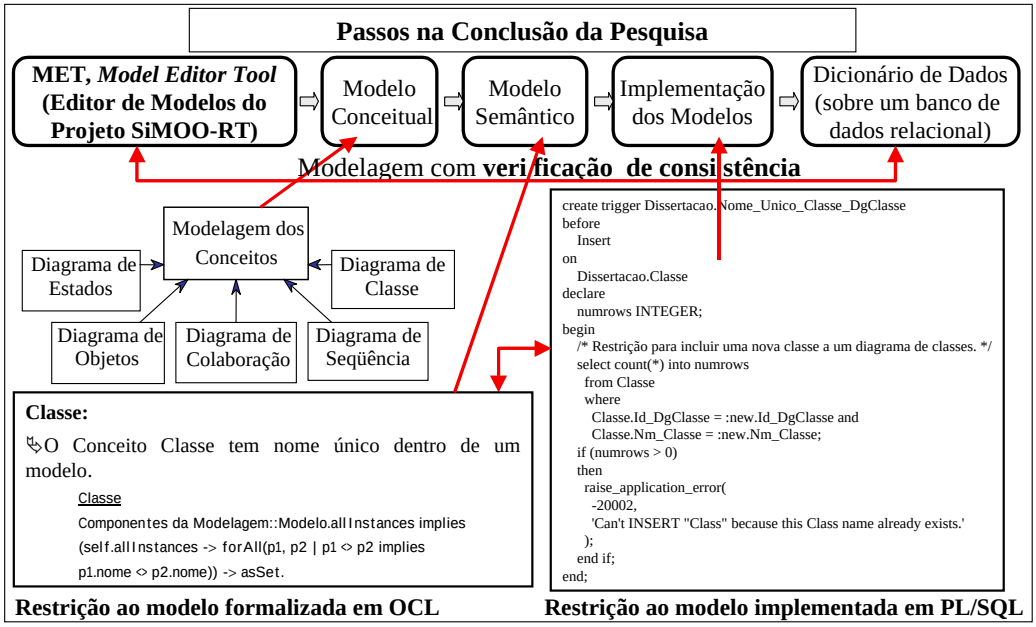


FIGURA 4.1 – Metamodelo e sua Prototipação no SiMOO-RT

Cliente/servidor é uma arquitetura versátil, surgiu com o objetivo de melhorar a usabilidade, flexibilidade, interoperabilidade e escalabilidade das aplicações. O cliente pode ser definido como o agente solicitante de serviços e servidor como o agente fornecedor de serviços. Uma máquina pode assumir o papel de cliente ou servidor, ou ambos, e a comunicação entre cliente/servidor é normalmente feita através de RPC(*Remote Procedure Call*) ou declarações SQL.

A arquitetura em dois níveis, na qual está estruturada o MET e o Dicionário de dados, é a divisão da aplicação em duas partes, como mostra a FIGURA 4.9. O ponto de divisão do processamento entre cliente/servidor é relativo a cada aplicação.

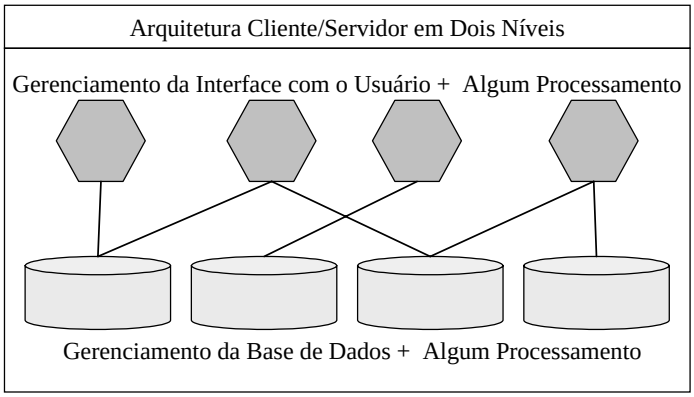


FIGURA 4.2 – Arquitetura Cliente/Servidor em Dois Níveis

Ao se utilizar esta tecnologia, obtem-se algumas vantagens, tais como: a base de dados independente patrocinando maior escalabilidade e flexibilidade; as ferramentas GUI (*Grafic User Interface*) permitem desenvolver a aplicação mais rapidamente e amigáveis ao usuário; os servidores já não necessitam de recursos em demasia; dentre outros.

4.1 Mapeamento do Modelo de Classes para o ER e a sua Implementação no Oracle

Existem diversas alternativas de mapeamento das classes e seus relacionamentos para tabelas em um banco de dados. Entretanto, além destas alternativas, é necessário acrescentar detalhes às tabelas que as classes não implementam. As chaves primárias e as chaves estrangeiras, que darão suporte à integridade do DD no SGBDR, são exemplos de informações que necessitam ser acrescentadas. A seguir, serão abordadas estas alternativas propostas por Rumbaugh [RUM 97], adotadas para mapear o metamodelo em um modelo de entidades e seus relacionamentos, ER, do dicionário de dados (DD).

Cada entidade do modelo deve possuir um elemento denominado de chave primária. Uma chave primária é um identificador formado por um ou mais atributos que localizarão uma única linha de dados, também chamada de tupla, na tabela.

Para cada tabela mapeada, a partir de uma classe do metamodelo, foi criado um identificador correspondente. Por exemplo, a tabela RELACIONAMENTO, mapeada a partir da classe Relacionamento, tem o identificador ID_RELACIONAMENTO. Da mesma forma a tabela TERMINADOR contém o seu identificador denominado ID_TERMINADOR, como mostra a FIGURA 4.3. As demais tabelas e seus identificadores podem ser consultados na seção 4.1.1 ou através das suas DDL (*Data Description Language*) para o dicionário de dados na seção 2 dos anexos.

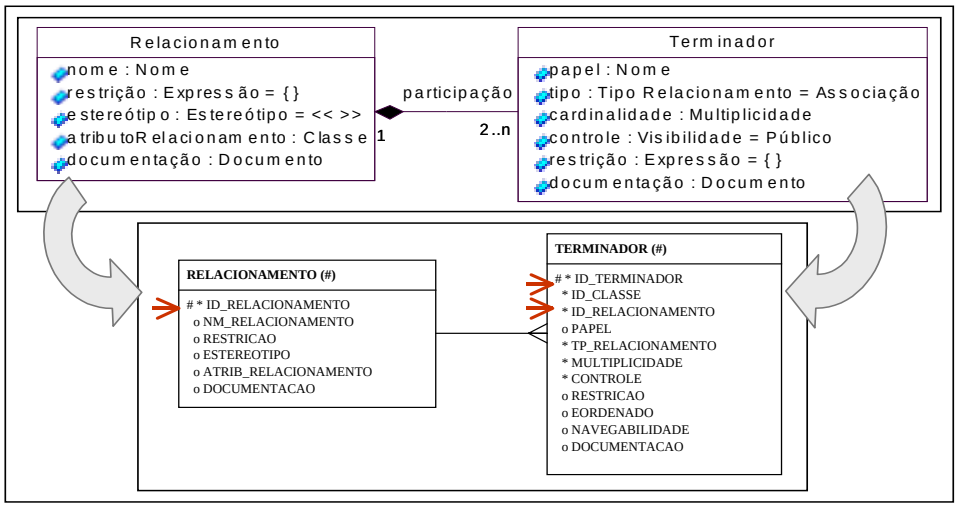


FIGURA 4.3 – Identificador para Chave Primária nas Tabelas Derivadas das Classes

Na situação em que as classes especializadas no relacionamento generalização herdam características de uma única classe genérica, diz-se que esta é uma herança simples. Para as heranças simples o mapeamento poderá ser uma tabela para cada classe.

Existem abordagens distintas que podem ser utilizadas para o mapeamento do relacionamento de generalização-especialização. Cada uma delas será comentada a seguir.

Em uma primeira abordagem, a tabela originária do mapeamento da classe genérica terá um identificador para localizar o subtipo. Estes subtipos correspondem às ocorrências nas tabelas, mapeadas a partir das classes especializadas neste tipo de relacionamento. As tabelas oriundas do mapeamento das classes especializadas terão a mesma chave primária que a tabela oriunda do mapeamento da classe genérica. Porém este tipo de abordagem incorre em uma desvantagem. O número de tabelas criadas é bastante elevado e isto pode tornar a navegação morosa, ou seja, o caminhamento entre os tipos e seus subtipos pode se tornar muito lento.

Em uma segunda abordagem, cada subclasse ou classe especializada será mapeada para uma tabela. Nesta abordagem, a classe genérica não será mapeada, pois as subclasses agregarão os atributos oriundos da classe genérica. Efetuar o mapeamento mediante esta abordagem, traz como vantagem a eliminação da navegação entre várias tabelas. E este fato se torna interessante quando se tem poucos atributos na classe genérica e muitos, nas subclasses.

A terceira abordagem para o mapeamento é praticamente o inverso da segunda. Nela, as subclasses deixam de existir e seus atributos são mapeados para a tabela oriunda da classe genérica. Esta abordagem se torna interessante quando o número de subclasses é inferior a três e possuem poucos atributos. Sua principal desvantagem consiste em violar a terceira forma normal. Entretanto esta terceira abordagem é utilizada no mapeamento de algumas ocorrências do relacionamento de generalização do metamodelo para o ER. No exemplo da FIGURA 4.4, a classe Classe, modelada na FIGURA 3.8 da seção 3.1.3.1, especializa as classes Concreta e Abstrata. Estas classes foram mapeadas para a tabela CLASSE. Os atributos

instância e *ativa* da classe Concreta passaram a ser atributos da tabela CLASSE e a classe Abstrata passou a ser o atributo booleano EABSTRATA da tabela mapeada.

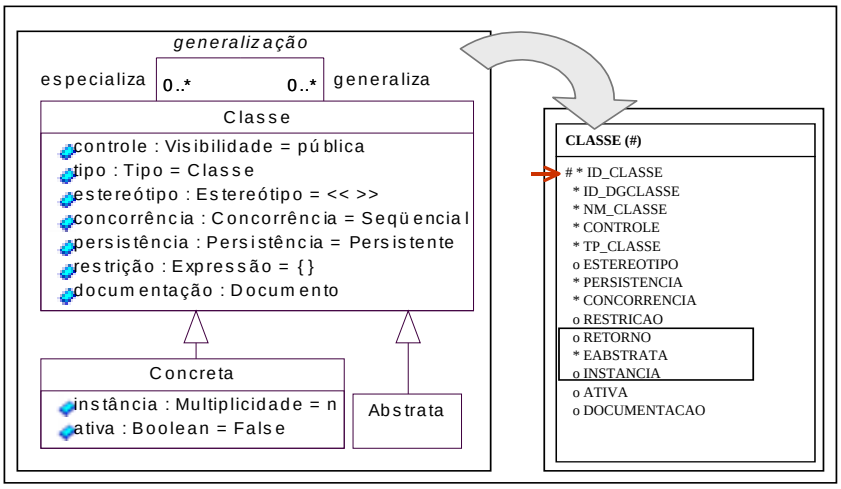


FIGURA 4.4 – Mapeamento do Relacionamento de Generalização

Ainda dentro do contexto do mapeamento dos conceitos presentes na FIGURA 3.8, a classe Atributo e suas subclasses - Atributo Objeto e Atributo Classe - foram mapeadas para a tabela ATRIBUTO no ER. Estas subclasses originaram os atributos ID_CLASSE e ID_OBJETO. Através deles será feita a identificação se o atributo é de classe ou de objeto.

De maneira semelhante, foi feito o mapeamento das classes Definição Método e suas subclasses Método Concreto e Método Abstrato. A classe Método Concreto especializa Método Concreto Objeto e Método Concreto Classe. O mesmo ocorre com a classe Método Abstrato que especializa Método Abstrato Objeto e Método Abstrato Classe vistas na FIGURA 3.9. Então, a classe Definição Método foi mapeada para a tabela DEFINIÇÃO_METODO. As subclasses Método Concreto e Método Abstrato foram mapeadas para o atributo ECONCRETO. Este é um atributo do tipo booleano. Se verdadeiro (*True*), o método é, então, dito concreto, caso contrário, é abstrato. Este mapeamento está à mostra na seção 4.1.1.

Na seqüência do mapeamento das classes presentes na FIGURA 3.9 para o ER, as classes Execução Método, suas subclasses Periódico e Aperiódico, juntamente com as classes *Jitter*, Tempo Execução, *Deadline*, *Deadline Hard*, *Deadline Soft*, *Deadline Firm*, foram mapeadas para a tabela EXECUÇÃO_MÉTODO. Neste mapeamento foi violada a terceira forma normal. Em contrapartida, o desempenho é melhorado uma vez que a busca pelos dados da execução do método é feita em uma única tabela.

A classe Mensagem e suas subclasses Mensagem Simples, Mensagem Concorrente e Mensagem *Broadcast* da FIGURA 3.16 foram mapeadas para a tabela MENSAGEM. Da mesma maneira, a classe Estado e suas subclasses vistas na FIGURA 3.18 foram mapeadas para a tabela ESTADO.

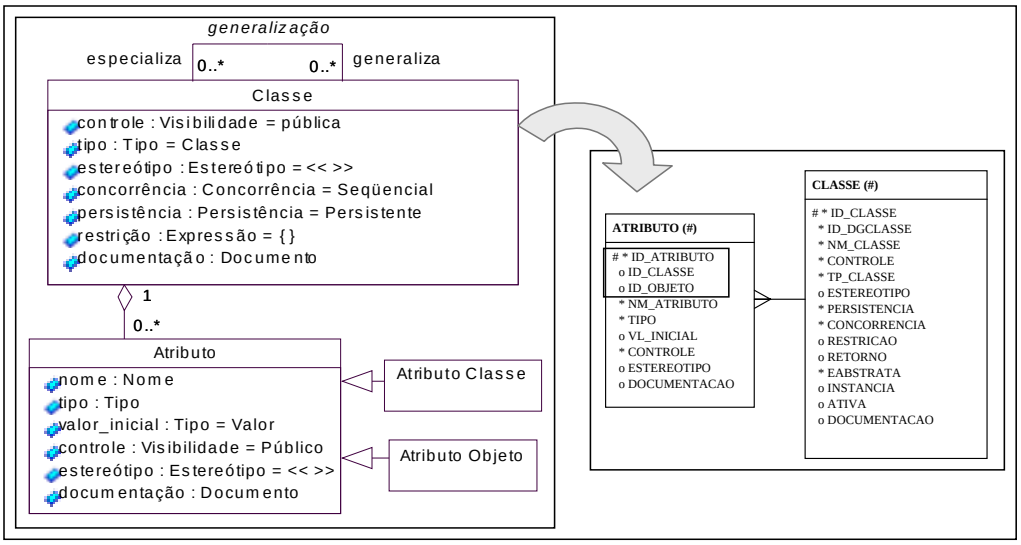


FIGURA 4.5 – Mapeamento do Relacionamento Associação e Agregação.

As regras utilizadas para mapear o relacionamento de associação são as mesmas utilizadas para mapear o relacionamento de associação por agregação.

Uma associação poderá ou não ser mapeada para uma tabela. Esta dependência está atrelada à multiplicidade do relacionamento. Outros tópicos como número de tabelas, desempenho no acesso ou flexibilidade quanto a futuras extensões também poderão influenciar esta decisão.

Quando uma associação é do tipo **muitos para muitos,** sempre deverá ser mapeada para uma tabela. As chaves primárias de cada uma das tabelas envolvidas no relacionamento, mais os possíveis atributos do relacionamento, formarão a tabela associativa. Com isso, a terceira forma normal nesta situação é satisfeita.

Quando uma associação é do tipo **um para muitos,** geralmente a chave primária da tabela, cujo terminador é **um**, estará mapeada como chave estrangeira na tabela em que a associação é **muitos**.

Nas situações em que a cardinalidade de uma associação for **um para um,** as tabelas pode ser reunidas em uma só. A incorporação deste relacionamento em uma única tabela melhora o desempenho e reduz as necessidades de armazenamento. Em contrapartida, há custos de capacidade de extensão e uma possível violação da terceira forma normal.

Deve ficar claro que as associações e associações por agregação, presentes no metamodelo que tiveram de ser mapeadas, tinham cardinalidade **um para muitos** ou **um para um**. Aquelas que implementavam cardinalidade **um para muitos** seguiram o exemplo mostrado na FIGURA 4.5 em que a chave primária, neste exemplo ID_CLASSE, da tabela, neste caso CLASSE, cujo terminador é **um,** estará mapeada como chave estrangeira na tabela, neste exemplo ATRIBUTO, em que a associação é **muitos**.

Durante o mapeamento do metamodelo para o ER do dicionário de dados, nas ocorrências em que a cardinalidade do relacionamento de associação era **um para um**, seu mapeamento foi feito em uma única classe. No mapeamento do relacionamento *especifica*, que descreve que um método Aperiódico agrega uma ou

nenhuma especificação de Tempo Execução, juntamente com o *Deadline* mostrado na FIGURA 3.9, foram mapeadas para uma única tabela. Neste caso, a EXECUÇÃO_MÉTODO.

4.1.1 Modelo ER

O resultado do mapeamento do metamodelo da notação UML-RT é o modelo ER. Este modelo foi descrito usando a ferramenta Case do Oracle – Oracle Designer. A notação usada para descrever o ER pode ser consultada em [NTS 90] e basicamente é constituída de um retângulo que caracteriza uma tabela. Tanto o nome desta tabela quanto seus atributos são descritos usando-se letras maiúsculas e são colocadas dentro do referido retângulo. Entretanto para o nome da tabela além de maiúsculas são utilizados caracteres em negrito. O nome da tabela também possuiu um posicionamento específico dentro do compartimento superior da figura geométrica.

4.1.1.1 Simbologia Utilizada

A fim de poder descrever os atributos são utilizadas simbologias. Cada uma delas agrega um significado específico para o atributo. Cada um dos símbolos utilizados é apresentado a seguir:

- # : Indica que o atributo é uma chave primária, ou seja, o identificador das tuplas da tabela.
- *: Indica que o atributo não pode ser nulo (*NOT NULL*).
- #*: Significa que o atributo é chave primária e não pode ser nulo. Esta é uma simbologia utilizada pela ferramenta Case do Oracle de forma redundante, uma vez que o atributo da chave primária não pode ser nulo.
- °: Indica que o atributo pode ser nulo (*NULL*).

Além das simbologias apresentadas até aqui, é necessário abrir mão de mais alguns símbolos para poder descrever os relacionamentos no ER.

O relacionamento em que simbologia que une as duas tabelas é uma linha com um terminador denominado “pé-de-galinha” e está ilustrada na FIGURA 4.6. O seu significado indica que cada tupla da tabela, onde o terminador é uma linha, é obrigatoriamente referenciada por uma ou mais tuplas onde o terminador é “pé-de-galinha”. Isto indica que a chave primária da tabela um será chave estrangeira na tabela dois, levando-se em consideração que a segunda tabela poderá ter **N** ocorrências da mesma chave estrangeira.

FIGURA 4.6 – Terminador "pé-de-galinha" Chave Estrangeira não Pode Ser Nula

Porém, se a linha que une as duas tabelas for tracejada, como a mostrada na FIGURA 4.7, indica que cada tupla da tabela onde o terminador é uma linha tracejada pode ou não (não é obrigatório) ser referenciada por uma ou mais tuplas onde o terminador é “pé-de-galinha”.

FIGURA 4.7 – Terminador "pé-de-galinha" Chave Estrangeira Pode Ser Nula

4.1.1.2 A Descrição do Modelo ER

Para facilitar a compreensão do modelo ER e permitir que fosse mostrado na sua totalidade, foi necessário dividi-lo em partes. Esta divisão foi feita de acordo com as persistências dos diagramas, resultando em quatro partes distintas como é ilustrado na FIGURA 4.8.

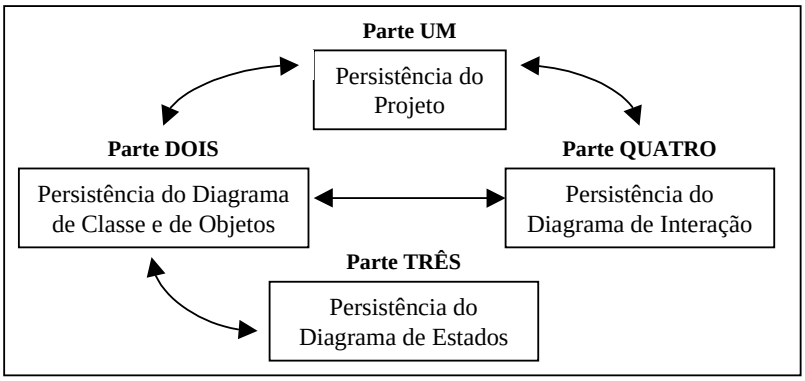


FIGURA 4.8 – Organização do Modelo ER.

A primeira parte trata da persistência das informações do projeto. Uma segunda parte mostra as tabelas referentes à persistência das informações sobre os diagramas de objetos e de classes. A terceira parte refere-se à persistência do diagrama de estados para as classes. A ultima destas partes mostra as classes que fazem a persistência dos diagramas de interação.

A FIGURA 4.9 mostra as tabelas que farão a persistência das informações de cada projeto desenvolvido, juntamente com sua documentação e diagramas que serão descritos.

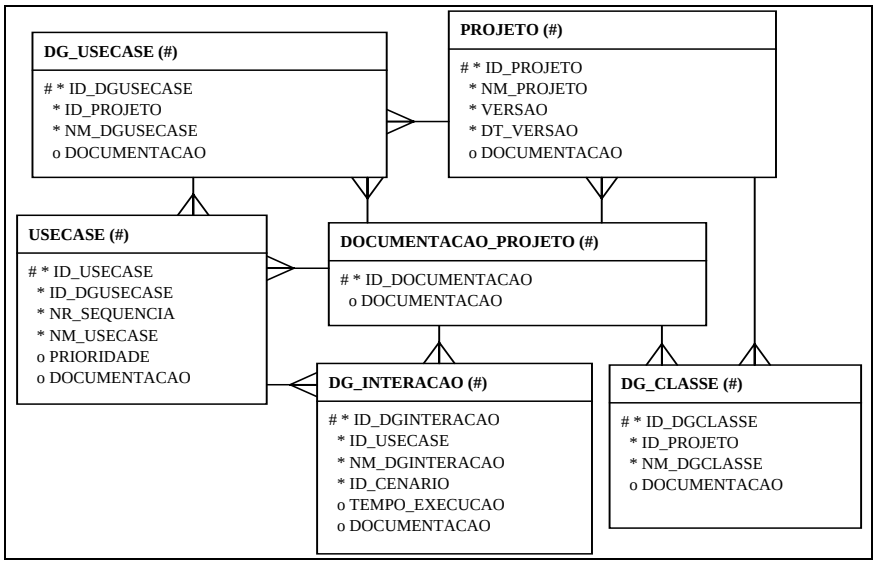


FIGURA 4.9 – Tabelas para Persistência das Informações do Projeto.

No metamodelo descrito na seção 3 não foi modelada a notação do diagrama Caso de Uso (*Use Case*). Através deste diagrama são descritas as funcionalidades do domínio do problema e sua interação com os Atores externos (entidades externas)

que interagem com domínio. Uma vez que os diagramas de interação descrevem uma funcionalidade do sistema, é necessário informar qual funcionalidade um determinado diagrama está descrevendo.

Para contornar isto, no modelo ER foram criadas as tabelas DG_USECASE e USECASE. A primeira faz a persistência das informações referentes ao diagrama Use Case e a segunda descreve as funcionalidades deste diagrama.

Cada diagrama de interação descreverá uma funcionalidade do sistema que está sendo projetado. Na FIGURA 4.9 é mostrada a tabela PROJETO associada à tabela DG_USECASE. Esta, por sua vez, está associada à tabela USECASE. Cada tupla da tabela DG_INTERACAO estará descrevendo uma funcionalidade que corresponde a uma tupla da tabela USECASE.

Um ponto positivo a ressaltar é o modo como estas duas tabelas foram descritas que permitirá às outras - com a persistência das demais informações do diagrama *Use Case* - serem também criadas no futuro.

Na FIGURA 4.10 estão as tabelas que foram mapeadas a partir dos conceitos modelados no pacote modelagem estática, FIGURA 3.7, da seção3.1.3. Esta figura também mostra as tabelas que farão a persistência de conceitos presentes no pacote componentes da modelagem da FIGURA 3.21, descritos na seção 3.1.5.

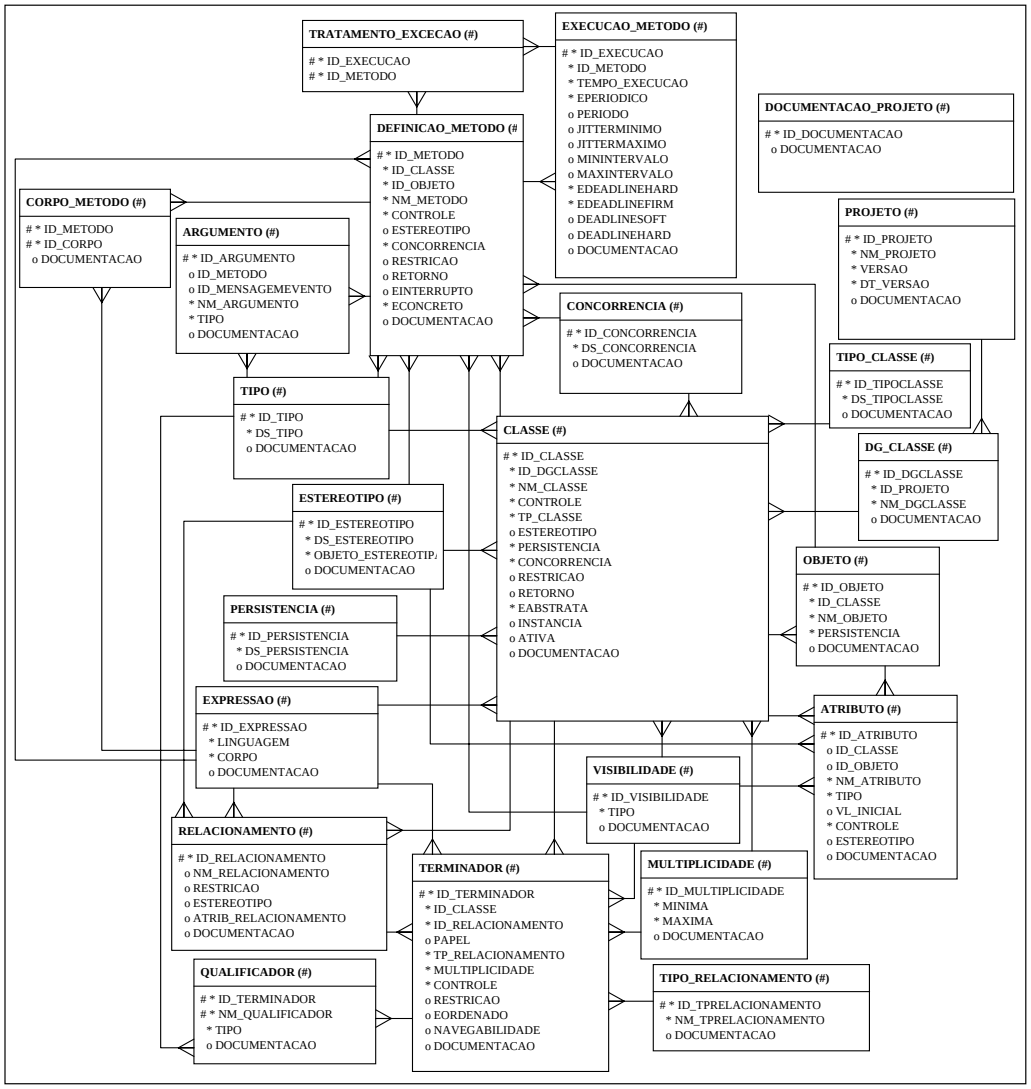


FIGURA 4.10 – Tabelas para a Persistência dos Diagramas de Classes e de Objetos.

Na FIGURA 4.11 são mostradas as tabelas que foram mapeadas a partir dos conceitos descritos nas seções 3.1.3 e 3.1.4. Estas serão responsáveis pela persistência dos possíveis diagramas de estados das classes do diagrama de classes.

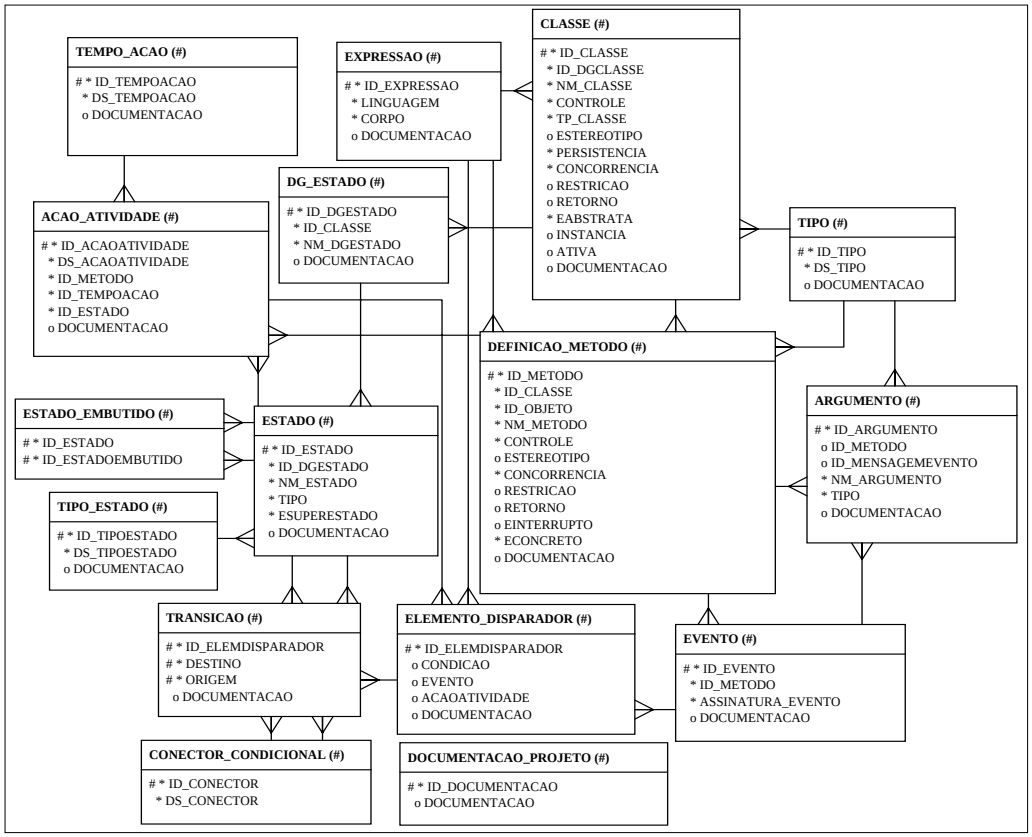


FIGURA 4.11 – Tabelas para a Persistência dos Diagramas de Estados das Classes.

Na FIGURA 4.12 estão as tabelas que serão responsáveis pela persistência dos diagramas de seqüência e de colaboração. Foram mapeadas a partir dos conceitos descritos nas seções 3.1.3 e 3.1.4.

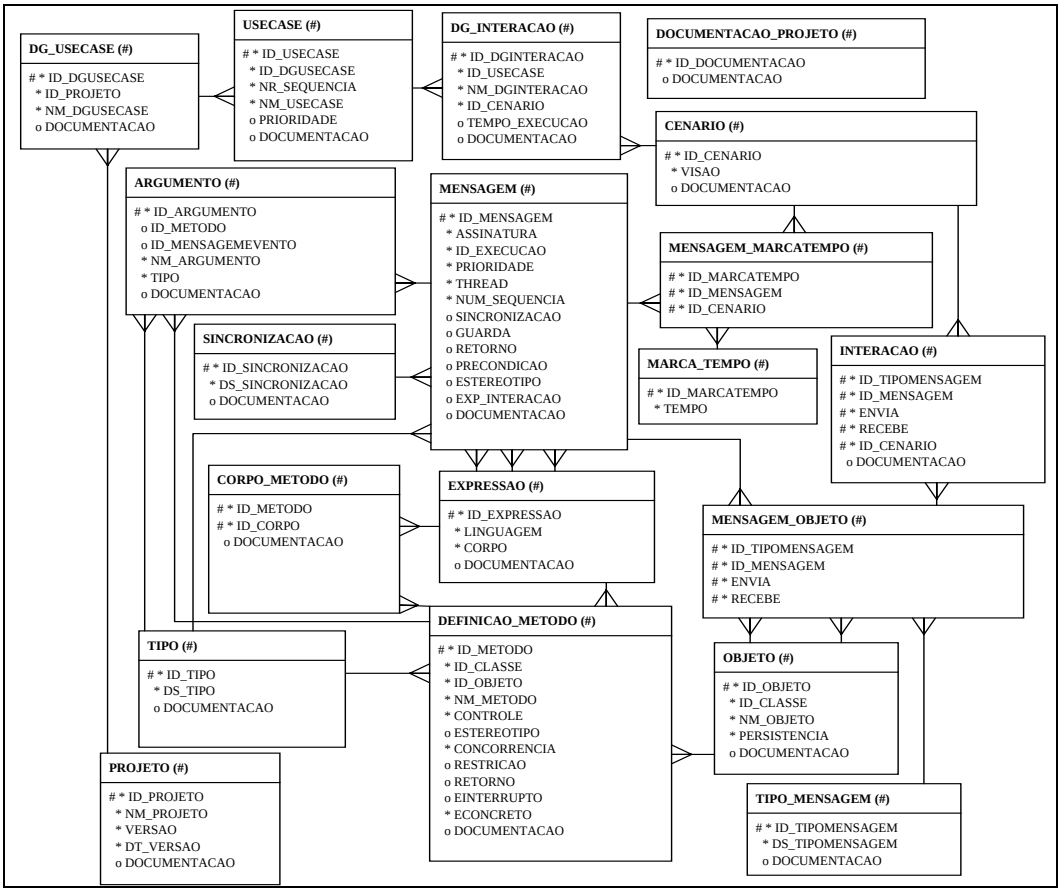


FIGURA 4.12 – Tabelas para a Persistência dos Diagramas de Interação.

4.1.2 Mapeamento das Regras de Consistência

Sobre os vários diagramas de classe que compõem o metamodelo, exposto na seção 3 , foram descritas regras em OCL, registradas com o objetivo de evitar ambigüidades no entendimento do metamodelo e descrever restrições cuja descrição gráfica não foi possível.

A semântica formulada em OCL tem como objetivo melhorar a precisão dos modelos criados com base nas descrições do metamodelo. Ou, sob outro ponto de vista, diminuir as inconsistências das instâncias geradas com base neste modelo conceitual.

No exercício de mapeamento do metamodelo para o modelo ER, feito na seção 4.1, foram descritas as tabelas que farão a persistência dos conceitos modelados pelas classes. Entretanto, as regras OCL não foram mapeadas. Conseqüentemente, o DD criado com base neste ER permitirá a ocorrência de inconsistências.

Para que isso seja evitado é necessário que as regras OCL também sejam mapeadas.

A maioria dos banco de dados implementam o conceito de gatilho (*trigger*). Baseado neste conceito, é possível descrever instruções a serem executadas nos momentos em que os dados do banco estejam sendo manipulados. Desta maneira, ao

inserir, excluir ou mesmo alterar uma determinada informação no banco de dados, instruções podem ser executadas para que a base de dados permaneça consistente.

Estes mecanismos de gatilho (*trigger*) funcionam da seguinte maneira:

SE <*condição*>

ENTÃO <*ação é executada*>

Estes gatilhos podem disparar funções ou procedimentos criados e compilados tanto em uma linguagem nativa do banco como em linguagens como C++ ou Java.

O banco de dados Oracle tem uma linguagem nativa PL/SQL na qual é possível a descrição de gatilhos, procedimentos e funções. Uma vez que a solução usada neste protótipo é este SGBD, as regras OCL, que foram mapeadas, utilizaram esta tecnologia.

4.1.3 Descrição da Ferramenta SiMOO-RT

A ferramenta SiMOO é especialmente desenvolvida para a construção de modelos de simulação interativos e visuais orientada a objetos. Através desse editor é possível descrever graficamente um modelo de simulação [JOR 98].

Esta ferramenta é composta por dois módulos básicos: a biblioteca de classes e o editor de modelos. O editor permite que se descreva graficamente a estrutura estática do modelo e a estrutura dinâmica em C++, utilizando os recursos definidos na sua biblioteca. Efetuada a descrição do modelo, o editor é capaz de gerar o código executável C++ que corresponde ao mesmo de maneira completa. Faz parte da biblioteca a interface padrão de interação com o modelo [COP 98]. A FIGURA 4.13 apresenta o esquema do SiMOO [JOR 98].

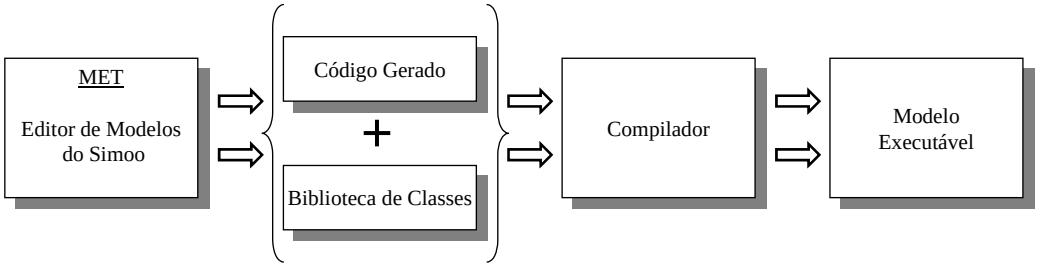


FIGURA 4.13 – Esquema geral de SiMOO.

MET - Model Editor Tool, é um editor gráfico e é um componente da ferramenta SiMOO. Modelos descritos com o MET são compostos por dois diagramas: o diagrama de classes e o diagrama de instâncias.

A especificação de uma entidade de simulação, no MET, sempre inicia pela sua definição no diagrama de classes - é similar aos diagramas tradicionalmente usados nas técnicas de análise orientada a objetos. Assim, permitem a descrição das classes que fazem parte de um sistema, bem como dos seus relacionamentos. O diagrama de instâncias é usado para especificar os elementos genéricos, descritos no diagrama de classes, e para definir a comunicação entre eles. Utilizando este editor, o programador se preocupa apenas em descrever os aspectos mais importantes sobre o sistema modelado e o MET gera código executável. A descrição do comportamento das entidades é feita usando-se rotinas definidas na biblioteca de classes do SiMOO.

O modelo gerado pode ser submetido aos processos de experimentação tradicionais [JOR 98].

SiMOO-RT [BEC 99, BEC 97] é uma extensão do SiMOO. Nesta extensão são disponibilizados conceitos e diagramas que contribuem para uma descrição mais completa dos modelos. A ferramenta SiMOO-RT implementa o diagrama de seqüência utilizando a notação da UML. Este aplicativo também disponibiliza mecanismos para a modelagem de restrições temporais, permitindo ao usuário modelar operações cíclicas e operações com *deadlines*.

A linguagem alvo trabalhada pelo SIMOO-RT é a AO/C++[BEC 99]. Esta foi a linguagem escolhida por disponibilizar comandos específicos para tratamento dos requisitos temporais. A linguagem trabalha com conceito de 'objetos ativos'[9], outro fator fundamental para sua escolha.

4.2 Integração do Dicionário de Dados com a Ferramenta SiMOO-RT

O SiMOO-RT foi desenvolvido na arquitetura em dois níveis, porque divide a aplicação em duas partes: interface e banco de dados.

A divisão do processamento, entre a interface e o banco de dados servidor, depende da forma que a aplicação foi projetada. No ambiente SiMOO-RT o servidor de banco de dados realiza a maior parte do processamento, constituído da lógica de acesso acrescida das regras-de-negócio. Conseqüentemente, o processamento da apresentação deste dados é feito na interface.

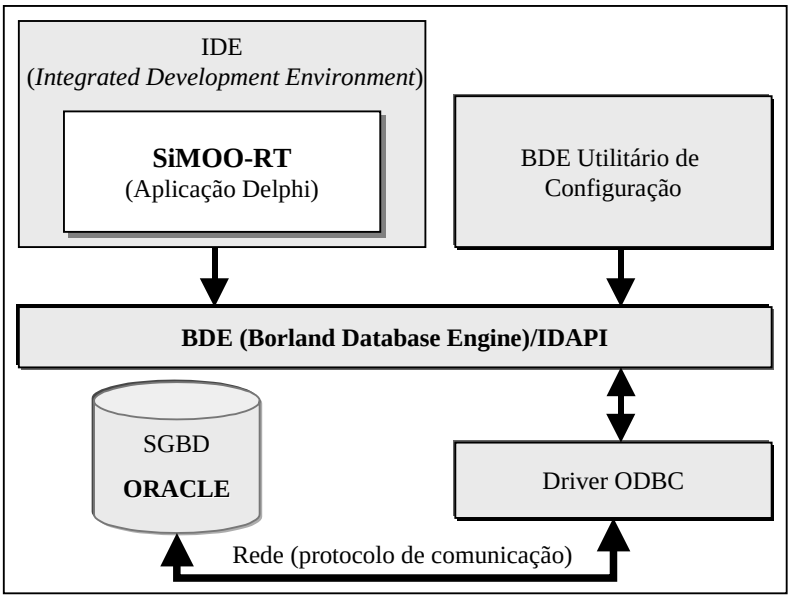


FIGURA 4.14 – Estrutura da Integração do SiMOO-RT com o DD.

Para implementar esta arquitetura foi utilizado o BDE (*Borland Database Engine*), também chamado IDAPI (*Integrated Database Application Program Interface*), juntamente com ODBC (*Microsoft Open Database Connectivity*), à mostra na FIGURA 4.14.

[9] Objetos ativos são objetos que tem sua própria *thread* de controle.

O BDE é um componente presente no ambiente de desenvolvimento de aplicações, Delphi da Borland. Este componente permite assistir os desenvolvedores de aplicações de banco de dados, funcionando como uma interface para múltiplos formatos de base de dados, como: Paradox, Oracle, Sybase e também ODBC. Isto significa que uma consulta a um conjunto de dados feita por uma aplicação Delphi é submetida ao banco via BDE. Esta consulta irá funcionar independente do SGBD em que os dados estão armazenados. A principal vantagem do BDE é que a aplicação não precisa ser reescrita se o banco de dados for modificado.

É necessário um mecanismo que faça a comunicação entre o BDE e o banco de dados. Este mecanismo é chamado de controlador (*driver*) de comunicação. Estes *drivers* podem ser nativos ao banco de dados. Isto significa que podem ser específicos para um determinado fornecedor de SGBD. Estes controladores, também, podem não ser nativos, como é o caso do ODBC, que permite a conexão com banco de dados de diferentes fornecedores.

No SiMOO-RT, para realizar a ligação entre o BDE e o DD implementado no banco de dados Oracle, foi utilizado o *driver* de comunicação ODBC, como mostra a FIGURA 4.14.

Anteriormente, a ferramenta SiMOO-RT armazenava os conceitos modelados, juntamente com a representação gráfica dos mesmos, em arquivos ASCII (*American Standard Code for Information Interchange*). Porém, para que a nova versão da ferramenta seja contemplada, é necessário que, além de gravar em arquivos ASCII, a mesma povoe o dicionário de dados.

A persistência nos dois sistemas: arquivos e banco de dados, é indispensável. Assim deve ser, uma vez que o dicionário de dados (banco de dados) ainda não está preparado para fazer a persistência das informações, referentes às várias representações gráficas que um conceito poderá ter. Atualmente, é feito a persitência dos conceitos em arquivos e dicionário de dados, sendo que a representação gráfica dos conceitos é somente armazenda em arquivos.

Uma vez concluída a versão protótipo em andamento, a ferramenta SiMOO-RT deverá armazenar as informações modeladas tanto dos conceitos quanto de suas respectivas representações, somente no dicionário de dados. Sobre este dicionário deverá estar uma interface, de forma que não só a ferramenta SiMOO-RT, mas outras - como, por exemplo, o EDI (Editor Diagramático na Internet) - poderão povoá-lo. Isto permitirá, também, que informações modeladas por outras ferramentas possam ser lidas pelo SiMOO-RT e vice-versa. Embora esta interface não seja o objetivo imediato deste trabalho, fica como sugestão a continuidade do mesmo em desenvolvimentos futuros.

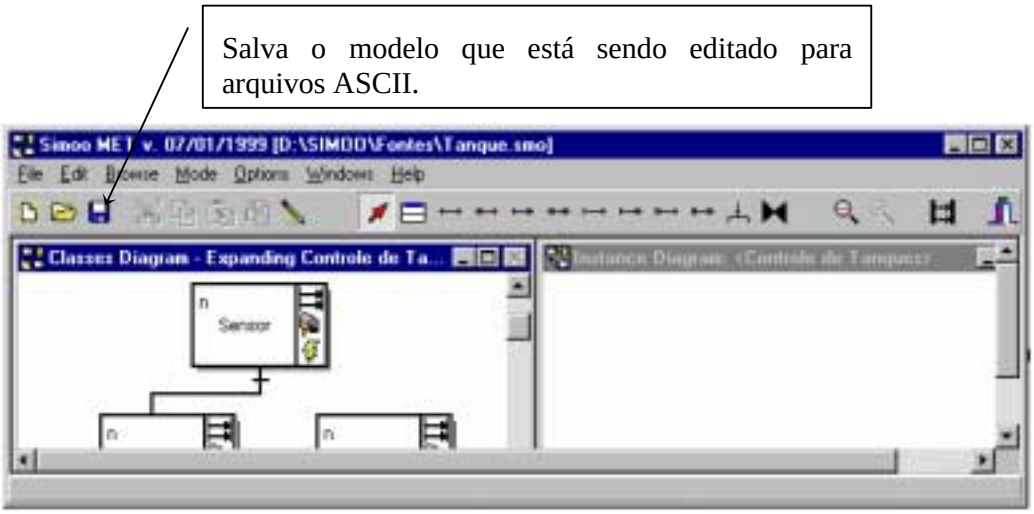


FIGURA 4.15 – Persistência em Arquivo ASCII.

A persistência nos arquivos ASCII é feita no momento em que o usuário da ferramenta for salvar seu trabalho de modelagem, clicando no ícone **Salvar,** da barra ícones , como mostra a FIGURA 4.15, ou no menu **File,** opção **Salvar.**

No banco de dados, a persistência é feita à medida que cada conceito é modelado. No exemplo da FIGURA 4.16, modelagem dos atributos, tanto de classe quantos os de objeto (neste exemplo, os atributos para classe Tanque), cada um destes é inserido no dicionário de dados no momento em que o botão **Insert** é clicado.

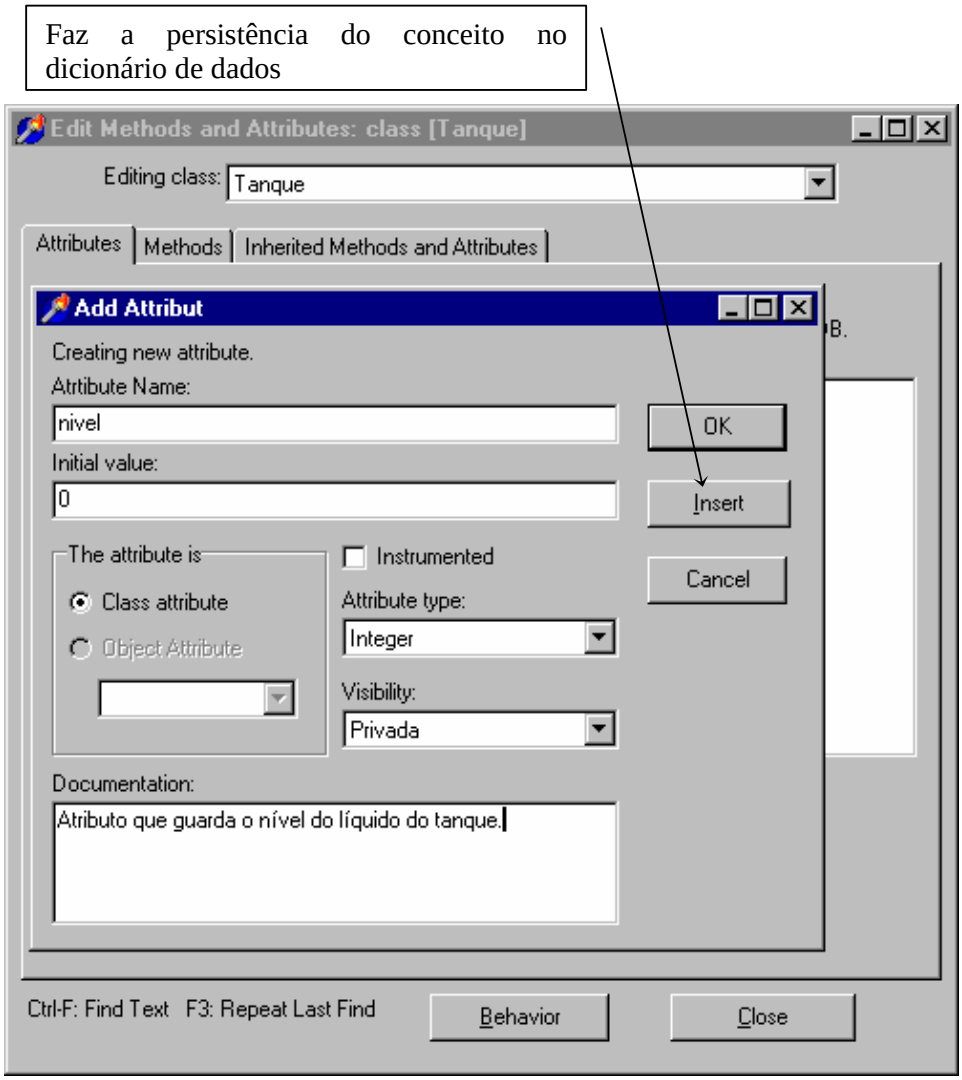


FIGURA 4.16 – Persistência no Dicionário de Dados.

No botão **Insert** é programado um código adequado, como mostra a TABELA 4, que enviará ao dicionário de dados os dados do atributo, com suas respectivas descrições, como, por exemplo: valor inicial, tipo do atributo, documentação, dentre outros.

TABELA 4 – Código Exemplo da Persistência de um Atributo no Dicionário de Dados.

```
// inicia a transação
  DMacessodados.Ora7.StartTransaction;
  seqAtributo:= 0;
  seqDoc:= 0;

// seta a base de dados, com a qual as querys irão trabalhar.
  DMacessodados.QSequencia.DatabaseName:= 'Ora';
  DMacessodados.Qdocumentacao.DatabaseName:= 'Ora';
  DMacessodados.QCompModel.DatabaseName:= 'Ora';

// se o conceito atributo foi documentado, prepara uma inserção na tabela Atributo e na tabela Documentação.
```

```
//busca ID da documentação na seqüência documentação.
   DMacessodados.QSequencia.Close;
   DMacessodados.QSequencia.SQL.Clear;
   DMacessodados.QSequencia.SQL.Add('select seq_documentacao.nextval from DUAL');
   DMacessodados.QSequencia.Open;
   seqDoc:= DMacessodados.QSequencia.Fields[0].asInteger;

//busca ID do Atributo na seqüência SEQ_CONCEITOS_NOTACAO.
   DMacessodados.QSequencia.Close;
   DMacessodados.QSequencia.SQL.Clear;
   DMacessodados.QSequencia.SQL.Add('select SEQ_CONCEITOS_NOTACAO.nextval from DUAL');
   DMacessodados.QSequencia.Open;
   seqAtributo:= DMacessodados.QSequencia.Fields[0].asInteger;

//insere as informações referentes à documentação do Atributo.
   DMacessodados.Qdocumentacao.Close;
   DMacessodados.Qdocumentacao.SQL.Clear;
   DMacessodados.Qdocumentacao.SQL.Add('insert into Documentacao_Projeto (id_documentacao,
documentacao)');
   DMacessodados.Qdocumentacao.SQL.Add('values (:IdDoc, :Doc)');
   DMacessodados.Qdocumentacao.Params[0].AsInteger:= seqDoc;
   DMacessodados.Qdocumentacao.Params[1].AsMemo:= MeDoc.Text;
   DMacessodados.Qdocumentacao.execSQL;

//insere as informações referentes ao Atributo e à sua documentação.
   DMacessodados.QCompModel.Close;
   DMacessodados.QCompModel.SQL.Clear;

//se for Atributo de Classe
     DMacessodados.QCompModel.SQL.Add('insert into atributo (id_atributo, id_classe, id_objeto, nm_Atributo, tipo,
vl_inicial, controle, estereotipo, documentacao)');
     DMacessodados.QCompModel.SQL.Add('values (:idAtributo, :idclasse, Null, :nmatributo, :tipo, :vlinicial,
:controle, Null, :iddoc)');
     DMacessodados.QCompModel.Params[0].AsInteger:= seqAtributo;
     DMacessodados.QCompModel.Params[1].asInteger:= PAtivo.iClasse;
     DMacessodados.QCompModel.Params[2].AsString:= EditAtrib.text;
     DMacessodados.QCompModel.Params[3].AsInteger:= DMacessodados.Qtipo.Fields[0].asInteger;
     DMacessodados.QCompModel.Params[4].AsString:= EdVlInicial.Text;
     DMacessodados.QCompModel.Params[5].AsInteger:= DMacessodados.Qvisibil.Fields[0].asInteger;
     DMacessodados.QCompModel.Params[6].AsInteger:= seqDoc;

//se for Atributo de Objeto
     DMacessodados.QCompModel.SQL.Add('insert into atributo (id_atributo, id_classe, id_objeto, nm_Atributo, tipo,
vl_inicial, controle, estereotipo, documentacao)');
     DMacessodados.QCompModel.SQL.Add('values (:idAtributo, Null, :idobejto, :nmatributo, :tipo, :vlinicial,
:controle, Null, :iddoc)');
     DMacessodados.QCompModel.Params[0].AsInteger:= seqAtributo;
     DMacessodados.QCompModel.Params[1].asInteger:= PAtivo.iObjeto;
     DMacessodados.QCompModel.Params[2].AsString:= EditAtrib.text;
     DMacessodados.QCompModel.Params[3].AsInteger:= DMacessodados.Qtipo.Fields[0].asInteger;
     DMacessodados.QCompModel.Params[4].AsString:= EdVlInicial.Text;
     DMacessodados.QCompModel.Params[5].AsInteger:= DMacessodados.Qvisibil.Fields[0].asInteger;
     DMacessodados.QCompModel.Params[6].AsInteger:= seqDoc;

   DMacessodados.QCompModel.execSQL;

//transação finalizada com sucesso.
  DMacessodados.Ora7.Commit;

//transação finalizada sem sucesso. Retorna as informações submentidas ao banco.
  DMacessodados.Ora7.Rollback;

END.
```

Uma vez que um evento, sendo de exclusão, alteração ou mesmo de inclusão de uma tupla, for disparado no banco, todos os gatilhos (*triggers*) associados à tabela/evento são disparados.

Seguindo o exemplo do atributo, nos eventos de inclusão de novos atributos no dicionário de dados, um dos gatilhos disparados é mostrado na TABELA 5.

TABELA 5 – Gatilho correspondente à regra de que uma classe não poderá conter atributos repetidos.

```
//REGRA: Uma Classe não tem atributos repetidos.

create trigger DDUMLRT.NM_UNICO_ATRIBUTO_CLASSE
before
   Insert
   Update
on    DDUMLRT.Atributo;
REFERENCING OLD AS old NEW AS new
declare   numrows INTEGER;
BEGIN
  /* Restrição para incluir um novo atributo na classe - deve ser único na classe. */
  If :new.Id_Objeto is null then
    select count(*) into numrows  from Atributo
      where Atributo.Id_Classe = :new.Id_Classe and
        Atributo.Nm_Atributo = :new.Nm_Atributo;
    if (numrows > 0)    then
      raise_application_error( -20002,
        'Cannot INSERT "Atribut" because this name already exists.' );
    end if;
  Else
   select count(*) into numrows  from Atributo
      where  Atributo.Id_Objeto = :new.Id_Objeto and
        Atributo.Nm_Atributo = :new.Nm_Atributo;
    if (numrows > 0)  then
      raise_application_error( -20002,
        'Cannot INSERT "Atribut" because this name already exists.'   );
    end if;
  end if;

 END;
```

A função deste gatilho é fazer uma verificação de forma que não seja permitido uma classe ou um objeto conter atributos repetidos. No exemplo do atributo, se for incluído um novo e este já estiver sido especificado anteriormente, como é o caso do atributo *volume* da classe Tanque, o gatilho detectará esta especificação e não deixará que o mesmo seja incluído novamente, como mostra a FIGURA 4.17.

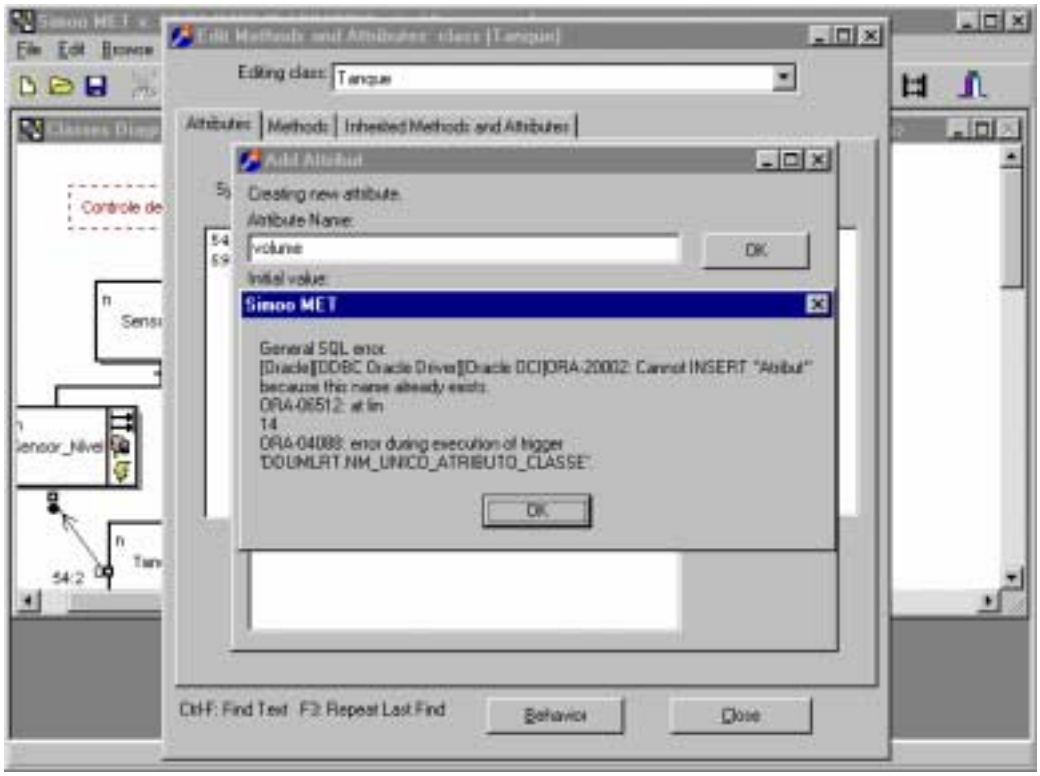


FIGURA 4.17 – Disparo do gatilho no evento inclusão de atributos no DD.

De maneira semalhante ao exemplo do atributo, descrito acima, se processa a inclusão, alteração e exclusão dos demais conceitos modelados. Isto faz com que o dicionário de dados se mantenha consistente e conseqüentemente o modelo como um todo.

5 Conclusões e Trabalhos Futuros

5.1 Considerações Gerais

Através das abordagens feitas aos sistemas de tempo real pode-se observar que tais aplicações agregam uma significativa complexidade em conseqüência dos requisitos de distribuição, de concorrência, de tempo de resposta, dentre outros. Apesar do paradigma da orientação a objetos mostrar-se adequado no tratamento de tais problemas, os benefícios obtidos por esta tecnologia na solução destas complexidades ainda não são suficientes para dar suporte ao ciclo de vida dos sistemas de tempo real.

Dentro deste contexto, muitos autores propuseram metodologias para dar suporte a todo ciclo de vida dos sistemas de tempo real. Estas metodologias são compostas por modelos que reunem conceitos e notação gráfica para estes conceitos, acompanhados por uma seqüência ordenada de etapas que permite a utilização destes modelos. Ainda assim, não é suficiente. Tais metodologias apresentaram carências de mecanismos que permitam descrições mais detalhadas dos requisitos temporais no desenvolvimento de sistemas tempo real.

Além do suporte metodológico, é importante que se tenha mecanismos para verificar se as descrições foram efetuadas de forma correta, ainda mais em se tratando de sistemas de tempo real, onde boa parcela dos erros é cometida na fase de análise. Verificações nas principais metodologias avaliaram a falta de meios que permitissem constatar a presença de inconsistências nas suas descrições. Outro ponto levado em consideração foi a indiscutível importância das ferramentas CASE no desenvolvimento de sistemas computacionais complexos; em especial, nos aspectos como visões gráficas e consistência entre modelos.

Os principais pontos abordados por esta dissertação envolvem a criação de um metamodelo para dar suporte à criação de um dicionário de dados para a modelagem de sistemas de tempo real utilizando a linguagem de modelagem gráfica UML-RT. Desta linguagem de modelagem, foram selecionados os diagramas de classe, de objetos, de estados, de seqüência e de colaboração. Foram feitos estudos de caso (Incubação Artificial de Ovos) para auxiliar a compreensão da sintaxe e da semântica dos conceitos trabalhados por estes diagramas. Posteriormente estes conceitos foram mapeados para um metamodelo.

O resultado deste mapeamento, mostrado na seção 3.1, foi muito produtivo. Os conceitos manipulados por mais de um diagrama foram possíveis de serem representados dentro do metamodelo, de forma única. Com isso, a consistência entre os diagramas foi alcançada. Entretanto, a semântica presente nos diagramas é de difícil representação gráfica em sua totalidade.

A solução encontrada para a descrição semântica foi a sua formalização através de regras OCL. Esta formalização permitiu o estabelecimento de restrições que garantem fazer uma descrição dos conceitos da maneira mais correta possível.

A soma destas duas soluções, propostas nesta dissertação, consentiu em alcançar um significativo melhoramento na ocorrência de descrições inconsistentes durante o desenvolvimento de sistemas de tempo real.

Outra vantagem significativa alcançada foi tornar possível tratar separadamente os conceitos das suas representações gráficas, no metamodelo. Todos os diagramas que modelarem o método e a sua execução poderão adotar representações gráficas distintas a fim de permitir encontrar uma representação que seja melhor compreendida pela equipe e que visualmente facilite a validação do que foi descrito.

A modelagem do conceito estereótipo é outro resultado positivo deste trabalho. Isso permite obter uma maior flexibilidade para os elementos mapeados no metamodelo, possibilitando a extensão destes para adequarem-se na modelagem de determinado domínio.

O metamodelo resultante também traz mapeado os conceitos que darão suporte à modelagem dos requisitos temporais, uma contribuição positiva deste estudo. Modelagem dos tempos de execução, *Deadlines soft*, *Deadlines hard*, *Deadlines firm*, descrição dos períodos em métodos periódicos, descrições dos intervalos máximo e mínimo de execução para os métodos aperiódicos, métodos atômicos e métodos não atômicos são alguns dos requisitos mapeados.

Outra contribuição significativa: o mapeamento do metamodelo para um modelo ER. No modelo Entidade&Relacionamento são descritas as tabelas que farão a persistência dos conceitos modelados através de classes do modelo conceitual. Torna-se muito simples a partir deste ER gerar os *scripts* que criarão o dicionário de dados no banco de dados.

5.2 Considerações ao Protótipo

Os passos seguidos na prototipação da pesquisa se iniciaram pela construção do modelo conceitual e as descrições semânticas em OCL. Posteriormente, foi realizado o mapeamento do metamodelo projetado para um modelo ER. Em seguida, foram gerados os *scripts* DDL usados na criação do dicionário de dados no banco de dados *Oracle*. As regras OCL foram mapeadas para gatilhos (*triggers*), que disparam no momento em que o dicionário de dados é manipulado.

O MET Editor do SiMOO-RT é a ferramenta diagramática que faz o povoamento dos dados no dicionário de dados, com base na arquitetura em dois níveis cliente/servidor.

Cliente/servidor é uma arquitetura versátil. Surgiu com o objetivo de melhorar a usabilidade, flexibilidade, interoperabilidade e escalabilidade das aplicações. O cliente pode ser definido como o agente solicitante de serviços e o servidor como o agente fornecedor de serviços. Uma máquina pode assumir o papel de cliente ou servidor, ou ambos, e a comunicação entre cliente/servidor é normalmente feita através de RPC(*Remote Procedure Call*) ou declarações SQL.

A arquitetura em dois níveis, na qual está estruturada o MET e o dicionário de dados, é a divisão da aplicação em duas partes. O ponto de divisão do processamento entre cliente/servidor é relativo a cada aplicação.

Ao se utilizar esta tecnologia, obtem-se algumas vantagens, tais como: a base de dados independente proporcionando maior escalabilidade e flexibilidade, as ferramentas GUI (*Grafic User Interface*) permitem desenvolver a aplicação mais rapidamente, interfaces gráficas amigáveis ao usuário e intuitivas para usar, os servidores já não necessitam de recursos em demasia, dentre outros.

Em aplicações que usam a tecnologia cliente/servidor em dois níveis, como MET e o dicionário de dados proposto, a portabilidade do mesmo pode ser dificultada. Isto acontece porque os fornecedores de banco de dados não utilizam um padrão para suas linguagens nativas ao banco de dados. Uma vez que as regras OCL são implementadas em PL/SQL, que é a linguagem nativa do banco de dados Oracle, então, ao ser trocado o fornecedor do banco, será necessário rescrever algumas particularidades para que as regras funcionem.

Teoricamente, por vezes que isto não é seguido, as DDL devem obedecer ao padrão ANSI. Conseqüentemente, os *scripts* com as descrições para a criação do esquema do dicionário de dados podem ser submetidos a qualquer banco de dados que o mesmo deveria ser criado. Isto não irá acontecer com as regras.

5.3 Trabalhos Futuros

Fica como sugestão para trabalhos a serem desenvolvidos futuramente criar uma terceira camada entre o banco e a interface que faria todo o processo de consistência e conexão com o banco. Esta terceira camada implementaria uma estrutura de classes que fariam o encapsulamento das tabelas do banco de dados, como também as bibliotecas de funções, com a verificação da consistência neste nível intermediário. Conceitos como de Framework e Patterns devem ser estudados ao propor esta solução.

Fica também como sugestão para trabalhos futuros o mapeamento do pacote **Gerenciamento da Representações Gráficas** para o ER e posteriormente para o dicionário de dados, que atualmente dá suporte ao pacote **Gerenciamento dos Modelos**, do metamodelo proposto na seção 3.1.

Finalmente, que possa ser modelado também o pacote **Gerenciamento da Implementação**, transformando o DD em uma enciclopédia que, segundo James Martin, é "... uma base de conhecimento que não armazena apenas informações de desenvolvimento, mas que ajuda a controlar a objetividade e a validação."

5.4 Conclusão

Os objetivos aos quais se propunha este trabalho tiveram conclusões satisfatórias, produzindo como resultado um metamodelo para os conceitos presentes na notação da UML-RT.

Através de um conjunto de regras formalizadas em OCL, foi descrito a semântica deste modelo, melhorando a precisão com a diminuição da presença de inconsistências e ambigüidades.

O metamodelo também mapeou conceitos que darão suporte à modelagem dos requisitos temporais, tais como: tempos de execução, *Deadlines soft, Deadlines hard, Deadlines firm*, descrição dos períodos em métodos periódicos, descrições dos intervalos máximo e mínimo de execução para os métodos aperiódicos, métodos atômicos e métodos não atômicos e descrições de tratamentos de execeções.

No exercício de prototipação desta pesquisa, realizou-se o mapeamento do metamodelo para um modelo ER. Foram descritas as tabelas que farão a persistência dos conceitos modelados pelas classes.

O banco de dados Oracle tem uma linguagem nativa PL/SQL na qual é possível a descrição de gatilhos, procedimentos e funções. Uma vez que a solução usada neste protótipo é este SGBD, as regras OCL mapeadas, se utilizaram desta tecnologia.

1 Anexo

1.1 Pacote das Representações Gráficas

O pacote do **Gerenciamento das Representações Gráficas**, FIGURA 3.3, mapeia a notação gráfica na qual serão visualizados os modelos. Esta notação obedecerá as especificações da UML-RT. Trabalhos futuros poderão se responsabilizar por extensões do dicionário de dados, agregando outras metodologias.

Os conceito mapeados para a FIGURA 3.5 – Modelos dos Conceitos da Notação UML-RT devem estar associados a, pelo menos, uma representação gráfica, caracterizada por uma Figura Geométrica, como mostra a FIGURA 2.1. Existem duas formas de mostrar o relacionamento do conceito com a sua representação gráfica:

a) A primeira, é importando a classe que modela a representação gráfica do pacote **Gerenciamento das Representações Gráficas** para modelo dos conceitos e criar o relacionamento de agregação entre ambas - classe Conceito e sua representação, como mostra a FIGURA 1.1. Esta opção tem a desvantagem de poluir o modelo dos conceitos. Isso se dá porque ao importamos as classes e estabelecermos os relacionamentos teremos um aumento no volume visual do modelo. Muitas classes e relacionamentos cruzados que podem dificultar o entendimento, comprometendo a legibilidade, além disso, tem o fato de estarmos representando conceitos semanticamente distintos em um mesmo local.

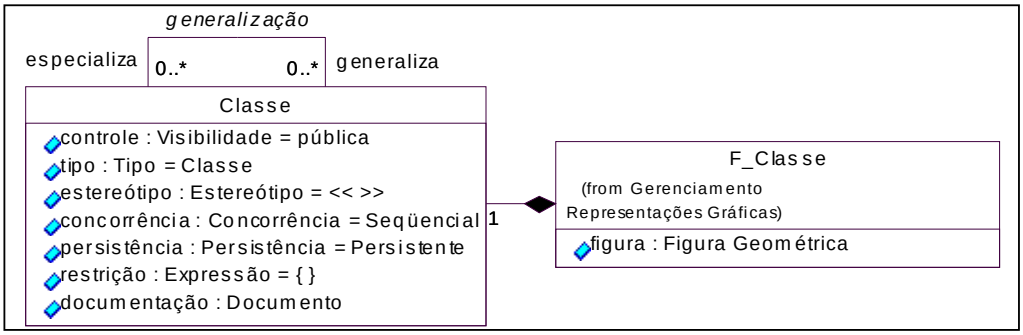


FIGURA 1.1 – Modelagem do Conceito e sua Representação por Relacionamento

b) Um relacionamento de agregação pode ser representado através de um atributo, como foi exposto na seção 3.1.1. Esta segunda opção torna o modelo menor mais conciso, conseqüentemente a legibilidade e o entendimento são melhorados, FIGURA 1.2.

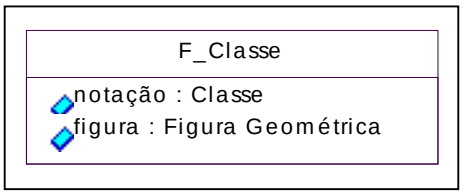


FIGURA 1.2 – Modelagem do Conceito e sua Representação por Atributo

Para descrever a representação gráfica dos conceitos vistos na seção 3.1 foi utilizado a segunda opção, de maneira que o relacionamento do conceito com a sua notação gráfica se deu por intermédio da definição de um atributo.

1.1.1 Figuras

A construção da notação gráfica dos conceitos da UML-RT se dá pela composição de Linha(s) e Forma(s). Esta determina as figuras geométricas modeladas na FIGURA 1.3. Um conjunto destas representações irá formar as visões para os modelos descritos na seção 3.1, ou seja, os diagramas.

1.1.1.1 Descrição da Modelagem das Figuras e Adornos

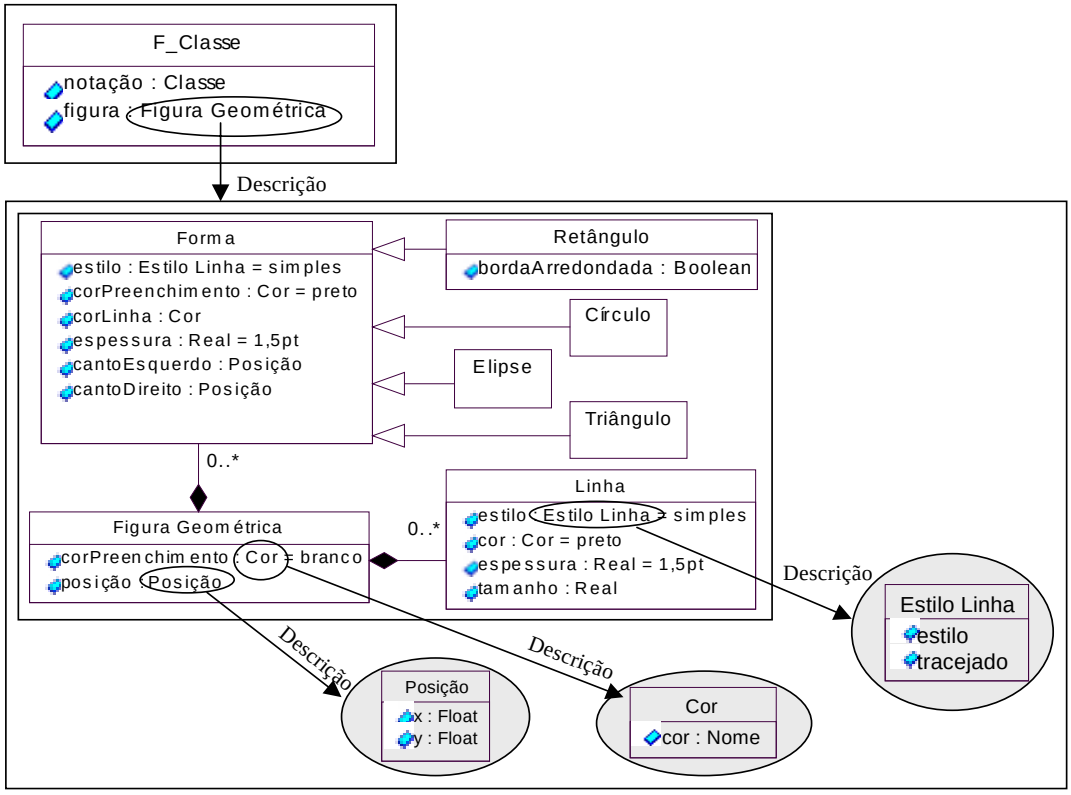


FIGURA 1.3 – Modelagem das Figuras e Adornos.

Figura Geométrica: É a composição de um símbolo[10] através da agregação de uma ou mais formas e/ou linhas.

Atributos:

corPreenchimento: Cor de preenchimento do símbolo. Ver Cor;

posição: Guarda a posição (x,y) do símbolo. Ver Posição.

Semântica:

[10] A palavra símbolo ou figuras são utilizadas no texto para fazer referência as formas geométricas representativas dos conceitos. Por exemplo em UML-RT o retângulo de cantos arredondados é representativo do conceito de estado de um objeto.

a) Uma Figura Geométrica é composta ou por Forma(s), ou por Linha(s), ou por ambas.

Linha: É um símbolos.

Atributos:

estilo: Estilo da linha. Uma instância da classe Estilo Linha;

cor: Cor da linha. Uma instância da classe Cor;

espessura: Espessura da linha;

tamanho: Tamanho do símbolo (linha).

Forma: É a classe genérica dos símbolos. Uma forma pode ser um Triângulo, uma Elipse, um Círculo ou um Retângulo (com os cantos arredondados, ou não).

Atributos:

estilo: Estilo da linha do símbolo. Ver Estilo Linha;

corPreenchimento: Cor do símbolo. Ver classe Cor;

corLinha: Cor da linha do símbolo. Ver classe Cor;

espessura: Espessura da linha do símbolo;

cantoEsquerdo e cantoDireito: Especificam o tamanho do símbolo. Ver Posição.

Semântica:

a) Uma Forma é uma classe genérica. Suas instâncias são ou uma Elipse, ou um Retângulo, ou um Círculo, ou um Triângulo.

Retângulo: Retângulo é um símbolo. Especifica o retângulo de uma classe ou o retângulo de um estado da classe, se seu atributo *bordaArredondada* for *true*.

Atributos:

bordaArredondada: É um atributo booleano que indica se as bordas do retângulo são ou não arredondadas.

Círculo, Elipse, Triângulo: Especificam, juntamente com a Linha, os demais símbolos da representação gráfica da UML-RT.

Cor: Classe que modela as cores.

Posição: Classe que contém a posição do símbolo.

Estilo Linha: Modela o estilo da linha.

Atributos:

estilo: Descreve o estilo da linha. Por exemplo, linha simples, linha dupla, linha tripla, entre outros.

tracejado: Descreve o tipo da linha. Por exemplo, linha pontilhada, linha tracejada, linha contínua, entre outros.

1.1.1.2 Descrição das Restrições em OCL

Figura Geométrica:

a) Uma Figura Geométrica é composta ou por Forma(s), ou por Linha(s), ou por ambas.

Figura Geométrica

```
self.allInstaces -> exists( ((self.forma -> size >=1) or
(self.linha -> size >=1)) or ((self.forma -> size >=1) and
(self.linha -> size >=1))).
```

Forma:

a) Uma Forma é uma classe genérica. Suas instâncias são ou uma Elipse, ou um Retângulo, ou um Círculo, ou um Triângulo.

Forma

```
self -> select(oclType = Elipse or
                oclType = Retângulo or
                oclType = Círculo or
                oclType = Triângulo).
```

1.1.2 Modelagem das Visões da Notação UML-RT

Para descrever a representação gráfica dos conceitos da UML_RT foi utilizada a notação do Diagrama de Classe da UML. Além disso, algumas outras convenções foram adotadas, como mostram os itens a seguir:

- Os conceito podem ou não ter uma representação gráfica. Para os que a possuem é modelado uma classe que representará esta visão.
- O nome da classe, que representa graficamente o conceito, foi padronizado da seguinte forma: F_Nome do Conceito, como mostra a FIGURA 1.4. A letra F significa Figura Geométrica e o 'Nome do Conceito', a classe que está sendo representada graficamente. Por exemplo, F_Atributo modela a representação gráfica do conceito Atributo.
- A classe que representa graficamente o conceito possui dois atributos: *notação* e *figura*, como mostra a FIGURA 1.4.
    - *notação*: Especifica qual o conceito cuja representação gráfica está sendo modelada. Por exemplo, *notação:*Atributo. Neste caso é a representação de atributo que está sendo modelada;
    - *figura*: Especifica a figura geométrica representativa do conceito. Por exemplo, *figura:*Figura Geométrica. A FIGURA 1.3 mostra a descrição dos componentes de uma figura geométrica.

    Resumindo, estes dois atributos, que descreve uma figura de um diagrama qualquer da UML-RT, representam uma agregação por composição, ou seja, uma classe F_Nome do Conceito, FIGURA 1.4, é composta por uma instância de Conceito e por uma instância de Figura Geométrica.

- Um conjunto de classes, que representa graficamente os conceitos e seus relacionamentos, é a visão de um diagrama.

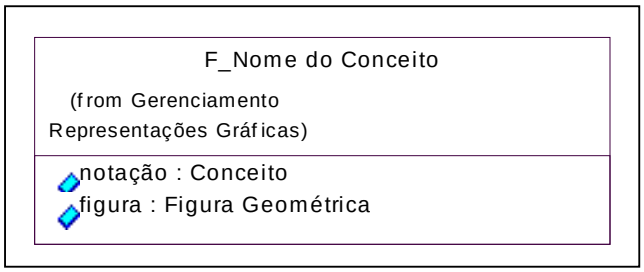


FIGURA 1.4 – Exemplo da Representação Gráfica de um Conceito.

1.1.2.1 Representação Gráfica do Diagrama de Classe

A FIGURA 1.5 modela a visão para os conceitos presentes no Diagrama de Classes.

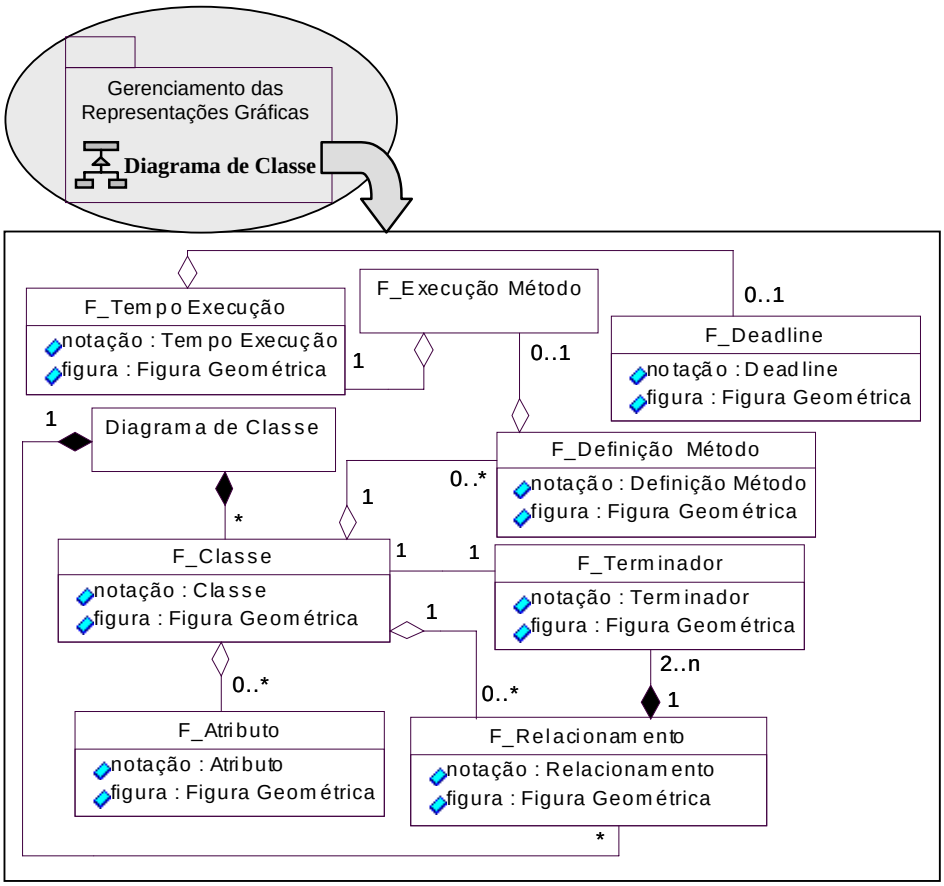


FIGURA 1.5 – Modelagem da Representação Gráfica do Diagrama de Classes.

1.1.2.2 Representação Gráfica do Diagrama de Objetos

A FIGURA 1.6 modela a visão para os conceitos presentes no Diagrama de Objetos.

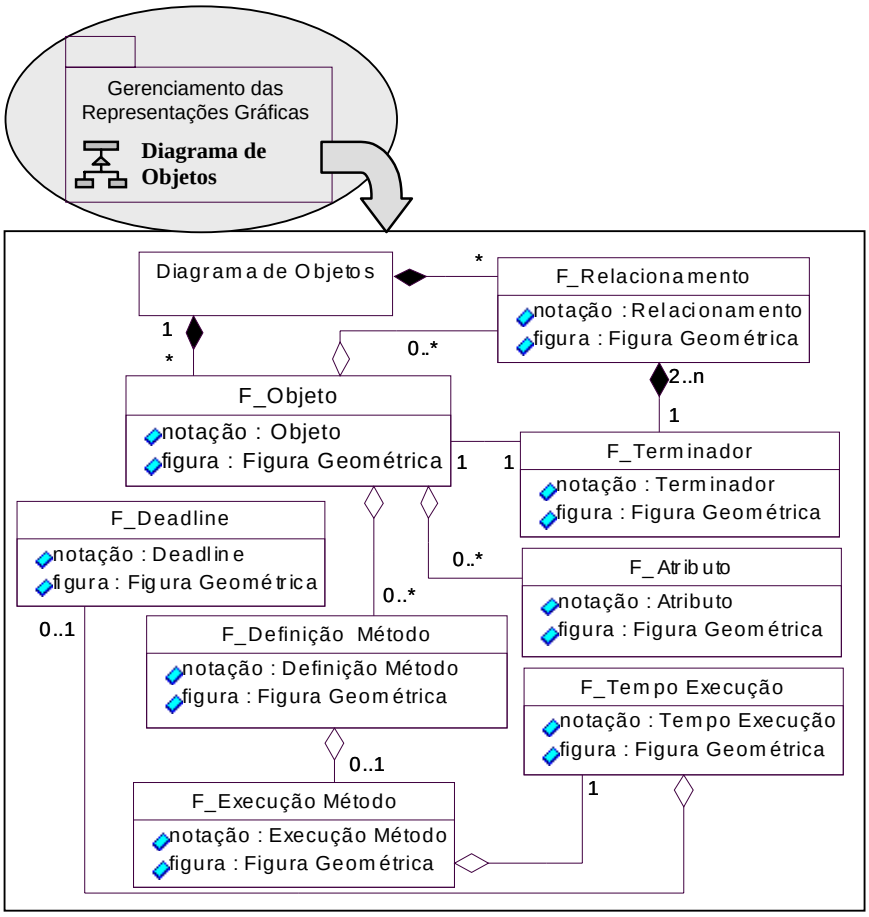


FIGURA 1.6 – Modelagem da Representação Gráfica do Diagrama de Objetos.

1.1.2.3 Representação Gráfica dos Diagramas de Interação

A FIGURA 1.7 modela a visão para os conceitos presentes nos Diagramas de Interação: Seqüência e Colaboração

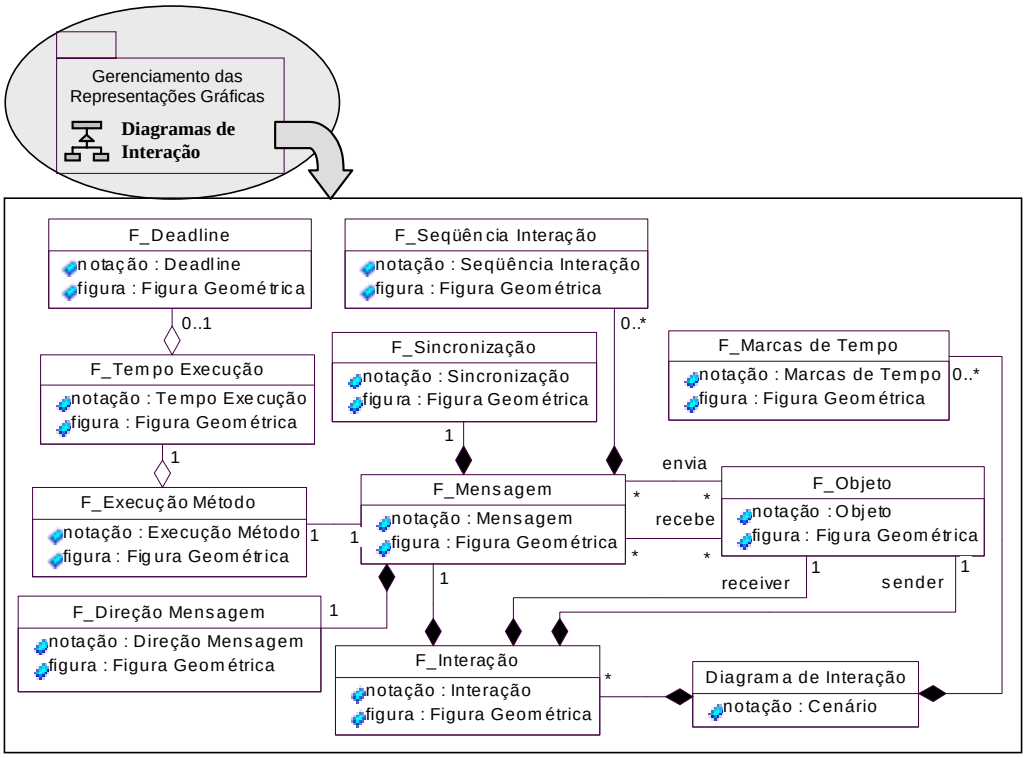


FIGURA 1.7 – Modelagem da Representação Gráfica do Diagrama de Interação.

1.1.2.4 Representação Gráfica do Diagrama de Estados

A FIGURA 1.8 modela a visão para os conceitos presentes no Diagrama de Estados do Objeto.

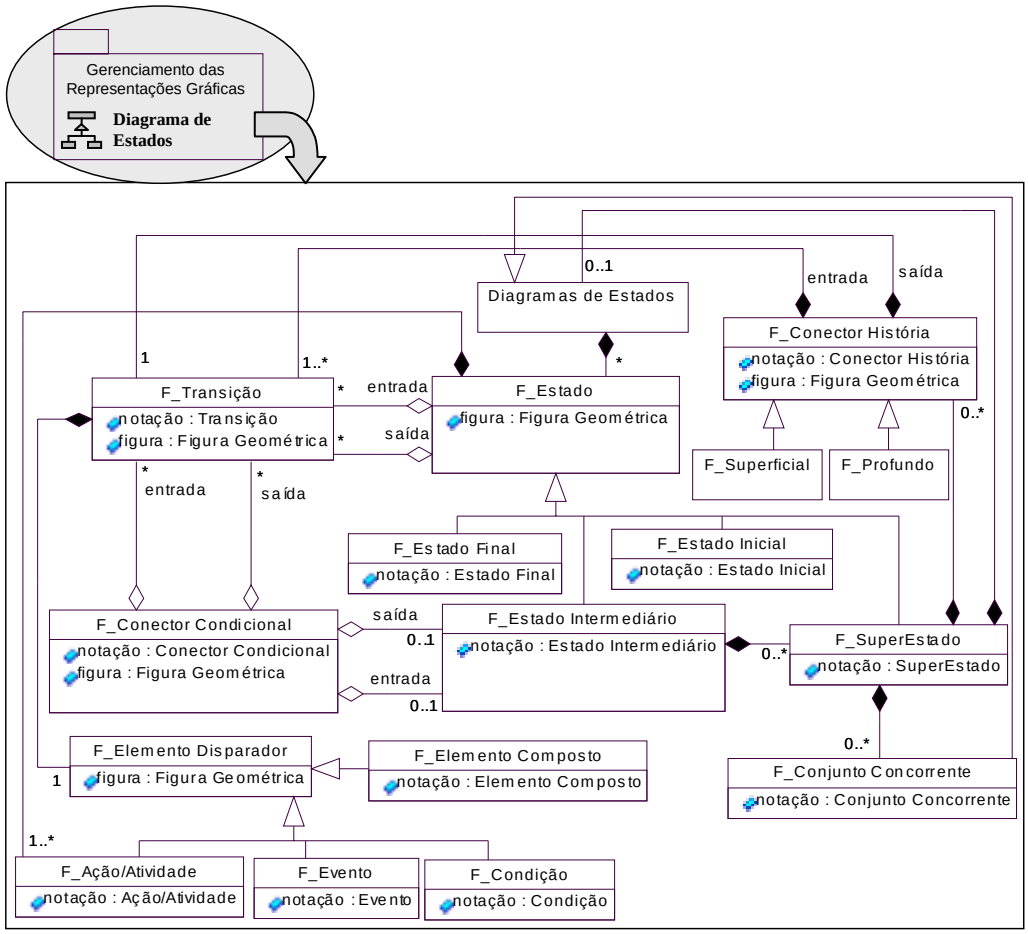


FIGURA 1.8– Modelagem da Representação Gráfica do Diagrama de Estados

2 Anexo

2.1 DDL do Dicionário de Dados

```
PROMPT Creating Table ACAO_ATIVIDADE
CREATE TABLE acao_atividade(
 id_acaoatividade          NUMBER(8)          NOT NULL,
 ds_acaoatividade          VARCHAR2(100)      NOT NULL,
 id_metodo                 NUMBER(8)          NOT NULL,
 id_tempoacao              NUMBER(8)          NOT NULL,
 id_estado                 NUMBER(8)          NOT NULL,
 documentacao              NUMBER(8)          NULL);

PROMPT Creating Table ARGUMENTO
CREATE TABLE argumento(
 id_argumento              NUMBER(8)          NOT NULL,
 id_metodo                 NUMBER(8)          NULL,
 id_mensagemevento         NUMBER(8)          NULL,
 nm_argumento              VARCHAR2(100)      NOT NULL,
 tipo                      NUMBER(8)          NOT NULL,
 documentacao              NUMBER(8)          NULL);

PROMPT Creating Table ATRIBUTO
CREATE TABLE atributo(
 id_atributo               NUMBER(8)          NOT NULL,
 id_classe                 NUMBER(8)          NULL,
 id_objeto                 NUMBER(8)          NULL,
 nm_atributo               VARCHAR2(100)      NOT NULL,
 tipo                      NUMBER(8)          NOT NULL,
 vl_inicial                VARCHAR2(100)      NULL,
 controle                  NUMBER(8)          NOT NULL,
 estereotipo               NUMBER(8)          NULL,
 documentacao              NUMBER(8)          NULL);

PROMPT Creating Table CENARIO
CREATE TABLE cenario(
 id_cenario                NUMBER(8)          NOT NULL,
 visao                     VARCHAR2(100)      NOT NULL,
 documentacao              NUMBER(8)          NULL);

PROMPT Creating Table CLASSE
CREATE TABLE classe(
 id_classe                 NUMBER(8)          NOT NULL,
 id_dgclasse               NUMBER(8)          NOT NULL,
 nm_classe                 VARCHAR2(100)      NOT NULL,
 controle                  NUMBER(8)          NOT NULL,
 tp_classe                 NUMBER(8)          NOT NULL,
 estereotipo               NUMBER(8)          NULL,
```

```
 persistencia              NUMBER(8)             NOT NULL,
 concorrencia              NUMBER(8)             NOT NULL,
 restricao                 NUMBER(8)             NULL,
 eabstrata                 CHAR(1)               NOT NULL,
 instancia                 NUMBER(8)             NULL,
 ativa                     CHAR(1)               NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Table CONCORRENCIA
CREATE TABLE concorrencia(
 id_concorrencia           NUMBER(8)             NOT NULL,
 ds_concorrencia           VARCHAR2(100)         NOT NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Table CONECTOR_CONDICIONAL
CREATE TABLE conector_condicional(
 id_conector               NUMBER(8)             NOT NULL,
 ds_conector               VARCHAR2(100)         NOT NULL);

PROMPT Creating Table CORPO_METODO
CREATE TABLE corpo_metodo(
 id_metodo                 NUMBER(8)             NOT NULL,
 id_corpo                  NUMBER(8)             NOT NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Table DEFINICAO_METODO
CREATE TABLE definicao_metodo(
 id_metodo                 NUMBER(8)             NOT NULL,
 id_classe                 NUMBER(8)             NOT NULL,
 id_objeto                 NUMBER(8)             NOT NULL,
 nm_metodo                 VARCHAR2(100)         NOT NULL,
 controle                  NUMBER(8)             NOT NULL,
 estereotipo               NUMBER(8)             NULL,
 concorrencia              NUMBER(8)             NOT NULL,
 restricao                 NUMBER(8)             NULL,
 retorno                   NUMBER(8)             NULL,
 einterrupto               CHAR(1)               NULL,
 econcreto                 CHAR(1)               NOT NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Table DG_CLASSE
CREATE TABLE dg_classe(
 id_dgclasse               NUMBER(8)             NOT NULL,
 id_projeto                NUMBER(8)             NOT NULL,
 nm_dgclasse               VARCHAR2(100)         NOT NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Table DG_ESTADO
CREATE TABLE dg_estado(
```

```
 id_dgestado              NUMBER(8)             NOT NULL,
 id_classe                NUMBER(8)             NOT NULL,
 nm_dgestado              VARCHAR2(100)         NOT NULL,
 documentacao             NUMBER(8)             NULL);

PROMPT Creating Table DG_INTERACAO
CREATE TABLE dg_interacao(
 id_dginteracao           NUMBER(8)             NOT NULL,
 id_usecase               NUMBER(8)             NOT NULL,
 nm_dginteracao           VARCHAR2(100)         NOT NULL,
 id_cenario               NUMBER(8)             NOT NULL,
 tempo_execucao           NUMBER(4,3)           NULL,
 documentacao             NUMBER(8)             NULL);

PROMPT Creating Table DG_USECASE
CREATE TABLE dg_usecase(
 id_dgusecase             NUMBER(8)             NOT NULL,
 id_projeto               NUMBER(8)             NOT NULL,
 nm_dgusecase             VARCHAR2(100)         NOT NULL,
 documentacao             NUMBER(8)             NULL);

PROMPT Creating Table DOCUMENTACAO_PROJETO
CREATE TABLE documentacao_projeto(
 id_documentacao          NUMBER(8)             NOT NULL,
 documentacao             VARCHAR2(1000)        NULL);

PROMPT Creating Table ELEMENTO_DISPARADOR
CREATE TABLE elemento_disparador(
 id_elemdisparador        NUMBER(8)             NOT NULL,
 condicao                 NUMBER(8)             NULL,
 evento                   NUMBER(8)             NULL,
 acaoatividade            NUMBER(8)             NULL,
 documentacao             NUMBER(8)             NULL);

PROMPT Creating Table ESTADO
CREATE TABLE estado(
 id_estado                NUMBER(8)             NOT NULL,
 id_dgestado              NUMBER(8)             NOT NULL,
 nm_estado                VARCHAR2(100)         NOT NULL,
 tipo                     NUMBER(8)             NOT NULL,
 esuperestado             CHAR(1)               NOT NULL,
 documentacao             NUMBER(8)             NULL);

PROMPT Creating Table ESTADO_EMBUTIDO
CREATE TABLE estado_embutido(
 id_estado                NUMBER(8)             NOT NULL,
 id_estadoembutido        NUMBER(8)             NOT NULL);

PROMPT Creating Table ESTEREOTIPO
```

```
CREATE TABLE estereotipo(
 id_estereotipo          NUMBER(8)          NOT NULL,
 ds_estereotipo          VARCHAR2(100)      NOT NULL,
 objeto_estereotipado    VARCHAR2(50)       NOT NULL,
 documentacao            NUMBER(8)          NULL);

PROMPT Creating Table EVENTO
CREATE TABLE evento(
 id_evento               NUMBER(8)          NOT NULL,
 id_metodo               NUMBER(8)          NOT NULL,
 assinatura_evento       VARCHAR2(100)      NOT NULL,
 documentacao            NUMBER(8)          NULL);

PROMPT Creating Table EXECUCAO_METODO
CREATE TABLE execucao_metodo(
 id_execucao             NUMBER(8)          NOT NULL,
 id_metodo               NUMBER(8)          NOT NULL,
 tempo_execucao          NUMBER(4,2)        NOT NULL,
 eperiodico              CHAR(1)            NOT NULL,
 periodo                 NUMBER(4,2)        NULL,
 jitterminimo            NUMBER(4,2)        NULL,
 jittermaximo            NUMBER(4,2)        NULL,
 minintervalo            NUMBER(4,2)        NULL,
 maxintervalo            NUMBER(4,2)        NULL,
 edeadlinehard           CHAR(1)            NOT NULL,
 edeadlinefirm           CHAR(1)            NOT NULL,
 deadlinesoft            NUMBER(4,2)        NULL,
 deadlinehard            NUMBER(4,2)        NULL,
 documentacao            NUMBER(8)          NULL);

PROMPT Creating Table EXPRESSAO
CREATE TABLE expressao(
 id_expressao            NUMBER(8)          NOT NULL,
 linguagem               VARCHAR2(100)      DEFAULT 'OCL'  NOT NULL,
 corpo                   VARCHAR2(1000)     NOT NULL,
 documentacao            NUMBER(8)          NULL);

PROMPT Creating Table INTERACAO
CREATE TABLE interacao(
 id_tipomensagem         NUMBER(8)          NOT NULL,
 id_mensagem             NUMBER(8)          NOT NULL,
 envia                   NUMBER(8)          NOT NULL,
 recebe                  NUMBER(8)          NOT NULL,
 id_cenario              NUMBER(8)          NOT NULL,
 documentacao            NUMBER(8)          NULL);

PROMPT Creating Table MARCA_TEMPO
CREATE TABLE marca_tempo(
 id_marcatempo           NUMBER(8)          NOT NULL,
```

```
 tempo                  NUMBER(4,2)          NOT NULL);

PROMPT Creating Table MENSAGEM
CREATE TABLE mensagem(
 id_mensagem            NUMBER(8)            NOT NULL,
 assinatura             VARCHAR2(100)        NOT NULL,
 id_execucao            NUMBER(8)            NOT NULL,
 prioridade             NUMBER(2)            NOT NULL,
 thread                 VARCHAR2(10)         NOT NULL,
 num_sequencia          NUMBER(5)            NOT NULL,
 sincronizacao          NUMBER(8)            NULL,
 guarda                 NUMBER(8)            NULL,
 retorno                NUMBER(8)            NULL,
 precondicao            NUMBER(8)            NULL,
 estereotipo            NUMBER(8)            NULL,
 exp_interacao          NUMBER(8)            NULL,
 documentacao           NUMBER(8)            NULL);

PROMPT Creating Table MENSAGEM_MARCATEMPO
CREATE TABLE mensagem_marcatempo(
 id_marcatempo          NUMBER(8)            NOT NULL,
 id_mensagem            NUMBER(8)            NOT NULL,
 id_cenario             NUMBER(8)            NOT NULL);

PROMPT Creating Table MENSAGEM_OBJETO
CREATE TABLE mensagem_objeto(
 id_tipomensagem        NUMBER(8)            NOT NULL,
 id_mensagem            NUMBER(8)            NOT NULL,
 envia                  NUMBER(8)            NOT NULL,
 recebe                 NUMBER(8)            NOT NULL);

PROMPT Creating Table MULTIPLICIDADE
CREATE TABLE multiplicidade(
 id_multiplicidade      NUMBER(8)            NOT NULL,
 minima                 VARCHAR2(3)          NOT NULL,
 maxima                 VARCHAR2(3)          NOT NULL,
 documentacao           NUMBER(8)            NULL);

PROMPT Creating Table OBJETO
CREATE TABLE objeto(
 id_objeto              NUMBER(8)            NOT NULL,
 id_classe              NUMBER(8)            NOT NULL,
 nm_objeto              VARCHAR2(100)        NOT NULL,
 persistencia           NUMBER(8)            NOT NULL,
 documentacao           NUMBER(8)            NULL);

PROMPT Creating Table PERSISTENCIA
CREATE TABLE persistencia(
 id_persistencia        NUMBER(8)            NOT NULL,
```

```
 ds_persistencia              VARCHAR2(100)          NOT NULL,
 documentacao                 NUMBER(8)              NULL);

PROMPT Creating Table PROJETO
CREATE TABLE projeto(
 id_projeto                   NUMBER(8)              NOT NULL,
 nm_projeto                   VARCHAR2(100)          DEFAULT 'Nome'  NOT NULL,
 versao                       VARCHAR2(4)            DEFAULT 'V1.0'  NOT NULL,
 dt_versao                    DATE                   DEFAULT SYSDATE  NOT NULL,
 documentacao                 NUMBER(8)              NULL);

PROMPT Creating Table QUALIFICADOR
CREATE TABLE qualificador(
 id_terminador                NUMBER(8)              NOT NULL,
 nm_qualificador              VARCHAR2(100)          NOT NULL,
 tipo                         NUMBER(8)              NOT NULL,
 documentacao                 NUMBER(8)              NULL);

PROMPT Creating Table RELACIONAMENTO
CREATE TABLE relacionamento(
 id_relacionamento            NUMBER(8)              NOT NULL,
 nm_relacionamento            VARCHAR2(100)          NULL,
 restricao                    NUMBER(8)              NULL,
 estereotipo                  NUMBER(8)              NULL,
 atrib_relacionamento         NUMBER(8)              NULL,
 documentacao                 NUMBER(8)              NULL);

PROMPT Creating Table SINCRONIZACAO
CREATE TABLE sincronizacao(
 id_sincronizacao             NUMBER(8)              NOT NULL,
 ds_sincronizacao             VARCHAR2(100)          NOT NULL,
 documentacao                 NUMBER(8)              NULL);

PROMPT Creating Table TEMPO_ACAO
CREATE TABLE tempo_acao(
 id_tempoacao                 NUMBER(8)              NOT NULL,
 ds_tempoacao                 VARCHAR2(100)          NOT NULL,
 documentacao                 NUMBER(8)              NULL);

PROMPT Creating Table TERMINADOR
CREATE TABLE terminador(
 id_terminador                NUMBER(8)              NOT NULL,
 id_classe                    NUMBER(8)              NOT NULL,
 id_relacionamento            NUMBER(8)              NOT NULL,
 papel                        VARCHAR2(100)          NULL,
 tp_relacionamento            NUMBER(8)              NOT NULL,
 multiplicidade               NUMBER(8)              NOT NULL,
 controle                     NUMBER(8)              NOT NULL,
 restricao                    NUMBER(8)              NULL,
```

```
 eordenado              CHAR(1)          NULL,
 navegabilidade         CHAR(1)          NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TIPO
CREATE TABLE tipo(
 id_tipo                NUMBER(8)        NOT NULL,
 ds_tipo                VARCHAR2(100)    NOT NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TIPO_CLASSE
CREATE TABLE tipo_classe(
 id_tipoclasse          NUMBER(8)        NOT NULL,
 ds_tipoclasse          VARCHAR2(100)    NOT NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TIPO_ESTADO
CREATE TABLE tipo_estado(
 id_tipoestado          NUMBER(8)        NOT NULL,
 ds_tipoestado          VARCHAR2(100)    NOT NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TIPO_MENSAGEM
CREATE TABLE tipo_mensagem(
 id_tipomensagem        NUMBER(8)        NOT NULL,
 ds_tipomensagem        NUMBER(8)        NOT NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TIPO_RELACIONAMENTO
CREATE TABLE tipo_relacionamento(
 id_tprelacionamento    NUMBER(8)        NOT NULL,
 nm_tprelacionamento    VARCHAR2(100)    NOT NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TRANSICAO
CREATE TABLE transicao(
 id_elemdisparador      NUMBER(8)        NOT NULL,
 destino                NUMBER(8)        NOT NULL,
 origem                 NUMBER(8)        NOT NULL,
 documentacao           NUMBER(8)        NULL);

PROMPT Creating Table TRATAMENTO_EXCECAO
CREATE TABLE tratamento_excecao(
 id_execucao            NUMBER(8)        NOT NULL,
 id_metodo              NUMBER(8)        NOT NULL);

PROMPT Creating Table USECASE
CREATE TABLE usecase(
 id_usecase             NUMBER(8)        NOT NULL,
```

```
 id_dgusecase              NUMBER(8)             NOT NULL,
 nr_sequencia              VARCHAR2(6)           NOT NULL,
 nm_usecase                VARCHAR2(100)         NOT NULL,
 prioridade                NUMBER(3)             DEFAULT 1  NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Table VISIBILIDADE
CREATE TABLE visibilidade(
 id_visibilidade           NUMBER(8)             NOT NULL,
 tipo                      VARCHAR2(100)         DEFAULT 'pública'  NOT NULL,
 documentacao              NUMBER(8)             NULL);

PROMPT Creating Sequence SEQ_CONCEITOS_NOTACAO
CREATE SEQUENCE seq_conceitos_notacao
 INCREMENT BY 1
 START WITH 1000
 NOMINVALUE
 NOMAXVALUE
 NOCYCLE
 CACHE 20
 ORDER;

REM Sequencia para gerar a chave primaria dos diagramas de um projeto
PROMPT Creating Sequence SEQ_DIAGRAMA
CREATE SEQUENCE seq_diagrama
 INCREMENT BY 1
 START WITH 1000
 NOMINVALUE
 NOMAXVALUE
 NOCYCLE
 CACHE 20
 ORDER;

REM Esta seqüência gera um número para codificar cada documentação dos conceitos na tabela DOCUMENTACAO_PROJETO.
PROMPT Creating Sequence SEQ_DOCUMENTACAO
CREATE SEQUENCE seq_documentacao
 INCREMENT BY 1
 START WITH 1000
 NOMINVALUE
 NOMAXVALUE
 NOCYCLE
 CACHE 20
 ORDER;

REM Sequencia que cria a chave primaria do projeto.
PROMPT Creating Sequence SEQ_PROJETO
CREATE SEQUENCE seq_projeto
 INCREMENT BY 1
 START WITH 1000
```

```
 NOMINVALUE
 NOMAXVALUE
 NOCYCLE
 CACHE 20
 ORDER;

PROMPT Creating User DDUMLRT
CREATE USER ddumlrt IDENTIFIED BY UMLRT;

PROMPT Granting Role To User DDUMLRT
GRANT DBA ,CONNECT ,RESOURCE TO ddumlrt;

PROMPT Granting Access On V_VISIBILIDADE TO PUBLIC
GRANT ALL PRIVILEGES  ON v_visibilidade TO PUBLIC
 WITH GRANT OPTION;

PROMPT Granting Object Privilege On ACAO_ATIVIDADE To DDUMLRT
GRANT ALL PRIVILEGES  ON acao_atividade TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On ARGUMENTO To DDUMLRT
GRANT ALL PRIVILEGES  ON argumento TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On ATRIBUTO To DDUMLRT
GRANT ALL PRIVILEGES  ON atributo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On CENARIO To DDUMLRT
GRANT ALL PRIVILEGES  ON cenario TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On CLASSE To DDUMLRT
GRANT ALL PRIVILEGES  ON classe TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On CONCORRENCIA To DDUMLRT
GRANT ALL PRIVILEGES  ON concorrencia TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On CONECTOR_CONDICIONAL To DDUMLRT
GRANT ALL PRIVILEGES  ON conector_condicional TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On CORPO_METODO To DDUMLRT
GRANT ALL PRIVILEGES  ON corpo_metodo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On DEFINICAO_METODO To DDUMLRT
GRANT ALL PRIVILEGES  ON definicao_metodo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On DG_CLASSE To DDUMLRT
GRANT ALL PRIVILEGES  ON dg_classe TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On DG_ESTADO To DDUMLRT
GRANT ALL PRIVILEGES  ON dg_estado TO ddumlrt WITH GRANT OPTION;
```

```
PROMPT Granting Object Privilege On DG_INTERACAO To DDUMLRT
GRANT ALL PRIVILEGES  ON dg_interacao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On DG_USECASE To DDUMLRT
GRANT ALL PRIVILEGES  ON dg_usecase TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On DOCUMENTACAO_PROJETO To DDUMLRT
GRANT ALL PRIVILEGES  ON documentacao_projeto TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On ELEMENTO_DISPARADOR To DDUMLRT
GRANT ALL PRIVILEGES  ON elemento_disparador TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On ESTADO To DDUMLRT
GRANT ALL PRIVILEGES  ON estado TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On ESTADO_EMBUTIDO To DDUMLRT
GRANT ALL PRIVILEGES  ON estado_embutido TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On ESTEREOTIPO To DDUMLRT
GRANT ALL PRIVILEGES  ON estereotipo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On EVENTO To DDUMLRT
GRANT ALL PRIVILEGES  ON evento TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On EXECUCAO_METODO To DDUMLRT
GRANT ALL PRIVILEGES  ON execucao_metodo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On EXPRESSAO To DDUMLRT
GRANT ALL PRIVILEGES  ON expressao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On INTERACAO To DDUMLRT
GRANT ALL PRIVILEGES  ON interacao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On MARCA_TEMPO To DDUMLRT
GRANT ALL PRIVILEGES  ON marca_tempo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On MENSAGEM To DDUMLRT
GRANT ALL PRIVILEGES  ON mensagem TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On MENSAGEM_MARCATEMPO To DDUMLRT
GRANT ALL PRIVILEGES  ON mensagem_marcatempo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On MENSAGEM_OBJETO To DDUMLRT
GRANT ALL PRIVILEGES  ON mensagem_objeto TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On MULTIPLICIDADE To DDUMLRT
GRANT ALL PRIVILEGES  ON multiplicidade TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On OBJETO To DDUMLRT
```

```
GRANT ALL PRIVILEGES  ON objeto TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On PERSISTENCIA To DDUMLRT
GRANT ALL PRIVILEGES  ON persistencia TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On PROJETO To DDUMLRT
GRANT ALL PRIVILEGES  ON projeto TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On QUALIFICADOR To DDUMLRT
GRANT ALL PRIVILEGES  ON qualificador TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On RELACIONAMENTO To DDUMLRT
GRANT ALL PRIVILEGES  ON relacionamento TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On SEQ_DOCUMENTACAO To DDUMLRT
GRANT ALL PRIVILEGES  ON seq_documentacao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On SINCRONIZACAO To DDUMLRT
GRANT ALL PRIVILEGES  ON sincronizacao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TEMPO_ACAO To DDUMLRT
GRANT ALL PRIVILEGES  ON tempo_acao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TERMINADOR To DDUMLRT
GRANT ALL PRIVILEGES  ON terminador TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TIPO To DDUMLRT
GRANT ALL PRIVILEGES  ON tipo TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TIPO_CLASSE To DDUMLRT
GRANT ALL PRIVILEGES  ON tipo_classe TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TIPO_ESTADO To DDUMLRT
GRANT ALL PRIVILEGES  ON tipo_estado TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TIPO_MENSAGEM To DDUMLRT
GRANT ALL PRIVILEGES  ON tipo_mensagem TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TIPO_RELACIONAMENTO To DDUMLRT
GRANT ALL PRIVILEGES  ON tipo_relacionamento TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TRANSICAO To DDUMLRT
GRANT ALL PRIVILEGES  ON transicao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On TRATAMENTO_EXCECAO To DDUMLRT
GRANT ALL PRIVILEGES  ON tratamento_excecao TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On USECASE To DDUMLRT
GRANT ALL PRIVILEGES  ON usecase TO ddumlrt WITH GRANT OPTION;
```

```
PROMPT Granting Object Privilege On VISIBILIDADE To DDUMLRT
GRANT ALL PRIVILEGES  ON visibilidade TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Object Privilege On V_VISIBILIDADE To DDUMLRT
GRANT ALL PRIVILEGES  ON v_visibilidade TO ddumlrt WITH GRANT OPTION;

PROMPT Granting Default Tablespace To DDUMLRT
ALTER USER  ddumlrt
 DEFAULT TABLESPACE SYSTEM
 TEMPORARY TABLESPACE SYSTEM
 DEFAULT ROLE ALL;

PROMPT Adding PRIMARY Constraint To ACAO_ATIVIDADE Table
ALTER TABLE ACAO_ATIVIDADE ADD (
    CONSTRAINT PK_ACAOATIVIDADE
    PRIMARY KEY (ID_ACAOATIVIDADE))/

PROMPT Adding PRIMARY Constraint To ARGUMENTO Table
ALTER TABLE ARGUMENTO ADD (
    CONSTRAINT PK_ARGUMENTO
    PRIMARY KEY (ID_ARGUMENTO))/

PROMPT Adding PRIMARY Constraint To ATRIBUTO Table
ALTER TABLE ATRIBUTO ADD (
    CONSTRAINT PK_ATRIBUTO
    PRIMARY KEY (ID_ATRIBUTO))/

PROMPT Adding PRIMARY Constraint To CENARIO Table
ALTER TABLE CENARIO ADD (
    CONSTRAINT PK_CENARIO
    PRIMARY KEY (ID_CENARIO))/

PROMPT Adding PRIMARY Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT PK_CLASSE
    PRIMARY KEY (ID_CLASSE))/

PROMPT Adding PRIMARY Constraint To CONCORRENCIA Table
ALTER TABLE CONCORRENCIA ADD (
    CONSTRAINT PK_CONCORRENCIA
    PRIMARY KEY (ID_CONCORRENCIA))/

PROMPT Adding PRIMARY Constraint To CONECTOR_CONDICIONAL Table
ALTER TABLE CONECTOR_CONDICIONAL ADD (
    CONSTRAINT PK_CONECTOR
    PRIMARY KEY (ID_CONECTOR))/

PROMPT Adding PRIMARY Constraint To CORPO_METODO Table
```

```
ALTER TABLE CORPO_METODO ADD (
    CONSTRAINT PK_CORPOMETODO
    PRIMARY KEY (ID_METODO,
            ID_CORPO))/

PROMPT Adding PRIMARY Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT PK_METODO
    PRIMARY KEY (ID_METODO))/

PROMPT Adding PRIMARY Constraint To DG_CLASSE Table
ALTER TABLE DG_CLASSE ADD (
    CONSTRAINT PK_DGCLASSE
    PRIMARY KEY (ID_DGCLASSE))/

PROMPT Adding PRIMARY Constraint To DG_ESTADO Table
ALTER TABLE DG_ESTADO ADD (
    CONSTRAINT PK_DGESTADO
    PRIMARY KEY (ID_DGESTADO))/

PROMPT Adding PRIMARY Constraint To DG_INTERACAO Table
ALTER TABLE DG_INTERACAO ADD (
    CONSTRAINT PK_DGINTERACAO
    PRIMARY KEY (ID_DGINTERACAO))/

PROMPT Adding PRIMARY Constraint To DG_USECASE Table
ALTER TABLE DG_USECASE ADD (
    CONSTRAINT PK_DGUSECASE
    PRIMARY KEY (ID_DGUSECASE))/

PROMPT Adding PRIMARY Constraint To DOCUMENTACAO_PROJETO Table
ALTER TABLE DOCUMENTACAO_PROJETO ADD (
    CONSTRAINT PK_DOCUMENTACAO
    PRIMARY KEY (ID_DOCUMENTACAO))/

PROMPT Adding PRIMARY Constraint To ELEMENTO_DISPARADOR Table
ALTER TABLE ELEMENTO_DISPARADOR ADD (
    CONSTRAINT PK_ELEMDISPARADOR
    PRIMARY KEY (ID_ELEMDISPARADOR))/

PROMPT Adding PRIMARY Constraint To ESTADO Table
ALTER TABLE ESTADO ADD (
    CONSTRAINT PK_ESTADO
    PRIMARY KEY (ID_ESTADO))/

PROMPT Adding PRIMARY Constraint To ESTADO_EMBUTIDO Table
ALTER TABLE ESTADO_EMBUTIDO ADD (
    CONSTRAINT PK_ESTADOEMBUTIDO
    PRIMARY KEY (ID_ESTADO,
```

```
        ID_ESTADOEMBUTIDO))/

PROMPT Adding PRIMARY Constraint To ESTEREOTIPO Table
ALTER TABLE ESTEREOTIPO ADD (
    CONSTRAINT PK_ESTEREOTIPO
    PRIMARY KEY (ID_ESTEREOTIPO))/

PROMPT Adding PRIMARY Constraint To EVENTO Table
ALTER TABLE EVENTO ADD (
    CONSTRAINT PK_EVENTO
    PRIMARY KEY (ID_EVENTO))/

PROMPT Adding PRIMARY Constraint To EXECUCAO_METODO Table
ALTER TABLE EXECUCAO_METODO ADD (
    CONSTRAINT PK_EXECUCAOMETODO
    PRIMARY KEY (ID_EXECUCAO))/

PROMPT Adding PRIMARY Constraint To EXPRESSAO Table
ALTER TABLE EXPRESSAO ADD (
    CONSTRAINT PK_EXPRESSAO
    PRIMARY KEY (ID_EXPRESSAO))/

PROMPT Adding PRIMARY Constraint To INTERACAO Table
ALTER TABLE INTERACAO ADD (
    CONSTRAINT PK_INTERACAO
    PRIMARY KEY (ID_TIPOMENSAGEM,
        ID_MENSAGEM,
        ENVIA,
        RECEBE,
        ID_CENARIO))/

PROMPT Adding PRIMARY Constraint To MARCA_TEMPO Table
ALTER TABLE MARCA_TEMPO ADD (
    CONSTRAINT PK_MARCATEMPO
    PRIMARY KEY (ID_MARCATEMPO))/

PROMPT Adding PRIMARY Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_MENSAGEM
    PRIMARY KEY (ID_MENSAGEM))/

PROMPT Adding PRIMARY Constraint To MENSAGEM_MARCATEMPO Table
ALTER TABLE MENSAGEM_MARCATEMPO ADD (
    CONSTRAINT PK_MENSA_MARCATEMPO
    PRIMARY KEY (ID_MARCATEMPO,
        ID_MENSAGEM,
        ID_CENARIO))/

PROMPT Adding PRIMARY Constraint To MENSAGEM_OBJETO Table
```

```
ALTER TABLE MENSAGEM_OBJETO ADD (
    CONSTRAINT PK_MENSAOBJETO
    PRIMARY KEY (ID_TIPOMENSAGEM,
            ID_MENSAGEM,
            ENVIA,
            RECEBE))/

PROMPT Adding PRIMARY Constraint To MULTIPLICIDADE Table
ALTER TABLE MULTIPLICIDADE ADD (
    CONSTRAINT PK_MULTIPLICIDADE
    PRIMARY KEY (ID_MULTIPLICIDADE))/

PROMPT Adding PRIMARY Constraint To OBJETO Table
ALTER TABLE OBJETO ADD (
    CONSTRAINT PK_OBJETO
    PRIMARY KEY (ID_OBJETO))/

PROMPT Adding PRIMARY Constraint To PERSISTENCIA Table
ALTER TABLE PERSISTENCIA ADD (
    CONSTRAINT PK_PERSISTENCIA
    PRIMARY KEY (ID_PERSISTENCIA))/

PROMPT Adding PRIMARY Constraint To PROJETO Table
ALTER TABLE PROJETO ADD (
    CONSTRAINT PK_PROJETO
    PRIMARY KEY (ID_PROJETO))/

PROMPT Adding PRIMARY Constraint To QUALIFICADOR Table
ALTER TABLE QUALIFICADOR ADD (
    CONSTRAINT PK_QUALIFICADOR
    PRIMARY KEY (ID_TERMINADOR,
            NM_QUALIFICADOR))/

PROMPT Adding PRIMARY Constraint To RELACIONAMENTO Table
ALTER TABLE RELACIONAMENTO ADD (
    CONSTRAINT PK_RELACIONAMENTO
    PRIMARY KEY (ID_RELACIONAMENTO))/

PROMPT Adding PRIMARY Constraint To SINCRONIZACAO Table
ALTER TABLE SINCRONIZACAO ADD (
    CONSTRAINT PK_SINCRONIZACAO
    PRIMARY KEY (ID_SINCRONIZACAO))/

PROMPT Adding PRIMARY Constraint To TEMPO_ACAO Table
ALTER TABLE TEMPO_ACAO ADD (
    CONSTRAINT PK_TEMPOACAO
    PRIMARY KEY (ID_TEMPOACAO))/

PROMPT Adding PRIMARY Constraint To TERMINADOR Table
```

```
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT PK_TERMINADOR
    PRIMARY KEY (ID_TERMINADOR))/

PROMPT Adding PRIMARY Constraint To TIPO Table
ALTER TABLE TIPO ADD (
    CONSTRAINT PK_TIPO
    PRIMARY KEY (ID_TIPO))/

PROMPT Adding PRIMARY Constraint To TIPO_CLASSE Table
ALTER TABLE TIPO_CLASSE ADD (
    CONSTRAINT PK_TIPOCLASSE
    PRIMARY KEY (ID_TIPOCLASSE))/

PROMPT Adding PRIMARY Constraint To TIPO_ESTADO Table
ALTER TABLE TIPO_ESTADO ADD (
    CONSTRAINT PK_TIPOESTADO
    PRIMARY KEY (ID_TIPOESTADO))/

PROMPT Adding PRIMARY Constraint To TIPO_MENSAGEM Table
ALTER TABLE TIPO_MENSAGEM ADD (
    CONSTRAINT PK_TIPOMENSAGEM
    PRIMARY KEY (ID_TIPOMENSAGEM))/

PROMPT Adding PRIMARY Constraint To TIPO_RELACIONAMENTO Table
ALTER TABLE TIPO_RELACIONAMENTO ADD (
    CONSTRAINT PK__TPRELACIONAMENTO
    PRIMARY KEY (ID_TPRELACIONAMENTO))/

PROMPT Adding PRIMARY Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT PK_TRANSICAO
    PRIMARY KEY (ID_ELEMDISPARADOR,
            ORIGEM,
            DESTINO))/

PROMPT Adding PRIMARY Constraint To TRATAMENTO_EXCECAO Table
ALTER TABLE TRATAMENTO_EXCECAO ADD (
    CONSTRAINT FK_TRATEXCECAO
    PRIMARY KEY (ID_EXECUCAO,
            ID_METODO))/

PROMPT Adding PRIMARY Constraint To USECASE Table
ALTER TABLE USECASE ADD (
    CONSTRAINT PK_USECASE
    PRIMARY KEY (ID_USECASE))/

PROMPT Adding PRIMARY Constraint To VISIBILIDADE Table
ALTER TABLE VISIBILIDADE ADD (
```

```
    CONSTRAINT PK_VISIBILIDADE
    PRIMARY KEY (ID_VISIBILIDADE))/

PROMPT Adding UNIQUE Constraint To ACAO_ATIVIDADE Table
ALTER TABLE ACAO_ATIVIDADE ADD (
    CONSTRAINT UN_ACAOATIVIDADE
    UNIQUE    (ID_METODO,
               ID_TEMPOACAO,
               ID_ESTADO))/

PROMPT Adding UNIQUE Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT UK_CLASSE
    UNIQUE    (ID_DGCLASSE,
               NM_CLASSE))/

PROMPT Adding UNIQUE Constraint To ESTEREOTIPO Table
ALTER TABLE ESTEREOTIPO ADD (
    CONSTRAINT UK_ESTEREOTIPO
    UNIQUE    (ID_ESTEREOTIPO,
               DS_ESTEREOTIPO,
               OBJETO_ESTEREOTIPADO))/

PROMPT Adding UNIQUE Constraint To TIPO Table
ALTER TABLE TIPO ADD (
    CONSTRAINT UK_DSTIPO
    UNIQUE    (DS_TIPO))/

PROMPT Adding FOREIGN Constraint To ACAO_ATIVIDADE Table
ALTER TABLE ACAO_ATIVIDADE ADD (
    CONSTRAINT FK_ESTACAOATIVIDADE
    FOREIGN KEY (ID_ESTADO)
    REFERENCES  ESTADO (
               ID_ESTADO))/

PROMPT Adding FOREIGN Constraint To ACAO_ATIVIDADE Table
ALTER TABLE ACAO_ATIVIDADE ADD (
    CONSTRAINT FK_METACAOATIVIDADE
    FOREIGN KEY (ID_METODO)
    REFERENCES  DEFINICAO_METODO (
               ID_METODO))/

PROMPT Adding FOREIGN Constraint To ACAO_ATIVIDADE Table
ALTER TABLE ACAO_ATIVIDADE ADD (
    CONSTRAINT FK_DOCACAOATIVIDADE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
               ID_DOCUMENTACAO))/
```

```
PROMPT Adding FOREIGN Constraint To ARGUMENTO Table
ALTER TABLE ARGUMENTO ADD (
    CONSTRAINT FK_TPARGUMENTO
    FOREIGN KEY (TIPO)
    REFERENCES  TIPO (
            ID_TIPO))/

PROMPT Adding FOREIGN Constraint To ARGUMENTO Table
ALTER TABLE ARGUMENTO ADD (
    CONSTRAINT FK_DEFMETARGUMENTO
    FOREIGN KEY (ID_METODO)
    REFERENCES  DEFINICAO_METODO (
            ID_METODO))/

PROMPT Adding FOREIGN Constraint To ARGUMENTO Table
ALTER TABLE ARGUMENTO ADD (
    CONSTRAINT FK_MENSARGUMENTO
    FOREIGN KEY (ID_MENSAGEMEVENTO)
    REFERENCES  MENSAGEM (
            ID_MENSAGEM))/

PROMPT Adding FOREIGN Constraint To ARGUMENTO Table
ALTER TABLE ARGUMENTO ADD (
    CONSTRAINT FK_EVENTOARGUMENTO
    FOREIGN KEY (ID_MENSAGEMEVENTO)
    REFERENCES  EVENTO (
            ID_EVENTO))/

PROMPT Adding FOREIGN Constraint To ARGUMENTO Table
ALTER TABLE ARGUMENTO ADD (
    CONSTRAINT FK_DOCARGUMENTO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO)
)/

PROMPT Adding FOREIGN Constraint To ATRIBUTO Table
ALTER TABLE ATRIBUTO ADD (
    CONSTRAINT FK_CLASSEATRIB
    FOREIGN KEY (ID_CLASSE)
    REFERENCES  CLASSE (
            ID_CLASSE))/

PROMPT Adding FOREIGN Constraint To ATRIBUTO Table
ALTER TABLE ATRIBUTO ADD (
    CONSTRAINT FK_OBJETOATRIB
    FOREIGN KEY (ID_OBJETO)
    REFERENCES  OBJETO (
            ID_OBJETO))/
```

```
PROMPT Adding FOREIGN Constraint To ATRIBUTO Table
ALTER TABLE ATRIBUTO ADD (
    CONSTRAINT FK_VISIBILATRIB
    FOREIGN KEY (CONTROLE)
    REFERENCES  VISIBILIDADE (
            ID_VISIBILIDADE))/

PROMPT Adding FOREIGN Constraint To ATRIBUTO Table
ALTER TABLE ATRIBUTO ADD (
    CONSTRAINT FK_ESTEREOATRIB
    FOREIGN KEY (ESTEREOTIPO)
    REFERENCES  ESTEREOTIPO (
            ID_ESTEREOTIPO))/

PROMPT Adding FOREIGN Constraint To ATRIBUTO Table
ALTER TABLE ATRIBUTO ADD (
    CONSTRAINT FK_DOCATRIBUTO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To CENARIO Table
ALTER TABLE CENARIO ADD (
    CONSTRAINT FK_DOCCENARIO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_CLASSE
    FOREIGN KEY (ID_DGCLASSE)
    REFERENCES  DG_CLASSE (
            ID_DGCLASSE))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_VISIBILIDADE
    FOREIGN KEY (CONTROLE)
    REFERENCES  VISIBILIDADE (
            ID_VISIBILIDADE))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_PERSISTENCIA
    FOREIGN KEY (PERSISTENCIA)
    REFERENCES  PERSISTENCIA (
            ID_PERSISTENCIA))/
```

```
PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_EXPRESSAO
    FOREIGN KEY (RESTRICAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_CONCORRENCIA
    FOREIGN KEY (CONCORRENCIA)
    REFERENCES  CONCORRENCIA (
            ID_CONCORRENCIA))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_ESTEREOTIPO
    FOREIGN KEY (ESTEREOTIPO)
    REFERENCES  ESTEREOTIPO (
            ID_ESTEREOTIPO))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_TIPOCLASSE
    FOREIGN KEY (TP_CLASSE)
    REFERENCES  TIPO_CLASSE (
            ID_TIPOCLASSE))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_MULTIPLICIDADE
    FOREIGN KEY (INSTANCIA)
    REFERENCES  MULTIPLICIDADE (
            ID_MULTIPLICIDADE))/

PROMPT Adding FOREIGN Constraint To CLASSE Table
ALTER TABLE CLASSE ADD (
    CONSTRAINT FK_DOCCLASSE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To CONCORRENCIA Table
ALTER TABLE CONCORRENCIA ADD (
    CONSTRAINT FK_DOCCONCORRENCIA
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/
```

```
PROMPT Adding FOREIGN Constraint To CORPO_METODO Table
ALTER TABLE CORPO_METODO ADD (
    CONSTRAINT FK_DEFMET_CORPOMET
    FOREIGN KEY (ID_METODO)
    REFERENCES  DEFINICAO_METODO (
            ID_METODO))/

PROMPT Adding FOREIGN Constraint To CORPO_METODO Table
ALTER TABLE CORPO_METODO ADD (
    CONSTRAINT FK_EXPCORPOMETODO
    FOREIGN KEY (ID_CORPO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To CORPO_METODO Table
ALTER TABLE CORPO_METODO ADD (
    CONSTRAINT FK_DOCCORPOMETODO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_TPDEFMETODO
    FOREIGN KEY (RETORNO)
    REFERENCES  TIPO (
            ID_TIPO))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_CLASSEDEFMETODO
    FOREIGN KEY (ID_CLASSE)
    REFERENCES  CLASSE (
            ID_CLASSE))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_CONCORDEFMET
    FOREIGN KEY (CONCORRENCIA)
    REFERENCES  CONCORRENCIA (
            ID_CONCORRENCIA))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_EXPDEFMETODO
    FOREIGN KEY (RESTRICAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/
```

```
PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_ESTEREODEFMETODO
    FOREIGN KEY (ESTEREOTIPO)
    REFERENCES  ESTEREOTIPO (
            ID_ESTEREOTIPO))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_VISIBILDEFMETODO
    FOREIGN KEY (CONTROLE)
    REFERENCES  VISIBILIDADE (
            ID_VISIBILIDADE))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_OBJETODEFMETODO
    FOREIGN KEY (ID_OBJETO)
    REFERENCES  OBJETO (
            ID_OBJETO))/

PROMPT Adding FOREIGN Constraint To DEFINICAO_METODO Table
ALTER TABLE DEFINICAO_METODO ADD (
    CONSTRAINT FK_DOCDEFMETODO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To DG_CLASSE Table
ALTER TABLE DG_CLASSE ADD (
    CONSTRAINT FK_DGCLASSE
    FOREIGN KEY (ID_PROJETO)
    REFERENCES  PROJETO (
            ID_PROJETO))/

PROMPT Adding FOREIGN Constraint To DG_CLASSE Table
ALTER TABLE DG_CLASSE ADD (
    CONSTRAINT FK_DOCDGCLASSE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To DG_ESTADO Table
ALTER TABLE DG_ESTADO ADD (
    CONSTRAINT FK_CLASSEDGESTADO
    FOREIGN KEY (ID_CLASSE)
    REFERENCES  CLASSE (
            ID_CLASSE))/
```

```
PROMPT Adding FOREIGN Constraint To DG_ESTADO Table
ALTER TABLE DG_ESTADO ADD (
    CONSTRAINT FK_DOCDGESTADO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To DG_INTERACAO Table
ALTER TABLE DG_INTERACAO ADD (
    CONSTRAINT FK_DGINTERACAO
    FOREIGN KEY (ID_USECASE)
    REFERENCES  USECASE (
            ID_USECASE))/

PROMPT Adding FOREIGN Constraint To DG_INTERACAO Table
ALTER TABLE DG_INTERACAO ADD (
    CONSTRAINT FK_CENARIO
    FOREIGN KEY (ID_CENARIO)
    REFERENCES  CENARIO (
            ID_CENARIO))/

PROMPT Adding FOREIGN Constraint To DG_INTERACAO Table
ALTER TABLE DG_INTERACAO ADD (
    CONSTRAINT FK_DOCDGINTERACAO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To DG_USECASE Table
ALTER TABLE DG_USECASE ADD (
    CONSTRAINT FK_DGUSECASE
    FOREIGN KEY (ID_PROJETO)
    REFERENCES  PROJETO (
            ID_PROJETO))/

PROMPT Adding FOREIGN Constraint To DG_USECASE Table
ALTER TABLE DG_USECASE ADD (
    CONSTRAINT FK_DOCDGUSECASE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To ELEMENTO_DISPARADOR Table
ALTER TABLE ELEMENTO_DISPARADOR ADD (
    CONSTRAINT FK_ACAOATIVELEMDISPARADOR
    FOREIGN KEY (ACAOATIVIDADE)
    REFERENCES  ACAO_ATIVIDADE (
            ID_ACAOATIVIDADE))/
```

```
PROMPT Adding FOREIGN Constraint To ELEMENTO_DISPARADOR Table
ALTER TABLE ELEMENTO_DISPARADOR ADD (
    CONSTRAINT FK_EVENTOELEMDISPARADOR
    FOREIGN KEY (EVENTO)
    REFERENCES  EVENTO (
            ID_EVENTO))/

PROMPT Adding FOREIGN Constraint To ELEMENTO_DISPARADOR Table
ALTER TABLE ELEMENTO_DISPARADOR ADD (
    CONSTRAINT FK_EXPELEMDISPARADOR
    FOREIGN KEY (CONDICAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To ELEMENTO_DISPARADOR Table
ALTER TABLE ELEMENTO_DISPARADOR ADD (
    CONSTRAINT FK_DOCELEMDISPARADOR
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To ESTADO Table
ALTER TABLE ESTADO ADD (
    CONSTRAINT FK_DGESTADOESTADO
    FOREIGN KEY (ID_DGESTADO)
    REFERENCES  DG_ESTADO (
            ID_DGESTADO))/

PROMPT Adding FOREIGN Constraint To ESTADO Table
ALTER TABLE ESTADO ADD (
    CONSTRAINT FK_TIPOESTADO
    FOREIGN KEY (TIPO)
    REFERENCES  TIPO_ESTADO (
            ID_TIPOESTADO))/

PROMPT Adding FOREIGN Constraint To ESTADO Table
ALTER TABLE ESTADO ADD (
    CONSTRAINT FK_DOCESTADO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To ESTADO_EMBUTIDO Table
ALTER TABLE ESTADO_EMBUTIDO ADD (
    CONSTRAINT FK_ESTADOESTADO
    FOREIGN KEY (ID_ESTADO)
    REFERENCES  ESTADO (
            ID_ESTADO))/
```

```
PROMPT Adding FOREIGN Constraint To ESTADO_EMBUTIDO Table
ALTER TABLE ESTADO_EMBUTIDO ADD (
    CONSTRAINT ESTADOEMBUTIDO
    FOREIGN KEY (ID_ESTADOEMBUTIDO)
    REFERENCES  ESTADO (
            ID_ESTADO))/

PROMPT Adding FOREIGN Constraint To ESTEREOTIPO Table
ALTER TABLE ESTEREOTIPO ADD (
    CONSTRAINT FK_DOCESTEREOTIPO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To EVENTO Table
ALTER TABLE EVENTO ADD (
    CONSTRAINT FK_METODOEVENTO
    FOREIGN KEY (ID_METODO)
    REFERENCES  DEFINICAO_METODO (
            ID_METODO))/

PROMPT Adding FOREIGN Constraint To EVENTO Table
ALTER TABLE EVENTO ADD (
    CONSTRAINT FK_DOCEVENTO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To EXECUCAO_METODO Table
ALTER TABLE EXECUCAO_METODO ADD (
    CONSTRAINT FK_DEFMETEXECUCAO
    FOREIGN KEY (ID_METODO)
    REFERENCES  DEFINICAO_METODO (
            ID_METODO))/

PROMPT Adding FOREIGN Constraint To EXECUCAO_METODO Table
ALTER TABLE EXECUCAO_METODO ADD (
    CONSTRAINT FK_DOCEXECMETODO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To EXPRESSAO Table
ALTER TABLE EXPRESSAO ADD (
    CONSTRAINT FK_DOCEXPRESSAO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/
```

```
PROMPT Adding FOREIGN Constraint To INTERACAO Table
ALTER TABLE INTERACAO ADD (
    CONSTRAINT FK_CENARIOINTERACAO
    FOREIGN KEY (ID_CENARIO)
    REFERENCES  CENARIO (
            ID_CENARIO))/

PROMPT Adding FOREIGN Constraint To INTERACAO Table
ALTER TABLE INTERACAO ADD (
    CONSTRAINT FK_MENSAOBJINTERACAO
    FOREIGN KEY (ID_TIPOMENSAGEM,
            ID_MENSAGEM,
            ENVIA,
            RECEBE)
    REFERENCES  MENSAGEM_OBJETO (
            ID_TIPOMENSAGEM,
            ID_MENSAGEM,
            ENVIA,
            RECEBE)
)/

PROMPT Adding FOREIGN Constraint To INTERACAO Table
ALTER TABLE INTERACAO ADD (
    CONSTRAINT FK_DOCINTERACAO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_SINCRONMENSA
    FOREIGN KEY (SINCRONIZACAO)
    REFERENCES  SINCRONIZACAO (
            ID_SINCRONIZACAO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_EXECMETMENSA
    FOREIGN KEY (ID_EXECUCAO)
    REFERENCES  EXECUCAO_METODO (
            ID_EXECUCAO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_GUARDA
    FOREIGN KEY (GUARDA)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/
```

```
PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_PRECONDICAO
    FOREIGN KEY (PRECONDICAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_TIPORETORNOMENSA
    FOREIGN KEY (RETORNO)
    REFERENCES  TIPO (
            ID_TIPO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_ESTEREOMENSA
    FOREIGN KEY (ESTEREOTIPO)
    REFERENCES  ESTEREOTIPO (
            ID_ESTEREOTIPO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_EXPINTERACAO
    FOREIGN KEY (EXP_INTERACAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM Table
ALTER TABLE MENSAGEM ADD (
    CONSTRAINT FK_DOCMENSAGEM
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM_MARCATEMPO Table
ALTER TABLE MENSAGEM_MARCATEMPO ADD (
    CONSTRAINT FK_MENSAMARCATEMPO
    FOREIGN KEY (ID_MENSAGEM)
    REFERENCES  MENSAGEM (
            ID_MENSAGEM))/

PROMPT Adding FOREIGN Constraint To MENSAGEM_MARCATEMPO Table
ALTER TABLE MENSAGEM_MARCATEMPO ADD (
    CONSTRAINT FK_CENARIOMARCATEMPO
    FOREIGN KEY (ID_CENARIO)
    REFERENCES  CENARIO (
            ID_CENARIO))/
```

```
PROMPT Adding FOREIGN Constraint To MENSAGEM_MARCATEMPO Table
ALTER TABLE MENSAGEM_MARCATEMPO ADD (
    CONSTRAINT FK_MARCATEMPMENSA
    FOREIGN KEY (ID_MARCATEMPO)
    REFERENCES  MARCA_TEMPO (
            ID_MARCATEMPO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM_OBJETO Table
ALTER TABLE MENSAGEM_OBJETO ADD (
    CONSTRAINT FK_OBJMENSAOBJ
    FOREIGN KEY (ENVIA)
    REFERENCES  OBJETO (
            ID_OBJETO))/

PROMPT Adding FOREIGN Constraint To MENSAGEM_OBJETO Table
ALTER TABLE MENSAGEM_OBJETO ADD (
    CONSTRAINT FK_MENSAMENSAOBJ
    FOREIGN KEY (ID_MENSAGEM)
    REFERENCES  MENSAGEM (
            ID_MENSAGEM))/

PROMPT Adding FOREIGN Constraint To MENSAGEM_OBJETO Table
ALTER TABLE MENSAGEM_OBJETO ADD (
    CONSTRAINT FK_TIPOMENSAOBJETO
    FOREIGN KEY (ID_TIPOMENSAGEM)
    REFERENCES  TIPO_MENSAGEM (
            ID_TIPOMENSAGEM))/

PROMPT Adding FOREIGN Constraint To MENSAGEM_OBJETO Table
ALTER TABLE MENSAGEM_OBJETO ADD (
    CONSTRAINT FK_OBJMENSARECEBE
    FOREIGN KEY (RECEBE)
    REFERENCES  OBJETO (
            ID_OBJETO))/

PROMPT Adding FOREIGN Constraint To MULTIPLICIDADE Table
ALTER TABLE MULTIPLICIDADE ADD (
    CONSTRAINT FK_DOCMULTIPLICIDADE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To OBJETO Table
ALTER TABLE OBJETO ADD (
    CONSTRAINT FK_CLASSEOBJETO
    FOREIGN KEY (ID_CLASSE)
    REFERENCES  CLASSE (
            ID_CLASSE))/
```

```
PROMPT Adding FOREIGN Constraint To OBJETO Table
ALTER TABLE OBJETO ADD (
    CONSTRAINT FK_PERSISTAOBJ
    FOREIGN KEY (PERSISTENCIA)
    REFERENCES  PERSISTENCIA (
            ID_PERSISTENCIA))/

PROMPT Adding FOREIGN Constraint To OBJETO Table
ALTER TABLE OBJETO ADD (
    CONSTRAINT FK_DOCOBJETO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To PERSISTENCIA Table
ALTER TABLE PERSISTENCIA ADD (
    CONSTRAINT FK_DOCCONSISTENCIA
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To PROJETO Table
ALTER TABLE PROJETO ADD (
    CONSTRAINT FK_DOCPROJETO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To QUALIFICADOR Table
ALTER TABLE QUALIFICADOR ADD (
    CONSTRAINT FK_RELACIONQUALIFIC
    FOREIGN KEY (ID_TERMINADOR)
    REFERENCES  TERMINADOR (
            ID_TERMINADOR))/

PROMPT Adding FOREIGN Constraint To QUALIFICADOR Table
ALTER TABLE QUALIFICADOR ADD (
    CONSTRAINT FK_TIPOQUALIFICADOR
    FOREIGN KEY (TIPO)
    REFERENCES  TIPO (
            ID_TIPO))/

PROMPT Adding FOREIGN Constraint To QUALIFICADOR Table
ALTER TABLE QUALIFICADOR ADD (
    CONSTRAINT FK_DOCQUALIFICADOR
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/
```

```
PROMPT Adding FOREIGN Constraint To RELACIONAMENTO Table
ALTER TABLE RELACIONAMENTO ADD (
    CONSTRAINT FK_ATRIBRELACIONAMENTO
    FOREIGN KEY (ATRIB_RELACIONAMENTO)
    REFERENCES  CLASSE (
            ID_CLASSE))/

PROMPT Adding FOREIGN Constraint To RELACIONAMENTO Table
ALTER TABLE RELACIONAMENTO ADD (
    CONSTRAINT FK_EXPRERELACION
    FOREIGN KEY (RESTRICAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To RELACIONAMENTO Table
ALTER TABLE RELACIONAMENTO ADD (
    CONSTRAINT FK_ESTERELACION
    FOREIGN KEY (ESTEREOTIPO)
    REFERENCES  ESTEREOTIPO (
            ID_ESTEREOTIPO))/

PROMPT Adding FOREIGN Constraint To RELACIONAMENTO Table
ALTER TABLE RELACIONAMENTO ADD (
    CONSTRAINT FK_DOCRELACIONAMENTO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To SINCRONIZACAO Table
ALTER TABLE SINCRONIZACAO ADD (
    CONSTRAINT FK_DOCSINCRONIZACAO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TEMPO_ACAO Table
ALTER TABLE TEMPO_ACAO ADD (
    CONSTRAINT FK_DOCTEMPOACAO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_RELACIONAMENTO
    FOREIGN KEY (ID_RELACIONAMENTO)
    REFERENCES  RELACIONAMENTO (
            ID_RELACIONAMENTO))/
```

```
PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_CLASSETERMIN
    FOREIGN KEY (ID_CLASSE)
    REFERENCES  CLASSE (
            ID_CLASSE))/

PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_TPRELACIONAMENTO
    FOREIGN KEY (TP_RELACIONAMENTO)
    REFERENCES  TIPO_RELACIONAMENTO (
            ID_TPRELACIONAMENTO))/

PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_VISIBILTERMIN
    FOREIGN KEY (CONTROLE)
    REFERENCES  VISIBILIDADE (
            ID_VISIBILIDADE))/

PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_MULTIPTERMIN
    FOREIGN KEY (MULTIPLICIDADE)
    REFERENCES  MULTIPLICIDADE (
            ID_MULTIPLICIDADE))/

PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_EXPRETERMIN
    FOREIGN KEY (RESTRICAO)
    REFERENCES  EXPRESSAO (
            ID_EXPRESSAO))/

PROMPT Adding FOREIGN Constraint To TERMINADOR Table
ALTER TABLE TERMINADOR ADD (
    CONSTRAINT FK_DOCTERMINADOR
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TIPO Table
ALTER TABLE TIPO ADD (
    CONSTRAINT FK_DOCTIPO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/
```

```
PROMPT Adding FOREIGN Constraint To TIPO_CLASSE Table
ALTER TABLE TIPO_CLASSE ADD (
    CONSTRAINT FK_DOCTPCLASSE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TIPO_ESTADO Table
ALTER TABLE TIPO_ESTADO ADD (
    CONSTRAINT FK_DOCTPESTADO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TIPO_MENSAGEM Table
ALTER TABLE TIPO_MENSAGEM ADD (
    CONSTRAINT FK_DOCTPMENSAGEM
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TIPO_RELACIONAMENTO Table
ALTER TABLE TIPO_RELACIONAMENTO ADD (
    CONSTRAINT FK_DOCTPRELACIONAMENTO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT FK_ESTADORIGEM
    FOREIGN KEY (ORIGEM)
    REFERENCES  ESTADO (
            ID_ESTADO))/

PROMPT Adding FOREIGN Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT FK_ESTADODESTINO
    FOREIGN KEY (DESTINO)
    REFERENCES  ESTADO (
            ID_ESTADO))/

PROMPT Adding FOREIGN Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT FK_CONECTORDESTINO
    FOREIGN KEY (DESTINO)
    REFERENCES  CONECTOR_CONDICIONAL (
            ID_CONECTOR))/
```

```
PROMPT Adding FOREIGN Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT FK_CONECTORORIGEM
    FOREIGN KEY (ORIGEM)
    REFERENCES  CONECTOR_CONDICIONAL (
            ID_CONECTOR))/

PROMPT Adding FOREIGN Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT FK_ELEMDISPTRANSICAO
    FOREIGN KEY (ID_ELEMDISPARADOR)
    REFERENCES  ELEMENTO_DISPARADOR (
            ID_ELEMDISPARADOR))/

PROMPT Adding FOREIGN Constraint To TRANSICAO Table
ALTER TABLE TRANSICAO ADD (
    CONSTRAINT FK_DOCTRANSICAO
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

PROMPT Adding FOREIGN Constraint To TRATAMENTO_EXCECAO Table
ALTER TABLE TRATAMENTO_EXCECAO ADD (
    CONSTRAINT FK_EXECMETTRATEXCECAO
    FOREIGN KEY (ID_EXECUCAO)
    REFERENCES  EXECUCAO_METODO (
            ID_EXECUCAO))/

PROMPT Adding FOREIGN Constraint To TRATAMENTO_EXCECAO Table
ALTER TABLE TRATAMENTO_EXCECAO ADD (
    CONSTRAINT FK_DEFMETTRATEXCECAO
    FOREIGN KEY (ID_METODO)
    REFERENCES  DEFINICAO_METODO (
            ID_METODO))/

PROMPT Adding FOREIGN Constraint To USECASE Table
ALTER TABLE USECASE ADD (
    CONSTRAINT FK_USECASE
    FOREIGN KEY (ID_DGUSECASE)
    REFERENCES  DG_USECASE (
            ID_DGUSECASE))/

PROMPT Adding FOREIGN Constraint To USECASE Table
ALTER TABLE USECASE ADD (
    CONSTRAINT FK_DOCUSECASE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/
```

```
PROMPT Adding FOREIGN Constraint To VISIBILIDADE Table
ALTER TABLE VISIBILIDADE ADD (
    CONSTRAINT FK_DOCVISIBILIDADE
    FOREIGN KEY (DOCUMENTACAO)
    REFERENCES  DOCUMENTACAO_PROJETO (
            ID_DOCUMENTACAO))/

REM Indice para o diagrama de estados
PROMPT Creating Index IK_DGESTADO on Table DG_ESTADO
CREATE UNIQUE INDEX IK_DGESTADO ON DG_ESTADO
(     id_dgestado )
PCTFREE  10;

PROMPT Creating Index IN_PROJETO on Table PROJETO
CREATE INDEX IN_PROJETO ON PROJETO
(     nm_projeto ,
    id_projeto )
PCTFREE  10;

PROMPT Creating Index IN_QUALIFICADOR on Table QUALIFICADOR
CREATE INDEX IN_QUALIFICADOR ON QUALIFICADOR
(     id_terminador )
PCTFREE  10;

PROMPT Creating Index IN_VISIBILIDADE on Table VISIBILIDADE
CREATE INDEX IN_VISIBILIDADE ON VISIBILIDADE
(     tipo )
PCTFREE  10;
```

Bibliografia

[AMB 98] AMBLER, Scott W. **Building Object Applications that Work:** Your Step by Step Handbook for Developing Robust Systems with Object Technology. [S.l.]: Cambridge University Press, 1998.

[AWA 96] AWAD, Maher et al. **Object-Oriented Technology for Real Time Systems:** A Pratical Approach. [S.l.]: Prentice Hall, 1996.

[BEC 97] BECKER, Leandro. **O Uso de Objetos Ativos para a Modelagem e Implementação de Sistemas Tempo-Real:** Análise do Estado da Arte e Estudo de Caso. Porto Alegre: CPGCC da UFRGS, 1897.

[BEC 99] BECKER, Leandro Buss. **Ambiente de Modelagem e Implementação de Sistemas de Tempo Real Usando o Paradigma de Orientação a Objetos**. Porto Alegre: CPGCC/UFRGS, 1999. Dissertação de Mestrado.

[BEN 94] BENNETT, Stuart. **Real-Time Computer Control:** An Introduction. [S.l.]: Prentice Hall, 1994.

[BOD 97] BODILY, Susan; Woodfield, N. Scott. **Integrating Object-Oriented Methods and Models**. [S.l.]: Brigham Young University, Computer Science Departament, 1997.

[BOO 94] BOOCH, Grady. **Object OrientedvAnalysis and Design, with Aplications.** 2.ed. [S.l.]: The Benjamin/Cummings, 1994.

[BRU 99] BRUEL, Jean-Michel; FRANCE, Robert B. **Transforming UML Models to Formal Specification**. Disponível por www em www.cs.colostate.edu/UML99 (1999).

[BUR 94] BURNS, A Wellings A. J. HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. **Real Time System**, [S.l.], v.6, n.1, Jan. 1994.

[BUS 96] BUSCHMANN, Frank et al. **A System of Patterns:** Patterns Oriented Software Architecture. [S.l.]: Addison-Wesley,1996.

[COA 92] COAD, Peter et al. **Análise Baseado em Objetos**. Rio de Janeiro: Campus,1993.

[COA 93] COAD, Peter et al. **Projeto Baseado em Objetos**. Rio de Janeiro: Campus,1993.

[COA 97] COAD, Peter; NORTH, David; MAYFIELD, Mark. **Objects Models:** Strategies, Patterns, e Applications. Results in COAD, OMT and UNIFIED. 2 ed. [S.l.]: Yourdon Press, 1997.

[COO 94] COOK, S. Daniels, J. **Designing Object Systems:** Object Oriented Modelling with Syntropy. [S.l.]: Prentice Hall, 1994

[COP 97] COPSTEIN, Bernardo. **SiMOO: Plataforma Orientada a Objetos para a Simulação Discreta Multi-Paradigma.** Porto Alegre: CPGCC da

UFRGS, 1997. Tese de Doutorado.

[COP 98] COPSTEIN, Bernardo et al. **SiMOO - Manual do Usuário**. Versão 1.0, Porto Alegre: CPGCC da UFRGS, 1998. Documentação SiMOO.

[CRA 99] CRANEFIELD, Stephen; PURVIS, Martin. **UML as na Ontology Modelling Language**. [S.l.]: Department of Information Science. University of Otago. 1999.

[DOU 97] DOUGLASS, Bruce. **UML for Real-Time Systems Design**. Disponível por www em www.rational.com (1997).

[DOU 98] DOUGLASS, Bruce Powel. **Real-Time UML:** Developing Efficient Objects for Embedded Systems.[S.l.]: Addison-Wesley, 1998

[EVA 99a] EVANS, And et al. **Meta-Modeling Semantics of UML**. Disponível por www em www.cs.colostate.edu/UML99 (1999).

[EVA 99b] EVANS, And at al. **The UML as Formal Modeling Notation**. Disponível por www em www.cs.colostate.edu/UML99 (1999).

[EVA 99c] EVANS, Andy at al. **Advanced Methods and Tools for a Precise UML**. Disponível por www em www.trireme.com (1999).

[FIC 92] FICHMAN, Robert G. Kemerer, Chris F. **Object-Oriented and Conventional Analysis and Design Methologies:** Comparison and Critique. [S.l.]: Massachusetts Institute of Technology, IEE Computers, 1992. p. 22-39.

[FOW 97a] FOWLER, Martin. **UML Distilled:** Applying the Standard Object Modeling Language. Massachusetts: Addison-Wesley, 1997. 179p.

[FOW 97b] FOWLER, Martin. **Analysis Patterns:** Reusable Object Models. California: Addison-Wesley, 1997. 357p.

[FRE 96] FREITAS, André L. Castro. **Um Estudo de Metodologias de Análise e Projeto Orientado a Objetos**. Porto Alegre: CPGCC da UFRGS, 1996.

[FUR 98] FURLAN, José Davi. **Modelagem de Objetos através da UML - The Unified Modeling Language**. São Paulo: Makron Books do Brasil, 1998.

[GAM 95] GAMMA, Erich; Helm at al. **Design Patterns: Elements of Reusable Object-Oriented Software.** [S.l.]: Addison-Wesley,1995.

[GOS 98] GOSSAIN, Sanjiv. **Object Modeling and Design Strategies**. [S.l.]: Cambridge University Press, 1998.

[HAR 88] HAREL, David. On Visual Formalisms. **Communication of the ACM**, New York, v.31, n.5, May 1988.

[JAC 92] JACOBSON, Ivar. **Object-Oriented Software Engineering**. [S.l.]: Addison-Wesley,1992.

[JOR 99] JORNADA, João Francisco Homrich. **MET - *Model Editor Tool*:** Documentação do Projeto. Porto Alegre: CPGCC da UFRGS, 1998. Documentação SiMOO.

[KEN 97] KENT, Stuart. Constraint Diagrams: Visualizing Invariants in Object Oriented Models. In: OOPSLA, 1997. **Proceedings...** Disponível por www em www.it.brighton.ac.uk/staff/Stuart.Kent.

[KEN 99] KENT, Stuart; GIL, Joseph. Visualising Action Contracts in Object-Oriented Modeling. In: OOPSLA, 1999. **Proceedings...** Disponível por www em www.it.brighton.ac.uk/staff/Stuart.Kent.

[LAN 93] LANG, Neil. Shlaer-Mellor Object Oriented Analysis Rules. ***Software* Engineering Notes**, New York, v.18, n.1, January 1993

[LAP 97] LAPLANTE, Phillip A. **Real-Time Systems Design and Analysis:** An Engeneer's Handbook. New York: IEEE Press, 1997.

[LEE 98] LEE, Michael M. **Shlaer-Mellor Object Oriented Development:** A Manager's Practical Guide. Disponível por www em www.projtech.com (1998).

[MRA 89] MRACK, Flávio Roberto. **Protótipo de um Dicionário de Dados para um Editor Diagramático Generalizado**: trabalho de conclusão de curso. Porto Alegre: UFRGS, 1989.

[OBJ 97] OBJECTIME LIMITED. **Toolset Guide.** Canada: Objectime, 1997. Manual do Objectime.

[OCL 98] OBJECT Constraint Language Specification. Disponível por www em http://www.*software*.ibm.com/ad/ocl, (1998).

[ODE 95] ODELL, James J. **Meta-Modeling**. [S.l.: s.n.], 1995.

[ODE 98] ODELL, James J. **Advanced Object-Oriented Analysis & Design Using UML**. [S.l.]: Cambridge University, 1998.

[OMG 97] OBJECT MANAGEMENT GROUP. **Meta Object Facility (MOF) Specification**. Disponível por www em www.omg.com (1997). Document ad/97-18-14.

[OMG 99] OMG Unified Modeling Language Specification. Disponível em http://www.omg.com (jan. 1999).

[PAR 99] PARDI, Wilson et al. **SiMOO-RT (*Simulation Modeling Object-Oriented: Real-Time*) Uma Ferramenta de Modelagem, Simulação e Supervisão.** Porto Alegre: CPGEE, 1999. Apresentação em PowerPoint.

[PAS 94] PASTOR, Esteban. **Estudo Comparativo de Metodologias de Desenvolvimento de Software.** Porto Alegre: CPGCC daUFRGS, 1994.

[PAS 96] PASTOR, Esteban. **Uso de Metamodelos para o Estudo e Comparação de Metodologias de Desenvolvimento de Software.** Porto Alegre: CPGCC da UFRGS, 1996.

[PER 96] PEREIRA, Carlos E. Métodos de Análise de Sistemas Tempo Real Usando Técnicas de Orientação a Objetos. In: SIMPÓSIO BRASILEIRO DE SOFTWARE, SBES, 10., 1996, São Carlos - SP. **Anais...** São Carlos: UFSCAR, 1996.

[POM 98] POMPERMAIER, Leandro Bento; PRICE, Roberto Tom. Considerações sobre o Desenvolvimento de Sistemas de Informação na Internet - Um Editor de Modelagem Distribuída. In: WORKSHOP LATINO AMERICANO DE SOFTWARE, IDEAS, 1.,1998, Torres-RS. **Anais...** Porto Alegre: UFRGS, 1998.

[PRE 95] PRESSMAN, Roger. **Engenharia de Software.** Rio de Janeiro: Makron Books, 1995.1056p.

[RAS 97] RATIONAL *SOFTWARE* CORPORATION. **UML Semantics**, Versão_1.1, 1997. Manual que Descreve a Semântica Linguagem de Modelagem UML. Disponível por www em www.rational.com (1997).

[RAT 97] RATIONAL *SOFTWARE* CORPORATION. **UML Notation Guide**, Versão 1.1, 1997. Manual da Linguagem de Modelagem UML. Disponível por www em www.rational.com (1997).

[RUM 97] RUMBAUGH, James et al. **Modelagem e Projetos Baseado em Objetos**: Rio de Janeiro: Campus, 1997.

[SEL 94] SELIC, Bran et al. **Real-Time Object-Oriented Modeling.** [S.l.]: John Wiley & Sons, 1994.

[SIL 96] SILVA, Ricardo Pereira. **Avaliação de Metodologias de Análise e Projeto Orientadas a Objetos Voltadas ao Desenvolvimento de Aplicações, sob a Ótica de sua Utilização no Desenvolvimento de Frameworks Orientado a Objetos**. Porto Alegre: CPGCC da UFRGS, 1996.

[SIS 95] SISTLA, A. Prosad; WOLFSON, Ouri. Temporal Conditions and Integrity Constrains in Active Database Systems. **Sigmod Record**, New York, v.24, n.2, p. 269-279, June 1995. Trabalho apresentado na ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1995, San Jon, CA.

[SOU 97] SOUZA, Isabel Fernandes. **Análise de Metodologias de Desenvolvimento de Software OO, Objetivando uma Proposta de Extensão da Linguagem Unificada - UML, para Modelagem de Sistemas de Tempo Real**. Porto Alegre: CPGCC da UFRGS, 1997.

[TAS 98] TASKON A/S; REICH TECHNOLOGIES; HUMANS AND TECHNOLOGY. **The OOram Meta-Model:** Combining role models, interfaces, and classes to support system centric and program centric modeling. Version 1.0. Disponível por www em www.projexion.com (1998).

[TKA 94] TKACH, Daniel; PUTTICK, Richard. **Object Technology in Application Development**. [S.l.]: The Benjamin/Cummings, 1994.

[UML 98] UNIFIED Modeling Language for Real Time. Disponível por www em www.rational.com (1998).

[VAZ 92] VAZ, Jorge Peil Marques. **Um Repositório para Case de Dados & Funções.** Porto Alegre: CPGCC da UFRGS, 1992.

[WAR 99] WARMER, Jos B. Kleppe, Anneke G. **Object Constraint Language: Precise Modeling with UML**. [S.l.]: Addison Wesley, 1999.

[WID 92] WIDOM, Jennifer. The Starbust Rule System: Language Design, Implementation, and Application. **IEE Data Engineering Bulletin**, Special Issue on Active Database, New York, p. 15 – 18, Dec. 1992.

[WID 93] WIDOM, Jennifer. Deductive and Active Database: Two Paradigms or Ends of a Spectrum. In: INTERNATIONAL WORKSHOP ON RULES IN DATABASE SYSTEM, 1., 1993. **Proceedings...** [S.l.:s.n.], 1993.