152495-3

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Estudo para Implementação do Modelo TF-ORM

por

EDUARDO HENRIQUE PEREIRA DE ARRUDA

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

Prof. José Palazzo M. de Oliveira
Orientador

Porto Alegre, maio de 1996.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Arruda, Eduardo Henrique Pereira de
Um estudo para implementação do modelo TF-ORM / Eduardo Henrique Pereira de Arruda. - Porto Alegre: CPGCC da UFRGS, 1996.
129f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR - RS, 1996. Orientador: Oliveira, José Palazzo M. de.

1. Banco de dados 2. Objetos 3. Papéis 4. ORM 5. Modelos I. Oliveira, José Palazzo M. de. II. Título.

Banco de Dados - SBU
Banco: Dados
Modelos: Dados temporais
Modelo: Papeis
CNPq 1.03.03.00-6

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA
N.º CHAMADA: 681.32.072(043) A779E
33811
29,12,97
ORIGEM: D
DATA: 18/12/97
R$ 30,00
FUNDO: II
FORN.: II

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Profª. Wrana Panizzi
Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann
Pró-Reitor de Pesquisa: Prof. Pedro Cezar Dutra Fonseca
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

AGRADECIMENTOS

Agradeço ao professor José Palazzo Moreira de Oliveira, pela compreensão e, especialmente, confiança depositadas em minha pessoa.

Agradeço à professora Nina Edelweiss, pela sempre cordial e pronta recepção, dirimindo dúvidas nos momentos em que estas teimavam em impedir o prosseguimento deste trabalho.

Agradeço aos amigos, pela esperança sempre sincera de sucesso. Aos mestres, que colaboraram para o êxito de meus estudos. Aos colegas, por enxergarem, entre as agruras conjuntas, nossas perspectivas de futuro.

Agradeço a Tatiana, pelo companheirismo, que muitas vezes toca tanto quanto a paixão, ou o carinho, pois nele se pode vislumbrar uma das mais belas faces do amor.

Agradeço, com um amor todo especial, a minha família. Os temores, a certeza ante as incertezas, sempre palavras de apoio. Desculpo-me por certos momentos, em que destas não retirei seu verdadeiro significado. Hoje, quando aquelas dificuldades são passado, antevejo em meu futuro a única e eterna certeza que, junto a vocês, encontrarei sempre abrigo. Meu lar, mesmo que fisicamente distante, será sempre ao seu lado. Amor.

MOD. 2.3.2

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

SUMÁRIO

LISTA DE ABREVIATURAS 9
LISTA DE FIGURAS 10
LISTA DE TABELAS 13
RESUMO 14
ABSTRACT 16
1 INTRODUÇÃO 17
2 O MODELO DE OBJETOS COM PAPÉIS 21
2.1 Perspectiva Histórica 21
2.2 O Conceito de Objeto 21
2.3 Hierarquia de Classes 22
2.4 Ciclo de Vida de Objetos 23
2.5 O Conceito de Papel 25
3 O MODELO TF-ORM 27
3.1 Classes e Papéis 28
3.1.1 Papel Básico 28
3.1.2 Hierarquia de Classes e Papéis 29
3.1.2.1 Especialização 29
3.1.2.1.1 Superclasse OBJECT 30
3.1.2.2 Agregação 30
3.2 Instâncias 30
3.2.1 Identidade 30
3.2.2 Propriedades 31
3.2.2.1 Valores Pré-Definidos 31
3.2.2.2 Propriedades Pré-Definidas 32
3.2.3 Estados 32
3.2.3.1 Estados Pré-Definidos 33
3.2.4 Mensagens 33
3.2.4.1 Mensagens Pré-Definidas 34
3.2.5 Regras 34
3.2.5.1 Regras de Transição de Estado 34
3.2.5.1.1 Regras de Transição de Estado Condicionadas a Decisões 35
3.2.5.2 Regras de Integridade 36
3.2.5.3 Regras Definidas no Papel Básico 36
3.3 Domínios de Propriedades 36
3.3.1 Pontos no Tempo 37
3.3.2 Intervalos 38

3.3.3 Durações......38
3.3.4 Tipos de Dados para Informações Temporais Incompletas......38

3.4 Linguagem de Lógica Temporal......38
3.4.1 Operadores......39
3.4.1.1 Operadores sobre Tipos de Dados Temporais......39
3.4.1.2 Operadores Genéricos......40
3.4.2 Predicados......40
3.4.2.1 Predicados sobre Tipos de Dados Temporais......40
3.4.2.2 Predicados sobre Valores de Propriedades......40
3.4.2.3 Predicados sobre Objetos e Instâncias de Papel......40
3.4.3 Funções......41
3.4.3.1 Funções sobre Tipos de Dados Temporais......41
3.4.3.2 Funções sobre Valores de Propriedades......42
3.4.3.3 Funções sobre Objetos e Instâncias de Papel......42

3.5 Linguagem de Consulta......42

4 O SISTEMA DE GERÊNCIA DE BANCOS DE DADOS O_2......43

4.1 Requisitos de SGBDs Orientados a Objetos......44
4.1.1 Gerência de Objetos e Regras......44
4.1.2 Ampliação das Funções de Gerência de Dados......44
4.1.3 Sistemas Abertos de Gerência de Dados......44

4.2 Os Conceitos de Tipo e Classe......45

4.3 Especificação de Esquemas de Dados O_2......45
4.3.1 Definição de Tipos......45
4.3.2 Definição de Classes......46
4.3.3 Manipulação da Hierarquia de Classes......47
4.3.3.1 Superclasse *object*......48
4.3.4 Manipulação e Persistência de Instâncias......49
4.3.5 Identidade de Objetos......50
4.3.6 Regras de Transição de Estado e Regras de Restrição de Integridade......50

4.4 Definição de Aplicações O_2......50
4.4.1 Métodos......51
4.4.2 Aplicações......51
4.4.3 Programas......51
4.4.4 Funções......52
4.4.5 Execução Direta de Comandos......52

5 IMPLEMENTAÇÃO O_2 DO MODELO TF-ORM......53

5.1 Abordagem para Implementação de Objetos com Papéis......53

5.2 Classes......56
5.2.1 Descritores de Classes......56
5.2.2 Criação de Objetos......57

5.3 Papéis......57

5.3.1 Descritores de Papéis 58
5.3.2 Criação de Instâncias de Papéis 58

5.4 Hierarquia de Classes e Papéis 59
5.4.1 Especialização e Agregação 60

5.5 Propriedades 61
5.5.1 Propriedades Estáticas....61
5.5.2 Propriedades Dinâmicas 61
5.5.3 Propriedades Pré-Definidas....63

5.6 Estados 63
5.6.1 Estados Pré-definidos 64

5.7 Mensagens 64

5.8 Domínios de Propriedades 65
5.8.1 Domínios Simples....65
5.8.2 Domínios Complexos....66
5.8.3 Domínios de Dados Temporais....66
5.8.3.1 Pontos no Tempo....67
5.8.3.2 Intervalos....69
5.8.3.3 Durações....69
5.8.3.4 Tipos de Dados para Informações Temporais Incompletas....70

5.9 Linguagem de Lógica Temporal.... 70

5.10 Linguagem de Consulta 70

6 IMPLEMENTAÇÃO O_2 DO MODELO DE REGRAS TF-ORM.... 71

6.1 Mecanismos de Suporte a Regras.... 71

6.2 Análise das Regras TF-ORM 75
6.2.1 Regras de Transição de Estado 75
6.2.2 Regras de Transição de Estado Condicionadas a Decisões 76
6.2.3 Regras de Integridade....77

6.3 Abordagem para Implementação de Regras TF-ORM 78

6.4 Mecanismo de Ativação e Execução de Regras 79

7 ESTUDO DE CASO: AMBIENTE DE PRODUÇÃO 81

7.1 Representação Gráfica de Requisitos TF-ORM.... 81
7.1.1 Diagrama de *Clusters*....82
7.1.2 Diagrama de Classes 82
7.1.3 Diagramas de Transição de Estados 83

7.2 Descrição do Problema 84

7.3 Diagramas de Classes 84
7.3.1 *Cluster* Administração de Vendas 84
7.3.2 *Cluster* Administração de Produção....85

7.3.3 *Cluster* Administração de Materiais......85
7.3.4 *Cluster* Administração de Compras......86

7.4 Especificação de Classes......86
7.4.1 Classes de Recurso......86
7.4.1.1 Classe Documento......86
7.4.1.1.1 Papel Básico......86
7.4.1.1.2 Papel Pedido de Cliente......87
7.4.1.1.3 Papel Ordem de Serviço......87
7.4.1.1.4 Papel Ordem de Compra......88
7.4.1.1.5 Papel Nota Fiscal......89
7.4.1.2 Classe Item de Estoque......90
7.4.1.2.1 Papel Básico......90
7.4.1.2.2 Papel Produto......91
7.4.1.2.3 Papel Insumo......91
7.4.2 Classes de Processo......92
7.4.2.1 Classe Administração de Vendas......92
7.4.2.1.1 Papel Básico......92
7.4.2.1.2 Papel Controle de Pedido......92
7.4.2.2 Classe Administração de Produção......93
7.4.2.2.1 Papel Básico......93
7.4.2.2.2 Papel Linha de Produção......94
7.4.2.3 Classe Administração de Materiais......94
7.4.2.3.1 Papel Básico......94
7.4.2.3.2 Papel Gerência de Almoxarifado......95
7.4.2.3.3 Papel Controle de Estoque......95
7.4.2.4 Classe Administração de Compras......96
7.4.2.4.1 Papel Básico......96
7.4.2.4.2 Papel Controle de Compra......97
7.4.3 Classes de Agentes......97
7.4.3.1 Classe Pessoa......97
7.4.3.1.1 Papel Básico......97
7.4.3.1.2 Papel Cliente......98
7.4.3.1.3 Papel Funcionário......98
7.4.3.2 Classe Fornecedor......99
7.4.3.2.1 Papel Básico......99

8 CONCLUSÕES......100

ANEXO 1 SINTAXE DA LINGUAGEM TF-ORM......103

ANEXO 2 SINTAXE RESUMIDA DOS COMANDOS *O_2*......110

ANEXO 2.1 Definição de Classes......110

ANEXO 2.2 Definição de *Named-Objects* e *Named-Values*......110

ANEXO 2.3 Importação de Classes......110

ANEXO 2.4 Definição de Aplicações, Programas, Transações e Funções ... 111

ANEXO 2.5 Definição de Código O_2C ... 111

ANEXO 2.6 Definições Gerais ... 111

ANEXO 3 ESTUDO DE CASO: CÓDIGO TF-ORM ... 112

ANEXO 3.1 Classes Recurso ... 112
ANEXO 3.1.1 Classe Documento ... 112
ANEXO 3.1.2 Classe Item de Estoque ... 113

ANEXO 3.2 Classes Processo ... 114
ANEXO 3.2.1 Classe Administração de Vendas ... 114
ANEXO 3.2.2 Classe Administração de Produção ... 114
ANEXO 3.2.3 Classe Administração de Materiais ... 115
ANEXO 3.2.4 Classe Administração de Compras ... 116

ANEXO 3.3 Classes Agente ... 117
ANEXO 3.3.1 Classe Pessoa ... 117
ANEXO 3.3.2 Classe Fornecedor ... 118

ANEXO 4 ESTUDO DE CASO: CÓDIGO O_2 ... 119

ANEXO 4.1 Classes Recurso ... 119
ANEXO 4.1.1 Classe Documento ... 119
ANEXO 4.1.2 Classe Item de Estoque ... 120

ANEXO 4.2 Classes Processo ... 121
ANEXO 4.2.1 Classe Administração de Vendas ... 121
ANEXO 4.2.2 Classe Administração de Produção ... 121
ANEXO 4.2.3 Classe Administração de Materiais ... 121
ANEXO 4.2.4 Classe Administração de Compras ... 122

ANEXO 4.3 Classes Agente ... 122
ANEXO 4.3.1 Classe Pessoa ... 122
ANEXO 4.3.2 Classe Fornecedor ... 123

ANEXO 5 DESCRIÇÃO DE UMA FERRAMENTA DE TRADUÇÃO TF-ORM PARA O_2 ... 124

BIBLIOGRAFIA ... 126

LISTA DE ABREVIATURAS

BNF	*Backus Naur Form*
CAD	*Computer Aided Design*
CAM	*Computer Aided Manufacturing*
CASE	*Computer Aided Software Engineering*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
F-ORM	*Functionality in Objects with Roles Model*
OMT	*Object Modeling Technique*
ORM	*Objects with Roles Model*
RECAST	*Requirements Collection And Specification Tool*
SGBD	Sistema de Gerência de Bancos de Dados
SIE	Sistema de Informação de Escritórios
SQL	*Structured Query Language*
TF-ORM	*Temporal Functionality in Objects with Roles Model*

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

LISTA DE FIGURAS

FIGURA 2.1 - Classificação e Instanciação ... 22
FIGURA 2.2 - Generalização e Especialização ... 23
FIGURA 2.3 - Herança Múltipla ... 23
FIGURA 2.4 - Diagrama de Classes da Realidade *Cliente* e *Funcionário*, com Herança Simples (utilizando notação baseada no modelo OMT [RUM 91]) ... 24
FIGURA 2.5 - Diagrama de Classes da Realidade *Cliente* e *Funcionário*, com Herança Múltipla (utilizando notação baseada no modelo OMT [RUM 91]) ... 24
FIGURA 2.6 - Diagrama de Classes da Realidade *Cliente* e *Funcionário* (utilizando notação baseada em [BEL 92]) ... 26
FIGURA 4.1 - Evolução dos Modelos de Dados (Adaptado de [TAK 90]) ... 43
FIGURA 4.2 - Esquema de uma Classe O_2 ... 46
FIGURA 4.3 - Definição de um Método O_2 ... 47
FIGURA 5.1 - Mapeamento por Herança ... 54
FIGURA 5.2 - Mapeamento por Atributos ... 55
FIGURA 5.3 - Implementação de uma Classe TF-ORM ... 56
FIGURA 5.4 - Classe O_2 *Descriptor* ... 56
FIGURA 5.5 - Descritor de Classe TF-ORM ... 56
FIGURA 5.6 - Implementação de um Papel TF-ORM ... 57
FIGURA 5.7 - Conjuntos de Instâncias de Papéis ... 57
FIGURA 5.8 - Implementação do Relacionamento entre Classes e Papéis ... 58
FIGURA 5.9 - Descritor de Papel TF-ORM ... 58
FIGURA 5.10 - Classe O_2 *Object_class* ... 59
FIGURA 5.11 - Classe O_2 *Role_class* ... 60
FIGURA 5.12 - Evolução dos Valores das Propriedades Dinâmicas ... 61
FIGURA 5.13 - Classe O_2 *Dynamic_domain_class* ... 62
FIGURA 5.14 - Implementação de um Domínio para Definição de Propriedades Dinâmicas TF-ORM ... 62
FIGURA 5.15 - Classe O_2 *State_class* ... 63
FIGURA 5.16 - Classe O_2 *Integer* ... 65
FIGURA 5.17 - Classe O_2 *Boolean* ... 65
FIGURA 5.18 - Classe O_2 *String* ... 66
FIGURA 5.19 - Estrutura Hierárquica da Classe O_2 *Instant* ... 67
FIGURA 5.20 - Implementação das Classes Raízes da Hierarquia de *Instant* ... 67
FIGURA 5.21 - Classe O_2 *Date* ... 68
FIGURA 5.22 - Classe O_2 *Time* ... 68
FIGURA 5.23 - Classe O_2 *Instant* ... 69
FIGURA 5.24 - Classe O_2 *Instant_interval* ... 69

FIGURA 5.25 - Classe O_2 *Minute_duration* 70
FIGURA 6.1 - Classe O_2 *Rule_class* 78
FIGURA 7.1 - Componentes do Diagrama de *Clusters* 82
FIGURA 7.2 - Componentes do Diagrama de Classes 82
FIGURA 7.3 - Componentes do Diagrama de Transição de Estado 83
FIGURA 7.4 - Estudo de Caso: Diagrama de *Clusters* 84
FIGURA 7.5 - Diagrama de Classes para o *Cluster* Administração de Vendas 85
FIGURA 7.6 - Diagrama de Classes para o *Cluster* Administração de Produção 85
FIGURA 7.7 - Diagrama de Classes para o *Cluster* Administração de Materiais 85
FIGURA 7.8 - Diagrama de Classes para o *Cluster* Administração de Compras 86
FIGURA 7.9 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Documento* 86
FIGURA 7.10 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Pedido de Cliente* da Classe *Documento* 87
FIGURA 7.11 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Ordem de Serviço* da Classe *Documento* 88
FIGURA 7.12 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Ordem de Compra* da Classe *Documento* 89
FIGURA 7.13 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Nota Fiscal* da Classe *Documento* 89
FIGURA 7.14 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Item de Estoque* 90
FIGURA 7.15 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Produto* da Classe *Item de Estoque* 91
FIGURA 7.16 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Insumo* da Classe *Item de Estoque* 91
FIGURA 7.17 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Vendas* 92
FIGURA 7.18 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Controle de Pedido* da Classe *Administração de Vendas* 93
FIGURA 7.19 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Produção* 93
FIGURA 7.20 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Linha de Produção* da Classe *Administração de Produção* 94
FIGURA 7.21 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Materiais* 95
FIGURA 7.22 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Gerência de Almoxarifado* da Classe *Administração de Materiais* 95
FIGURA 7.23 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Controle de Estoque* da Classe *Administração de Materiais* 96
FIGURA 7.24 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Compras* 96

FIGURA 7.25 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Controle de Compra* da Classe *Administração de Compras*............................97
FIGURA 7.26 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Pessoa*...98
FIGURA 7.27 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Cliente* da Classe *Pessoa*...98
FIGURA 7.28 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Funcionário* da Classe *Pessoa*...99
FIGURA 7.29 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Fornecedor*..99
FIGURA A-5.1 - Esquema do Banco de Dados da Ferramenta *TFORM2O₂* (utilizando notação baseada em [YOU 92]) 124

LISTA DE TABELAS

TABELA 7.1 - Estudo de Caso: *Papel Básico* da Classe *Documento*........................86
TABELA 7.2 - Estudo de Caso: Papel *Pedido de Cliente* da Classe *Documento*.......87
TABELA 7.3 - Estudo de Caso: Papel *Ordem de Serviço* da Classe *Documento*.......87
TABELA 7.4 - Estudo de Caso: Papel *Ordem de Compra* da Classe *Documento*......88
TABELA 7.5 - Estudo de Caso: Papel *Nota Fiscal* da Classe *Documento*................89
TABELA 7.6 - Estudo de Caso: *Papel Básico* da Classe *Item de Estoque*................90
TABELA 7.7 - Estudo de Caso: Papel *Produto* da Classe *Item de Estoque*..............91
TABELA 7.8 - Estudo de Caso: Papel *Insumo* da Classe *Item de Estoque*...............91
TABELA 7.9 - Estudo de Caso: *Papel Básico* da Classe *Administração de Vendas*...92
TABELA 7.10 - Estudo de Caso: Papel *Controle de Pedido* da Classe *Administração de Vendas*...92
TABELA 7.11 - Estudo de Caso: *Papel Básico* da Classe *Administração de Produção*...93
TABELA 7.12 - Estudo de Caso: Papel *Linha de Produção* da Classe *Administração de Produção*...94
TABELA 7.13 - Estudo de Caso: *Papel Básico* da Classe *Administração de Materiais*...94
TABELA 7.14 - Estudo de Caso: Papel *Gerência de Almoxarifado* da Classe *Administração de Materiais*...95
TABELA 7.15 - Estudo de Caso: Papel *Controle de Estoque* da Classe *Administração de Materiais*...95
TABELA 7.16 - Estudo de Caso: *Papel Básico* da Classe *Administração de Compras*..96
TABELA 7.17 - Estudo de Caso: Papel *Controle de Compra* da Classe *Administração de Compras*..97
TABELA 7.18 - Estudo de Caso: *Papel Básico* da Classe *Pessoa*............................97
TABELA 7.19 - Estudo de Caso: Papel *Cliente* da Classe *Pessoa*............................98
TABELA 7.20 - Estudo de Caso: Papel *Funcionário* da Classe *Pessoa*.....................98
TABELA 7.21 - Estudo de Caso: *Papel Básico* da Classe *Fornecedor*......................99

RESUMO

O modelo de especificação de requisitos em sistemas de informação TF-ORM estende o conceito de orientação a objetos, dividindo o comportamento destes nos diversos papéis que podem desempenhar. Adicionalmente, introduz o suporte a propriedades com variação dinâmica de valor e definidas sobre domínios temporais, bem como a regras de transição de estado e restrição de integridade. Este trabalho apresenta uma abordagem para implementação TF-ORM sobre o sistema de gerência de bancos de dados orientado a objetos O_2.

A utilização de modelos orientados a objetos na especificação de sistemas de informação promove mudanças radicais na maneira como estes são analisados, projetados e implementados. No entanto, certos aspectos relacionados à evolução dinâmica do comportamento e dos valores das propriedades dos objetos não são plenamente satisfeitos através destes modelos. TF-ORM introduz novos conceitos, estendendo o modelo de orientação a objetos a fim de suportá-los.

Advindo do modelo de objetos com papéis (ORM), TF-ORM divide o comportamento dos objetos nos diversos papéis que estes podem desempenhar. São introduzidos conceitos para modelagem dos aspectos tempo-dependentes das aplicações, incluindo utilização de marcas de tempo associadas aos objetos e instâncias de papéis, e aos valores das propriedades com variação dinâmica, bem como a definição de domínios temporais e uma linguagem de lógica temporal.

TF-ORM permite a definição de restrições sobre o comportamento dinâmico das instâncias, através de regras de transição de estado, e sobre os valores das propriedades, através de regras de restrição de integridade.

Uma abordagem de implementação TF-ORM deve levar em consideração o conjunto destes aspectos, em especial: (i) suporte ao conceito de papéis, permitindo a criação de instâncias múltiplas e paralelas; (ii) suporte aos conceitos tempo-dependentes, incluindo a definição de *timestamps* e domínios de dados temporais; e (iii) suporte ao mecanismo de regras de transição de estado e regras de integridade.

Este trabalho apresenta um estudo para implementação do modelo TF-ORM sobre o sistema de gerência de banco de dados orientado a objetos O_2, o qual suporta plenamente os conceitos do modelo de orientação a objetos e os conceitos básicos de bancos de dados de segunda geração. Dentro deste estudo são analisadas as possibilidades de mapeamento do modelo de objetos com papéis para o modelo tradicional de orientação a objetos. Da mesma forma, requisitos para implementação de aspectos temporais são analisados.

O modelo de regras TF-ORM, baseado no estudo de diversos modelos de implementação, é mapeado para um modelo de regras E-C-A (evento-condição-ação), as quais permitem a transformação do O_2 em um sistema de bancos de dados

ativo, capaz de responder a estímulos não diretamente ligados a requisições de usuários.

A abordagem de implementação apresentada permite a especificação de grande parte da funcionalidade do modelo TF-ORM. A fim de certificar sua correção, é proposta uma ferramenta de tradução e desenvolvido um estudo de caso utilizando notação gráfica para especificação de requisitos TF-ORM.

PALAVRAS-CHAVES: Banco de dados, objetos, papéis, ORM, modelos.

TITLE: "A STUDY FOR TF-ORM IMPLEMENTATION"

ABSTRACT

The information systems' requirements specification model TF-ORM extends the object-oriented model, splitting the object behavior in different roles that it can perform. In addition, introduces the support to dynamic properties, temporal domains, state transition rules, and constraints. This work presents a TF-ORM implementation approach to the object-oriented database management system O_2.

The use of object-oriented models in information systems' specification radically changes the manner in which this systems are analyzed, designed, and implemented. However, some aspects related to dynamic behavior and property value evolution are not fully satisfied through these models. TF-ORM introduces a set of new concepts, extending object-oriented model to support these aspects.

Originated in the object with roles model (ORM), TF-ORM divides the objects behavior in the different roles that it can perform. Are introduced concepts to model time-dependent aspects, including timestamps associated to objects and roles instances, and to values of dynamic properties, as well as the temporal domain specification and a temporal logic language are supported.

TF-ORM allows the definition of constraints over the dynamic behavior of the instances, through state transition rules, and over the property values, through integrity constraints.

A TF-ORM implementation approach must consider all of these aspects, specially: (i) support to roles concept, allowing the creation of multiple and parallel instances; (ii) support to time-dependents concepts, including timestamps definition and temporal data domains; and (iii) support to state transition rules and integrity constraints.

This work presents a study to implement the TF-ORM model over O_2, an object-oriented database management system that supports entirely object-oriented and databases' second generation requirements. This study analyses the possibilities to mapping roles to traditional object-oriented model, and temporal aspects implementation requirements.

The TF-ORM rules model, based on the study of many implementation models, is mapped to a E-C-A (event-condition-action) rules model. E-C-A rules allow transforming O_2 in an active database, able to answer impulses not directly generated by users' requirements.

The implementation approach presented allows the specification of multiples aspects of the TF-ORM functionality. To certificate its correctness, is proposed a translate tool, and developed a study of case, using a graphical notation to TF-ORM requirements specification.

KEYWORDS: Databases, objects, roles, ORM, models.

1 INTRODUÇÃO

O processo de desenvolvimento de sistemas tem, na especificação de requisitos, uma de suas mais importantes fases. A fim de possibilitar a adequada representação da funcionalidade de um sistema, uma linguagem de especificação deve ser capaz de representar não somente as características estáticas, mas também as características dinâmicas e evolutivas das informações envolvidas. Por outro lado, deve possibilitar a correta implementação da especificação sobre um sistema que garanta a persistência dos dados.

A utilização de modelos orientados a objetos [BOO 91, COA 92, HEV 92, RUM 91, TAK 90, WIR 90] na especificação de requisitos em sistemas de informação, tem recebido progressiva atenção, seja pela redução da distância existente entre uma realidade e o modelo de sua representação, seja pelo conceito intrínseco de objeto. A utilização de noções de hierarquia de classes, identidade de objetos e encapsulamento de dados promovem mudanças radicais na maneira como são analisados, projetados e implementados os novos sistemas [HEV 92].

Embora esteja caracterizado pela flexibilidade e generalidade, o modelo de orientação a objetos imputa certa dificuldade à modelagem do comportamento dinâmico dos objetos, em especial à representação dos diferentes contextos de execução que estes podem desempenhar. Durante o ciclo de vida de um objeto, este reage aos estímulos do meio, representados por mensagens recebidas. Um mesmo objeto, no entanto, inserido em dois contextos (meios) distintos de execução, responderá de forma diferenciada aos estímulos que recebe. Desta forma, um mesmo objeto pode, durante sua existência, desempenhar diferentes *papéis*, possuindo um comportamento específico em cada um destes.

O modelo de objetos com papéis ORM [PER 90], e o modelo derivado F-ORM [DEA 91, DEA 92], estendem o conceito de objetos através da divisão de seus comportamentos em relação aos diversos contextos de execução nos quais estes podem estar inseridos. Com isso, é possível modelar, através de um objeto identificado univocamente, instâncias de papéis com execuções simultâneas e comportamentos distintos, ou mesmo múltiplas execuções de um mesmo comportamento.

Não apenas o comportamento de um objeto evolui ao longo de seu ciclo de vida; também as informações associadas a este podem assumir múltiplos valores. Da mesma forma, cada comportamento leva o objeto através de uma seqüência de estados abstratos, considerando os valores de suas propriedades [BOO 91].

A representação de informações e comportamentos que possuam uma evolução dinâmica requer que o modelo de especificação utilizado suporte a modelagem de características tempo-dependentes. TF-ORM [EDE 92, EDE 94] baseia-se no modelo ORM, introduzindo diversas noções relacionadas ao suporte de informações temporais e à compreensão da realidade de sistemas de informação.

Em TF-ORM, são associadas informações temporais: (i) aos objetos e seus papéis, indicando os instantes de criação e destruição, e os intervalos em que sua execução foi suspensa; (ii) às propriedades dos papéis, as quais podem ser definidas sobre domínios de dados temporais; e (iii) às propriedades cujos valores evoluem dinamicamente, armazenando sua história pregressa e associando a cada valor

(passado, presente ou futuro) rótulos de tempo. Adicionalmente são definidas uma linguagem de lógica temporal, utilizada na modelagem de condições associadas ao comportamento de objetos e seus papéis e uma linguagem de consulta, capaz de realizar a recuperação de valores simples ou temporais.

O modelo TF-ORM prevê, também, a modelagem do conjunto de mensagens às quais um objeto pode reagir em cada um de seus papéis, ocasionando uma mudança em seu estado, a criação de outras instâncias de papéis, ou mesmo a remoção de instâncias existentes. Estas mensagens podem ser associadas a regras de transição de estado, governando a evolução de um objeto. Regras de integridade definem restrições aos valores das propriedades, podendo ser definidas através da construção de expressões em lógica temporal.

TF-ORM considera três tipos distintos de objetos: (i) aqueles que modelam as informações relevantes a um sistema, representando *recursos* utilizados; (ii) *processos*, representando a parcela estruturada do trabalho, ou seja, as rotinas básicas que compõem a funcionalidade de um sistema; e (iii) *agentes*, responsáveis pela parcela não estruturada do trabalho, ou seja, o processo de tomada de decisões.

O Modelo F-ORM deu origem a uma técnica de modelagem, suportada pela ferramenta CASE denominada RECAST (*Requirements Collection And Specification Tool*) [BEL 92]. Tanto o modelo F-ORM como a ferramenta, foram desenvolvidos por pesquisadores do Dipartamento di Elettronica e Informazione do Politecnico di Milano, no âmbito do projeto ITHACA (*Integrated Toolkit for Highly Advanced Computer Applications*), visando a modelagem de sistemas de informação de escritórios. A notação e as ferramentas gráficas utilizadas no presente trabalho foram adaptadas daquelas propostas na ferramenta RECAST, e utilizadas na modelagem de um estudo de caso, visando o levantamento de pontos relevantes e a posterior certificação das estruturas propostas no modelo de implementação TF-ORM.

Como um modelo para especificação de sistema de informação, TF-ORM pressupõe a existência de um sistema de gerência de banco de dados (SGBD) que propicie a persistência e a segurança necessárias às informações. SGBDs orientados a objetos [ATK 89, COM 91, FRE 90], devem suportar uma série de requisitos característicos de outras gerações de sistemas ou relacionados diretamente ao conceito de objetos. O_2 [LEC 90, O2T 91, O2T 91a, O2T 91b] é um SGBD orientado a objetos, com características de suporte aos conceitos básicos de SGBDs relacionais, como transações de bancos de dados, em adição àqueles específicos da orientação a objetos, como a possibilidade de construção de tipos complexos e de hierarquia de classes.

A implementação do modelo de especificação TF-ORM sobre um SGBD como o O_2 encontra, nas características do conceito de objetos, sua maior vantagem e, simultaneamente, seu maior empecilho. A vantagem reside na possibilidade de modelar hierarquias de classes e, através destas, representar classes, papéis, regras e propriedades dinâmicas. O empecilho é representando pela impossibilidade de uma implementação natural, visto que o conceito de papéis não é suportado diretamente no modelo dito "tradicional" de orientação a objetos. Estruturas adicionais devem, portanto, ser definidas, a fim de estabelecer o suporte à criação de objetos e, para cada um destes, de múltiplas instâncias de papéis.

Devem ser previstas, igualmente, estruturas que dêem suporte à definição de propriedades com variação dinâmica no valor, armazenando, para cada uma destas, todos os valores já assumidos bem como os instantes de sua definição e do início de sua validade. O suporte a estas informações temporais requer a implementação de um banco de dados bitemporal, que permita a manipulação e recuperação dos valores válidos destas propriedades em todos os instantes [EDE 94]. Domínios temporais de propriedades devem ser suportados e, para cada um destes, devem ser providas operações de manipulação, recuperação e conversão de valores.

A implementação do mecanismo de regras TF-ORM, incluindo regras de transição de estado e regras de integridade, requer a definição de um banco de dados ativo, ou seja, capaz de ativar operações sem a intervenção direta de um usuário [BAU 91]. Regras [HUL 91] e gatilhos [SIE 92] têm sido propostos como uma maneira de estender a funcionalidade de SGBDs passivos, permitindo a ativação automática de determinadas operações.

Regras E-C-A (evento-condição-ação) podem ser consideradas um mecanismo genérico de prover capacidade ativa a SGBDs [DÍA 91]. Assim, um possível modelo de implementação das regras TF-ORM pode basear-se no mapeamento destas para regras E-C-A.

Existem diversas abordagens para a implementação de mecanismos de suporte a regras em SGBDs orientados a objetos. Algumas inserem o código das regras em métodos, outras definem uma hierarquia especial de classes modelando regras. Em alguns casos, o mecanismo de regras consiste em uma nova camada do SGBD, requerendo a alteração de sua estrutura interna [DÍA 91]. A segunda abordagem permite uma implementação uniforme, já que tira proveito dos próprios conceitos de orientação a objetos sem necessitar a alteração do código do SGBD [BAU 91, DAY 88, DÍA 91].

Diversos trabalhos têm sido realizados visando implementar mecanismos de regras sobre SGBDs orientados a objetos, utilizando diferentes abordagens e priorizando diferentes aspectos de implementação, como desempenho ou facilidade de utilização. Modelos como HiPAC [DAY 88] e ADAM [DÍA 91], utilizam a abordagem de hierarquia de classes, permitindo a manipulação de regras através de operações básicas sobre objetos. No modelo de regras O_2, apresentado em [BAU 91], são fornecidas novas operações do SGBD para suportar o conceito de regras, sem, contudo, afastar-se da implementação de regras como objetos.

Alguns trabalhos dão especial atenção a questões de desempenho na busca e execução de regras, como em Ariel [HAN 91], ou à ativação concorrente de regras [HSU 88] e às características de terminação e confluência de conjuntos de regras [VOO 91a].

O objetivo deste trabalho é apresentar uma abordagem para implementação TF-ORM sobre um SGBD passivo orientado a objetos, em especial, o O_2, cobrindo os seguintes aspectos: (i) suporte à noção de classes e papéis; (ii) suporte à definição de propriedades com evolução dinâmica e domínios temporais de propriedades; e (iii) suporte ao mecanismo de regras de transição de estado e regras de integridade.

Em trabalhos anteriores foram realizadas análises do modelo TF-ORM [ARR 92], bem como propostas soluções para alguns aspectos de sua implementação

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

[ARR 94, ARR 95]. Estudos para implementação do modelo TF-ORM foram realizados em [PAL 95]. Este trabalho sedimenta os conceitos apresentados nos trabalhos anteriores, descrevendo uma abordagem completa, incluindo o suporte ao mecanismo de regras de transição de estado e regras de integridade, possibilitando, desta forma, a implementação da maior parte das características TF-ORM.

O trabalho está organizado como segue. No capítulo 2 são analisados conceitos relacionados ao modelo de orientação a objetos, estabelecendo um paralelo entre este e o modelo de objetos com papéis. O modelo de objetos com papéis é apresentado como uma solução para modelagem de sistemas de informação. No capítulo 3, o modelo de especificação de requisitos TF-ORM é apresentado.

O capítulo 4 realiza uma breve análise dos requisitos a serem atendidos por SGBDs orientados a objetos, situando o SGBD O_2 frente a estes. As principais características deste sistema são descritas.

No capítulo 5, a abordagem proposta para a implementação TF-ORM é apresentada, enfocando, em especial, o suporte ao conceito de papéis, estados, propriedades dinâmicas e domínios de valores temporais. O capítulo 6 realiza um estudo sobre mecanismos de suporte ao conceito de regras, introduzindo uma proposta de implementação das regras TF-ORM sobre o SGBD O_2.

O capítulo 7 apresenta um estudo de caso utilizando as representações gráficas de requisitos TF-ORM, adaptadas de [BEL 92]. Nos anexos podem ser encontradas a sintaxe da linguagem de especificação TF-ORM e da linguagem de definição de dados do SGBD O_2, bem como o código TF-ORM e o código O_2 para o estudo de caso desenvolvido no capítulo 7. Uma ferramenta automática de tradução de esquemas TF-ORM para esquemas O_2 é descrita no Anexo 5.

2 O MODELO DE OBJETOS COM PAPÉIS

A crescente difusão do modelo de orientação a objetos [COA 92, BOO 91, HEV 92, RUM 91, TAK 90, WIR 90], em especial devido aos constantes avanços nas linguagens de desenvolvimento, suportados pelos novos conceitos em ambientes e sistemas operacionais, além de ressaltar suas qualidades, apresenta algumas inadequações no que se refere à modelagem de sistemas de informação.

Este capítulo apresenta uma análise do modelo de orientação a objetos, destacando suas principais características e analisando sua aplicação na modelagem de sistemas de informação. O modelo de objetos com papéis [DEA 91, DEA 92, PER 90] é apresentado como uma solução para a modelagem mais natural das características evolutivas existentes neste tipo de realidade.

2.1 Perspectiva Histórica

Um grande número de métodos, técnicas e ferramentas têm sido desenvolvidas a fim de fornecer suporte à grande demanda por sistemas de informação, especialmente nas áreas comercial e industrial. O modelo de orientação a objetos introduz uma nova perspectiva para o desenvolvimento destes sistemas, não mais os percebendo simplesmente como um conjunto de processos, fluxos de dados e fluxos de controle, mas sim como coleções de objetos identificados e interagentes. Tal abordagem promove uma mudança radical na maneira pela qual os sistemas são analisados, projetados e implementados [HEV 92].

Segundo Booch [BOO 91] os seguintes eventos contribuíram para o estabelecimento do atual modelo de orientação a objetos:

- avanços na arquitetura dos computadores incluindo características de suporte de hardware e sistema operacionais;
- avanços nas linguagens de programação, como o caso de Simula, Smalltalk, CLU e Ada;
- avanços nas metodologias de programação, incluindo modularização e ocultamento de informação;
- avanços nos modelos de bancos de dados;
- pesquisas em inteligência artificial;
- avanços na filosofia e ciências cognitivas.

Pelo fato dos conceitos de orientação a objetos terem sido derivados de diferentes áreas da ciência, não existe, ainda hoje, uma terminologia padrão, no entanto a maior parte dos conceitos já encontra consenso.

2.2 O Conceito de Objeto

Hevner [HEV 92] define um *objeto* como "alguma coisa perceptível". O ser humano possui uma habilidade cognitiva para perceber tanto objetos tangíveis (percebidos pelos sentidos) como intangíveis (dependentes de construções mentais). Baseado neste fato, é possível concluir que o ser humano percebe a realidade através

de objetos, generalizando tal conceito para incluir não apenas objetos propriamente ditos, mas também pessoas, documentos, etc., e os relacionamentos entre estes. Sendo assim, um possível modelo baseado em objetos, no qual os componentes do espaço dos problemas (realidade a ser modelada) fossem mapeados diretamente para o espaço das soluções (implementação), seria a maneira mais natural para o desenvolvimento de sistemas [TAK 90].

O atual modelo de orientação a objetos caracteriza objetos a partir de três conceitos básicos [BOO 91]:

- o *estado* do objeto, baseado no contexto atual de sua execução, levando em consideração os valores atuais de um conjunto de *propriedades*;
- o *comportamento* do objeto, ou como este responde aos estímulos do ambiente, através de alterações no seu estado e conseqüentes respostas a este ambiente. Geralmente o comportamento é modelado como um conjunto de *métodos*, cada um destes ativado após o recebimento de uma *mensagem* específica;
- a *identidade* do objeto, propriedade inerente a todos e que os distingue univocamente[1]. Cabe notar que, naturalmente, esta é uma limitação imposta pelo modelo, já que os objetos reais são diferenciados, em sua maioria, por características subjetivas, advindas da combinação de informações obtidas pelos sentidos.

É possível, ainda, identificar um quarto conceito, relacionado com a propriedade que os objetos possuem de agrupar-se segundo suas características comuns: uma *classe* reúne objetos com as mesmas propriedades e métodos. Assim, um objeto pode ser compreendido como uma *instância* de uma classe. Na Figura 2.1, *Pedro* e *Paulo* são instâncias da classe *Cliente* e, como tais, possuem as propriedades e o métodos definidos para os objetos desta classe.

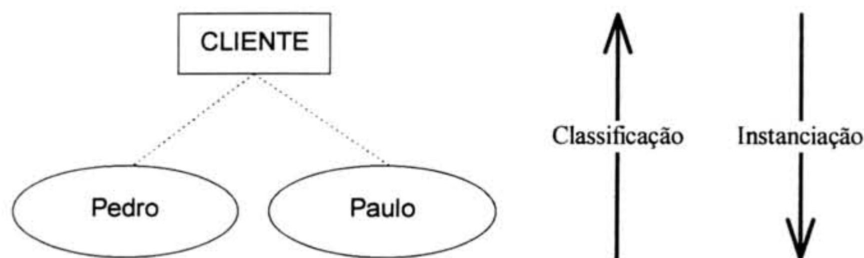


FIGURA 2.1 - Classificação e Instanciação

2.3 Hierarquia de Classes

Um conjunto de classes pode possuir características semelhantes, sendo *generalizadas* em uma classe de nível superior, a qual reúne as propriedades e métodos comuns às demais. É possível, também, refinar uma determinada classe, criando uma

[1] A partir deste momento, os identificadores unívocos de objetos passam a ser referidos como *oId* (*object identifier*).

classe *especializada*, com propriedades e métodos particulares, além daqueles já definidos.

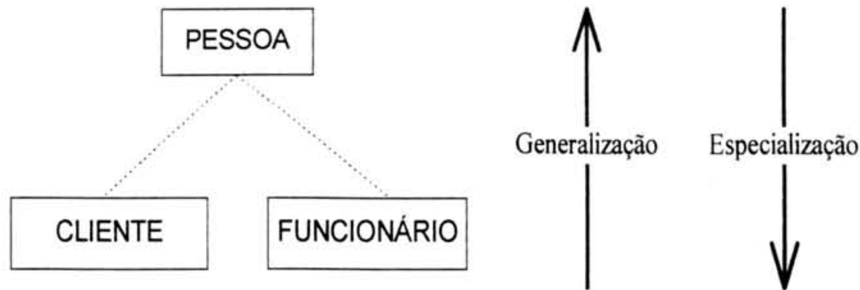


FIGURA 2.2 - Generalização e Especialização

Com isso, pode ser estabelecida uma hierarquia de classes, na qual as classes de um nível inferior (*subclasses*) "herdam" as propriedades e os métodos da classe de nível imediatamente superior (*superclasse*). Este mecanismo é conhecido como *herança simples*, sendo possível definir novas propriedades e métodos ou mesmo redefinir aqueles da superclasse. Na Figura 2.2, as subclasses *Cliente* e *Funcionário* herdam as propriedades e os métodos da superclasse *Pessoa*, possivelmente definindo novas propriedades e novos métodos específicos, ou mesmo redefinindo aqueles da classe *Cliente*.

FIGURA 2.3 - Herança Múltipla

Em alguns casos, é possível estabelecer uma relação de herança com mais de uma superclasse. Este mecanismo é conhecido como *herança múltipla*. Na Figura 2.3, a classe *Funcionário Especial* reúne todas aquelas *Pessoas* que são tanto *Funcionários* como *Clientes*, ou seja, possuem propriedades e métodos tanto de *Funcionários* como de *Clientes* e, possivelmente, propriedades e métodos específicos ou redefinidos[2].

2.4 Ciclo de Vida de Objetos

Um objeto possui um *ciclo de vida*: é criado, interage com o ambiente e com os demais objetos segundo seu comportamento, evoluindo ao longo de uma seqüência de estados e, finalmente, é eliminado.

---

[2] Não são tratadas neste texto, porém de suma relevância, considerações sobre os conflitos decorrentes da herança múltipla, como a herança repetida e os conflitos entre nomes de propriedades e métodos. Para maiores informações, consulte [O2T 91, RUM 91, WIR 90].

Ao ser criado, um objeto é instanciado em uma classe específica, recebendo um identificador unívoco (identidade de objeto) e possuindo as propriedades e os métodos definidos para esta classe. Durante o seu ciclo de vida, um objeto pode ser especializado, ou seja, instanciado em uma sucessão de classes de nível inferior, desde que, cada uma destas, seja subclasse da classe atual e assim sucessivamente.

Ao ser instanciado em uma classe, um objeto será regido pelo comportamento descrito para os objetos desta classe, ou seja, poderá responder somente às mensagens definidas para esta. Igualmente, os estados pelos quais irá evoluir, serão caracterizados pelos valores das propriedades definidas em sua classe.

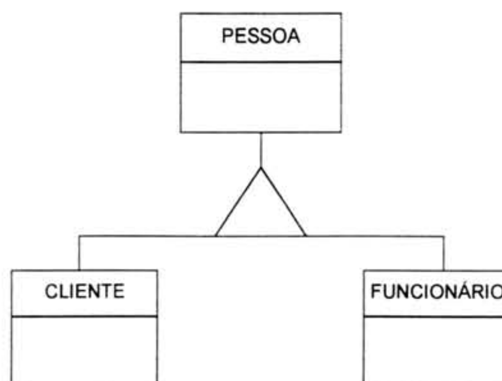


FIGURA 2.4 - Diagrama de Classes da Realidade *Cliente* e *Funcionário*, com Herança Simples (utilizando notação baseada no modelo OMT [RUM 91])

A Figura 2.4 representa a realidade de *Pessoas*, *Clientes* e *Funcionários*.

Um objeto que represente um funcionário, ao ser criado, será instanciado na classe *Funcionário*, ou na classe *Pessoa* e posteriormente especializado em *Funcionário*. Ao longo de sua existência este objeto (o qual possui um identificador unívoco específico e determinado) será sempre um funcionário, ou seja, será sempre uma instância da classe *Funcionário*.

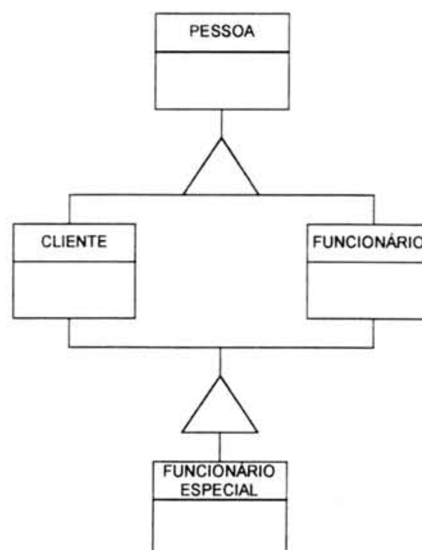


FIGURA 2.5 - Diagrama de Classes da Realidade *Cliente* e *Funcionário*, com Herança Múltipla (utilizando notação baseada no modelo OMT [RUM 91])

Caso exista a possibilidade de pessoas comportarem-se como clientes e funcionários simultaneamente, a realidade pode ser modelada como na Figura 2.5. Desta forma, as pessoas que são clientes e funcionários simultaneamente serão instâncias da classe *Funcionário Especial*.

Considerando outra situação: uma mesma pessoa trabalha em duas empresas e, em cada uma destas, possui existência distinta e paralela. Neste caso, para modelar tal pessoa, seria necessário criar duas instâncias da classe *Funcionário*, uma representando cada existência.

Nesta situação, um mesmo objeto real (a pessoa sendo modelada), estará sendo representada por dois objetos distintos (com identificadores distintos), o que contraria a idéia de que a orientação a objetos seria uma solução natural para a modelagem de sistemas de informação, pois um mesmo objeto real estaria sendo representado por dois objetos distintos.

Com isso, conclui-se que o modelo atual de orientação a objetos não permite a modelagem natural da evolução dinâmica das propriedades e do comportamento dos objetos reais, já que não permite a modelagem de existências paralelas de um mesmo comportamento de objeto.

2.5 O Conceito de Papel

O modelo de objetos com papéis tem, como principal característica, a modelagem dos aspectos dinâmicos da evolução dos objetos. Neste modelo, o comportamento definido em cada classe é dividido, de acordo com os diferentes contextos em que é possível a execução de um objeto, isto é, cada classe é dividida nos diversos *papéis* que podem ser desempenhados pelos objetos instanciados nesta [PER 90].

Cada papel especifica as características de um determinado tipo de comportamento. Um objeto instanciado em uma classe pode possuir instâncias de diferentes papéis, ou seja, pode apresentar comportamentos alternativos dependentes do contexto de sua execução e dos estímulos que recebe do ambiente. Com base nisto, é possível caracterizar um papel a partir de três conceitos:

- um *estado* do papel, baseado no contexto de execução do objeto, levando em consideração os valores que o objeto apresenta para as propriedades definidas para este papel;
- um *comportamento* de papel, descrevendo como este objeto reage aos estímulos do ambiente quando desempenhando este papel;
- uma *identidade* de papel, que identifica univocamente as instâncias de papéis desempenhados por um mesmo objeto, e as instâncias de papéis desempenhados por diferentes objetos[3].

Cada classe possui um *papel básico*, reunindo as características comuns a todos os demais papéis, e modelando um comportamento padrão para os objetos instanciados nesta.

---

[3] A partir deste momento, os identificadores unívocos de instâncias de papéis passam a ser referidos como *rId* (*role identifier*).

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Ao ser criada uma instância de determinada classe, o objeto recebe um identificador e é imediatamente instanciado no papel básico, ou seja, passa a possuir as características e o comportamento definidos neste papel. A partir deste momento, podem ser criadas instâncias dos demais papéis e, cada uma, receberá um identificador de instância de papel.

Utilizando o modelo de objetos com papéis, o exemplo de clientes e funcionários poderia ser modelado através de uma classe *Pessoa*, dividida em dois papéis, *Cliente* e *Funcionário* (Figura 2.6).

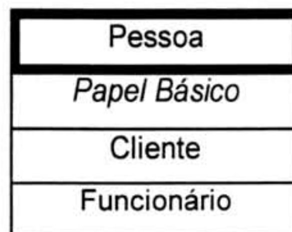


FIGURA 2.6 - Diagrama de Classes da Realidade *Cliente* e *Funcionário* (utilizando notação baseada em [BEL 92])

Através desta solução é possível modelar as situações em que uma pessoa é cliente e funcionário simultaneamente ou é funcionário de duas empresas. A criação do objeto que representa a pessoa corresponde à criação de uma instância (objeto) da classe *Pessoa* e à decorrente instanciação no papel básico. A partir deste momento, este objeto possui as características comuns a todas as instâncias da classe *Pessoa* (por exemplo, nome, endereço, etc.) e o comportamento padrão definido para estas (por exemplo, métodos de atualização de endereço, de modificação do número de dependentes, etc.). A situação em que uma pessoa é cliente e funcionário é modelada através da criação de uma instância do papel *Cliente* e uma instância do papel *Funcionário*. A situação em que uma pessoa é funcionário de duas empresas é modelada através da criação de duas instâncias do papel *Funcionário*, com ciclos de vida independentes e paralelos.

A caracterização de um objeto dentro do modelo de objetos com papéis pode ser feita, portanto, a partir dos seguintes conceitos:

- um *estado do objeto*;
- um *comportamento padrão* para os objetos;
- uma *identidade de objeto* (*oId*);
- um *conjunto de papéis*, caracterizando cada um dos comportamentos alternativos. Cada papel caracteriza-se por:
    - um *estado do papel*;
    - um *comportamento* para as instâncias do papel;
    - uma *identidade de instância de papel* (*rId*).

3 O MODELO TF-ORM

TF-ORM [EDE 92, EDE 94] é um modelo conceitual, orientado a objetos, no qual o conceito de papéis fornece um mecanismo básico de abstração para a especificação de sistemas de informação. Este modelo é uma extensão do modelo de dados F-ORM [DEA 91, DEA 92], o qual visa especificamente a representação de sistemas de informação de escritórios, através da extensão do modelo de dados ORM [PER 90].

Através do modelo TF-ORM, o desenvolvedor pode construir o esquema de dados de um sistema de informação utilizando classes para encapsular aspectos estáticos e dinâmicos da aplicação, modelar os aspectos tempo-dependentes utilizando domínios temporais e linguagem de lógica temporal, regras de transição de estado e restrições de integridade.

Convém salientar que, como um modelo de especificação de dados, TF-ORM não considera aspectos do projeto e implementação de aplicações, salvo aqueles relacionados às transições de estado e às restrições de integridade. Desta forma, após a especificação de um esquema de dados TF-ORM, fases adicionais de projeto e implementação devem definir aplicações sobre este esquema, possivelmente através da especificação de novos métodos (associados a mensagens) nas classes e papéis [DEA 91].

Por estar voltado à modelagem de sistemas de informação de escritórios, TF-ORM baseia a análise das aplicações em certos conceitos, utilizados para qualificar os diferentes componentes do escritório [DEA 92]:

- *área funcional*: uma divisão da organização com respeito às similaridades e diferenças de objetivos / produtos / resultados do trabalho a ser executado;
- *atividade*: trabalho a ser realizado para o qual um objetivo significativo e/ou um conjunto facilmente identificado de entradas e saídas pode ser identificado;
- *agente*: uma qualificação ou cargo organizacional;
- *tarefa* (atividade elementar): uma parcela de uma atividade designada a um agente;
- *operação*: unidade elementar de trabalho relevante para a descrição da organização (a granularidade depende da profundidade da análise realizada);
- *procedimento*: um conjunto de tarefas descrevendo o comportamento de um grupo de agentes;
- *informação/documentos*: documentos ou objetos envolvidos nas operações / tarefas / procedimentos.

Este capítulo apresenta o modelo TF-ORM, incluindo as definições de classes e papéis, bem como as extensões para o suporte à modelagem de informações tempo-dependentes.

3.1 Classes e Papéis

Uma classe TF-ORM é definida por um nome *cn*, que a identifica univocamente em toda a hierarquia de classes da aplicação, e por um conjunto de papéis, representando os diferentes comportamentos dos objetos desta classe.

$$class = (cn, R_0, R_1, ..., R_i, ..., R_n)$$

São identificados três tipos distintos de classes: *classes recurso*, *classes agente* e *classes processo*. As classes recurso definem a estrutura das *informações/documentos* manipulados, em termos dos papéis que estes podem desempenhar durante o seu ciclo de vida. As classes agente representam os *agentes*, executores de tarefas, aos quais compete a parcela não estruturada do trabalho: o processo de tomada de decisões. As classes processo integram as classes recurso e agente, permitindo a descrição do trabalho realizado no escritório em termos de sua organização e da cooperação entre os agentes envolvidos. Cada classe processo modela um *procedimento* do escritório.

$$R_i = (Rn_i, P_i, S_i, M_i, Ru_i)$$

Cada papel R_i é representado por um nome Rn_i, que o identifica univocamente dentro da classe a que pertence; um conjunto de *propriedades* P_i, implementadas como atributos; um conjunto de *estados* S_i representando os contextos de execução possíveis para as instâncias deste papel; um conjunto de *mensagens* M_i que as instâncias deste papel podem enviar ou receber; um conjunto de *regras de transição de estado* e *regras de integridade* Ru_i, descrevendo o comportamento dinâmico das instâncias do papel, através de restrições aplicadas às transições de estado, e o comportamento estático, através de restrições aos valores possíveis das propriedades.

$$R_i = (Rn_i, P_i, S_i, M_i, \mathbf{D_i}, Ru_i)$$

Nos papéis definidos em classes agente são modeladas, adicionalmente, as *decisões* D_i que podem ser tomadas pelas instâncias deste papel. As decisões aparecem nas regras de transição de estado e têm por finalidade condicioná-las a uma tomada de decisão pelo agente.

3.1.1 Papel Básico

Toda classe TF-ORM inclui a definição de um papel especial, denominado *papel básico*, o qual descreve as propriedades comuns a todos os papéis da classe e regula a evolução do objeto através dos demais papéis. Suas regras geralmente coordenam a criação e remoção de instâncias de papéis. Ao ser criada uma instância de classe, é, automaticamente, instanciado o seu papel básico, que, ao contrário dos demais, pode possuir apenas uma instância.

3.1.2 Hierarquia de Classes e Papéis

3.1.2.1 Especialização

Uma classe TF-ORM pode ser definida como subclasse de uma (herança simples) ou mais classes (herança múltipla). A herança é identificada pela cláusula *is_a*.

<classe> is_a (<classe>, <classe>, ...)

Quanto às especificações dos papéis da subclasse, cinco situações são previstas no modelo[4]:

- *papéis herdados sem modificações*: todos os papéis da(s) superclasse(s) que não forem listados na definição de papéis da subclasse (com exceção do papel básico) são herdados sem modificações. Pode ser utilizada a cláusula *inherits* para enumerar estes papéis (opcional);

  inherits (<superclasse>.<papel>,<superclasse>.<papel>,...)

- *papéis estendidos*: um papel pode ser estendido através da inclusão de novas propriedades, novos estados, novas mensagens e novas regras. Este papéis são identificados pela cláusula *extends*;

  extends (<superclasse>.<papel>,<superclasse>.<papel>,...)

- *papéis totalmente redefinidos*: um papel é totalmente redefinido quando a subclasse define novamente este papel e seu nome não está incluído na cláusula *extends*;
- *papéis novos*: são papéis definidos na subclasse e cuja definição não consta de nenhuma superclasse;
- *papéis não herdados*: os papéis não herdados da(s) superclasse(s) são identificados pela cláusula *not_inherits*.

  not_inherits (<superclasse>.<papel>,...)

As propriedades do papel básico da subclasse não necessitam ser redefinidas, sendo formadas pela união das propriedades definidas nos papéis básicos das superclasses. As regras do papel básico da subclasse devem ser totalmente redefinidas, a fim de refletir as modificações na estrutura dos papéis.

Quando duas superclasses apresentarem papéis com nomes iguais, o conflito pode ser resolvido de duas maneiras: (i) os papéis a serem herdados são especificados na cláusula *inherits*, acompanhados do nome da superclasse; ou (ii) os papéis que não devem ser herdados são especificados na cláusula *not_inherits*, igualmente acompanhados do nome da superclasse. Quando, na hierarquia de herança, duas superclasses de níveis diferentes apresentarem papéis de nomes iguais, o papel herdado será o da superclasse de nível mais baixo.

---

[4] Nas cláusulas que seguem, a identificação da superclasse não é necessária, a menos que exista conflito de nomes de papéis.

3.1.2.1.1 Superclasse OBJECT

O modelo TF-ORM pré-define uma classe denominada *OBJECT*, raiz da hierarquia de classes, da qual todas as demais são especializações. As propriedades desta classe são herdadas por todas as demais, sejam do tipo agente, recurso ou processo. O papel básico da classe *OBJECT* possui duas propriedades dinâmicas pré-definidas denominadas *object_instance* e *end_object* herdadas por todos os papéis básicos das demais classes.

A classe *OBJECT* possui um papel pré-definido denominado *ROLE*, no qual são definidas duas propriedades dinâmicas *role_instance* e *end_role*, herdadas, sem possibilidade de redefinição, por todos os demais papéis.

As mensagens da classe *OBJECT* são pré-definidas: *create_object*, *suspend_object*, *resume_object* e *kill*, coordenando as instâncias de classes; *add_role*, *suspend_role*, *resume_role* e *terminate_role*, coordenando as instâncias de papéis; *forbid_role*, *allow_role*, *forbid_op*, *allow_op*, *forget* e *recall*, coordenando aspectos da execução das instâncias de papel.

3.1.2.2 Agregação

A agregação no modelo TF-ORM é permitida somente para classes do tipo *recurso*. A agregação é identificada pela cláusula *composed_of*.

<classe> composed_of (<classe>, <classe>, ...)

Determinados papéis das classes podem ser excluídos da agregação, através da cláusula *not_include*.

not_include (<superclasse>.<papel>,...)

3.2 Instâncias

O modelo TF-ORM prevê a existência de dois tipos distintos de instâncias: instâncias de classes, também denominadas *objetos*, e *instâncias de papéis*. Dependendo da classe a que pertença, um objeto pode corresponder a um procedimento (classe processo), um agente (classe agente) ou a um recurso (classe recurso).

3.2.1 Identidade

O modelo TF-ORM utiliza um mecanismo de identificação unívoca de instâncias, tanto de classes quanto de papéis, a fim de permitir sua manipulação, especialmente nas regras de transição de estado e regras de integridade e nas condições associadas a estas. De forma similar ao modelo de objetos com papéis, é definido um identificador unívocos de objeto (*oId*) e um identificador unívoco de instância de papel (*rId*). Assim, um objeto fica perfeitamente caracterizado pelo seu *oId*, ao passo que uma instância de papel, em referências externas ao seu objeto, é caracterizada pelo par *oId.rId*.

3.2.2 Propriedades

Cada propriedade p_{ij} pertencente ao conjunto de propriedades P_i definido em um papel, é descrita através de um nome pn_{ij} e de um domínio d_{ij}.

$$p_{ij} \in P_i \mid p_{ij} = (pn_{ij},\ d_{ij})$$

O modelo TF-ORM identifica dois tipos básicos de propriedades: as *propriedades estáticas* e as *propriedades dinâmicas*. Uma propriedade é considerada estática se o seu valor não possui variação constante durante o ciclo de vida da instância ou se os seus valores pregressos não são relevantes para o modelo. Desta forma, quando uma propriedade estática assume um valor, este provavelmente se manterá invariável durante a existência da instância.

As propriedades dinâmicas são aquelas que podem apresentar diversos valores durante a existência das instâncias, e o conhecimento destes valores possui relevância para o modelo. Todos os valores assumidos por uma propriedade dinâmica ficam armazenados indefinidamente, tendo associados a si a informação temporal sobre sua validade, pressupondo a existência de um banco de dados temporal. Com isso, a história completa de um objeto ou instância de papel pode ser recuperada através dos valores de suas propriedades dinâmicas.

São considerados dois tipos de informações temporais associadas às propriedades dinâmicas: o *tempo de transação* e o *tempo de validade*. O tempo de transação corresponde ao instante em que uma informação é armazenada na base de dados, sendo controlado implicitamente pelo SGBD. O tempo de validade corresponde ao instante de tempo a partir do qual a informação passa a modelar a realidade, sendo fornecido explicitamente pelo usuário quando o valor é definido.

Os tempos de transação e de validade podem ser diferentes: a modificação de uma determinada alíquota de imposto foi realizada no dia 5 de dezembro do ano corrente (tempo de transação), mas tal valor será utilizado apenas a partir do primeiro dia útil do próximo ano (tempo de validade).

Seja uma propriedade dinâmica p_{ij}, definida sobre um domínio d_{ij} qualquer, o valor v desta propriedade dinâmica em um dado instante t é implicitamente definido pela tupla:

$$p_{ij(t)} = (v;\ t_t;\ t_v) \mid (v \in d_{ij})$$

onde t_t e t_v são instantes representando, respectivamente, o tempo de transação e o tempo de validade do valor v.

3.2.2.1 Valores Pré-Definidos

O modelo TF-ORM introduz valores pré-definidos, que podem ser assumidos pelas propriedades durante certos intervalos de tempo:

- *null*: representa a falta de valoração, ou seja, a propriedade não possui um valor neste instante;
- *nonull*: representa a indefinição de valor, ou seja, a propriedade possui um valor indefinido qualquer, não nulo. Este conceito é

utilizado na manipulação das propriedades *object_instance*, *end_object*, *role_instance* e *end_role* (ver seção 3.2.2.2).

No momento da criação da instância, todas as suas propriedades (estáticas e dinâmicas) recebem o valor *null*. As propriedades estáticas, ao receberem um valor não nulo, permanecerão com este durante provavelmente toda a existência da instância. As propriedades dinâmicas, ao receberem um valor, permanecem com este até a definição de outro, ou até o instante em que o tempo de validade indicar nenhum valor válido, quando a propriedade passa novamente a ter o valor *null*. O valor *null* está implicitamente definido em todos os domínios.

3.2.2.2 Propriedades Pré-Definidas

O modelo TF-ORM associa informações temporais às instâncias, a fim de modelar o seu comportamento dinâmico, com base nos instantes (tempo de transação e tempo de validade) de sua criação e destruição e nos instantes das eventuais suspensões na execução e posteriores reativações.

As informações temporais associadas aos objetos são armazenadas através das propriedades dinâmicas pré-definidas *object_instance* e *end_object*, armazenadas no papel básico de cada classe. As informações temporais associadas às instâncias de papéis são armazenadas através das propriedades dinâmicas pré-definidas *role_instance* e *end_role*, armazenadas em cada papel.

Após a criação de um objeto, o valor da propriedade *object_instance* recebe *nonull*, seu tempo de transação indica o instante em que o objeto foi criado e seu tempo de validade, o instante a partir do qual inicia a execução deste objeto. O recebimento de uma mensagem *kill* torna o valor da propriedade *object_instance* em *null* (não alterando os tempos de transação e validade), o valor da propriedade *end_object* recebe *nonull*, seu tempo de transação indica o instante de recebimento da mensagem e seu tempo de validade, o instante em que o objeto deixa de existir.

Quando uma mensagem *suspend_object* é recebida, o valor da propriedade *object_instance* passa a ser *null*, seu tempo de transação marca o instante de recebimento da mensagem e o tempo de validade, o instante a partir do qual a suspensão se inicia. Ao receber uma mensagem *resume_object*, a propriedade *object_instance* passa novamente a ter um valor *nonull*, seu tempo de transação marca o instante de recebimento da mensagem e seu tempo de validade, o instante em que o objeto reinicia sua execução.

O comportamento das propriedades pré-definidas *role_instance* e *end_role* é análogo ao das propriedades *object_instance* e *end_object*.

3.2.3 Estados

Os estados $s_{ij} \in S_i$ são definidos explicitamente no modelo TF-ORM e representam um estado concreto da instância. Um estado de uma determinada instância de papel indica o atual contexto de execução do objeto naquele papel.

Os estados podem ser simples, representando apenas contextos isolados de execução, como *aguardando_liberação* e *liberado*, ou complexos, representando associações entre diversos contextos, definidos sobre relações entre conjuntos de estados.

Seja um dado papel R_i, denominado *Recurso*, representando recursos utilizados em determinado processo produtivo e cujo conjunto de estados está definido como segue:

states = ((ativo, suspenso), (alocado, livre))

Neste caso, o primeiro conjunto de estados (*ativo* e *suspenso*) representa o contexto de execução da instância no que se refere a possibilidade ou não da utilização deste recurso, o segundo conjunto representa o contexto relacionado à alocação deste recurso. Uma instância deste papel terá, em um dado instante, um estado definido sobre uma relação entre os dois conjuntos de estados: (i) (*ativo, alocado*), o recurso está operacional e alocado a um processo produtivo; (ii) (*ativo, livre*), o recurso está operacional e não alocado; (iii) (*suspenso, alocado*), o recurso está em manutenção, mas alocado a um processo; ou (iv) (*suspenso, livre*), o recurso encontra-se em manutenção, não alocado a nenhum processo.

Não é definida nenhuma relação formal entre os estados definidos explicitamente e os estados abstratos decorrentes dos valores das propriedades das instâncias, no entanto, o desenvolvedor deve levar em consideração tais relações, sendo aconselhável a associação de uma anotação livre (comentário) indicando os relacionamentos importantes entre os valores das propriedades e um dado estado [DEA 92].

3.2.3.1 Estados Pré-Definidos

Os estados pré-definidos no papel básico controlam a execução dos objetos e das instâncias de papéis:

- *active*, quando o objeto está em execução;
- *suspended*, quando o objeto ou a instância de papel está impedida de enviar e receber mensagens.

3.2.4 Mensagens

As mensagens $m_{ij} \in M_i$ descrevem as mensagens que podem ser enviadas e recebidas pela instância do papel. Somente mensagens relevantes para a descrição do comportamento da instância, através da ativação de mudanças de estado, são consideradas no modelo. Eventualmente, após a especificação de um esquema de dados TF-ORM, mensagens relacionadas especificamente às aplicações devem ser definidas, e os métodos associados a estas, implementados.

As mensagens são caracterizadas por um nome mn_{ij}, uma lista de parâmetros (mp_{ij_1}, mp_{ij_2}, ..., mp_{ij_n}), definidos por um nome mpn_{ij_1} e um domínio mpd_{ij_1} e relacionados ao envio e recebimento de valores, e por uma direção, ou seja, a caracterização como mensagem de entrada (recebida pela instância) ou mensagem de saída (enviada pela instância). A direção é especificada na declaração (assinatura) da mensagem através dos termos *to* (mensagem de saída) e *from* (mensagem de entrada):

<mensagem> (<parâmetros>) to <papel>
<mensagem> (<parâmetros>) from <papel>

onde *papel* refere-se respectivamente ao papel para o qual a mensagem será enviada ou do qual será recebida. É possível especificar o recebimento ou envio de mensagens para a própria instância (utilizando o termo *itself* como papel fonte ou destino da mensagem) ou para o ambiente (utilizando o termo *external_world*).

Pode-se, também, definir que o envio ou recebimento de uma mensagem esteja condicionado à cooperação de duas ou mais instâncias de papel, através da especificação de uma lista de papéis destino ou fonte da mensagem. Se uma mensagem é definida com papéis cooperantes, tais papéis devem enviá-la sincronizadamente (a instância receberá uma só mensagem).

3.2.4.1 Mensagens Pré-Definidas

As mensagens responsáveis pelo controle da criação e remoção de objetos e instâncias de papéis, bem como pelo controle de mensagens e regras, estão pré-definidas no papel básico das classes:

- *create_object (<classe>, <oId>)*, cria uma instância (objeto) de *classe*, retornando seu *oId*;
- *suspend_object (<oId>)* e *resume_object (<oId>)*, suspende e prossegue a execução do objeto identificado por *oId*, transicionando-o entre os estados *active* e *suspended*;
- *kill (<oId>)*, termina a execução do objeto identificado por *oId*;
- *add_role (<oId>, <papel>, <rId>)*, cria uma instância de *papel* para o objeto identificado por *oId*, retornando seu *rId*;
- *suspend_role (<oId>, <rId>)* e *resume_role (<oId>, <rId>)*, suspende e prossegue a execução da instância de papel *rId* do objeto *oId*, transicionando-a entre os estados *active* e *suspended*;
- *terminate_role (<oId>, <rId>)*, termina a execução da instância *rId* do objeto *oId*;
- *forbid_role (<oId>, <papel>)* e *allow_role (<oId>, <papel>)*, proíbe e permite novamente a instanciação de *papel* no objeto *oId*;
- *forbid_op (<oId>, <direção>, <mensagem>)* e *recall (<oId>, <direção>, <mensagem>)*, impede e libera novamente o recebimento ou envio de *mensagem* no objeto *oId*.
- *forget (<oId>, <regra>)* e *recall (<oId>, <regra>)*, suspende e retoma novamente a verificação de *regra* no objeto *oId*.

3.2.5 Regras

3.2.5.1 Regras de Transição de Estado

As regras de transição de estado governam o comportamento ativo (dinâmico) dos objetos e instâncias de papel, modelando as transições de estado possíveis para estes. Cada regra de transição $ru_{ij} \in Ru_i$ é identificada por um nome run_{ij} e define uma possível transição de um estado inicial s_1 para um estado final s_2, condicionada à recepção de uma mensagem m_1 e podendo enviar uma mensagem m_2:

$$run_{ij}: \ state(oId_1.rId_1,\ s_1),\ msg(\leftarrow oId_2.rId_2,\ m_1) \Rightarrow$$
$$msg(\rightarrow oId_3.rId_3,\ m_2),\ state(oId_1.rId_1,\ s_2);\ (<\text{condição}>)$$

O primeiro predicado do lado esquerdo da regra testa se a instância de papel[5] rId_1 do objeto identificado por oId_1 encontra-se no estado s_1. O segundo predicado indica o recebimento de uma mensagem m_1, proveniente da instância de papel identificada por rId_2 do objeto oId_2. A transição de estado será habilitada se a instância de papel rId_1 do objeto oId_1 estiver no estado s_1 e receber a mensagem m_1. O primeiro predicado do lado direito indica que, caso a transição seja habilitada, será enviada à instância de papel rId_3 do objeto oId_3 a mensagem m_2. O segundo predicado indica que se a transição for habilitada a instância de papel identificada por rId_1 do objeto oId_1 irá transicionar para o estado s_2.

Os predicados das regras são opcionais, gerando quatro tipos distintos de regras:

- *nenhum estado especificado do lado esquerdo da regra*: o lado direito da regra pode ser executado em qualquer estado, desde que seja recebida a mensagem m_1 (permite a definição de *objetos ativos*);
- *nenhuma mensagem especificada no lado esquerdo da regra*: o lado direito da regra é executada sempre que o objeto ou instância de papel transicionar para o estado s_1 (a regra comporta-se como um gatilho — *trigger*);
- *nenhum estado especificado no lado direito da regra*: a regra causa somente o envio de uma mensagem, sem transição de estado;
- *nenhuma mensagem especificada no lado direito da regra*: a regra causa somente uma transição de estado, sem envio de mensagem;

A *condição* especificada após a lado direito da regra pode ser uma condição temporal, expressa através de uma linguagem de lógica temporal de primeira ordem. Esta condição é avaliada logo após a habilitação da regra de transição: se a condição for verdadeira a transição e/ou o envio da mensagem são executados.

Nas classes do tipo recurso e agente, as regras são utilizadas para modelar os papéis que, respectivamente, as informações/documentos e os agentes podem desempenhar, baseado no comportamento que apresentam em determinados contextos. Já nas classes do tipo *processo*, que modelam os procedimentos do escritório, as regras modelam as diferentes tarefas que compõem estes procedimentos, cada uma envolvendo um agente e um ou mais recursos.

3.2.5.1.1 Regras de Transição de Estado Condicionadas a Decisões

As classes do tipo *agente* permitem a representação do processo de tomada de decisões, condicionando a habilitação das regras de transição de estado à interação junto ao agente no momento da ativação. Uma regra de transição de estado definida nas classes agente pode, portanto, possuir a forma alternativa:

$$run_{ij}: \ state(oId_1.rId_1, s_1),\ decision(\langle decisão\rangle\ (parâmetros)) \Rightarrow$$
$$msg(\rightarrow oId_3.rId_3, m_2),\ state(oId_1.rId_1, s_2);\ (\langle condição\rangle)$$

[5] Caso não sejam especificados os identificadores de instância de papel (rId_n), os predicados serão avaliados para o papel básico do objeto oId_n.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

A transição de estado será habilitada se a decisão for tomada e a instância estiver no estado s_1.

3.2.5.2 Regras de Integridade

As regras de integridade governam o comportamento passivo (estático) dos objetos e instâncias de papel. Uma regra de integridade $ru_{ij} \in Ru_i$ especifica uma restrição a ser satisfeita por todas as instâncias do papel, sendo identificada por um nome run_{ij} e definida pela expressão:

$$run_{ij}: constraint\ (pré\text{-}condição \Rightarrow pós\text{-}condição)$$

Sempre que um objeto ou instância de papel satisfaz a *pré-condição* deve satisfazer a *pós-condição*. Caso a *pós-condição* não seja satisfeita, as modificações realizadas pela última operação[6] devem ser desfeitas, o estado anterior deve ser restabelecido e as propriedades modificadas devem ter seus valores anteriores restaurados, retornando o banco de dados ao estado consistente imediatamente anterior. As restrições de integridade devem ser avaliadas, portanto, após a criação de um novo estado do banco de dados, ou seja, logo após a modificação do estado ou de alguma propriedade de uma instância.

Tanto a pré-condição quanto a pós-condição podem ser expressas em linguagem de lógica temporal, permitindo assim a definição de regras de integridade sobre domínios temporais e propriedades dinâmicas.

3.2.5.3 Regras Definidas no Papel Básico

As regras de transição de estado definidas no papel básico possuem, geralmente, as seguintes atribuições: (i) governar a criação e remoção de instâncias de papéis; (ii) indicar o estado inicial e as condições a serem respeitadas pelas instâncias recém criadas; (iii) identificar quais os papéis, além do básico, devem ser instanciados após a criação de um novo objeto.

As regras de integridade definidas no papel básico descrevem condições genéricas sobre as propriedades, as quais devem ser respeitadas por todas as instâncias de todos os papéis definidos para a classe.

3.3 Domínios de Propriedades

Quatro diferentes conceitos são utilizados para as definições tempo-dependentes: (i) *pontos no tempo* (*timestamps*) associados aos objetos e às instâncias de papéis, bem como às propriedades que possuem variação dinâmica no tempo; (ii) um valor *null* especial, para representar os períodos em que as propriedades não possuem um valor definido; (iii) um conjunto de tipos de dados, operações e funções temporais; e (iv) condições temporais escritas em lógica modal de primeira ordem associadas a regras de transição de estado e restrição de integridade.

---

[6] O modelo não determina a granularidade desta operação. É possível interpretar uma operação como: (i) uma alteração individual de uma propriedade (através de um método); (ii) um conjunto de alterações efetuadas por um método; ou (iii) todas as alterações realizadas por um conjunto de métodos, reunidos em uma transação de banco de dados.

Os domínios das propriedades podem ser *simples* ou *complexos*. Um domínio simples é definido em termos dos tipos de dados pré-definidos ou classes. Um domínio complexo é definido em termos de agregações de domínios simples. São previstas duas estruturas básicas para construção de domínios complexos: *conjuntos* e *listas*.

O modelo TF-ORM apresenta domínios de dados convencionais (como por exemplo, inteiros), e domínios de dados temporais. O elemento temporal primitivo adotado no modelo TF-ORM foi o *ponto no tempo* e o menor intervalo entre quaisquer pontos no tempo (*chronon*) é definido em minutos, por ser esta a menor granularidade temporal utilizada no domínio dos sistemas de informação de escritórios [EDE 94]. Com isso, a definição completa de um ponto no tempo pode ser feita pela concatenação das informações de uma data (ano, mês, dia) e de um horário (horas, minutos).

Os domínios simples pré-definidos no modelo são: *integer*, *real*, *boolean*, *string*, *text*, *place*, *title* e *image*. São identificados três domínios de dados temporais: pontos no tempo (*instant*, *date*, *time*, *year*, *month*, *day*, *hour*, *minute*, *week*, *semester*, *century* e *weekday*), intervalos (*interval*) e duração (*span*). São definidos, ainda, tipos de dados para informações temporais incompletas (*limit*).

3.3.1 Pontos no Tempo

instant ::= <ano> "/" <mês> "/" <dia> "," <hora> ":" <minuto>

Demais tipos de pontos no tempo podem ser obtidos através de restrições ao tipo *instant*:

date ::= <ano> "/" <mês> "/" <dia>
time ::= <hora> ":" <minuto>
year ::= <ano>
month :: <mês>
day ::= <dia>
hour ::= <hora>
minute ::= <minuto>
week ::= <semana>
semester ::= <semestre>
century ::= <século>
weekday ::= <dia da semana>

Cada um destes tipos apresenta restrições implícitas de valores, por exemplo: $1 \leq month \leq 12$.

É definido um ponto no tempo especial *now*, que corresponde ao instante atual. Este valor pode ser associado a propriedades definidas em qualquer domínio temporal, assumindo o tipo para o qual está sendo utilizado, considerado em comparações, regras de integridade e nas condições presentes nas regras.

3.3.2 Intervalos

Intervalos de tempo são utilizados para definir todos os instantes entre dois pontos no tempo, devendo o instante de tempo definido como primeiro limite ser anterior ao segundo limite. Os tipos de dados temporais que podem ser utilizados como limites de intervalos temporais são: *instant*, *date*, *time*, *year*, *month*, *day*, *hour* e *minute*.

Podem ser definidos seis tipos de intervalos quanto a pertinência ou não dos limites a este: (i) *intervalo fechado*, quando ambos os limites participam do intervalo; (ii) *intervalo semiaberto abaixo*, quando o limite inferior não pertence ao intervalo; (iii) *intervalo semiaberto acima*, quando o limite superior não participa do intervalo; (iv) *intervalo aberto*, quando ambos os limites não participam do intervalo; (v) *intervalo flutuante abaixo*, quando o limite inferior é o instante atual; e (vi) *intervalo flutuante acima*, quando o limite superior é o instante atual.

3.3.3 Durações

Uma duração representa o número de unidades no tempo que durou uma determinada atividade. Esta informação é representada por um número inteiro seguido de alguma unidade de tempo (por exemplo, horas, dias, semanas, etc.), especificando sua granularidade.

3.3.4 Tipos de Dados para Informações Temporais Incompletas

Muitas vezes é necessária a manipulação de informações temporais incompletas, como as noções de *antes* e *depois*, associadas a um evento e a um instante no tempo. O modelo TF-ORM apresenta dois tipos de dados a fim de modelar estas situações:

limite ::= after (<instante>)
limite ::= before (<instante>)

Este tipo de dado pode ser encarado como um intervalo com limite superior (*after*) ou limite inferior (*before*) infinito.

3.4 Linguagem de Lógica Temporal

As condições presentes nas regras de transição de estado e regras de restrição de integridade são escritas em uma linguagem de lógica temporal de primeira ordem e avaliam três tipos de informações: (i) valores de propriedades estáticas e dinâmicas; (ii) valores de rótulos temporais associados às propriedades dinâmicas, objetos e instâncias de papéis; e (iii) estados de objetos e instâncias de papéis. Os símbolos utilizados nas fórmulas das condições temporais são: (i) *proposições atômicas*, referindo-se a valores de propriedades estáticas e dinâmicas; (ii) *operadores relacionais*; (iii) conectivos lógicos *and*, *or* e *not*; (iv) quantificadores existencial (*exist*) e universal (*forall*); (v) valores transmitidos como *parâmetros* de mensagens recebidas e decisões; e (vi) um conjunto de *operadores lógicos temporais*. As proposições atômicas podem envolver *predicados* e *funções*.

3.4.1 Operadores

3.4.1.1 Operadores sobre Tipos de Dados Temporais

As seguintes operações envolvendo operandos temporais são definidas:

- *operações aritméticas soma e subtração*: podem ser aplicadas quando (i) um operando é um ponto no tempo (instante, data ou horário) e o segundo uma duração, resultando um valor do primeiro tipo; (ii) os dois operandos são do tipo duração, representados pela mesma unidade; (iii) quando os dois operandos são do tipo *week*, *semester* e *century*; e (iv) quando os dois operandos são intervalos de mesma granularidade, resultando um intervalo ou um resultado *null* (intervalos disjuntos).
- *operações aritméticas divisão, multiplicação e potenciação*: podem ser aplicadas quando um operando for um valor numérico e o outro (i) uma duração, resultando um valor com sua granularidade; (ii) um valor com granularidade *minute*, *hour*, *day*, *month*, *year*, *week*, *semester* ou *century*, sendo o resultado dado pela granularidade do segundo operando.
- *operadores relacionais* “<”, “>”, “=”, “≤”, “≥”, “≠” , aplicáveis na comparação de dois pontos no tempo ou duas durações;
- *operações sobre conjuntos* a serem aplicadas sobre intervalos, como (i) *union*; e (ii) *intersection*, resultando intervalos ou valores *null* e (iii) *contains*, resultando um valor lógico.
- operador unário *sometime past*: verdadeiro se o operando valeu em algum instante do passado;
- operador unário *immediately past*: verdadeiro se o operando valeu no instante imediatamente anterior;
- operador unário *always past*: verdadeiro se o operando valeu em todos os instantes passados;
- operador unário *sometime future*: o operando será válido em algum instante futuro;
- operador unário *immediately future*: o operando será válido no instante imediatamente posterior;
- operador unário *always future*: o operando será válido em todos os instantes futuros;
- operador binário *since*: o primeiro operando valeu em todos os instantes desde que o segundo operando passou a valer;
- operador binário *until*: o primeiro operando será válido em todos os instantes, até que o segundo operando passe a valer;
- operador binário *before*: o primeiro operando valeu em algum instante anterior ao que o segundo operando valeu;
- operador binário *after*: o primeiro operando será válido em algum instante posterior ao que o segundo operando valeu;

3.4.1.2 Operadores Genéricos

- operadores aritméticos *soma*, *subtração*, *multiplicação*, *divisão* e *potenciação*;
- *operadores relacionais* "<", ">", "=", "≤", "≥", "≠";
- operadores sobre conjuntos *union* e *intersection*;
- operadores lógicos *and*, *or* e *not*;
- quantificadores existencial (*exist*) e universal (*forall*);

3.4.2 Predicados

3.4.2.1 Predicados sobre Tipos de Dados Temporais

Os predicados sobre tipos de dados temporais comparam a posição temporal relativa de dois pontos no tempo ou dois intervalos:

- *before*: retorna verdadeiro quando o primeiro instante é anterior ao segundo;

  before (<instante>:<instante>)

- *equal*: retorna verdadeiro quando os instantes forem iguais;

  equal (<instante>:<instante>)

- *belongs*: retorna verdadeiro quando *instante* pertencer a *intervalo*;

  belongs (<instante>:<intervalo>)

- *contains*: retorna verdadeiro quando o primeiro intervalo está contido no segundo;

  contains (<intervalo>:<intervalo>)

3.4.2.2 Predicados sobre Valores de Propriedades[7]

- *is_valid*: verdadeiro se o valor válido atual de *propriedade* é *valor* para o objeto identificado por *oId*;

  is_valid (<oId>, <propriedade>, <valor>) |
  is_valid (<oId>.<rId>, <propriedade>, <valor>)

- *is_valid_at*: verdadeiro se o valor válido de *propriedade* em *instante* era *valor*.

  is_valid_at (<oId>, <propriedade>, <valor>, <instante>) |
  is_valid_at (<oId>.<rId>,<propriedade>,<valor>,<instante>)

3.4.2.3 Predicados sobre Objetos e Instâncias de Papel

- *has_class_instance*: verdadeiro se existe pelo menos uma instância de *classe*, retornando em *oId* o identificador da primeira instância encontrada;

---

[7] Construções alternativas são separadas pelo símbolo "|".

has_class_instance (<classe>, <oId>)

- *has_role_instance*: verdadeiro se existe pelo menos uma instância de *papel* para o objeto *oId*, retornando em *rId* o identificador da primeira instância encontrada;

  has_role_instance (<oId>, <papel>, <rId>)

- *active_class* e *active_role*: verdadeiros se a instância está ativa;

  active_class (<oId>)
  active_role (<oId>, <rId>)

- *active_class_at* e *active_role_at*: verdadeiros se a instância estava ativa em *instante*;

  active_class_at (<oId>, <instante>)
  active_role_at (<oId>, <rId>, <instante>)

- *out_role*: verdadeiro se o objeto identificado por *oId* não possui nenhuma instância de *papel*;

  out_role (<oId>, <papel>)

- *role*: verdadeiro se a instância de papel *rId* pertence ao objeto *oId*;

  role (<oId>, <rId>)

3.4.3 Funções

3.4.3.1 Funções sobre Tipos de Dados Temporais

O modelo TF-ORM apresenta uma série de *funções* pré-definidas para a manipulação de informações com granularidades temporais diferentes. As funções estão divididas em:

- *funções que modificam a granularidade* de uma informação:

to_minutes (<instante | horário>)	converte para minutos;
to_months (<instante | data>)	converte para meses;
to_days (<instante | data>)	converte para dias;

- *funções que retornam uma informação temporal*, calculada a partir de um valor temporal fornecido:

year (<instante | data>)	extrai os anos (truncando);
month (<instante | data>)	extrai os meses (truncando);
day (<instante | data>)	extrai o dia (truncando);
hour (<instante | horário>)	extrai as horas (truncando);
minute (<instante | horário>)	extrai os minutos (truncando);
weekday (<instante | data>)	dia da semana;
lower_bound (<intervalo>)	limite inferior de um intervalo;
upper_bound (<intervalo>)	limite superior de um intervalo;
duration (<intervalo>)	duração de um intervalo;
interval (<instante>, <instante>)	intervalo entre dois instantes;

3.4.3.2 Funções sobre Valores de Propriedades

- *value*: retorna o valor atual de *propriedade*;

  value (<oId>, <rId>, <propriedade>)

- *value_at*: retorna o valor válido de *propriedade* em *instante*;

  value_at (<oid>, <rId>, <propriedade>, <instante>)

- *valid_time*: retorna o tempo de validade para o valor de *propriedade*;

  valid_time (<oId>, <rId>, <propriedade>)

- *transaction_time*: retorna o tempo de transação para o valor de *propriedade*;

  transaction_time (<oId>, <rId>, <propriedade>)

3.4.3.3 Funções sobre Objetos e Instâncias de Papel

- *class_creation_time* e *role_creation_time*: retornam, respectivamente, os instantes de criação de um objeto ou de uma instância de papel;

  class_creation_time (<oId>)
  role_creation_time (<oId>, <rId>)

- *class_end_time* e *role_end_time*: retornam, respectivamente, os instantes de encerramento da execução de um objeto ou de uma instância de papel;

  class_end_time (<oId>)
  role_end_time (<oId>, <rId>)

- *state*: retorna o estado atual de um objeto ou instância de papel;

  state (<oId>) |
  state (<oId>, <rId>)

- *state_at*: retorna o estado de um objeto ou instância de papel em *instante*;

  state_at (<oId>, <instante>) |
  state_at (<oId>, <rId>, <instante>)

3.5 Linguagem de Consulta

O modelo TF-ORM possui uma linguagem de consulta temporal [EDE 94] que permite: (i) recuperar valores de propriedades com domínios simples ou temporais; (ii) referir-se a determinado instante ou intervalo temporal; e (iii) recuperar valores com base em restrições temporais. A análise mais detalhada da linguagem de consulta foge ao escopo deste trabalho.

4 O SISTEMA DE GERÊNCIA DE BANCOS DE DADOS O_2

O_2 [LEC 90, O2T 91, O2T 91a, O2T 91b] é um sistema de gerência de bancos de dados *orientado a objetos*. O grande interesse que têm despertado os sistemas de gerência de bancos de dados orientados a objetos deve-se, principalmente, à necessidade crescente da representação de dados para aplicações não convencionais, como Projeto e Produção Auxiliada por Computador (CAD/CAM, CASE), Inteligência Artificial e Sistemas de Informação de Escritórios (SIE). Tais aplicações tiram proveito das vantagens deste novo conceito, a saber, da possibilidade de modelar as entidades conceituais com o conceito único de objetos e utilizar as noções de hierarquia de classes e herança de propriedades [FRE 90].

Os sistemas de bancos de dados hierárquicos e em rede prevaleceram durante toda a década de 70, sendo considerados a *primeira geração* de SGBDs, pois foram os primeiros a oferecerem um número substancial de funções de gerência de bancos de dados unificadas em um sistema com linguagens de definição e manipulação de dados para coleções de registros. Nesta categoria enquadram-se os sistemas CODASYL e IMS. Na década de 80, os sistemas da primeira geração foram amplamente suplantados pelos SGBDs relacionais, considerados a *segunda geração*. Sistemas típicos desta geração são DB2, CA-INGRES, SYBASE SQL-SERVER, Microsoft SQL-Server, PROGRESS, NON-STOP SQL, ORACLE e Rdb/VMS [COM 90].

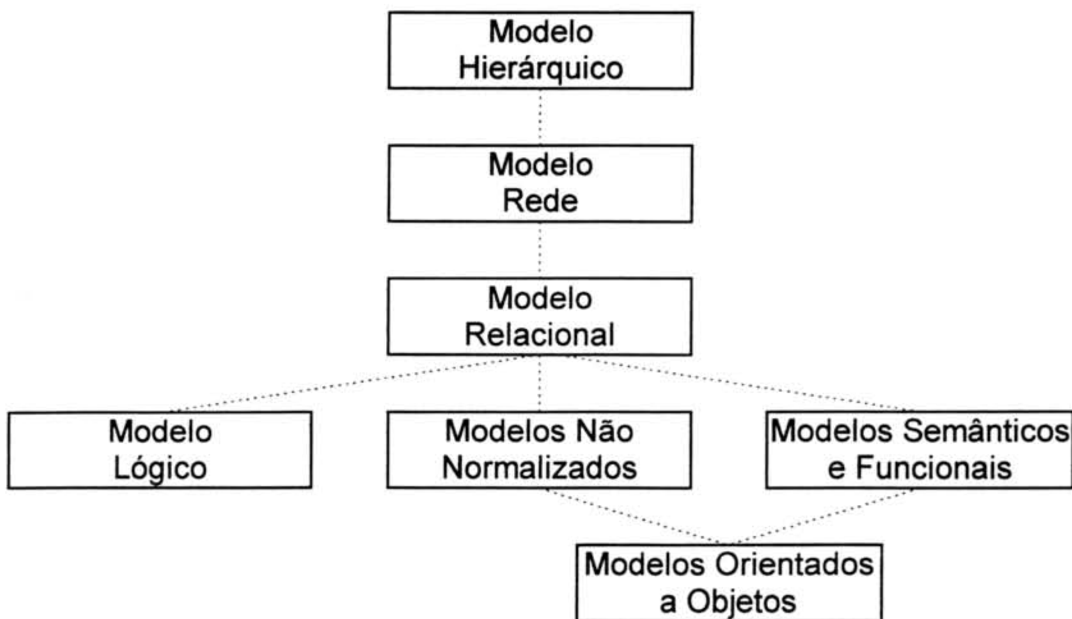


FIGURA 4.1 - Evolução dos Modelos de Dados (Adaptado de [TAK 90])

Transações longas, podendo durar horas ou dias e envolvendo diversos usuários, manipulação de objetos estruturalmente complexos e não facilmente representáveis através de tabelas, modelagem de tipos complexos com poucas instâncias ao invés de tipos mais simples com maior número de instâncias, identidade forte de objetos que atravessam sucessivas transformações e a necessidade de interfaces gráficas para a manipulação direta de objetos, são os principais requisitos

exigidos para os novos SGBDs, os quais procuram ser atendidos através da orientação a objetos [TAK 90]. Os novos sistemas orientados a objetos podem ser considerados componentes da *terceira geração* de SGBDs.

Uma série de requisitos amplamente discutidos em diversos trabalhos [ATK 89, COM 90, FRE 90] é destacada como fundamental para os novos sistemas de gerência de terceira geração. O objetivo deste capítulo é situar o SGBD O_2 frente a estes requisitos e apresentar suas características relevantes.

4.1 Requisitos de SGBDs Orientados a Objetos

Um SGBD orientado a objetos deve satisfazer a dois conjuntos de critérios: ao que caracteriza um sistema de gerência de bancos de dados e ao que caracteriza um sistema orientado a objetos. É possível dividir estes critérios em três grupos de proposições: (i) proposições referentes à *gerência de objetos e regras*; (ii) proposições referentes à *ampliação das funções de gerência de dados*; e (iii) proposições referentes à necessidade de *sistemas abertos de gerência de dados* [COM 90].

4.1.1 Gerência de Objetos e Regras

Um SGBD orientado a objetos deve ter capacidade para manipular grandes quantidades de informações, sendo estas informações tão complexas (ou não convencionais) quanto exigir a aplicação. Para isto é necessário o suporte à definição de objetos complexos e dados não convencionais, bem como à definição de regras, incluindo regras de transição de estado e regras de restrição de integridade.

4.1.2 Ampliação das Funções de Gerência de Dados

Os SGBDs orientados a objetos devem suportar algumas das características dos sistemas da segunda geração, principalmente no que se refere às linguagens de consulta, à especificação de conjuntos de elementos e à independência de dados.

Estes sistemas devem possibilitar que toda a manipulação da informação possa ser executada através de uma linguagem de alto nível, presente tanto em consultas *ad hoc* como na programação das aplicações. Devem existir dois métodos de expressar os membros de uma coleção de dados: através da enumeração de seus elementos ou através da especificação de conjuntos, via linguagem de consulta de alto nível.

4.1.3 Sistemas Abertos de Gerência de Dados

Os SGBDs de terceira geração devem suportar uma linguagem de quarta geração, ferramentas de auxílio à tomada de decisões, acesso a partir de diversas linguagens de programação de alto nível, acesso a aplicativos de função genérica (editores de textos, planilhas, etc.), interfaces para pacotes gráficos, a execução das aplicações sobre diferentes plataformas e distribuição de dados. Estas características possuem duas implicações: (i) os sistemas de gerência orientados a objetos devem suportar a maioria das ferramentas descritas acima e (ii) estes sistemas

devem ser abertos, ou seja, deve ser facilitada a adição de novas ferramentas, em uma variedade de plataformas.

4.2 Os Conceitos de Tipo e Classe

Existem duas grandes categorias de modelos orientados a objetos: aqueles que suportam o conceito de *tipos* e os que suportam o conceito de *classes* [COM 90].

Um tipo sumariza os aspectos comuns de um conjunto de objetos com as mesmas características, correspondendo à noção de tipo abstrato de dado e possuindo duas partes: interface e implementação. A interface corresponde às propriedades e às mensagens a que responde o objeto, e a implementação, ao código associado às mensagens, através do qual é possível manipular o valor das propriedades. A manipulação dos objetos é realizada através de comandos externos a sua estrutura. Modelos característicos desta categoria são os utilizados por C++, Simula, Trellis/Owl, Vbase e o próprio O_2.

No sistema O_2, os objetos, ao serem criados, através do comando *new*, são associados a um determinado tipo, que define sua estrutura durante todo o seu ciclo de vida, salvo sua especialização em uma subclasse definida sobre um subtipo da classe original (ver seção 4.3.3).

O conceito de classe diferencia-se do conceito de tipo, sua especificação é a mesma, mas sua noção é associada ao tempo de execução. Uma classe divide-se em uma *fábrica* e em um *depósito* de objetos. A fábrica pode ser utilizada para criar novos objetos, e o depósito é o conjunto de instâncias da classe, sobre o qual podem ser executadas operações. Uma classe pode ser modificada em tempo de execução permitindo uma grande flexibilidade. Modelos característicos desta categoria são os utilizados por Smalltalk, Gemstone, Vision, Orion, Flavors, G-base e Lore.

4.3 Especificação de Esquemas de Dados O_2

O SGBD O_2 baseia-se em dois conceitos fundamentais: (i) esquemas, onde são definidas classes, tipos, valores, objetos persistentes e aplicações; e (ii) bases de dados, associadas a esquemas com a finalidade de armazenar valores e objetos persistentes, possibilitando a execução de consultas. A manipulação de esquemas é realizada através de uma DDL (*Data Definition Language*) e a manipulação das bases de dados, através de uma DML (*Data Manipulation Language*), acessível através da execução de código *ad hoc* ou de aplicações.

Esta seção apresenta um resumo dos principais conceitos associados ao modelo de dados O_2, em especial aqueles manipulados através da DDL e alguns, associados à criação de objetos, manipulados através da DML.

4.3.1 Definição de Tipos

Um SGBD orientado a objetos deve fornecer suporte à especificação de objetos (ou tipos) complexos, ou seja, àqueles construídos a partir de objetos mais simples, através da aplicação de construtores. Objetos simples são definidos sobre

tipos simples, como inteiros, reais, strings, etc., ou referências a classes. Objetos complexos são definidos sobre tipos complexos, incluindo *tuplas* (registros), *vetores*, *listas* (seqüências), *conjuntos*, tipo *função*, construtor *união* e a composição recursiva destes elementos. Cada tipo complexo é construído através de um construtor específico, sendo o menor conjunto de construtores desejável formado por conjuntos, listas e tuplas.

O sistema O_2 suporta o conjunto mínimo de construtores de tipos complexos. A existência de conjuntos é importante pois estes representam naturalmente as propriedades de uma entidade. Já as listas possibilitam a inserção de uma relação de ordem entre seus componentes. Tuplas, por fim, têm sido consideradas, desde a segunda geração, como a maneira mais natural de construção de objetos complexos [COM 90]. O conjunto de tipos simples, os quais podem ser combinados na definição de tipos complexos, inclui: (i) inteiros (*integer*); (ii) caracteres (*char*); (iii) valores lógicos (*boolean*); (iv) valores em ponto flutuante (*real*); (v) seqüências de caracteres (*string*); e (vi) seqüências de bytes (*bit*).

- *tuplas*: são formadas pela reunião de diversos valores de diferentes tipos (simples ou complexos);

  tuple (<nome de atributo>:<tipo>,
  <nome de atributo>:<tipo>, ...)

- *conjuntos*: formados pela reunião de valores de um mesmo tipo;

  set (<tipo>)

- *listas*: formados pela reunião de valores de um mesmo tipo, mas, ao contrário dos conjuntos, permite a replicação de valores e o estabelecimento de uma relação de ordem entre os elementos.

  list (<tipo>)

O suporte a objetos complexos engloba também a existência de um conjunto de operações de manipulação e consulta aplicáveis a cada um dos tipos complexos. O sistema O_2 fornece suporte a diversas operações sobre estes tipos, como por exemplo inserção, remoção, contagem e busca de elementos em conjuntos e listas, cópias profundas (com replicação dos objetos aninhados) e cópias superficiais (sem replicação) de objetos complexos, etc. (ver seção 4.3.3.1).

4.3.2 Definição de Classes

```
class   <nome da classe>
        inherit <nome de classe>, <nome de classe>, ...

        type tuple (
                        <nome do atributo>: <tipo>,
                        <nome do atributo>: ...
        )

        method <nome do método> (<parâmetros>): <tipo retornado>,
        method <nome do método> (<parâmetros>): ...
end
```

FIGURA 4.2 - Esquema de uma Classe O_2

Uma classe O_2 (Figura 4.2) é completamente definida por (i) um *nome*, que a identifica univocamente na hierarquia de classes; (ii) uma *lista de superclasses*, das quais "herda" as definições de atributos e métodos; (iii) uma especificação de *tipo*, descrevendo os atributos que caracterizam os objetos da classe; e (iv) um conjunto de métodos, modelando o comportamento dos objetos desta classe.

Em geral, as classes são definidas em termos de um tipo *tupla*, correspondendo aproximadamente ao conceito de registros em outros sistemas, sendo este formado por uma combinação de atributos de diferentes tipos.

Os atributos podem ser definidos sobre tipos simples ou complexos. Os métodos são caracterizados por sua assinatura (*signature*) descrevendo os parâmetros e o valor retornado pelo método e por um corpo (*body*), descrevendo sua implementação na linguagem de programação O_2C. A assinatura dos métodos é associada à classe, sendo definida juntamente a esta (Figura 4.2). O corpo de um método é definido separadamente, através do comando *method body* (Figura 4.3).

```
method body   <nome do método> ( <parâmetro>:<tipo>, <parâmetro>:...)
              in class <nome de classe>
{
      <código O2C>
}
```

FIGURA 4.3 - Definição de um Método O_2

Os atributos e métodos de um objeto podem ser acessados através do operador "->".

<oId> -> <atributo>	retorna o valor do atributo;
<oId> -> <método>	ativa a execução do método.

Tanto os atributos como os métodos podem ser definidos como públicos (*public*), somente para leitura (*read*) ou privados (*private*). Itens públicos podem ser lidos (ativados, no caso de métodos) ou escritos por qualquer objeto, programa ou função da aplicação. Itens somente para leitura podem ser lidos por qualquer objeto, programa ou função da aplicação, mas seu valor pode ser alterado somente através dos métodos do objeto. Itens privados podem ser acessados (lidos ou alterados) somente pelos métodos do objeto.

4.3.3 Manipulação da Hierarquia de Classes

Em SGBDs orientados a objetos, a noção de hierarquia de classes e do mecanismo de herança permitem uma maior facilidade de modelagem, tornando mais precisa a abstração do sistema real e facilitando assim a aplicação do mecanismo de reutilização de classes. A possibilidade do estabelecimento de uma hierarquia de classes, incluindo herança múltipla, é considerada essencial [COM 90].

No sistema O_2, a herança é definida como um método de construir novas classes a partir de refinamentos de classes existentes [*O2T* 91], sendo assim, os objetos de uma *subclasse* são igualmente objetos da *superclasse*. O tipo que define a estrutura de dados da subclasse deve ser um subtipo daquele que define a superclasse, ou seja, deve respeitar as seguintes regras:

- Sejam T1 e T2 tipos tupla. T2 será um subtipo de T1 se cada atributo de T2 corresponder a um atributo do mesmo tipo ou de um subtipo em T1, ou se este não possuir correspondente em T1;
- Sejam T1 e T2 tipos conjunto ou lista. T2 será um subtipo de T1 se o tipo que define os componentes de T2 for subtipo do que define os componentes de T1.

A manipulação da hierarquia de classes é realizada através da inclusão e exclusão de relacionamentos de herança, durante a especificação das classes (Figura 4.2) ou posteriormente, após sua criação. Um relacionamento de herança é definido através do comando *inherit* e pode ser removido através do comando *delete inherit*.

Após sua criação e em tempo de execução, um objeto pode ser especializado em uma subclasse, através da função pré-definida *specialize*.

specialize (<old>, <nome da nova classe>)

O mecanismo de herança múltipla é suportado e os conflitos de nomes de atributos e métodos são resolvidos através da operação automática de mudança de nomes na subclasse (provida pelo sistema O_2) ou através da definição de novos nomes através do comando *rename*. No entanto, atributos e métodos presentes na superclasse não podem ser excluídos ou inibidos na subclasse.

rename attribute <nome atual> as <nome novo>
in class <nome da classe>
rename method <nome atual> as <nome novo>
in class <nome da classe>

A possibilidade de redefinição (*overriding*) de métodos e atributos em SGBDs orientados a objetos é fundamental, pois facilita a programação de aplicações, já que reduz o vocabulário de mensagens que o programador deve conhecer. O sistema O_2 permite tanto a redefinição de métodos como de atributos, no entanto, as seguintes regras devem ser respeitadas:

- a nova definição de um método deve possuir o mesmo número de parâmetros que a anterior, e estes parâmetros devem ser do mesmo tipo ou de subtipos dos parâmetros do método original. Se o método original retornava um valor, a nova definição deve igualmente retornar um valor, do mesmo tipo ou de um subtipo do original;
- A nova definição de um atributo deve ser um subtipo do atributo original.

4.3.3.1 Superclasse *object*

O sistema O_2 possui uma classe abstrata pré-definida denominada *object*, raiz da hierarquia de classes e que fornece métodos genéricos de manipulação da estrutura de dados, inicialização, apresentação, etc.. Dentre estes métodos podem ser destacados:

- *copy*: retorna um novo objeto cujos valores são iguais aos valores do objeto receptor do método. Se estes valores fazem referência a outros objetos, então estas referências são mantidas.

  <old> = <old> -> copy

- *deep_copy*: retorna um novo objeto cujos valores são iguais aos valores do objeto receptor do método. Se estes valores fazem referência a outros objetos, então novos objetos são criados com valores iguais aos referenciados, recursivamente.
- *equal*: retorna um valor lógico verdadeiro se o objeto receptor e o definido como parâmetro possuem os mesmos valores. Se estes valores fazem referência a outros objetos, então as referências devem ser as mesmas.
- *deep_equal*: retorna um valor lógico verdadeiro se o objeto receptor e o definido como parâmetro possuem os mesmos valores. Se estes valores fazem referência a outros objetos, então os objetos referenciados devem possuir os mesmos valores, recursivamente.
- *type_of*: retorna um identificador implícito do tipo da classe do objeto.
- *class_of*: retorna um identificador implícito da classe do objeto.

4.3.4 Manipulação e Persistência de Instâncias

Após a definição de uma classe, podem ser criadas instâncias (objetos) desta classe. As instâncias são criadas através da definição de uma variável, cujo tipo associado corresponde à classe, e da posterior execução do comando *new*, que ativa o método construtor da classe, denominado *init*, retornando, na variável, o identificador do objeto (*old*):

o2 <nome da classe> <nome da variável>
<nome da variável> = new <nome da classe>

O conjunto de instâncias de uma classe é denominado, no sistema O_2, de extensão (*extension*) da classe. Porém, não é automaticamente garantida a persistência da extensão de uma classe. No entanto, esta exigência é evidente do ponto de vista de um SGBD.

Desta forma, o sistema O_2 requer a definição explícita dos repositórios onde serão armazenados objetos individuais ou os dados genéricos persistentes (denominados *named objects* e *named values*, respectivamente). Cada *named object* ou *named value* garante que seus componentes serão persistentes. É aconselhável que, para cada classe, seja definido um *named value*, como um conjunto, no qual serão inseridos todos os objetos da classe aos quais se deseja garantir a persistência:

name <nome do repositório>: set (<nome da classe>)

Podem ser definidos *named values* constantes, a fim de garantir persistência a informações relevantes:

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

constant name <nome da variável>: <tipo>

4.3.5 Identidade de Objetos

Um sistema orientado a objetos deve suportar o conceito de identidade de objeto, ou seja, um objeto deve possuir um identificador unívoco através do qual será referenciado. Com isso, os objetos, ao contrário do que acontece nos SGBDs tradicionais, não necessitam ter os valores de seus atributos inicializados pelas aplicações durante sua criação para estarem aptos a ser referenciados em atributos, operações e consultas.

O sistema O_2, quando da criação de um objeto, associa-o a um identificador implícito, inacessível aos usuários e programadores, retornado na variável definida no instante da execução do comando *new*. Além disso, as propriedades do objeto são inicializadas com valores *default*, ou seja, existe um valor *default* para cada objeto simples, os conjuntos e listas são inicializados vazios, as tuplas têm os valores de seus componentes inicializados como *default* e objetos aninhados são inicializados com um valor especial *nil*, que indica a não existência do objeto.

Devido à noção de identidade de objetos, dois objetos com valores de atributos idênticos podem ser diferenciados, pois possuem identificadores diferentes. O sistema O_2 suporta dois tipos de operações de comparação entre objetos: (i) uma para determinar se ambos são idênticos (são o mesmo objeto, logo possuem o mesmo identificador); (ii) outra para determinar se ambos são iguais (são objetos diferentes, com identificadores distintos, mas possuem todos os atributos com os mesmos valores) (ver seção 4.3.3.1).

A existência de identificadores unívocos permite o compartilhamento de objetos sem a necessidade de replicação de informação. Isto facilita, também, a atualização dos objetos compartilhados, já que é necessário alterar apenas o valor do objeto compartilhado para que esta modificação venha a refletir-se em todas as referências a este.

4.3.6 Regras de Transição de Estado e Regras de Restrição de Integridade

O mecanismo de estabelecimento de regras de transição de estado e regras de restrição de integridade é considerado fundamental para os SGBDs orientados a objetos [COM 90].

Embora não contasse, originalmente, com um mecanismo de suporte a regras, pesquisas posteriores [BAU 91] desenvolveram um completo mecanismo de regras *E-C-A* (Evento-Condição-Ação), permitindo a modelagem tanto das regras de transição de estados quanto das regras de restrição de integridade. No entanto, a implementação deste mecanismo não está incluída na versão disponível do sistema O_2.

4.4 Definição de Aplicações O_2

Após a definição de um esquema O_2, através dos comandos da DDL, é necessário especificar as aplicações que manipularão os dados definidos neste esquema. A programação no ambiente O_2 baseia-se na definição de (i) *métodos*, associados às classes definidas na base de dados; (ii) *aplicações*, as quais são conjuntos

de *programas* que possuem o intuito de realizar um grupo relacionado de operações sobre a base de dados; e (iii) funções, que podem ser invocadas a partir de qualquer um dos anteriores.

Qualquer um dos itens acima descritos são desenvolvidos utilizando-se uma DML baseada em uma extensão da linguagem C padrão, com suporte a manipulação de objetos e persistência, denominada O_2C. Qualquer código escrito em O_2C pode possuir embutida uma consulta à base de dados, escrita em O_2Query, um dialeto SQL (*Structured Query Language*) [DAT 91, KOR 93].

4.4.1 Métodos

Métodos no ambiente O_2 são módulos de programa que descrevem um aspecto do comportamento de uma classe de objetos a qual estão associados [*O2T* 91]. A definição dos métodos divide-se claramente em duas partes: a assinatura (*signature*), descrição dos parâmetros de entrada e saída do método, e o corpo (*body)*, no qual é escrito o código O_2C que implementa a função desejada para o método (ver seção 4.3.2).

4.4.2 Aplicações

Uma aplicação é composta por um conjunto de *programas* e, opcionalmente, um conjunto de *variáveis*. Os programas componentes de uma aplicação podem ser públicos ou privados. Programas públicos podem ser acionados explicitamente pelos usuários através de um menu associado a cada aplicação.

4.4.3 Programas

Um programa de aplicação, no ambiente O_2, pode ser encarado de duas maneiras distintas: simplesmente como um *programa* ou como uma *transação*. Aqui o conceito de transação corresponde exatamente ao conceito genérico de transação utilizado em bancos de dados, ou seja, um grupo de modificações sobre a base de dados que deve ser executado completamente ou completamente desfeito. Logo, a diferença básica entre um programa e uma transação é que, caso alguma falha ocorra, a transação é desfeita, ou seja, qualquer modificação que ela tenha procedido sobre a base de dados será desfeita. São definidos dois comandos especiais: (i) *commit*, para garantir a persistência das alterações efetuadas por uma transação; e (ii) *abort*, para desfazê-las em caso de falha.

A definição dos programas de aplicação acontece em dois níveis: a inserção deste em um conjunto de programas de uma determinada aplicação e a definição de seu corpo (*body)*, ou seja, o código O_2C que o implementa. Durante a definição do corpo do programa de aplicação é especificado se este será um programa ou uma transação.

Os programas podem ser acionados explicitamente pelo usuário (*programas públicos*) através do menu de programas da aplicação a qual pertencem, ou implicitamente (*programas privados*) a partir da linguagem de programação O_2C .

Dois programas especiais podem ser definidos para inicializar (*init*) e terminar (*exit*) a execução de uma aplicação.

4.4.4 Funções

Funções são trechos de código não incluídos em qualquer aplicação, mas que podem ser acionados a partir de métodos, aplicações, programas ou outras funções. A definição de uma função é feita em dois níveis: a definição dos *parâmetros* de entrada e saída e a definição de seu corpo (*body)*, ou seja, o código O_2C que a implementa.

4.4.5 Execução Direta de Comandos

É possível executar trechos de código O_2C externamente a aplicações, programas ou funções. Estes trechos de código devem ser precedidos pelo comando *run body*, da seguinte forma:

run body { <código O_2*C*> *}*

5 IMPLEMENTAÇÃO O_2 DO MODELO TF-ORM

A implementação do modelo TF-ORM sobre um SGBD, requer a definição de diversas estruturas de dados e trechos de código necessários ao suporte de suas principais características.

Destaca-se, dentre estas, o conceito de papéis, ou seja, o suporte à evolução dinâmica de um objeto através de diferentes contextos de execução e o decorrente suporte à instanciação e manutenção deste objeto nos diferentes papéis. Da mesma forma, o suporte às propriedades dinâmicas e aos tipos de dados temporais, bem como à linguagem de lógica temporal, são fundamentais.

Este capítulo apresenta estruturas propostas para a implementação O_2 de esquemas de dados TF-ORM. Não são previstas, no entanto, estruturas de verificação da correção do esquema de dados sendo implementado. Tal verificação deve ser realizada pela ferramenta de geração de código (ver Anexo 5), durante a definição do esquema.

5.1 Abordagem para Implementação de Objetos com Papéis

A descrição de uma abordagem de implementação TF-ORM sobre um SGBD baseado no paradigma de orientação a objetos requer a definição de uma estratégia de mapeamento do modelo de papéis para as construções disponíveis no modelo tradicional de orientação a objetos.

Duas propriedades básicas do modelo de objetos com papéis devem ser consideradas: (i) um mesmo objeto pode desempenhar diferentes papéis simultaneamente; e (ii) um mesmo objeto pode possuir duas ou mais instâncias de um mesmo papel, paralela e simultaneamente, com contextos de execução individuais.

Uma primeira solução de mapeamento envolve o conceito de *herança* (Figura 5.1): seja uma classe TF-ORM denominada *C1*, que possua dois papéis *P1* e *P2*, sua implementação no modelo de orientação a objetos poderia ser realizada da seguinte forma:

- A classe TF-ORM *C1* é mapeada para a classe *C1_class*, cujos atributos e métodos (mensagens) são os definidos no papel básico de *C1*;
- Os papéis TF-ORM *P1* e *P2* são mapeados para as classes *P1_class* e *P2_class*, respectivamente, cujos atributos e métodos são os definidos nos papéis *P1* e *P2*;
- As classes *P1_class* e *P2_class* são definidas como subclasses de *C1_class*, estabelecendo a correspondência entre a classe e seus papéis;
- A classe TF-ORM *OBJECT* é mapeada para a classe *Object_class*, cujos atributos e métodos são os do papel básico de *OBJECT;*
- O papel *ROLE* da classe TF-ORM *OBJECT* é mapeado para a classe *Role_class*, cujos atributos e métodos são os do papel *ROLE*;

- A classe *C1_class* é definida como subclasse de *Object_class* e as classes *P1_class* e *P2_class* são definidas como subclasses de *Role_class*, herdando atributos e métodos pré-definidos.

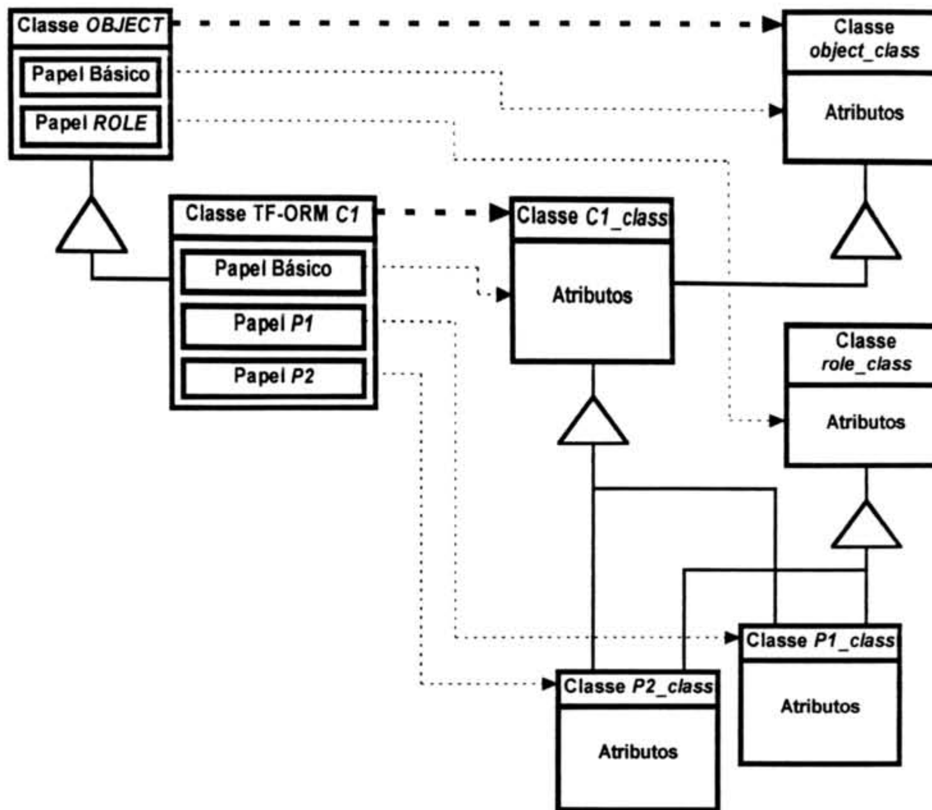


FIGURA 5.1 - Mapeamento por Herança

Este mapeamento, no entanto, não suporta as duas propriedades básicas do modelo de objetos com papéis:

- não é possível a criação de instâncias de dois ou mais papéis diferentes para um mesmo objeto: o modelo de orientação a objetos permite que um objeto seja especializado em uma subclasse da classe atual, ou seja, não é possível especializar um objeto para duas ou mais subclasses distintas, já que, por exemplo, *P1_class* não é uma subclasse de *P2_class* e vice-versa;
- não é possível a criação de duas ou mais instâncias do mesmo papel: o conceito de papel pressupõe que cada instância evolua independentemente, logo, seria necessário a especialização do objeto em duas ou mais instâncias diferentes da mesma subclasse, por exemplo, *P1*_class, o que não é permitido no modelo de orientação a objetos.

Uma solução mais adequada leva em consideração as características dinâmicas da evolução dos objetos através dos papéis, sendo implementada através do conceito de *atributos* (Figura 5.2):

- A classe TF-ORM *C1* é mapeada para a classe *C1_class*, cujos atributos e métodos (mensagens) são os definidos no papel básico de *C1*;

- Os papéis TF-ORM *P1* e *P2* são mapeados para as classes *P1_class* e *P2_class*, respectivamente, cujos atributos e métodos são os definidos nos papéis *P1* e *P2*;
- A classe *P1_class* possui dois atributos especiais *P1* e *P2*, definidos como conjuntos de instâncias das classes *P1_class* e *P2_class*, respectivamente, estabelecendo a correspondência entre a classe e seus papéis;
- A classe TF-ORM *OBJECT* é mapeada para a classe *Object_class*, cujos atributos e métodos são os do papel básico de *OBJECT;*
- O papel *ROLE* da classe TF-ORM *OBJECT* é mapeado para a classe *Role_class*, cujos atributos e métodos são os do papel *ROLE*;
- A classe *C1_class* é definida como subclasse de *Object_class* e as classes *P1_class* e *P2_class* são definidas como subclasses de *Role_class*, herdando atributos e métodos pré-definidos.

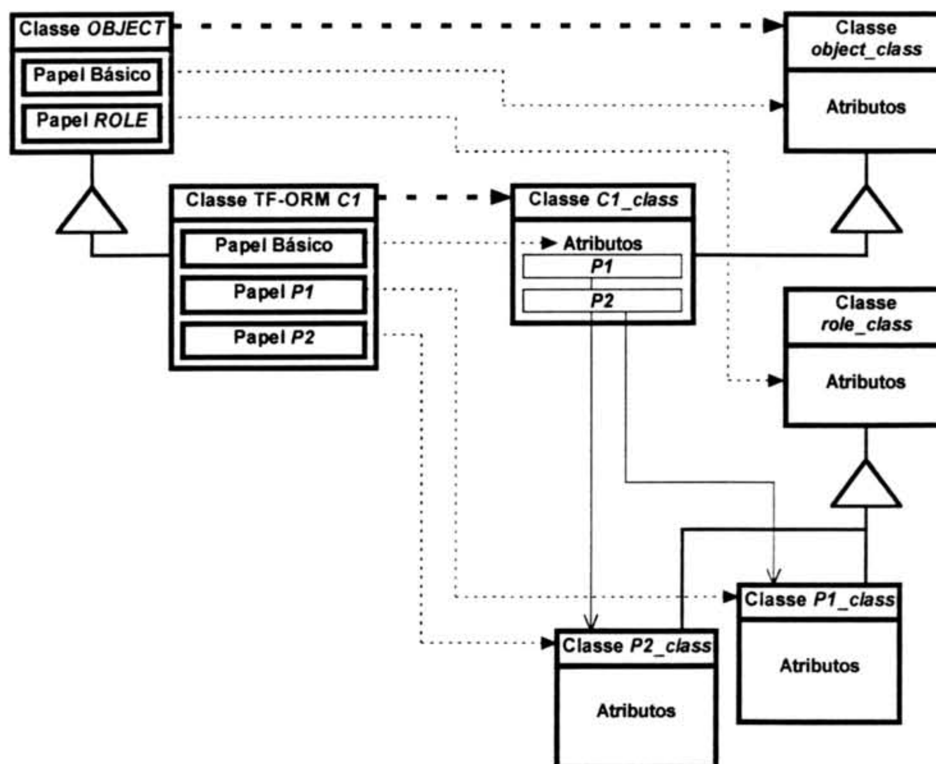


FIGURA 5.2 - Mapeamento por Atributos

Utilizando este mapeamento, a instanciação e a manutenção dos papéis pode ser realizada através de operações de criação de objetos e inserção e remoção de elementos dos conjuntos de instâncias de papéis. Desta forma, as duas propriedades básicas do modelo de objetos com papéis são suportadas:

- a criação de instâncias de dois papéis diferentes para um mesmo objeto é permitida, pois existe um conjunto de instâncias para cada papel;
- a criação de duas ou mais instâncias do mesmo papel pode ser realizada através da inserção de diversos elementos no conjunto de instâncias deste papel.

5.2 Classes

```
class  <nome da classe> ...
       type tuple  (
                     <atributos do papel básico> )
       method
                     ...
end
```

FIGURA 5.3 - Implementação de uma Classe TF-ORM

Utilizando o mapeamento por atributos, as classes definidas no modelo TF-ORM, independente de seu tipo, são implementadas como classes O_2 (Figura 5.3) e todos os objetos de uma classe TF-ORM, são instâncias da respectiva classe O_2. Os atributos da classe O_2 são os definidos para o papel básico da classe TF-ORM, desta forma, ao ser criado, um objeto será automaticamente instanciado no papel básico.

5.2.1 Descritores de Classes

```
class  Descriptor
       type tuple (  name: integer,
                     transition_rules: unique set (Rule_class),
                     integrity_rules: unique set (Rule_class) )
       method
                     public init (name: integer): Descriptor
end
```

FIGURA 5.4 - Classe O_2 *Descriptor*

Cada classe TF-ORM implementada em O_2 possui um *objeto descritor*, responsável pela criação, armazenamento (persistência) e remoção das instâncias desta classe, bem como pela manutenção do conjunto de regras definidas para esta.

```
constant name <nome da classe>: integer;
run body {<nome da classe> = <n>;}

class  Descriptor_<nome da classe> inherit Descriptor
       type tuple (  instances: unique set (<nome da classe>) )
       method
                     public init (name: integer): Descritor_<nome da classe>,

                     public create_object: <nome da classe>,
                     public kill (old: <nome da classe>): boolean
end

name <nome da classe>: Descriptor_<nome da classe>
```

FIGURA 5.5 - Descritor de Classe TF-ORM

É definida uma classe *Descriptor* (Figura 5.4), raiz da hierarquia de descritores, e uma série de classes, cada qual definindo um descritor específico de

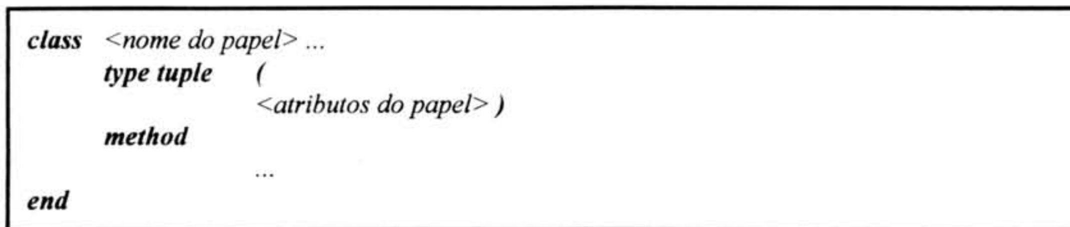
classe TF-ORM (Figura 5.5). Os descritores de classe incluem um nome único para esta classe[8] e um conjunto de regras de transição de estado e regras de integridade.

Cada descritor de classe é associado a um *named value*, a fim de garantir a persistência deste descritor e, conseqüentemente, das instâncias da classe (ver seção 4.3.4).

5.2.2 Criação de Objetos

A criação dos objetos é realizada através (i) da declaração de uma variável do tipo da classe e (ii) do envio de uma mensagem *create_object* para o descritor da classe. O método *create_object* efetivamente cria a instância (através de um comando *new*), inicializa a propriedade pré-definida *object_instance*, refletindo os tempos de transação e validade do objeto, e inclui a instância no conjunto de instâncias da classe. A variável recebe, então, o identificador do objeto (*oId*).

5.3 Papéis



```
class  <nome do papel> ...
       type tuple   (
                        <atributos do papel> )
       method
                        ...
end
```

FIGURA 5.6 - Implementação de um Papel TF-ORM

Os papéis definidos no modelo TF-ORM (com exceção dos papéis básicos) são implementados como classes O_2 (Figura 5.6), sendo, os atributos destas classes, aqueles definidos para o papel TF-ORM correspondente.

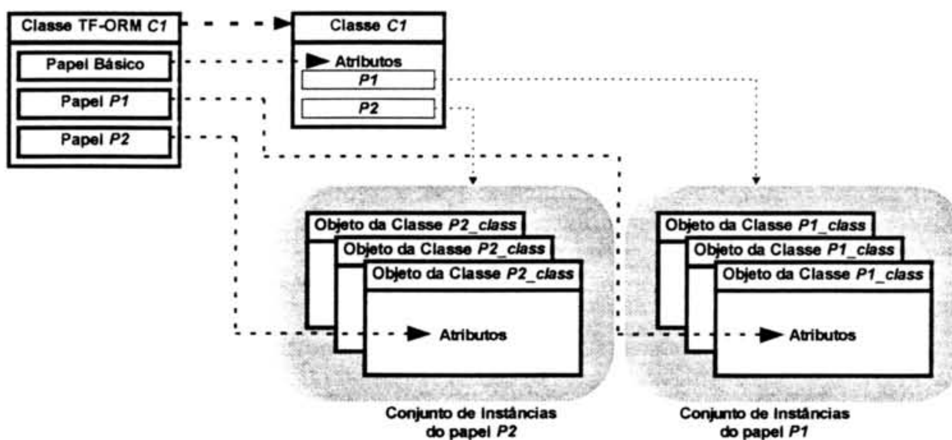


FIGURA 5.7 - Conjuntos de Instâncias de Papéis

O relacionamento dos papéis com a classe a que pertencem é realizado através de atributos especiais, recebendo o nome do papel e reunindo todas as suas instâncias (Figura 5.7). É definido, também, um atributo *state_of_roles*, como uma

[8] A partir deste momento, os nomes únicos para classes, definidos como identificadores inteiros, passam a ser referidos como *ClassName*.

lista de valores lógicos (*boolean*). Cada valor corresponde a um determinado papel definido para esta classe, e modela a possibilidade ou não da criação de instâncias deste papel.

O valor do atributo *state_of_roles* é alterado através do recebimento das mensagens pré-definidas *forbid_role (<RoleName>)* e *allow_role (<RoleName>)*: *forbid_role* faz o elemento de *state_of_roles* correspondente a *RoleName* ser *false*, impedindo a criação de novas instâncias deste papel, e *allow_role* torna o elemento *true*, permitindo novamente a instanciação.

Desta forma, uma classe TF-ORM implementada sobre O_2, possui a estrutura definida na Figura 5.8.

```
class  <nome da classe> ...
       type tuple  (
                    <atributos do papel básico>
                    <nome do papel>: unique set (<nome do papel>),
                    <nome do papel>: unique set (<nome do papel>),
                    state_of_roles: list (boolean) )
       method
                    ...
end
```

FIGURA 5.8 - Implementação do Relacionamento entre Classes e Papéis

5.3.1 Descritores de Papéis

Cada papel TF-ORM implementado em O_2, a exemplo das classes, possui um *objeto descritor* (Figura 5.9). Os descritores de papéis incluem um nome único para o papel[9].

```
constant name <nome do papel>: integer;
run body {<nome do papel> = <n>};

class  Descriptor_<nome do papel> inherit Descriptor
       method
                    public init (name: integer): Descriptor_<nome do papel>
end
```

FIGURA 5.9 - Descritor de Papel TF-ORM

5.3.2 Criação de Instâncias de Papéis

A criação de uma instância de papel é realizada através (i) da declaração de uma variável do tipo da classe que implementa o papel e (ii) do envio de uma mensagem *add_role (<RoleName>)* para o objeto a que pertencerá esta instância. O método *add_role* do objeto cria, efetivamente, uma instância da classe O_2, inicializa sua propriedade *role_instance* e a insere no conjunto de instâncias deste papel.

---

[9] A partir deste momento, os nomes únicos para papéis, definidos como identificadores inteiros, passam a ser referidos como *RoleName*.

5.4 Hierarquia de Classes e Papéis

A classe TF-ORM *OBJECT* é implementada como uma especialização da classe O_2 *object*. A classe O_2 gerada é denominada *Object_class* (Figura 5.10), sendo raiz da hierarquia das classes O_2 que implementam classes TF-ORM.

Os atributos e métodos de *Object_class* são pré-definidos e correspondem aos atributos e mensagens definidas para a classe TF-ORM *OBJECT* [10], e a consultas de atributos necessários à implementação de predicados e funções da linguagem de lógica temporal.

```
class   Object_class
        type tuple   (
                         state: State_class,
                         object_instance: Dynamic_char,
                         end_object: Dynamic_char,
                         state_of_rules: list (boolean) )
        method
                         public init: Object_class,

                         public suspend_object,
                         public resume_object,

                         public add_role (RoleName: integer),
                         public suspend_role (rid: Role_class),
                         public resume_role (rid: Role_class),
                         public terminate_role (rid: Role_class),

                         public allow_role  (RoleName: integer),
                         public forbid_role  (RoleName: integer),

                         public forget (RuleName: integer),
                         public recall (RuleName: integer),

                         public has_class_instance,
                         public has_role_instance (RoleName: integer),
                         public active_class,
                         public active_class_at (instant: Instant),
                         public out_role (RoleName: integer),
                         public role (rid: Role_class),

                         public class_creation_time,
                         public class_end_time,
                         public state,
                         public state_at (instant: Instant)
end
```

FIGURA 5.10 - Classe O_2 *Object_class*

O papel *ROLE*, da classe TF-ORM *OBJECT*, é implementado como uma especialização da classe O_2 *object*, denominada *Role_class* (Figura 5.11). Todas

[10] As mensagens *forbid_op* e *allow_op*, definidas no modelo TF-ORM, não foram implementadas, devido a restrições impostas pelo O_2 no tratamento dos métodos associados às classes, impossibilitando sua inibição, o que dificulta a modelagem natural destas mensagens.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

as classes O_2 que implementam papéis TF-ORM são definidas como subclasses de *Role_class*. Da mesma forma que *Object_class*, os atributos e métodos desta classe são pré-definidos.

```
class  Role_class
       type tuple   (
                    state: State_class,
                    role_instance: Dynamic_integer,
                    end_role: Dynamic_integer,
                    state_of_rules: list (boolean) )
       method
                    public init: Role_class,

                    public active_role,
                    public active_role_at (instant: Instant),

                    public role_creation_time,
                    public role_end_time,
                    public state,
                    public state_at (instant: Instant)
end
```

FIGURA 5.11 - Classe O_2 *Role_class*

5.4.1 Especialização e Agregação

O mecanismo de especialização do modelo TF-ORM não é suportado na abordagem de implementação O_2, pois existem limitações no que se refere a definição de restrições na herança de atributos e métodos. O sistema O_2 não permite, por exemplo, que um atributo de uma superclasse seja eliminado da definição da subclasse (cláusula *not_inherits* do modelo TF-ORM). Atributos podem ser redefinidos somente se mantiveram-se como subtipos do atributo da superclasse, e métodos devem manter a mesma estrutura de parâmetros do método da superclasse [O2T 91a]. Com isso, fica impossibilitada a utilização da cláusula *not_inherits* e a redefinição total de papéis. A herança de papéis sem modificação (cláusula *inherits*), a extensão da definição de papéis (cláusula *extends*) e a inclusão da definição de novos papéis na subclasse são possíveis.

Da mesma forma, o mecanismo de agregação de classes TF-ORM não é suportado, pois, no sistema O_2, uma classe não pode ser formada pela agregação de classes, ou seja, o mecanismo de agregação não é suportado. No entanto, é possível definir uma classe a partir da *composição* dos tipos de outras classes [O2T 91a]. Este tipo de construção permitiria implementar parcialmente o mecanismo de agregação.

Pesquisas mais detalhadas sobre a implementação do mecanismo de especialização e agregação TF-ORM não são objeto de estudo neste trabalho.

5.5 Propriedades

5.5.1 Propriedades Estáticas

Propriedades estáticas são implementadas como atributos nas classes O_2, recebendo a mesma denominação TF-ORM. Estas propriedades podem ser definidas sobre domínios TF-ORM ou sobre referências a classes e papéis. A recuperação e manipulação do seu valor é realizada através dos métodos especificados para o domínio sobre o qual está definida (ver seção 5.8).

Assim, dentro de uma classe ou papel, a definição de uma propriedade estática é realizada pelos comandos:

<nome da propriedade estática>: <Domínio>

5.5.2 Propriedades Dinâmicas

A fim de caracterizar a evolução de objetos e instâncias de papéis, é necessário armazenar não só o valor atual de uma propriedade dinâmica, mas todos os valores apresentados por esta em sua história pregressa. Além disso, o modelo TF-ORM prevê a implementação de um *banco de dados bitemporal* [EDE 94], ou seja, para cada propriedade dinâmica, é armazenado tanto o *tempo de transação* — o instante em que o valor foi armazenado — como o *tempo de validade* — o instante em que este valor passa a modelar a realidade (ver seção 3.2.2).

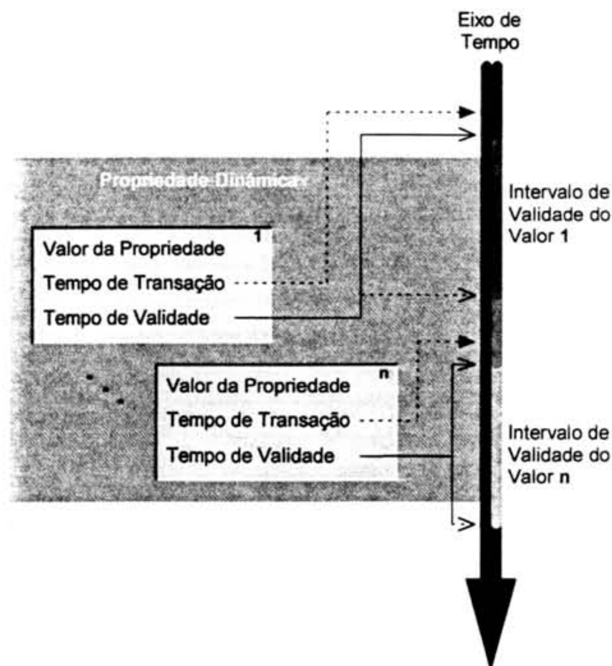


FIGURA 5.12 - Evolução dos Valores das Propriedades Dinâmicas

Desta forma, os domínios das propriedades dinâmicas devem indicar não somente o valor, mas também suas informações temporais (Figura 5.12). Cada

domínio definido no modelo TF-ORM é, portanto, implementado como uma classe O_2, a fim de modelar as propriedades dinâmicas definidas sobre este.

As classes que modelam domínios de propriedades dinâmicas recebem a denominação *Dynamic_<nome do domínio>*, sendo definidas através de uma *lista de tuplas*. Cada tupla representa um valor para esta propriedade e é formada por:

- um atributo denominado *value*, definido sobre o domínio original, modelando o valor da propriedade em um dado instante;
- um atributo instante de tempo, denominado *transaction_time*, que modela o tempo de transação do valor *value*;
- um atributo instante de tempo denominado *valid_time*, que modela o tempo de validade do valor *value*.

Assim, dentro de um papel ou classe, a definição de uma propriedade dinâmica é realizada pelos comandos:

<nome da propriedade dinâmica>: <Nome da Classe da Propriedade Dinâmica>

Todas as classes que modelam domínios dinâmicos são subclasses de *Dynamic_domain_class* (Figura 5.13), na qual estão definidas os atributos *transaction_time* e *valid_time*, bem como os métodos que manipulam estes atributos.

```
class  Dynamic_domain_class
       type list ( tuple (
              transaction_time: Instant,
              valid_time: Instant ) )
       method
              public init: Dynamic_domain_class,

              public transaction_time: Instant,
              public valid_time: Instant
end
```

FIGURA 5.13 - Classe O_2 *Dynamic_domain_class*

```
class  Dynamic_<domínio> inherit Dynamic_domain_class
       type list ( tuple (
              value: <domínio>)
       method
              public init (value: <domínio>, valid_time: Instant): Dynamic_<domínio>,
              public set_value (value: <domínio>, valid_time: Instant),

              public is_valid (value:<domínio>): boolean,
              public is_valid_at (value:<domínio>, instant: Instant): boolean,

              public value: <domínio>,
              public value_at (instant: Instant): <domínio>
end
```

FIGURA 5.14 - Implementação de um Domínio para Definição de Propriedades Dinâmicas TF-ORM

Uma série de métodos estão pré-definidos para as propriedades dinâmicas. Estes métodos têm a finalidade de suportar predicados e funções da linguagem de lógica temporal TF-ORM (Figura 5.14).

5.5.3 Propriedades Pré-Definidas

As seguintes propriedades são pré-definidas:

- *object_instance* e *end_object*: propriedades definidas na classe *Object_class* como *Dynamic_char*;
- *role_instance* e *end_role*: propriedades dinâmicas definidas na classe *Role_class* como *Dynamic_char*;

As propriedades *object_instance*, *end_object*, *role_instance* e *end_role* podem assumir os valores *null* ou *nonull*. Na abordagem de implementação, o valor *null* é representado como Ø e *nonull* como 1. Para as demais propriedades, apenas o valor *null* é considerado, sendo este implementado como o valor padrão fornecido pelo SGBD O_2.

5.6 Estados

Cada classe O_2, implementando uma classe ou um papel TF-ORM, possui um atributo *state*, definido sobre uma classe *State_class* (Figura 5.15). A classe *State_class* é definida como uma lista de tuplas representando todos os estados já ocupados pela instância. Cada tupla da lista contém:

- um atributo denominado *state*, definido como um inteiro, modelando o estado da classe ou papel em um dado instante;
- um atributo intervalo de tempo denominado *valid_time*, que modela o tempo de validade do estado *state*. O instante inicial do intervalo coincide com o instante da definição do estado, e o instante final, com o instante da definição do estado seguinte.

Adicionalmente é mantido um atributo inteiro *actual_state*, indicando o estado atual da instância, a fim de permitir sua determinação sem a necessidade de pesquisar a lista de estados e os tempos de validade.

```
class   State_class
        type tuple (
                        states: list ( tuple (  state: integer,
                                                valid_time: Interval ) ),
                        actual_state: integer)
        method
                        public init: State_class,
                        public change_state (new_state: integer),

                        public state: integer,
                        public state_at (instant: Instant): integer
end
```

FIGURA 5.15 - Classe O_2 *State_class*

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Os valores numéricos dos estados são definidos como *named values*. A seqüência de numeração dos estados é única e independe de classes e papéis, permitindo, desta forma, a correta verificação das transições de estado. Logo, os estados possíveis para as instância de um papel serão definidos como:

constant name <estado 1>: integer;
run body { <estado 1> = <n>; }
...

Estados complexos não são suportados pelo modelo de implementação. A existência de estados complexos pode, no entanto, ser simulada através de estados simples: cada estado composto possível pode ser mapeado para um estado simples.

5.6.1 Estados Pré-definidos

Os estados pré-definidos são: *active*, que pode ser ocupado pelos papéis básicos das classes TF-ORM, e *suspended*, que pode ser ocupado por todos os papéis. O estado *active* equivale ao valor inteiro Ø e o estado *suspended*, ao valor inteiro 1

5.7 Mensagens

O sistema O_2 considera as mensagens como sendo ativações de métodos. Uma referência do tipo "enviar uma mensagem para um objeto receptor" é considerada a ativação do método cujo nome coincide com o seletor de método da mensagem e cujo código está implementado na classe do objeto receptor ou em uma de suas superclasses [O2T 91]. No modelo TF-ORM, somente mensagens relevantes para a descrição do comportamento da instância, através da ativação de mudanças de estado, são consideradas [DEA 92].

Desta forma, as mensagens de entrada para classes e papéis TF-ORM são implementadas como métodos O_2, definidos na classe receptora e possuindo a mesma estrutura de assinatura (nome e definição de parâmetros). As mensagens de saída não são definidas, visto que a definição de métodos deve estar associada aos receptores das mensagens. No entanto, a verificação da possibilidade de envio de uma mensagem é realizada, durante a especificação das regras, pela ferramenta de geração de código (ver Anexo 5).

Após a definição do esquema TF-ORM sobre o SGBD O_2, métodos suplementares visando a construção de aplicações, programas e transações, devem ser especificados.

5.8 Domínios de Propriedades

5.8.1 Domínios Simples

```
class  Integer
       type tuple ( value: integer)
       method
              public init (value: integer): Integer,
              public set_value (value: integer),
              public value: integer,
              public add (value: Integer): Integer,
              public sub (value: Integer): Integer,
              public mul (value: Integer): Integer,
              public div (value: Integer): Integer,
              public pot (value: Integer): Integer,
              public less (value: Integer): Boolean,
              public greater (value: Integer): Boolean,
              public equal (value: Integer): Boolean
end
```

FIGURA 5.16 - Classe O_2 *Integer*

Os domínios simples, definidos no modelo TF-ORM, são implementados como classes O_2. Desta forma, pode ser definida uma interface de mensagens padrão para os operadores, facilitando a conversão das expressões em lógica temporal (ver Anexo 5). Todas as classes possuem métodos de inicialização e manipulação dos valores, bem como métodos implementando operadores aritméticos, relacionais e lógicos.

As classes *integer* e *real* possuem definições similares, variando apenas o domínio (Figura 5.16). A classe *boolean* (Figura 5.17) implementa métodos para os operadores lógicos *and*, *or* e *not*. A classe *string* (Figura 5.18), apresenta somente métodos de inicialização e manipulação dos valores.

```
class  Boolean
       type tuple ( value: boolean)
       method
              public init (value: boolean): Integer,
              public set_value (value: boolean),
              public value: boolean,

              public and (value: Boolean): Boolean,
              public or (value: Boolean): Boolean,
              public not: Boolean,
              public equal (value: Boolean): Boolean
end
```

FIGURA 5.17 - Classe O_2 *Boolean*

As classes *text* e *image* foram definidas como as classes O_2 *Text* e O_2 *Bitmap*, pré-definidas em um esquema de dados O_2 denominado *o2kit* e importadas através do comando:

import schema o2kit class Text, Bitmap

As classes *Text* e *Bitmap* já possuem métodos para inicialização, armazenamento e edição de textos e imagens. As classes TF-ORM *place* e *title* tiveram pouca utilização durante a construção dos estudos de caso, não sendo implementadas.

```
class  String
       type tuple ( value: string)
       method
              public init (value: string): Integer,
              public set_value (value: string),
              public value: string,

              public equal (value: String): Boolean
end
```

FIGURA 5.18 - Classe O_2 *String*

Dentro dos métodos suplementares, criados após a definição do esquema TF-ORM para a construção de aplicações, programas e transações, os valores das propriedades definidas sobre estes domínios podem ser manipulados através de operadores genéricos da linguagem O_2C, após o desencapsulamento dos valores através do método *value*. Por exemplo:

```
salário = valor ->value * 1,1;
valor->set_value(salário);
```

5.8.2 Domínios Complexos

Os domínios complexos TF-ORM conjuntos (*set_of*) e listas (*list_of*) são implementados através dos construtores de tipos complexos O_2 *set* e O_2 *list*. Desta forma, os operadores sobre conjuntos TF-ORM união (*union*) e interseção (*intersection*) podem ser implementados através de operações básicas O_2 para manipulação de conjuntos:

<conjunto 3> = <conjunto 1> + <conjunto 2>	realiza a união de *<conjunto 1>* e *<conjunto 2>*, retornando *<conjunto 3>*;
*<conjunto 3> = <conjunto 1> * <conjunto 2>*	realiza a intersecção entre *<conjunto 1>* e *<conjunto 2>*, retornando *<conjunto 3>*.

5.8.3 Domínios de Dados Temporais

A definição de pontos no tempo pressupõe a existência de informações sobre data e hora. O esquema de dados O_2 *o2kit* inclui a definição de uma classe *Date*. No entanto, a fim de implementar completamente os operadores, funções e predicados

da linguagem de lógica temporal TF-ORM, optou-se por não importar esta classe, definindo uma estrutura de informações temporais própria.

5.8.3.1 Pontos no Tempo

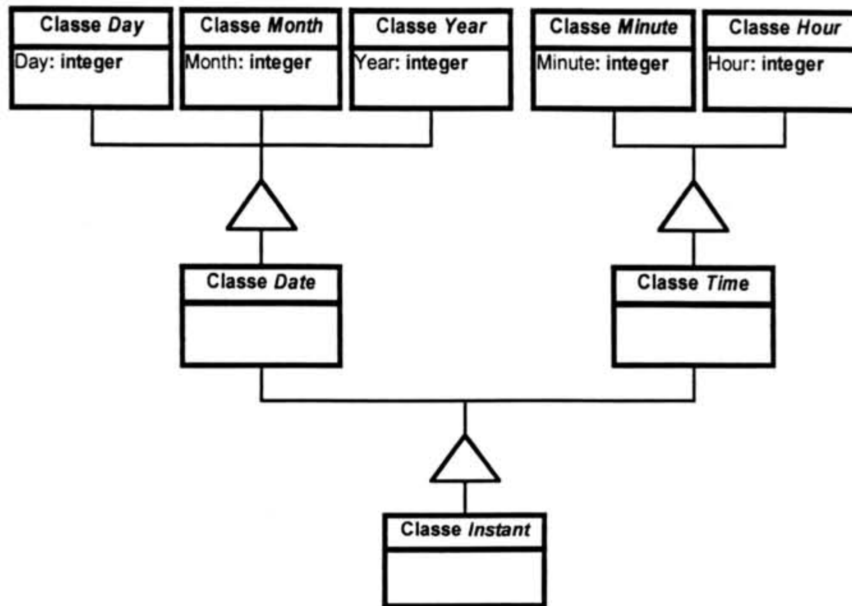
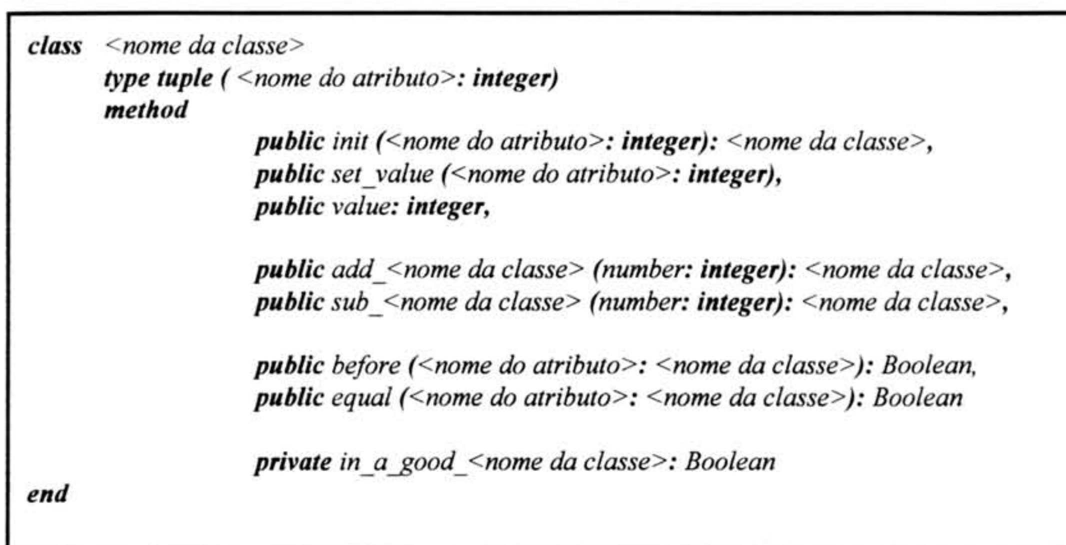


FIGURA 5.19 - Estrutura Hierárquica da Classe O_2 *Instant*

Pontos no tempo são modelados através de uma hierarquia de classes, representada na Figura 5.19. Através desta hierarquia, são definidas as superclasses *Day*, *Month*, *Year*, *Minute* e *Hour*, através das quais são supridos os métodos básicos para a inicialização e manipulação destes valores, operadores aritméticos e operadores de comparação. Por herança múltipla, são definidas as classes *Date* e *Time*, e finalmente a classe *Instant*. Os domínios *semester*, *century*, *week* e *weekday*, não foram implementados.



```
class <nome da classe>
      type tuple ( <nome do atributo>: integer)
      method
                public init (<nome do atributo>: integer): <nome da classe>,
                public set_value (<nome do atributo>: integer),
                public value: integer,

                public add_<nome da classe> (number: integer): <nome da classe>,
                public sub_<nome da classe> (number: integer): <nome da classe>,

                public before (<nome do atributo>: <nome da classe>): Boolean,
                public equal (<nome do atributo>: <nome da classe>): Boolean

                private in_a_good_<nome da classe>: Boolean
end
```

FIGURA 5.20 - Implementação das Classes Raízes da Hierarquia de *Instant*

Esta hierarquia de classes permite a definição de instantes com qualquer granularidade, como está previsto no modelo TF-ORM, bem como permite a aplicação de operadores e funções sobre estes valores. As classes raízes da hierarquia (*Day*, *Month*, *Year*, *Minute* e *Hour*) possuem estrutura semelhante, sendo baseada no modelo

descrito na Figura 5.20. A classe *Date* (Figura 5.21) renomeia os métodos de recuperação de valores (*value*) das classes *Day*, *Month*, *Year* para, respectivamente, *day*, *month* e *year*, e redefine os métodos *before* e *equal*.

```
class  Date inherit Day, Month, Year
       method
              public init (day: integer, month: integer, year: integer): Date,
              public set_value (day: integer, month: integer, year: integer): boolean,
              public set_to_current_date: Date,
              public value: type(Date),

              public day: integer,
              public month: integer,
              public year: integer,
              public weekday: string,

              public add (date: Date): Date,
              public sub (date: Date): Date,

              public before (date: Date): Boolean,
              public equal (date: Date): Boolean,

              private in_a_leap_year: boolean,
              private days_to_Date (days: integer),
              private in_a_good_date: boolean
end
```

FIGURA 5.21 - Classe O_2 *Date*

As classes *Time* (Figura 5.22) e *Instant* (Figura 5.23) possuem estrutura e métodos similares à classe *Date*.

```
class  Time inherit Minute, Hour
       method
              public init (minute: integer, hour: integer): Time,
              public set_value (minute: integer, hour: integer): boolean,
              public set_to_current_time: Time,
              public value: type(Time),

              public minute: integer,
              public hour: integer,

              public add (time: Time): Time,
              public sub (time: Time): Time,

              public before (time: Time): Boolean,
              public equal (time: Time): Boolean,

              private minutes_to_Time (minutes: integer),
              private in_a_good_time: boolean
end
```

FIGURA 5.22 - Classe O_2 *Time*

É prevista uma função *now*, a qual retorna o instante atual.

```
class  Instant inherit Time, Date
       method
            public init (minute: integer, hour: integer, day: integer, month: integer, year:
                    integer): Instant,
            public set_value (minute: integer, hour: integer, day: integer, month: integer, year:
                    integer): boolean,
            public set_to_current_instant: Instant,
            public value: type(Instant),

            public add (instant: Instant): Instant,
            public sub (instant: Instant): Instant,

            public before (instant: Instant): Boolean,
            public equal (instant: Instant): Boolean,

            public interval (instant: Instant): Interval,

            private in_a_good_instant: boolean
end
```

FIGURA 5.23 - Classe O_2 *Instant*

5.8.3.2 Intervalos

A implementação dos domínios de intervalos segue a mesma estrutura da implementação de instantes, ou seja, uma hierarquia composta pelas seguintes classes: *Day_interval*, *Month_interval*, *Year_interval*, *Minute_interval* e *Hour_interval*, e pelas suas subclasses, *Date_interval* e *Time_interval*, e finalmente a classe *Instant_interval* (Figura 5.24).

```
constant name closed=0, open=1, open_down=2, open_up=3, floating_down=4, floating_up=5

class  Instant_interval
       type tuple (  beggining: Instant,
                     ending: Instant )
       method
                     public init (beggining: Instant, ending: Instant): Instant_interval,
                     public set_value (beggining: Instant, ending: Instant): boolean,
                     public value: type(Instant_interval),

                     public lower_bound: Instant,
                     public upper_bound: Instant,

                     public duration: Minute_duration,

                     public belongs(instant: Instant): Boolean,
                     public contains(instant: Instant): Boolean
end
```

FIGURA 5.24 - Classe O_2 *Instant_interval*

5.8.3.3 Durações

Os diferentes domínios para durações são implementados através de uma hierarquia de classes, composta por uma superclasse denominada *Duration*, definindo um atributo inteiro *value*, e cinco subclasses *Minute_duration* (Figura 5.25),

Hour_duration, *Day_duration*, *Month_duration* e *Year_duration*, cada qual fornecendo métodos para interpretação e conversão da granularidade da duração.

```
class  Minute_duration
       method
              public init (value: integer): Minute_duration,
              public set_value (value: integer),
              public value: integer,

              public to_hours: integer,
              public to_days: integer,
              public to_months: integer
              public to_years: integer
end
```

FIGURA 5.25 - Classe O_2 *Minute_duration*

5.8.3.4 Tipos de Dados para Informações Temporais Incompletas

Os tipo de dados *after* e *before* não foram implementados. No entanto, podem ser considerados intervalos com limite superior (*after*) ou limite inferior (*before*) flutuantes (*floating*) ou infinitos, com a granularidade especificada. Desta forma, o modelo é simplificado, pela utilização dos mesmos métodos da classe *Instant* para sua manipulação.

5.9 Linguagem de Lógica Temporal

Os operadores definidos na linguagem de lógica temporal, sejam eles genéricos ou sobre domínios de dados temporais, são implementados como métodos das classes que definem os domínios TF-ORM sobre o SGBD O_2. Os operadores sobre conjuntos *union* e *intersection* são implementados através de operadores O_2.

Operadores temporais, como *sometime past*, *immediately past*, *since*, *before*, etc., são implementados como funções O_2. Os predicados são implementados como métodos das classes de domínios, sejam eles simples ou temporais.

5.10 Linguagem de Consulta

A linguagem de consulta temporal do modelo TF-ORM não é objeto de estudo deste trabalho. No entanto, por utilizar, em sua maior parte, os operadores, predicados e funções da linguagem de lógica temporal, grande parte de sua funcionalidade já é suportada.

6 IMPLEMENTAÇÃO O_2 DO MODELO DE REGRAS TF-ORM

O_2 pode ser considerado um SGBD *passivo*, ou seja, executa somente operações explicitamente especificadas nas requisições de usuários ou no código das aplicações, programas e transações [BEE 91].

SGBDs *ativos* estendem a funcionalidade de SGBDs passivos através da adição de elementos [SCH 91] como gatilhos (*triggers*) [SIE 92] e, mais genericamente, regras [HUL 91]. Desta forma, respondem automaticamente a eventos gerados interna ou externamente ao sistema, sem a intervenção do usuário [BAU 91].

Regras podem ser especializadas em restrições (*constraints*), através das quais, transações que levam o banco de dados a estados inconsistentes, são abortadas. Regras também têm sido utilizadas em SGBDs relacionais e, mais recentemente, em SGBDs orientados a objetos, para garantir restrições de integridade referencial, eventualmente procedendo a atualizações e remoções de instâncias "em cascata" e controlando a precedência na inclusão de instâncias [MAR 90].

Este capítulo apresenta uma análise de mecanismos de suporte a regras existentes na literatura, situando as regras do modelo TF-ORM dentro deste contexto. Uma abordagem para implementação destas regras é proposta.

6.1 Mecanismos de Suporte a Regras

A integração de regras a SGBDs orientados a objetos apresenta diferentes alternativas [DÍA 91]: (i) integração do mecanismo de ativação e execução de regras ao código dos métodos; (ii) definição de uma hierarquia especial de classes de objetos modelando regras; e (iii) integração de estruturas adicionais ao próprio modelo de dados do SGBD, visando suportar a definição de regras.

A integração de regras ao código dos métodos torna a identificação e manipulação individual das regras, visando modificações em sua natureza e estrutura, extremamente difíceis, dificultando a atividade do projetista de sistemas. Adicionalmente, a interação existente entre diversas regras, bem como aspectos de terminação, confluência e prioridades, não são facilmente representados. Regras complexas podem ter o seu código desmembrado em diferentes métodos, contrariando o paradigma da orientação a objetos, que encoraja o encapsulamento, e mesmo regras simples comprometem severamente a possibilidade de redefinição (*overriding*) de métodos.

A segunda alternativa permite uma implementação uniforme, já que regras são implementadas como parte integrante da hierarquia de classes [DAY 88, DÍA 91]. As principais vantagens desta abordagem residem, justamente, no aproveitamento das características inerentes ao modelo de orientação a objetos, como: (i) associação de regras a objetos baseado em critérios de contexto, reduzindo o escopo de pesquisa de ativação; (ii) possibilidade de definição de propriedades (atributos e métodos) de regras, tornando sua estrutura mais flexível; e (iii) criação, manipulação e remoção de regras através dos mecanismos usuais de instanciação.

A terceira alternativa é, possivelmente, a mais eficiente do ponto de vista de desempenho na verificação e ativação das regras [BAU 91], no entanto, é igualmente a mais limitada, pois está condicionada à modificação das estruturas internas do SGBD.

A implementação de mecanismos de regras em SGBDs requer a resolução de uma série de problemas, a saber [HAN 91]:

- projeto de uma linguagem de definição de regras, considerando sua estrutura em termos de condições de ativação e execução de ações;
- projeto de um mecanismo de teste de condições eficiente, que não cause restrições no desempenho das transações;
- integração do mecanismo de teste e execução de regras ao SGBD.

Regras *E-C-A* (evento-condição-ação — *event-condition-action*), propostas no modelo HiPAC em [DAY 88], podem ser consideradas um mecanismo genérico para prover capacidade ativa a SGBDs [DÍA 91].

O modelo HiPAC implementa regras E-C-A como objetos de uma classe *regra*, definida através dos seguintes atributos:

- *Identificador* da regra: como todo objeto, cada regra possui um identificador unívoco *oId*, através do qual pode ser referenciado em atributos e métodos;
- *Evento*: evento associado à ativação da regra. Por tratar-se de uma implementação sobre SGBD orientado a objetos, toda a manipulação de dados é realizada através de métodos, sendo estes, ou a sua composição, considerados eventos no modelo. Opcionalmente podem ser definidos atributos para os eventos;
- *Condição*: condição a ser testada logo após a detecção do evento. A condição é uma consulta ao banco de dados. Se esta consulta retornar valores não nulos, a regra será executada. Os valores retornados pela consulta podem ser repassados à ação;
- *Modo de Acoplamento E-C*: define o modo de acoplamento entre o evento e a condição. São definidos quatro acoplamentos: (i) *imediato*, no qual a regra é avaliada e possivelmente executada logo após a detecção do evento e a transação que ocasionou a ativação do evento aguarda o término da execução da ação; (ii) *diferido*, onde a regra é avaliada após o final da transação; (iii) *desligado e dependente-causal*, a regra é avaliada em uma transação independente, após o fim da transação que originou o evento; e (iv) *desligado e independente-causal*, a regra é avaliada em uma transação independente.
- *Ação*: operação a ser executada caso a condição se verifique;
- *Modo de acoplamento C-A*: equivalente aos modos de acoplamento E-C para condição e ação;
- *Restrições temporais*: limites de tempo, prioridades ou urgências;
- *Planos de contingência*: ação alternativa caso as restrições temporais sejam violadas;

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

- *Atributos*: atributos adicionais para as regras, podendo ser definidos sobre tipos simples ou complexos.

São previstas operações para manipulação de regras: *create*, *delete*, *enable*, *disable* e *fire*.

ADAM [DÍA 91] implementa, da mesma forma que HiPAC, regras como objetos, definindo atributos e métodos que implementam operações de gerência de regras. O suporte a regras está baseado na existência de três componentes: (i) a regra propriamente dita; (ii) um evento, que dispara a verificação das regras; e (iii) um gerador de eventos, representando qualquer possível fonte interna (por exemplo, um sinal de relógio) ou externa (por exemplo, uma aplicação de usuário) de mensagens.

Regra são definidas através de uma classe denominada *rule*. Várias instâncias desta classe, ou seja, várias regras, podem estar associadas a cada classe (e não a cada objeto), através de um atributo conjunto denominado *class-rules*. Da mesma forma, cada regra possui um atributo *active-class*, através do qual é possível estabelecer o contexto de execução da regra, e um atributo designando o evento que a ativa. Cada regra pode ser habilitada ou desabilitada através de métodos especiais.

Eventos são também definidos como instâncias, de uma classe denominada *event*, sendo possível, desta forma, definir atributos e métodos associados a estes. A definição do evento inclui a especificação da mensagem (método) associada a sua ocorrência. Com isso, eventos não estão restritos a operações de atualizações: qualquer mensagem pode ativar a execução de uma regra.

O relacionamento bilateral existente entre as classes e suas regras torna-se uma importante característica, determinando facilidade de implementação e ganhos de desempenho no teste e ativação de regras [DÍA 91].

Em [BAU 91] é descrito um sistema de gerência de regras de produção, integrado ao SGBD O_2, permitindo a definição explícita e a manipulação direta das regras. O mecanismo de suporte às regras está integrado ao SGBD, não constituindo uma camada extra.

As regras são implementadas como objetos de uma hierarquia especial de classes, mantida através de comandos especiais do SGBD. Estes objetos possuem os seguintes atributos: (i) um *nome*, identificando univocamente a regra; (ii) uma expressão descrevendo o *evento* que pode acionar a regra; (iii) uma *consulta* a ser realizada ao banco de dados, especificando a condição de ativação da regra e cujo resultado fica disponível aos métodos desta regra; (iv) um conjunto de operações representando a *ação* a ser executada; (v) um *tipo*, identificando se a regra está associada ao recebimento de mensagens ou a eventos temporais; (vi) um descritor de *prioridade*; e (vii) um descritor de *estado*, identificando se a regra está ou não habilitada. Na implementação, as regras são mapeadas para objetos mais simples, e consulta de condição e ação, reunidas em um só método.

São previstas diversas operações de atualização de regras: *add*, *delete*, *enable*, *disable*, *fire*, *connect* (conecta uma regra a um evento), *disconnect* e *change_priority*.

O algoritmo de execução é composto por uma fase de detecção de evento, a qual inicia um processo de seleção de regras habilitadas que estejam

conectadas ao evento e a conseqüente execução do método de consulta e ação. Não são considerados aspectos relativo ao desempenho na busca e execução das regras.

Em Ariel [HAN 91], é apresentada uma linguagem de definição de regras baseada no seguinte predicado:

define rule <rule name> [in <rule set name>]
[priority <priority>]
[on <event>]
[if <condition>]
then <action>

Regras são completamente definidas, portanto, por um *nome* único, um nível explícito de *prioridade*, utilizado para inserir uma relação de ordem na execução das regras, um *evento* ativador da regra, uma *condição*, a ser testada após a ativação e condicionando a execução da ação, e a *ação* propriamente dita. Adicionalmente, regras podem ser reunidas em conjuntos; no entanto, este conceito é utilizado somente em nível de definição, não sendo considerado no momento dos testes de ativação das regras.

São previstos três tipos de eventos: inserções, remoções e atualizações de tuplas. As ocorrências em cadeia dos eventos, são interpretadas como eventos únicos, simplificando o mecanismo de ativação. Desta forma, o projetista é encorajado a utilizar o conceito de blocos de comandos, similar ao conceito de transação de banco de dados. As ações podem ser comandos simples ou blocos de comandos.

O modelo prevê um algoritmo cíclico simplificado de ativação de regras, após a ocorrência de um evento:

```
match
while (rules left to run ∧ not halt)
      conflict resolution
      act
      match
end
```

O passo *match* identifica as regras associadas ao evento, armazenando-as em estruturas de dados especiais, estas regras são executadas enquanto não é acionado um comando de parada explícito. A execução consiste na resolução de conflitos, teste de condições e execução da ação propriamente dita. O algoritmo dá especial atenção aos aspectos de desempenho na busca e teste de regras.

Em [HSU 88] é discutido o mecanismo de execução de regras, no que se refere à ativação assíncrona de regras e à semântica de sua execução concorrente. Este modelo restringe HiPAC, definindo dois tipos de regras: (i) regras de consistência, relacionadas à manutenção da integridade; e (ii) regras de automação. São suportadas, também, construções temporais sobre os predicados das condições.

Ode [GEH 91] é um SGBD ativo orientado a objetos que suporta o conceito de restrições e gatilhos. Restrições estão associadas a classes de objetos, e têm por finalidade manter sua integridade, através de ações de recuperação ativadas em transações que levam os objetos a estados inconsistentes. Restrições podem ser

checadas a nível de mensagens, garantindo que em nenhum momento o objeto viola alguma destas restrições, ou a nível de transações. Gatilhos são utilizados para ativação automática de ações quando certas condições tornam-se verdadeiras.

Somente as ativações de métodos públicos das classes são considerados eventos, já que toda a alteração em um objeto deve ser realizada através de seus métodos, a fim de não ferir o princípio do encapsulamento. Desta forma, quando um método é ativado, o objeto receptor da mensagem é incluído em uma lista de *objetos a serem verificados*, onde serão verificadas as restrições e o acionamento de gatilhos.

Uma extensão ao modelo Ode baseia-se nas regras E-C-A. No entanto, define as condições como parte integrante do evento, resultando em um modelo E-A [GEH 92], eliminando a necessidade dos diferentes tipos de acoplamento entre evento, condição e ação. Este modelo prevê a existência de quatro tipos de eventos: (i) manipulação (criação, remoção e atualização) de objetos; (ii) execução de métodos; (iii) eventos temporais; e (iv) eventos associados a transações (início, término com sucesso ou fracasso). Adicionalmente, são suportados construtores de eventos complexos. A detecção dos eventos é realizada através de um autômato finito.

Alert [SCH 91] define uma série de acoplamentos para regras: (i) *acoplamento de transação*, especificando se a ação associada a uma regra executa ou não em uma transação separada à que ocasionou o evento; (ii) *acoplamento de tempo*, especificando se a transação que ocasionou o evento deve ou não esperar o término da ação associada à regra; e (iii) *modo de asserção*, especificando se a ação é executa imediatamente após a ocorrência do evento.

[SIE 92] apresenta algumas características de mecanismos de suporte a gatilhos, especialmente no que se refere à possibilidade de ativação simultânea de múltiplos gatilhos, o que levaria a situações imprevisíveis. A fim de evitar incongruências, introduz a noção de prioridade. A prioridade de cada gatilho é dada através de uma função, a qual pode ser: (i) independente do estado do banco de dados, estabelecendo uma ordem total entre os gatilhos; ou (ii) dependente do estado, onde a prioridade é calculada a partir do estado atual do banco de dados. A primeira alternativa simplifica a análise de prioridade. No entanto, as prioridades entre os gatilhos devem, geralmente, ser consideradas dependentes do estado.

Em [VOO 91] é realizado um estudo dos modelos existentes de regras e gatilhos, dando especial atenção às características de desempenho. Em [VOO 91a] é realizada uma análise das características de terminação e confluência de conjuntos de regras, ou seja, a determinação se as execuções das regras em um dado conjunto de regras (sendo que as ativações podem ocorrer "em cascata") atingem sempre um estado final, e se a ordem de execução destas regras não influi no resultado final de suas aplicações.

6.2 Análise das Regras TF-ORM

6.2.1 Regras de Transição de Estado

Regras de transição de estado TF-ORM (ver seção 3.2.5.1) possuem a seguinte forma:

run_{ij}: $state(old_1.rld_1, s_1), msg(\leftarrow old_2.rld_2, m_1) \Rightarrow$
$msg(\rightarrow old_3.rld_3, m_2), state(old_1.rld_1, s_2)$; (<condição>)

Considerando cada par *old.rld* como um identificador único de objeto (*old*)[11], a regra run_{ij} pode ser reestruturada:

if ((old₁ ->state = s₁) ∧ (<condição>))
old_3 -> m_2
old_1 -> state = s_2

O evento associado a esta regra consiste no recebimento da mensagem m_1. Com isso, as regras de transição de estado TF-ORM podem ser consideradas regras E-C-A, com a seguinte interpretação:

- *Evento*: recebimento de uma mensagem m_1;
- *Condição*: a instância old_1 deve estar no estado s_1 e a *<condição>* definida na regra deve ser satisfeita;
- *Ação*: a instância old_1 deve enviar a mensagem m_2 à instância old_3 e passar para o estado s_2.

No entanto, regras podem ser restritas, não contento a definição de todos os seus elementos, neste caso, a interpretação desta como uma regra E-C-A varia ligeiramente:

- *nenhum estado especificado do lado esquerdo da regra*: a regra pode ser executada em qualquer estado, desta forma, a condição da regra não inclui o teste de estado atual;
- *nenhuma mensagem especificada no lado esquerdo da regra*: a regra é executada tão logo a instância passe para o estado especificado. Como os eventos são considerados ativações de métodos, e a manipulação dos atributos das classes deve ser realizada sempre através de métodos, o estado poderá ser alterado somente através do método *change_state*, definido na classe *State_class*. Este método deverá incluir, portanto, uma verificação das regras da instância;
- *nenhum estado especificado no lado direito da regra*: a ação não inclui uma transição de estado;
- *nenhuma mensagem especificada no lado direito da regra*: a ação não inclui o envio de uma mensagem.

6.2.2 Regras de Transição de Estado Condicionadas a Decisões

Regras de transição de estado condicionadas a decisões diferem das regras de transição convencionais devido à inclusão de uma cláusula de decisão. Desta forma, a regra será ativada caso a decisão seja tomada e a instância esteja no estado s_1 especificado.

---

[11] Esta consideração pode ser realizada pois o modelo de implementação prevê que, tanto objetos como papéis, serão instâncias de classes, logo, identificados univocamente por um *old*.

run_{ij}: *state(oId₁.rId₁, s₁), decision(<decisão> (parâmetros))* $\Rightarrow$
msg(→oId₃.rId₃, m₂), state(oId₁.rId₁, s₂); (<condição>)

Podemos interpretar este tipo de regra como uma regra E-C-A cujo evento consiste na tomada da decisão. Logo, como eventos devem ser ativações de métodos, as decisões devem ser modeladas como métodos da classe. A ativação deste método dispara a verificação da regra.

6.2.3 Regras de Integridade

Regras de Integridade TF-ORM (ver seção 3.2.5.2) possuem a seguinte forma:

run_{ij}: *constraint (pré-condição* $\Rightarrow$ *pós-condição)*

Estas regras garantem que, caso a *pré-condição* se verifique, a *pós-condição* deverá igualmente valer, mantendo restrições de integridade sobre os valores das propriedades.

Novamente, todas as alterações impostas às propriedades devem ser realizadas através de métodos definidos nas classes. Por outro lado, conjuntos de alterações de atributos consistem em alterações no banco de dados da aplicação e, como tal, devem ser consideradas uma *transação* de banco de dados.

Logo, aplicações desenvolvidas sobre especificações de dados TF-ORM devem utilizar o conceito de transação, as quais tomam a seguinte forma:

Begin Transaction

...

oId_n -> m_n

...

oId_m -> m_m

...

commit

Em caso de alguma falha no decorrer da transação, esta poderá ser totalmente desfeita, respeitando o princípio da atomicidade de transações [DAT 91, KOR 93], através de um comando do tipo *abort*.

No SGBD O_2, o comando *abort* pode ser acionado no corpo da própria transação ou em algum dos métodos que esta ativa. Com isso, qualquer método pode ser responsável pelo cancelamento da transação.

As restrições de integridade devem ser verificadas, portanto, após cada execução de método. Se alguma restrição for violada, um comando *abort* deve ser executado. Estas regras podem ser representadas da seguinte forma:

if ((<pré-condição>) $\wedge \neg$ *(<pós-condição>))*
abort

Assim, regras de integridade podem ser consideradas regras E-C-A com a seguinte interpretação:

- *Evento*: término de um método m_i;
- *Condição*: a *pré-condição* deve ser satisfeita e a *pós-condição* não o deve;
- *Ação*: execução do comando *abort*.

6.3 Abordagem para Implementação de Regras TF-ORM

A abordagem para implementação de regras TF-ORM sobre o SGBD O_2 baseia-se integralmente no conceito de regras E-C-A, implementando tanto regras de transição de estado como regras de integridade através de uma abordagem homogênea, utilizando conceitos próprios do modelo de orientação a objetos.

É definida uma classe *Rule_class* (Figura 6.1), composta por um atributo identificando o nome literal da regra, e um identificador unívoco de regra[12]. São definidos três métodos: (i) *fire*: dispara a regra, checando se a condição se verifica e ativando a ação correspondente; (ii) *condition*: testa a condição, retornando *true* ou *false*; e (iii) *action*: executa a ação.

```
constant name <nome da regra>: integer;
run body {<nome da regra> = <n>};
...

class  Rule_class
       type tuple (
            name: string,
            RuleName: integer )
       method
            public init (name: string, RuleName: integer): Rule_class

            public fire,
            private condition: boolean,
            private action
end
```

FIGURA 6.1 - Classe O_2 *Rule_class*

Cada regra TF-ORM é definida como uma instância da classe *Rule_class*. Os objetos representando as regras definidas para uma classe ou papel são reunidos no conjunto de regras do descritor desta classe ou papel.

Tanto classes que implementam classes TF-ORM como papéis TF-ORM, definem um atributo lista de valores lógicos (*boolean*), denominado *state_of_rules*, para indicar a habilitação ou não das regras. O valor do atributo *state_of_rules* é alterado através do recebimento das mensagens pré-definidas *forget (<RuleName>)* e *recall (<RuleName>)*: *forget* faz o elemento de *state_of_rules*

[12] A partir deste momento, os nomes únicos para regras, definidos como identificadores inteiros, passam a ser referidos como *RuleName*

correspondente a *RuleName* ser *false*, desabilitando a verificação da regra, e *recall* torna o elemento *true*, habilitando novamente a regra.

6.4 Mecanismo de Ativação e Execução de Regras

O mecanismo de ativação e execução de regras implementado baseia-se parcialmente no modelo definido em [DÍA 91], no qual os relacionamentos existentes entre objetos e regras são utilizados como um mecanismo de indexação visando a melhoria do desempenho na seleção de regras.

Cada classe O_2 que implementa classes ou papéis TF-ORM, possui um método especial denominado *check_transition_rules*, o qual tem por finalidade selecionar, entre as regras presentes no conjunto de regras de transição de estado do descritor da classe ou papel, aquelas que devem ser executadas. Outro método, denominado *check_integrity_rules* seleciona as regras de integridade. A seleção é realizada mediante o teste do atributo *state_of_rules*: toda a regra habilitada tem o seu método *fire* acionado.

O método *check_transition_rules* é acionado no início de cada método correspondente a uma mensagem definida explicitamente na especificação TF-ORM. O método *check_integrity_rules* é acionado no fim de qualquer um dos métodos definidos para a classe ou papel.

Logo, seja m_1 uma mensagem de entrada definida explicitamente na especificação TF-ORM: m_1 possivelmente será considerada em alguma das regras de transição de estado. O método acionado por esta mensagem terá a forma:

```
method m1 (<parâmetros>)
{
        self ->check_transition_rules;
        ...
}
```

O método *change_state* da classe *State_class* ativa, da mesma maneira, o método *check_trasition_rules*.

Seja m_2 um método pré-definido ou definido pelo programador de aplicações sobre uma especificação TF-ORM: m_2 possivelmente realiza alguma alteração nos atributos do objeto, logo, terá a forma:

```
method m2 (<parâmetros>)
{
        ...
        self ->check_integrity_rules;
}
```

O método *change_state* da classe *State_class* ativa ambos os métodos *check_transition_rules* e *check_integrity_rules*.

Este algoritmo de execução possui como principais vantagens a redução do número de testes a serem realizados na busca das regras, pois somente regras habilitadas serão consideradas e, mesmo assim, somente aquelas regras associadas à

classe do objeto em execução e que se enquadrem na categoria associada ao evento (início do método que implementa mensagem de entrada — regra de transição de estado — final de método de atualização — regra de integridade).

Não são considerados aspectos de terminação e confluência de cada conjunto de regras, ficando isto a cargo do projetista encarregado da especificação do modelo. Igualmente são desconsideradas observações sobre a execução concorrente de regras.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

7 ESTUDO DE CASO: AMBIENTE DE PRODUÇÃO

A abordagem de implementação proposta permite a implementação de grande parte das construções TF-ORM. Desta forma, é possível desenvolver, no sistema O_2, uma ampla gama de aplicações, definidas sobre o modelo TF-ORM. Com o intuito de exemplificar e, em especial, certificar as proposições realizadas, foi desenvolvido um estudo de caso, baseado em um ambiente de produção hipotético e simplificado.

Este capítulo apresenta uma adaptação dos elementos de representação gráfica de requisitos F-ORM, definidos em [BEL 92], para refletir os conceitos TF-ORM, e os utiliza na definição da aplicação exemplo. Adicionalmente, o ANEXO 3 Estudo de Caso: Código TF-ORM, apresenta a definição da aplicação utilizando a sintaxe da linguagem TF-ORM.

7.1 Representação Gráfica de Requisitos TF-ORM

Requisitos TF-ORM podem ser especificados formalmente através de sua linguagem de definição (ver Anexo 1). No entanto, durante a modelagem de aplicações, torna-se fundamental a utilização de representações diagramáticas que, além de facilitar a construção do modelo da aplicação, permitem sua decomposição hierárquica, representando a subdivisão do problema em diversos subproblemas com progressivos níveis de detalhe [BEL 92].

A representação diagramática utilizada neste trabalho baseia-se nos elementos gráficos utilizados pela ferramenta RECAST (*Requirements Collection And Specification Tool*) [FUG 91] *Apud* [BEL 92]. RECAST permite o desenvolvimento de aplicações F-ORM baseado nos seguintes conceitos:

- Representação diagramática dos componentes da aplicação, incluindo sua estrutura funcional, classes, papéis, mensagens, regras e estados;
- Reutilização de componentes, através de pesquisas direcionadas a um banco de dados de classes F-ORM, metaclasses e elementos de programação.

A ferramenta RECAST baseia-se na metodologia de análise F-ORM [DEA 91], voltada especialmente à reutilização de componentes. Esta metodologia busca identificar e representar a dinâmica da realização de atividades dentro de uma organização, ou seja, o que é feito, quem está envolvido, como o trabalho é realizado, não somente em termos da informação utilizada ou produzida, mas levando em consideração as regras que governam o comportamento do sistema de informação.

Por estar voltada à modelagem de aplicações F-ORM, as organizações são compreendidas como coleções de *agentes*, *recursos*, *procedimentos*, *tarefas* e *operações*, elementos básicos do modelo TF-ORM. Desta forma, embora a ferramenta RECAST não compreenda aspectos temporais, seus elementos gráficos podem ser utilizados para a representação de grande parte da funcionalidade de aplicações TF-ORM. Os principais elementos gráficos utilizados em RECAST são: *Diagramas de Clusters*, *Diagramas de Classes* e *Diagramas de Transição de Estados*.

Adicionalmente, podem ser utilizados elementos auxiliares, como *Diagramas de Cooperação entre Clusters* e *Diagramas de Hierarquia de Classes*, no entanto, tais elementos apenas representam mais objetivamente informações presentes nos demais.

7.1.1 Diagrama de *Clusters*

Um *cluster* pode ser definido como uma área funcional ou atividade de uma organização que envolve, no mínimo, um agente, um recurso e um processo. O conceito de *cluster* é utilizado para permitir a divisão hierárquica da funcionalidade da aplicação, facilitando, desta forma, a modelagem de cada área funcional ou atividade individual. Cada *cluster* sumariza o comportamento geral dos procedimentos relacionados à determinada atividade e, em especial, as dependências desta com as demais.

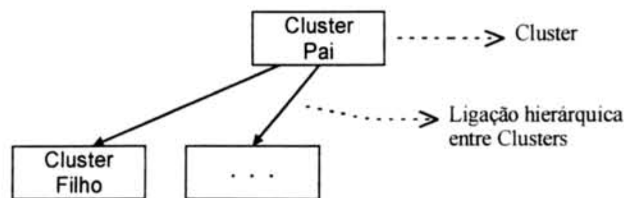


FIGURA 7.1 - Componentes do Diagrama de *Clusters*

7.1.2 Diagrama de Classes

Cada *cluster* possui associado um Diagrama de Classes, representando a composição do *cluster* em suas classes processo e recurso, as dependências entre estas classes e papéis e a relação com outros *clusters*. Não foi originalmente prevista a representação diferenciada das classes agente pela ferramenta RECAST, sendo estas consideradas classes recurso. Neste trabalho, a fim de facilitar a identificação destes agentes, foi adotada uma representação particular.

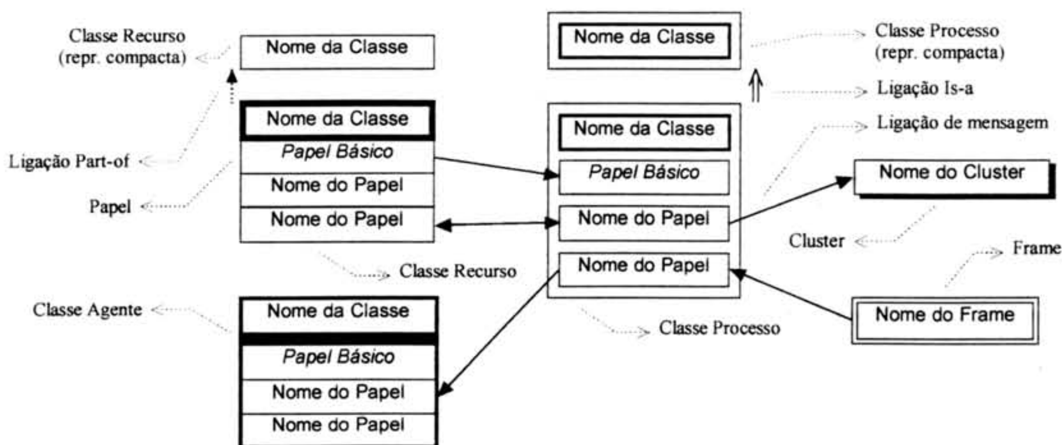


FIGURA 7.2 - Componentes do Diagrama de Classes

Cada classe F-ORM é representada através de um *componente classe*. A representação inclui o nome da classe e um nível variável de detalhe, com representações diferenciadas:

- *representação básica*: inclui o nome da classe e os papéis definidos para esta classe;

- *representação compacta*: representa somente o nome da classe;
- representação detalhada: é semelhante a representação básica, no entanto, os papéis apresentam à especificação das mensagens de entrada e saída.

Cada componente classe pode ser representado em diversos diagramas, associados a diferentes *clusters*, em geral com diferentes níveis de detalhe.

Os papéis são representados como *componentes papéis*, interconectados através de *ligações de mensagem*. Uma ligação de mensagem entre dois papéis sumariza todas as mensagens trocadas entre estes papéis, podendo ser uni ou bidirecional.

São representadas, adicionalmente, as relações com os demais *clusters*, através de *componentes cluster* e as relações com componentes externos à aplicação, através de *componentes frame*. As relações de especialização e agregação entre classes são representadas através de *ligações is-a* e *ligações part-of*, respectivamente.

A metodologia de análise F-ORM não prevê a representação gráfica das propriedades estáticas e dinâmicas dos papéis, sendo necessário, portanto, uma ferramenta adicional que identifique e descreva tais propriedades. Neste trabalho, tabelas representam e descrevem a natureza das propriedades de cada classe e papel.

7.1.3 Diagramas de Transição de Estados

Os Diagramas de Transição de Estados são construídos para cada papel definido nos Diagramas de Classes. A representação das transições de estado na metodologia F-ORM difere da representação utilizada em [RUM 91], pois compreende não só as transições propriamente ditas (estados e mensagens), mas também as entidades participantes, ou seja, os papéis fonte/destino das mensagens. Cada regra de transição é definida como um componente regra separado, a fim de representar todos os seus elementos.

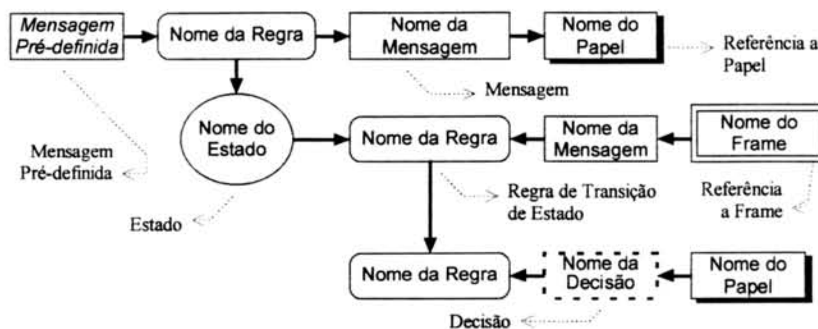


FIGURA 7.3 - Componentes do Diagrama de Transição de Estado

Um componente regra possui ligações de entrada provenientes de um *componente estado* e/ou de um *componente mensagem*, que especificam as pré-condições para a ativação da regra. Uma ligação de saída de um componente regra a um componente estado representa uma transição. Uma ligação de saída de um componente regra a um componente mensagem representa o envio desta mensagem. Algumas das ligações podem ser omitidas, permitindo a modelagem dos 4 tipos distintos de regras (ver seção 3.2.5.1). Não foi prevista originalmente uma

representação para as decisões tomadas pelos agentes, já que o modelo F-ORM não apresenta tais estruturas, neste trabalho, no entanto, foi adotada uma representação especial.

Os Diagramas de Transição de Estados podem conter, ainda, referências a papéis externos, que trocam mensagens de entrada e/ou saída com o papel sendo especificado. Estes componentes são denominados *componentes referência-a-papel*. Caso a classe de um componente referência-a-papel seja diferente da classe do papel sendo descrito, o nome desta classe deve ser incluído.

7.2 Descrição do Problema

O estudo de caso baseia-se em uma aplicação de ambiente de produção simplificado. Neste ambiente são processados pedidos provenientes de clientes pré-cadastrados. Para cada pedido recebido, é emitida uma ordem de serviço e designado um funcionário para executá-la. O processo produtivo é simples, composto de uma seqüência fixa de passos.

Antes de iniciar a produção propriamente dita, é necessário buscar, junto ao Almoxarifado, os insumos necessários à montagem do produto. Caso ocorra a falta de algum destes insumos, é emitida uma ordem de compra para algum dos fornecedores.

A partir de análise preliminar do problema, foram identificadas quatro áreas funcionais: *Administração de Vendas*, *Administração de Produção*, *Administração de Materiais* e *Administração de Compras*. Cada área funcional do ambiente produtivo é representada como um *cluster*.

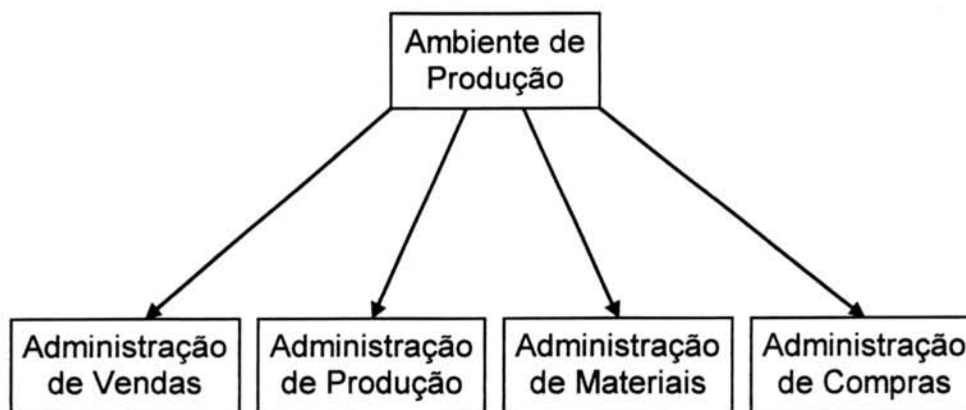


FIGURA 7.4 - Estudo de Caso: Diagrama de *Clusters*

7.3 Diagramas de Classes

7.3.1 *Cluster* Administração de Vendas

A Administração de Vendas compreende o recebimento de pedidos de clientes, seu atendimento, através de produtos em estoque (caso exista quantidade suficiente) ou através de processo produtivo, e a decorrente emissão da nota fiscal de venda.

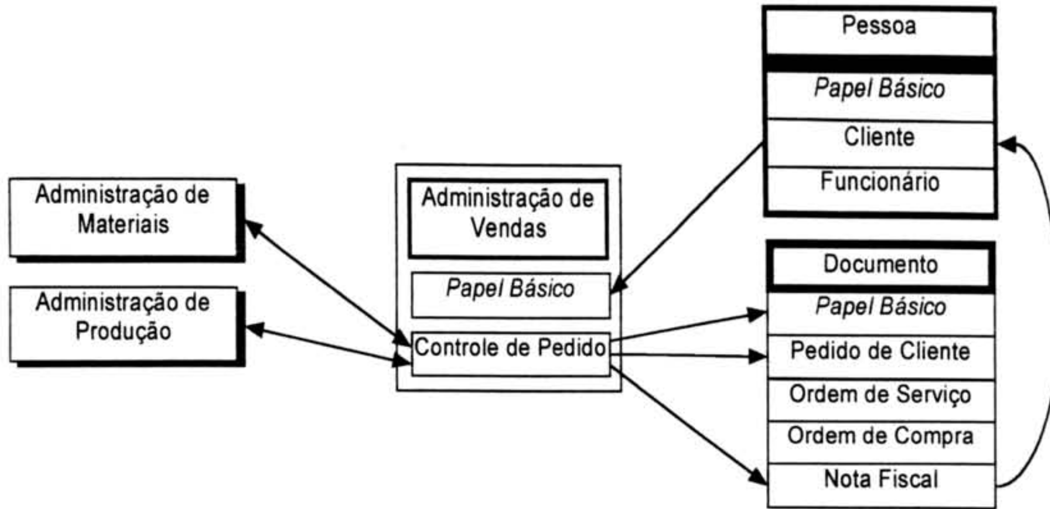


FIGURA 7.5 - Diagrama de Classes para o *Cluster* Administração de Vendas

7.3.2 *Cluster* Administração de Produção

A Administração de Produção controla o processo produtivo, desde a emissão das ordens de serviço, alocação de funcionários para as linhas de produção, acompanhamento das etapas da produção, até o encerramento da ordem.

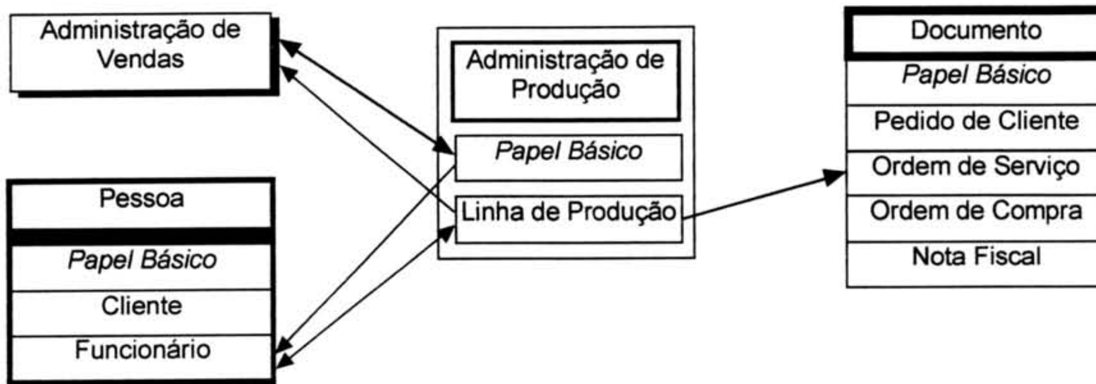


FIGURA 7.6 - Diagrama de Classes para o *Cluster* Administração de Produção

7.3.3 *Cluster* Administração de Materiais

A Administração de Materiais compreende a separação de insumos para a produção e o controle físico de estoque.

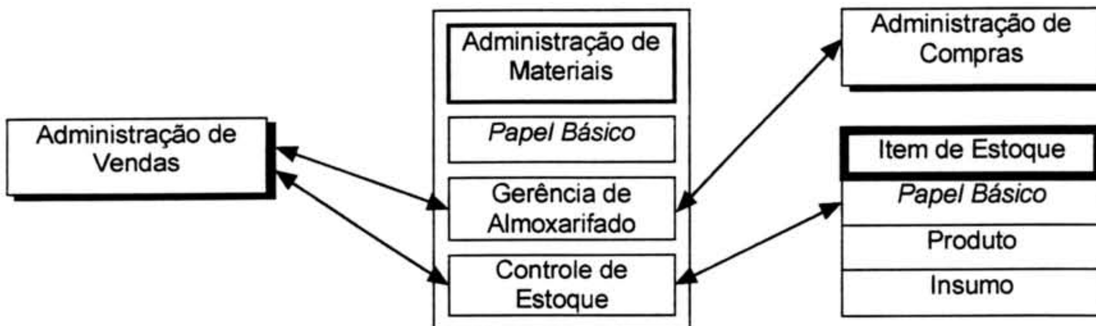


FIGURA 7.7 - Diagrama de Classes para o *Cluster* Administração de Materiais

7.3.4 *Cluster* Administração de Compras

A Administração de Compras é requisitada sempre que detectada a falta de algum insumo em estoque, controlando a emissão de ordens de compra e a interação com os fornecedores.

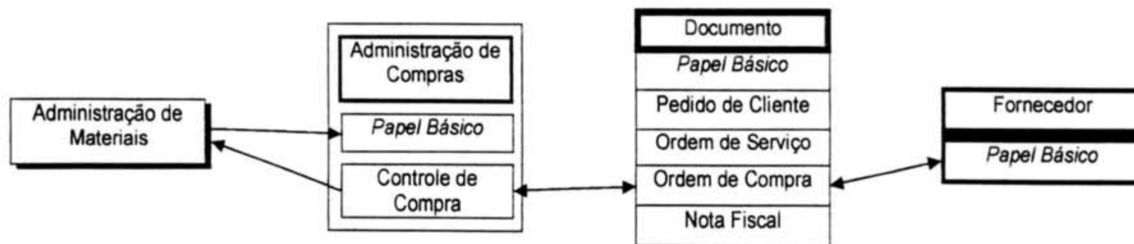


FIGURA 7.8 - Diagrama de Classes para o *Cluster* Administração de Compras

7.4 Especificação de Classes

7.4.1 Classes de Recurso

7.4.1.1 Classe Documento

Os documentos representam toda a informação relacionada ao processo produtivo, incluindo o pedido do cliente, a ordem de serviço que o executa, as ordens de compra necessárias para adquirir os insumos em falta e a nota fiscal emitida após a venda. Todos os documentos permanecem armazenados mesmo após seu encerramento.

7.4.1.1.1 Papel Básico

As propriedades presentes no *papel básico* da classe Documento têm a finalidade de permitir a localização de documentos armazenados, através de sua identificação por um número único, independente do tipo de documento, da data em que foi processado ou do funcionário que o criou.

TABELA 7.1 - Estudo de Caso: *Papel Básico* da Classe *Documento*

Nome da Propriedade	Descrição	Tipo
Número	Número do documento.	Estática
Emissão	Instante em que este documento foi emitido.	Estática
Responsável	Funcionário responsável pelo processamento.	Dinâmica

Um documento é criado e encerrado sempre a partir de uma atividade.

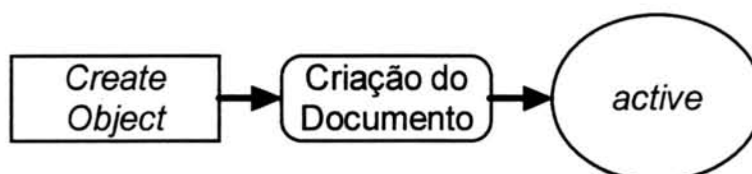


FIGURA 7.9 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Documento*

7.4.1.1.2 Papel Pedido de Cliente

Cada pedido de cliente é emitido pela Administração de Vendas, a partir de negociação prévia.

TABELA 7.2 - Estudo de Caso: Papel *Pedido de Cliente* da Classe *Documento*

Nome da Propriedade	Descrição	Tipo
Data de Confirmação	Data da confirmação deste pedido	Estática
Cliente	Cliente para o qual destina-se o pedido.	Estática
Produto	Produto solicitado neste pedido.	Dinâmica
Quantidade do Produto	Quantidade solicitada neste pedido.	Dinâmica
Data de Entrega	Data prevista para a entrega do pedido.	Dinâmica

Após o atendimento do pedido, este é encerrado.

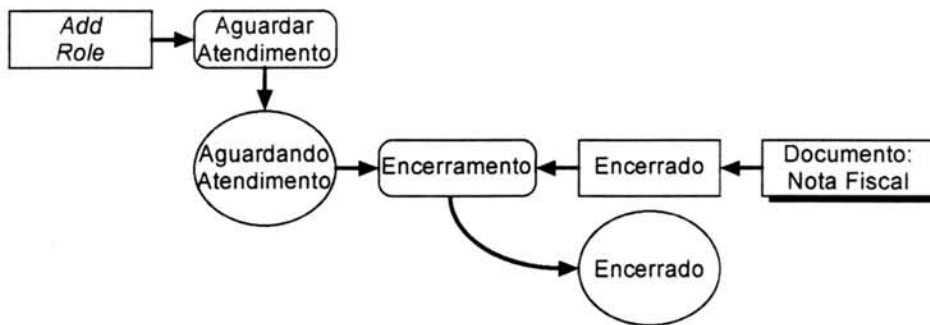


FIGURA 7.10 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Pedido de Cliente* da Classe *Documento*

7.4.1.1.3 Papel Ordem de Serviço

Cada ordem de serviço é criada para atender a um pedido de cliente e, possivelmente, gerar um estoque de determinado produto.

TABELA 7.3 - Estudo de Caso: Papel *Ordem de Serviço* da Classe *Documento*

Nome da Propriedade	Descrição	Tipo
Estágio	Setor no qual a ordem de serviço se encontra. Através desta propriedade dinâmica é possível verificar a evolução do processo produtivo.	Dinâmica
Intervalo de Produção	Intervalo decorrido entre a abertura da ordem e o seu encerramento.	Dinâmica

As informações da ordem de serviço são enviadas a um funcionário, que será responsável pela execução de todo o processo produtivo, após o qual informará o término da produção, encerrando esta ordem. Após o término da produção, parte da quantidade produzida será utilizada para atender ao Pedido de Cliente e o eventual excedente é registrado como entrada de produto em estoque.

A produção segue uma seqüência fixa e pré-definida de passos, sendo o encerramento de cada qual comunicado pelo funcionário responsável.

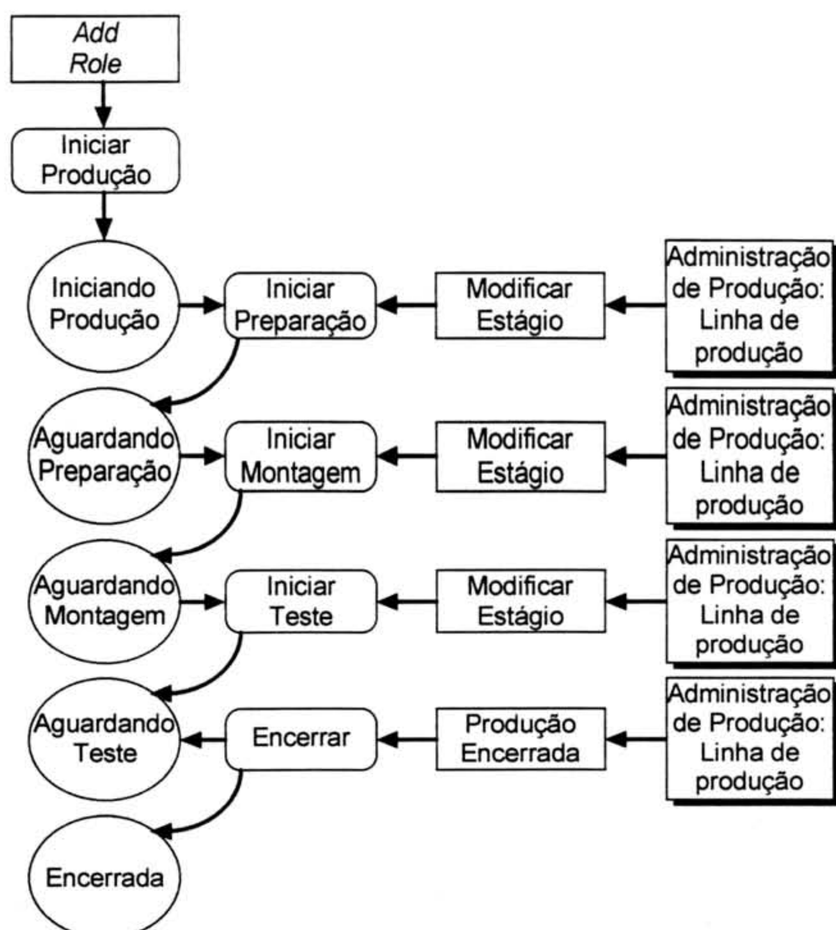


FIGURA 7.11 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Ordem de Serviço* da Classe *Documento*

7.4.1.1.4 Papel Ordem de Compra

Uma ordem de compra é emitida para suprir as faltas de insumos detectadas durante o processo de separação de materiais para a produção.

TABELA 7.4 - Estudo de Caso: Papel *Ordem de Compra* da Classe *Documento*

Nome da Propriedade	Descrição	Tipo
Instante de Emissão	Instante de emissão da ordem de compra.	Estática
Fornecedor	Fornecedor do insumo.	Dinâmica
Data Prevista	Data prevista para a entrega dos insumos.	Dinâmica
Intervalo de Espera	Intervalo de espera pelo recebimento dos produtos.	Estática
Insumo	Insumo a ser adquirido.	Estática
Quantidade	Quantidade a ser adquirida.	Dinâmica
Valor	Valor unitário do insumo.	Dinâmica

Após a emissão da ordem de compra, esta é enviada ao fornecedor que deve efetuar a entrega dos insumos obrigatoriamente até a data prevista, caso contrário estará sujeito a decréscimo em seu índice de qualidade.

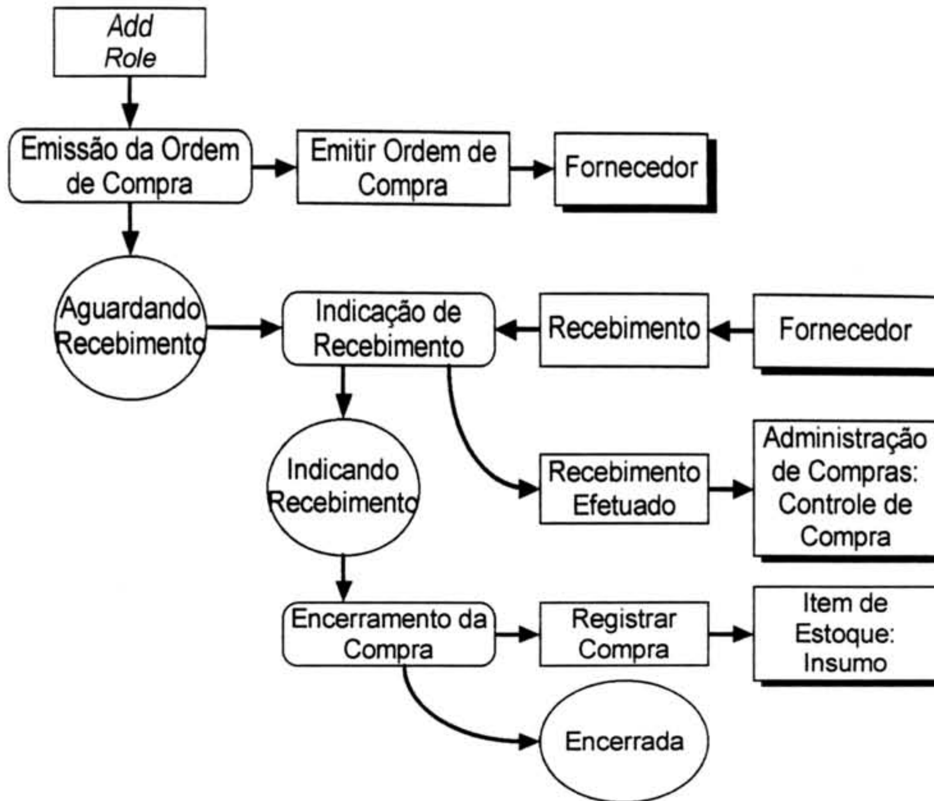


FIGURA 7.12 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Ordem de Compra* da Classe *Documento*

7.4.1.1.5 Papel Nota Fiscal

Uma nota fiscal é emitida após o processamento do pedido e enviada para o cliente juntamente com o produto.

TABELA 7.5 - Estudo de Caso: Papel *Nota Fiscal* da Classe *Documento*

Nome da Propriedade	Descrição	Tipo
Data de Emissão	Data de emissão e arquivamento de cópia da Nota Fiscal.	Estática
Valor	Valor total da Nota Fiscal.	Dinâmica

Antes de emitir a nota fiscal, é necessário armazenar, junto ao cliente, as informações sobre o pedido atendido e informá-lo que o produto e a nota fiscal impressa já se encontram a disposição. A partir das informações sobre os vários pedidos é possível determinar os principais clientes em volume de compras efetuadas e suas preferências.

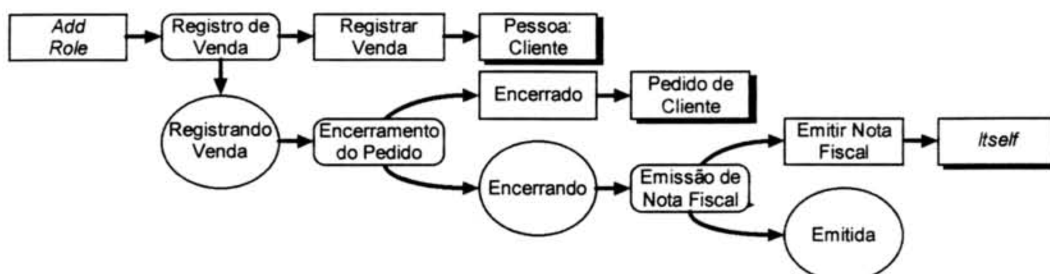


FIGURA 7.13 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Nota Fiscal* da Classe *Documento*

7.4.1.2 Classe Item de Estoque

Cada Item de Estoque pode representar um insumo a ser adquirido dos fornecedores ou um produto produzido na empresa, desde que fisicamente armazenados no Almoxarifado.

7.4.1.2.1 Papel Básico

As propriedades definidas no *papel básico* da classe Item de Estoque permitem a identificação unívoca de uma peça, através do seu código, e sua caracterização, através de uma descrição, valor (de compra, no caso de insumos; ou venal, no caso de produtos) e a quantidade disponível em estoque.

TABELA 7.6 - Estudo de Caso: *Papel Básico* da Classe *Item de Estoque*

Nome da Propriedade	Descrição	Tipo
Código da Peça	Código unívoco de identificação de peças.	Estática
Descrição	Descrição textual da peça.	Estática
Valor	Valor médio unitário desta peça. Para Produtos, este é o valor venal médio e, para insumos, este é o preço de compra médio.	Dinâmica
Composição	Propriedade definida sobre domínio composto (conjunto), indicando quais peças compõe a atual.	Dinâmica
Quantidade	Quantidade disponível em estoque.	Dinâmica

Uma peça pode ser composta de diversas outras, formando uma estrutura hierárquica, cuja raiz representa um produto final e as folhas, insumos adquiridos dos fornecedores.

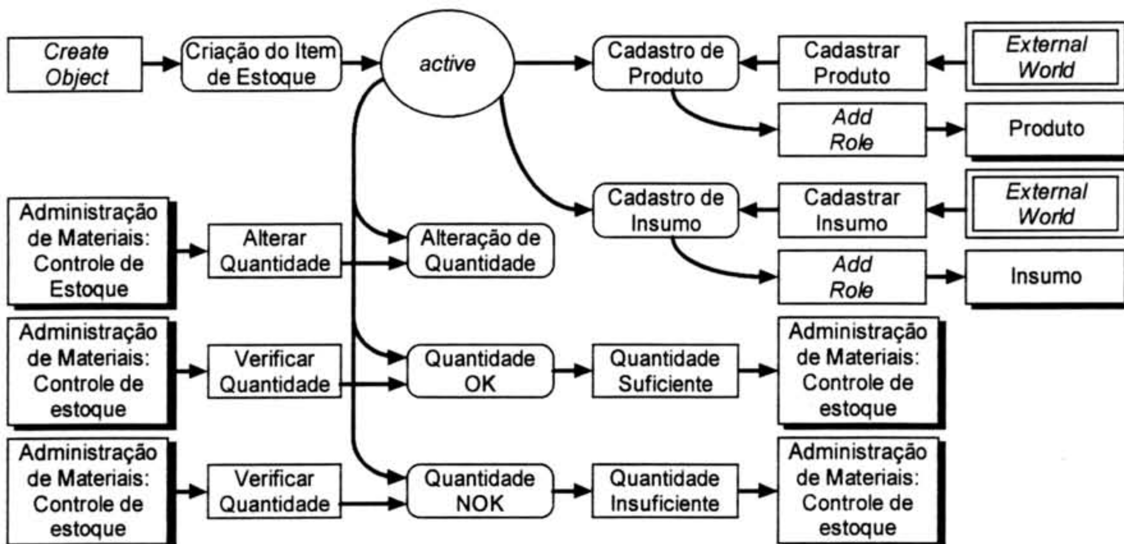


FIGURA 7.14 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Item de Estoque*

A quantidade disponível de determinado Item de Estoque pode ser alterada pelo Setor de Controle de Estoque. No caso de um produto, esta quantidade é reduzida ao ser atendido um pedido de cliente ou aumentada caso tenha sido concluída uma ordem de serviço com excedente de produção. No caso de um insumo, a

quantidade é reduzida durante a separação de insumos para produção e aumentada após o recebimento dos itens adquiridos através de ordens de compra.

7.4.1.2.2 Papel Produto

Produtos são todos os itens de estoque que podem ser vendidos a um cliente. Um produto pode possuir um nome fantasia, pelo qual será conhecido.

TABELA 7.7 - Estudo de Caso: Papel *Produto* da Classe *Item de Estoque*

Nome da Propriedade	Descrição	Tipo
Nome Fantasia	Nome fantasia do produto.	Dinâmica
Custo	Custo médio total de produção	Dinâmica
Ciclo de Produção	Duração média do ciclo produtivo.	Dinâmica

A fim de possibilitar sua análise, é necessário contabilizar os custos de produção, incluindo insumos utilizados e pagamento de horas de trabalho aos funcionários responsáveis.

FIGURA 7.15 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Produto* da Classe *Item de Estoque*

7.4.1.2.3 Papel Insumo

As propriedades dos insumos incluem a lista de possíveis fornecedores e propriedades dinâmicas indicando todos os valores já pagos por este insumo bem como todos os fornecedores dos quais este já foi adquirido.

TABELA 7.8 - Estudo de Caso: Papel *Insumo* da Classe *Item de Estoque*

Nome da Propriedade	Descrição	Tipo
Fornecedores	Lista de fornecedores desta peça.	Dinâmica
Valor	Valor da última compra.	Dinâmica
Fornecedor	Fornecedor da última compra.	Dinâmica

Os insumos são adquiridos através da emissão de uma ordem de compra. Após o encerramento desta, são atualizadas as propriedades dinâmicas a fim de refletirem a última compra efetuada.

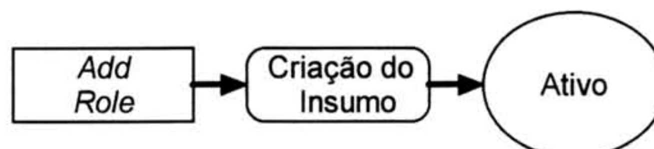


FIGURA 7.16 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Insumo* da Classe *Item de Estoque*

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

7.4.2 Classes de Processo

7.4.2.1 Classe Administração de Vendas

7.4.2.1.1 Papel Básico

O papel básico da classe Administração de Vendas executa a criação dos processos de controle individual de pedido.

TABELA 7.9 - Estudo de Caso: *Papel Básico* da Classe *Administração de Vendas*

Nome da Propriedade	Descrição	Tipo
Responsável	Funcionário responsável pelo setor.	Dinâmica
Data de Posse	Data da posse do responsável.	Dinâmica

A cada pedido de cliente recebido, a Administração de Vendas inicia uma atividade de controle de pedido que, efetivamente, acompanha o processamento e evolução deste pedido.

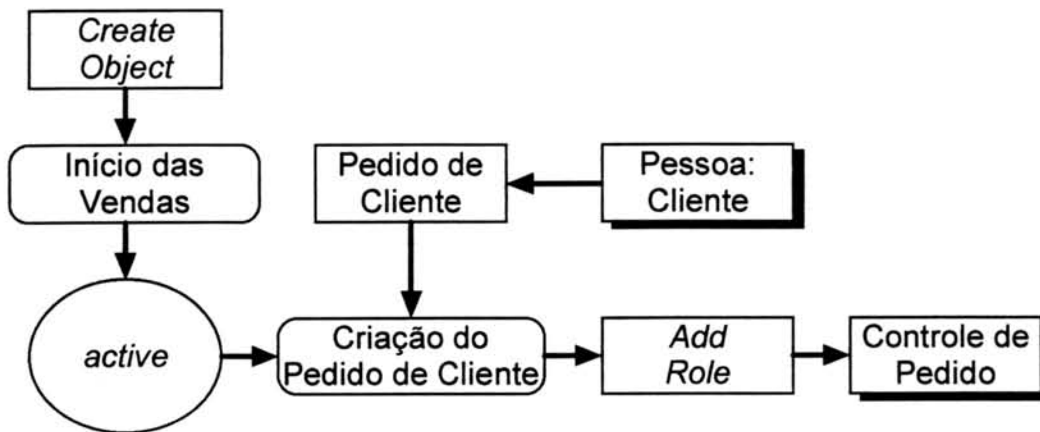


FIGURA 7.17 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Vendas*

7.4.2.1.2 Papel Controle de Pedido

Cada instância de controle de pedido é responsável pelo acompanhamento de um pedido de cliente em particular.

TABELA 7.10 - Estudo de Caso: Papel *Controle de Pedido* da Classe *Administração de Vendas*

Nome da Propriedade	Descrição	Tipo
Documento	Documento do pedido controlado por esta instância.	Estática
Instante de Início	Instante de início do processamento do pedido.	Estática

Após a recepção de um pedido, é criada a documentação necessária ao seu atendimento. Um pedido pode ser atendido de duas maneiras: através de produtos em estoque, caso exista quantidade suficiente deste produto no Almoxarifado, ou através de produção via ordem de serviço. Em ambos os casos, após o atendimento, é emitida uma nota fiscal.

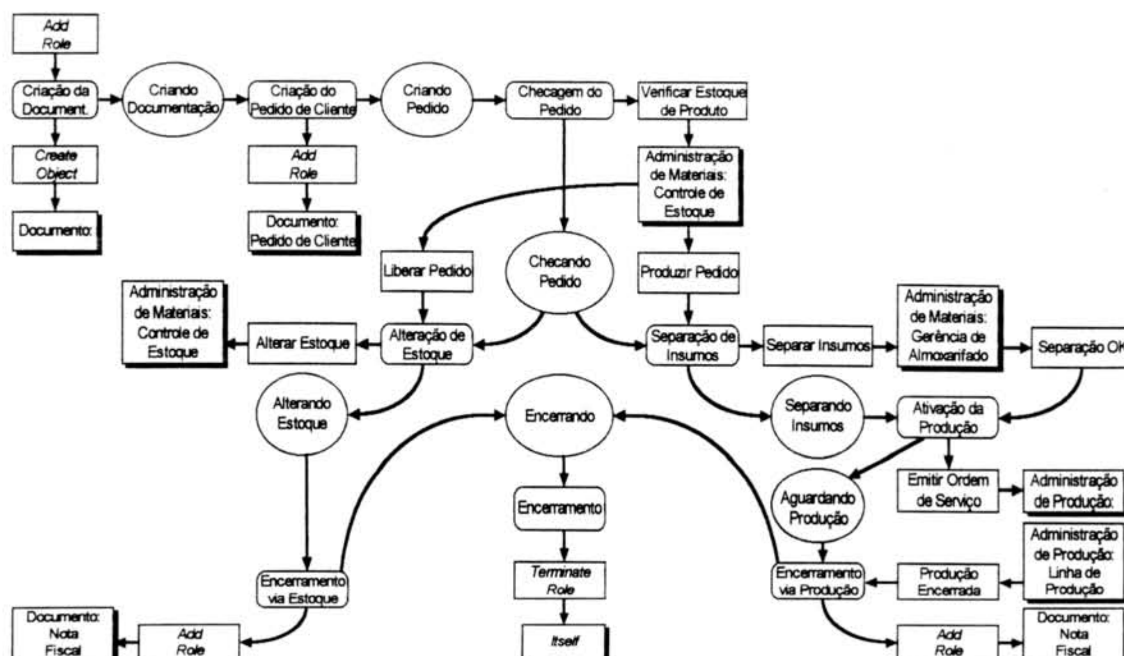


FIGURA 7.18 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Controle de Pedido* da Classe *Administração de Vendas*

7.4.2.2 Classe Administração de Produção

7.4.2.2.1 Papel Básico

O papel básico da classe Administração de Produção possui as informações necessárias ao controle da emissão de ordens de serviço, a fim de que estas não excedam o limite da capacidade produtiva.

TABELA 7.11 - Estudo de Caso: *Papel Básico* da Classe *Administração de Produção*

Nome da Propriedade	Descrição	Tipo
Ocupação	Percentual de ocupação das linhas de produção.	Dinâmica
Limite	Percentual limite de ocupação, a partir do qual é emitido um aviso para o funcionário responsável.	Dinâmica
Responsável	Funcionário responsável pelo setor.	Dinâmica
Data de Posse	Data da posse do responsável.	Dinâmica

A cada emissão de ordem de serviço é computado o percentual de ocupação das linhas de produção, as quais são criadas dinamicamente, e comparado com o limite estipulado. Em caso de nível crítico de ocupação, é comunicado o funcionário responsável pela linha de produção e a ordem de serviço aguarda a desocupação.

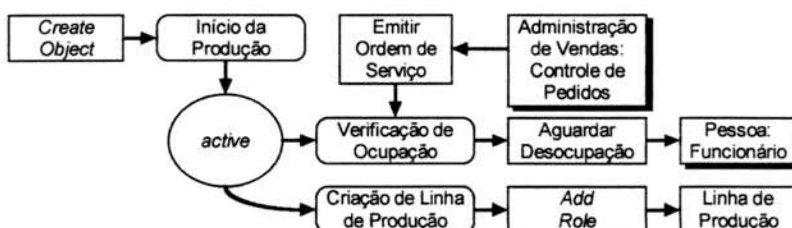


FIGURA 7.19 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Produção*

7.4.2.2.2 Papel Linha de Produção

Cada linha de produção é criada para processar um pedido de cliente. Com isso, pode ser identificada pelo número da documentação que inclui o pedido ou pelo número do funcionário responsável.

TABELA 7.12 - Estudo de Caso: Papel *Linha de Produção* da Classe *Administração de Produção*

Nome da Propriedade	**Descrição**	**Tipo**
Documento	Documento que inclui o Pedido de Cliente que esta linha visa atender.	Estática
Responsável	Funcionário responsável pela produção.	Dinâmica

O processamento de um pedido compreende a verificação do estoque do produto solicitado, se este for suficiente, é emitida a Nota Fiscal e entregue o produto; caso contrário, é produzida quantidade *não inferior* à necessária para cobrir o pedido, ou seja, é possível produzir unidades adicionais, a fim de permitir a manutenção de quantidade em estoque.

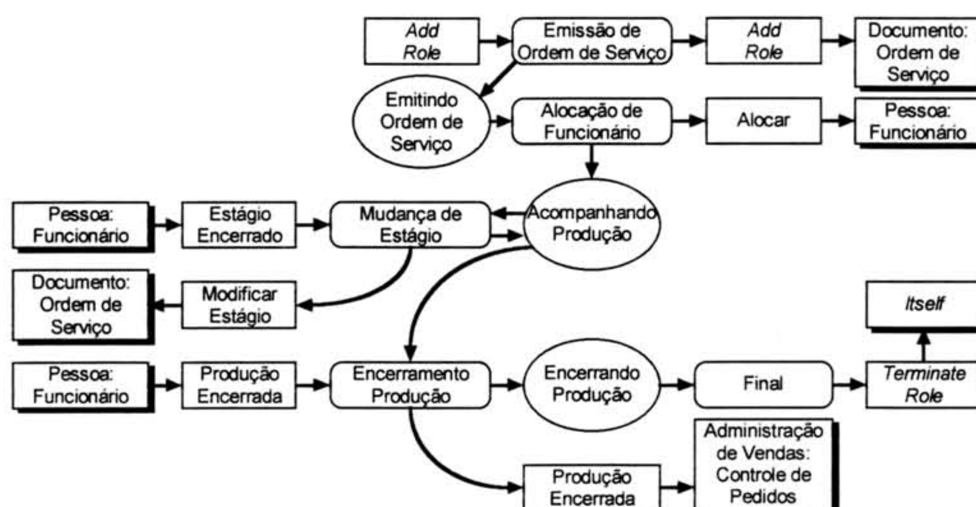


FIGURA 7.20 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Linha de Produção* da Classe *Administração de Produção*

7.4.2.3 Classe Administração de Materiais

7.4.2.3.1 Papel Básico

A Administração de Materiais é responsável pelo controle dos materiais (insumos e produtos) no Almoxarifado.

TABELA 7.13 - Estudo de Caso: *Papel Básico* da Classe *Administração de Materiais*

Nome da Propriedade	**Descrição**	**Tipo**
Responsável	Funcionário responsável pelo setor.	Dinâmica
Data de Posse	Data da posse do responsável.	Dinâmica

Existem, basicamente, duas rotinas na Administração de Materiais: controle físico de estoque e separação de insumos para a produção.

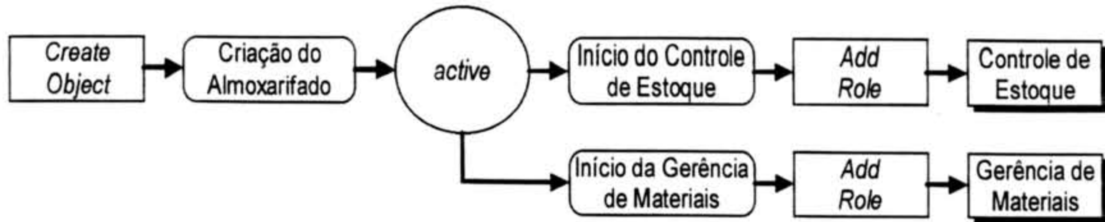


FIGURA 7.21 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Materiais*

7.4.2.3.2 Papel Gerência de Almoxarifado

A Gerência de Almoxarifado é responsável pela separação dos insumos para a produção.

TABELA 7.14 - Estudo de Caso: Papel *Gerência de Almoxarifado* da Classe *Administração de Materiais*

Nome da Propriedade	Descrição	Tipo
Ocupação	Percentual de ocupação dos funcionários do setor.	Dinâmica

A separação de insumos consiste em sucessivas consultas e alterações no estoque físico. Caso o insumo requerido possua quantidade suficiente, este é separado, caso contrário, é emitida uma ordem de compra a fim de suprir a falta de estoque e, eventualmente, manter um estoque mínimo.

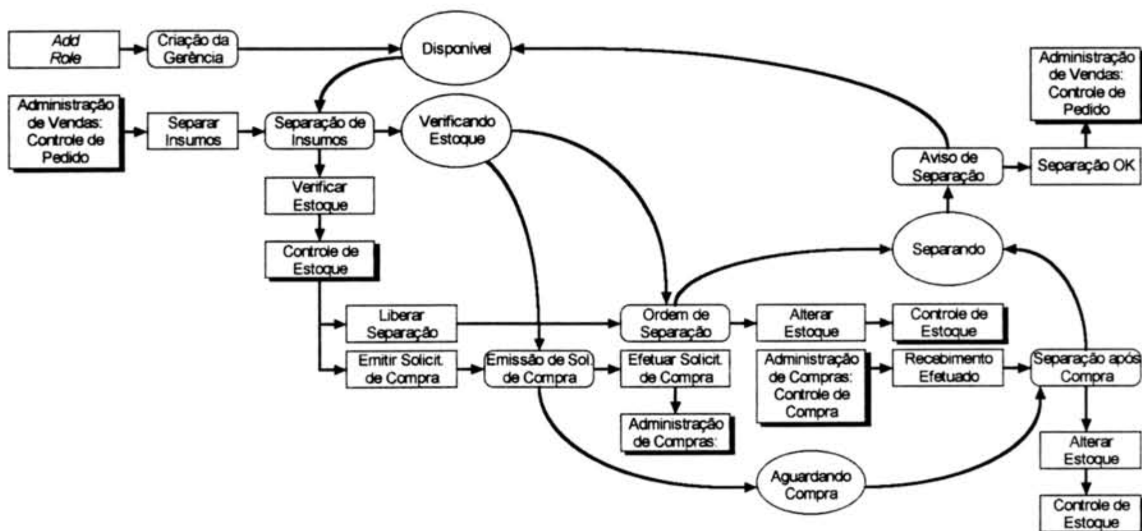


FIGURA 7.22 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Gerência de Almoxarifado* da Classe *Administração de Materiais*

7.4.2.3.3 Papel Controle de Estoque

O controle de estoque realiza o controle físico das quantidades de produtos e insumos no Almoxarifado.

TABELA 7.15 - Estudo de Caso: Papel *Controle de Estoque* da Classe *Administração de Materiais*

Nome da Propriedade	Descrição	Tipo
Data de Verificação	Data da última contagem física de estoque.	Dinâmica

Basicamente, são solicitadas ao controle de estoque (i) verificações de quantidade disponível de produtos, para atender a pedidos de cliente, ou de insumos, para atender à separação de materiais para ordens de serviço; e (ii) alteração da quantidade de produtos e insumos após estas atividades.

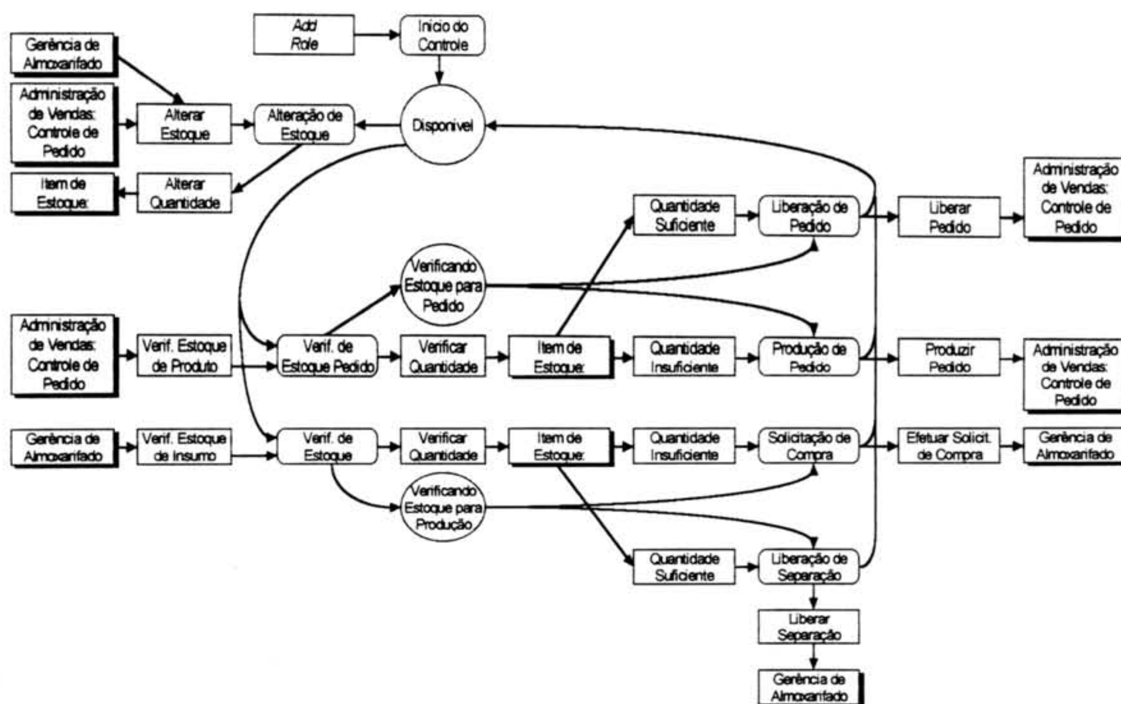


FIGURA 7.23 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Controle de Estoque* da Classe *Administração de Materiais*

7.4.2.4 Classe Administração de Compras

7.4.2.4.1 Papel Básico

A Administração de Compras controla o processo de emissão de ordens de compra e recebimento dos materiais adquiridos dos fornecedores.

TABELA 7.16 - Estudo de Caso: *Papel Básico* da Classe *Administração de Compras*

Nome da Propriedade	Descrição	Tipo
Responsável	Funcionário responsável pelo setor.	Dinâmica
Data de Posse	Data da posse do responsável.	Dinâmica

Adicionalmente, é realizado o cálculo do índice de qualidade dos fornecedores, de acordo com a eficiência na entrega e a qualidade dos materiais recebidos.

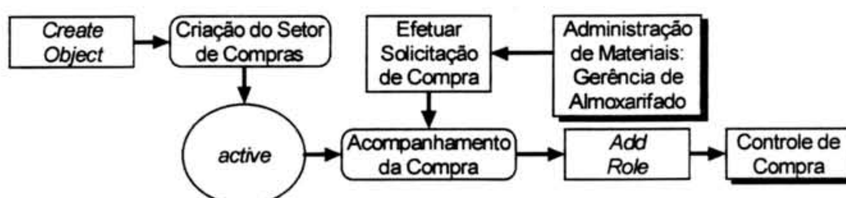


FIGURA 7.24 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Administração de Compras*

7.4.2.4.2 Papel Controle de Compra

Para cada ordem de compra emitida é designado um controle de compra a cargo de um funcionário da Administração de Compras.

TABELA 7.17 - Estudo de Caso: Papel *Controle de Compra* da Classe *Administração de Compras*

Nome da Propriedade	Descrição	Tipo
Documento	Documento que inclui a ordem de compra a ser atendida.	Estática
Responsável	Funcionário responsável pela negociação e recebimento dos insumos.	Dinâmica

O recebimento dos insumos é comunicado à gerência de almoxarifado que, desta forma, pode suprir as faltas de material para produção.

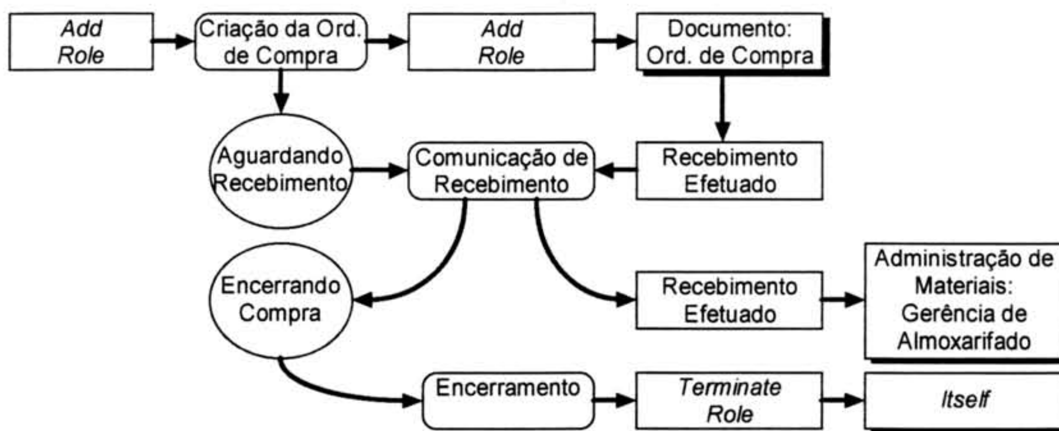


FIGURA 7.25 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Controle de Compra* da Classe *Administração de Compras*

7.4.3 Classes de Agentes

7.4.3.1 Classe Pessoa

7.4.3.1.1 Papel Básico

A classe pessoa inclui funcionários e clientes da empresa, já que todos estes são pessoas físicas, identificados pelo seu CPF.

TABELA 7.18 - Estudo de Caso: *Papel Básico* da Classe *Pessoa*

Nome da Propriedade	Descrição	Tipo
CPF	CPF da pessoa.	Estática
Nome	Nome da Pessoa.	Dinâmica
Data de Nascimento	Data de nascimento da pessoa.	Estática
Endereço	Endereço da pessoa.	Dinâmica

É admissível que uma mesma pessoa possa trabalhar na empresa e simultaneamente enviar pedidos como cliente.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

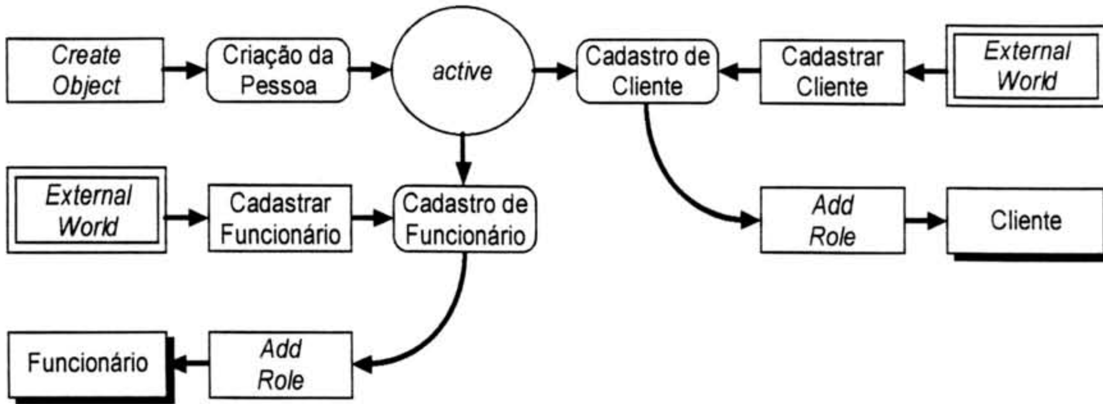


FIGURA 7.26 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Pessoa*

7.4.3.1.2 Papel Cliente

Os clientes são os responsáveis pela solicitação de produtos, através de pedidos de cliente.

TABELA 7.19 - Estudo de Caso: Papel *Cliente* da Classe *Pessoa*

Nome da Propriedade	Descrição	Tipo
Último Documento	Último documento de pedido emitido para este cliente.	Dinâmica

Através de uma propriedade dinâmica que armazena o último documento associado a este cliente, é possível obter diversas informações sobre este: total de compras, produtos preferenciais, etc.

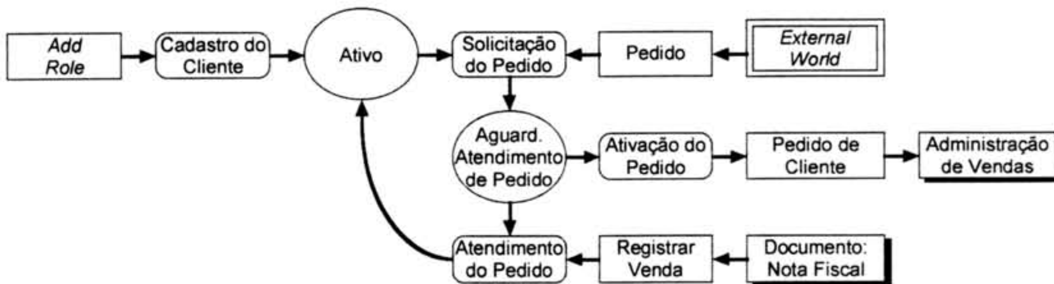


FIGURA 7.27 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Cliente* da Classe *Pessoa*

7.4.3.1.3 Papel Funcionário

Os funcionários são responsáveis tanto pelo controle dos setores como pela execução do processo produtivo.

TABELA 7.20 - Estudo de Caso: Papel *Funcionário* da Classe *Pessoa*

Nome da Propriedade	Descrição	Tipo
Setor	Setor em que atua este funcionário.	Dinâmica
Salário	Valor pago por hora de trabalho.	Dinâmica
Número de Horas	Número de horas de trabalho no mês corrente.	Dinâmica
Atuação	Indica se o funcionário atua diretamente no setor de produção.	Dinâmica
Estado	Indica se o funcionário está livre ou alocado.	Dinâmica

Funcionários que atuem no setor de produção podem ser alocados para linhas de produção. No entanto, um funcionário já alocado não pode ser responsável por outra linha de produção.

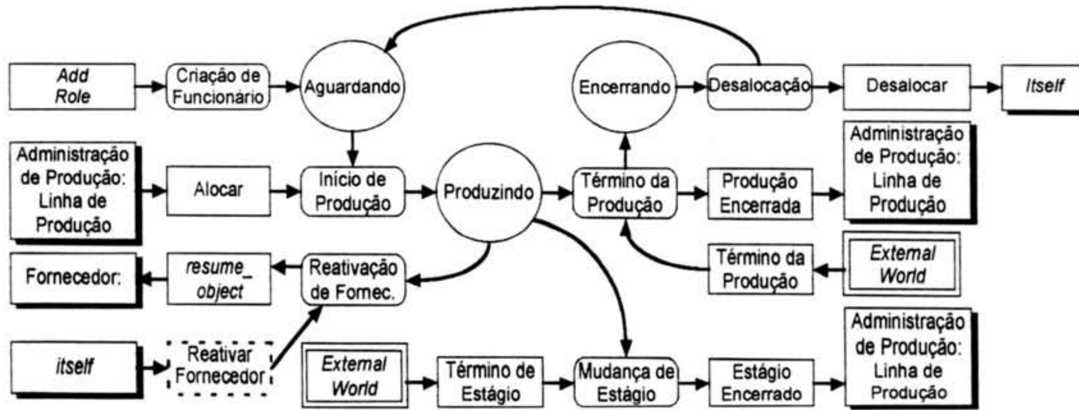


FIGURA 7.28 - Estudo de Caso: Diagrama de Transição de Estado para o Papel *Funcionário* da Classe *Pessoa*

7.4.3.2 Classe Fornecedor

7.4.3.2.1 Papel Básico

Fornecedores são responsáveis pelo provimento de insumos para produção, mediante o atendimento de ordens de compra.

TABELA 7.21 - Estudo de Caso: *Papel Básico* da Classe *Fornecedor*

Nome da Propriedade	Descrição	Tipo
CGC	CGC do fornecedor.	Estática
Nome	Razão social do fornecedor.	Dinâmica
Endereço	Endereço do fornecedor.	Dinâmica
Telefone	Telefone do fornecedor.	Dinâmica
Índice de Qualidade	Índice de qualidade de prazo e recebimento.	Dinâmica

Ao acusar o recebimento de uma ordem de compra pendente, o fornecedor é avaliado através de um índice de qualidade. Caso este índice torne-se inferior ao limite pré-estabelecido de 100, este fornecedor será suspenso, sendo novamente ativado somente após uma decisão do funcionário responsável pela Administração de Compras.

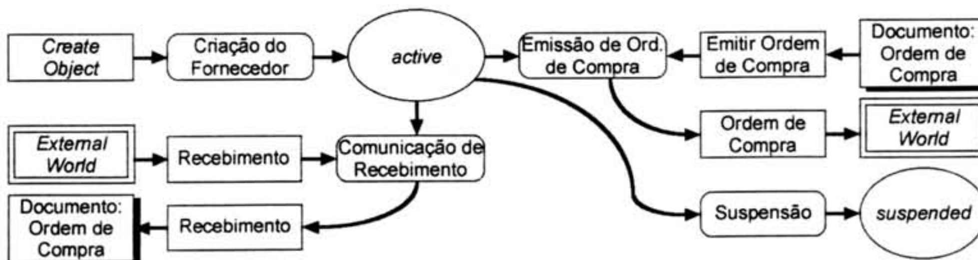


FIGURA 7.29 - Estudo de Caso: Diagrama de Transição de Estado para o *Papel Básico* da Classe *Fornecedor*

8 CONCLUSÕES

A abordagem apresentada neste trabalho permite a implementação de grande parte da funcionalidade representada em especificações de requisitos TF-ORM. Desta forma, uma extensa variedade de aplicações TF-ORM podem ser construídas sobre o SGBD O_2, utilizando os conceitos de papéis, evolução dinâmica de valores, domínios de propriedades e regras de transição de estado e regras de integridade.

Modelos de objetos com papéis, dentre estes o modelo TF-ORM, estendem o modelo de orientação a objetos, permitindo a especificação de requisitos de sistemas de informação sob uma nova perspectiva: um objeto não possui uma evolução seqüencial, ou seja, através de sua existência, pode estar inserido em diferentes e paralelos contextos de execução.

O modelo tradicional de orientação a objetos pressupõe que cada objeto esteja associado, em um dado instante, a uma determinada classe, que descreva seus atributos e governe o seu comportamento. A evolução do conjunto de atributos e do comportamento do objeto está diretamente vinculada a sua instanciação em uma subclasse da original. Desta forma, situações em que um mesmo objeto pode possuir atributos e comportamento diferentes e/ou simultâneos, não são corretamente modeladas.

A divisão dos atributos e do comportamento dos objetos nos diversos papéis que estes podem desempenhar torna a representação dos requisitos de sistemas de informação mais intuitiva e, portanto, mais próxima da realidade. No entanto, é necessário que estas especificações, construídas utilizando-se o modelo TF-ORM, possam ser implementadas sobre algum SGBD atualmente disponível. O projetista de especificações TF-ORM deve ser capaz de realizar o processo de definição dos esquemas de dados de forma transparente, mesmo no momento de sua especificação sobre o SGBD alvo. A ferramenta de tradução TF-ORM desenvolvida permite a conversão de especificações TF-ORM para esquemas de dados O_2.

A implementação de modelos de objetos com papéis não encontra, ainda hoje, um suporte transparente através dos modelos de orientação a objetos existentes. O mapeamento de papéis adotado neste trabalho permite sua implementação de uma forma homogênea, e o suporte completo aos mecanismos de criação de objetos e instâncias de papéis através de operações normalmente encontradas em qualquer sistema.

A evolução dos objetos ao longo dos papéis, bem como a completa caracterização de suas existências a partir dos valores das propriedades, tornam a utilização de aspectos tempo-dependentes fundamental. Com isso, a associação de marcas de tempo a instâncias e propriedades permite a recuperação adequada de informações, sejam estas válidas no momento atual, como em qualquer outro, passado ou futuro.

A utilização de marcas de tempo, por sua vez, requer a definição de métodos especiais para recuperação de informações a partir de condições temporais. Além disso, a representação dos valores pregressos das propriedades dinâmicas TF-ORM requerem a existência de um banco de dados bitemporal, através do qual seja possível o armazenamento e recuperação dos tempos de transação e validade associados a cada valor. A definição de classes de propriedades dinâmicas, as quais,

além dos valores, armazenam os instantes de definição (tempo de transação) e de validade (tempo de validade), permite a realização de quaisquer consultas, retornando valores simples ou temporais, através de métodos pré-definidos.

A implementação dos domínios temporais de propriedades TF-ORM encontra, na vasta gama de granularidades possíveis, sua maior dificuldade. A hierarquia de classes adotada para a implementação dos tipos instante de tempo, intervalo e duração, permite a modelagem homogênea destes domínios, e sua manipulação através de métodos associados às classes. Esta última característica revela-se de grande importância para a implementação da linguagem de lógica temporal utilizada nas regras TF-ORM.

A linguagem de lógica temporal TF-ORM, composta por operadores, predicados e funções, é plenamente suportada pelo modelo de implementação, através de métodos e funções pré-definidas. A ferramenta de tradução desenvolvida converte as condições TF-ORM para chamadas a estes métodos e funções, permitindo a construção transparente de regras que incluam condições de ativação.

Regras de transição de estado e restrições de integridade TF-ORM consistem um mecanismo adequado para modelar e restringir a evolução do comportamento e a valoração das propriedades das instâncias. A abordagem adotada prioriza a implementação homogênea do mecanismo de regras, utilizando para isto uma hierarquia especial de classes.

Embora não tenha sido dada especial atenção a aspectos de desempenho na busca e execução, o mapeamento das regras TF-ORM, sejam elas de transição de estado ou integridade, para regras E-C-A (evento-condição-ação) permitiu a implementação de um algoritmo naturalmente seletivo. A criação de descritores de classes e papéis, e a definição dos conjuntos de regras junto a estes, limitou a busca de regras somente àquelas pertencentes ao conjunto associado ao objeto ou instância de papel em execução, desde que estejam habilitadas para esta instância em especial.

Adicionalmente, a consideração de que, por tratar-se de um modelo orientado a objetos, a manipulação dos valores das propriedades TF-ORM somente pode ser realizada através da ativação de métodos das classes, permitiu a integração do algoritmo de busca e execução de regras a estes métodos. Mensagens de entrada definidas para instâncias de classes e papéis são referenciadas em regras de transição de estado, logo, sua recepção pode ser responsável pela ativação de um conjunto de regras de transição de estado. Da mesma forma, qualquer método definido em uma classe pode ser responsável pela alteração dos atributos de uma instância, logo, o resultado de sua aplicação deve ser confrontado com as regras de integridade definidas para esta. Com a utilização da ferramenta de tradução, a inserção das chamadas aos métodos de seleção e execução das regras é realizada de forma transparente para o projetista, bastando para este definir o conjunto de regras para as classes e papéis.

O mecanismo de implementação de regras logrou êxito ao transformar o SGBD O_2 em um sistema ativo, capaz de reagir automaticamente (e em cadeia) a alterações nos atributos dos objetos, sem a necessidade da intervenção direta de usuários. O aproveitamento do conceito de transação, suportado por este sistema, permitiu a implementação, embora sem a consideração de aspectos de execução concorrente, das restrições de integridade, onde alterações que causam violação devem ser desfeitas.

Uma deficiência atual da abordagem de implementação reside na carência de mecanismos para verificação da terminação e da confluência dos conjuntos de regras das classes e papéis. Desta forma, a ativação de uma regra poderia levar à ativação em cadeia das demais sem, no entanto, atingir um resultado final. Embora seja possível afirmar que a responsabilidade pela terminação e confluência dos conjuntos de regras seja do projetista, a implementação de algoritmos de verificação constitui um interessante tópico de pesquisas futuras. Da mesma forma, mecanismos para verificação das execuções concorrentes de conjuntos de regras poderiam ser implementados.

Outra deficiência consiste na não implementação de certos aspectos do modelo TF-ORM, em especial das mensagens pré-definidas *forbid_op* e *allow_op*, para, respectivamente, inibir e permitir novamente o envio e/ou recebimento de determinada mensagem. Embora a implementação seja possível, a carência de comandos O_2 que permitam a inibição de atributos e métodos das classes, levou a optar-se pela não implementação. Esta e outras características do modelo de dados O_2 foram consideradas para a não implementação do mecanismo de herança do modelo TF-ORM. Estas deficiências consistem em novos tópicos de pesquisas, a serem explorados futuramente.

Durante o desenvolvimento do estudo de caso, ficou clara a necessidade de um desenvolvimento mais aprimorado da metodologia de análise e projeto de sistemas utilizando TF-ORM, bem como a adequação das ferramentas gráficas propostas à modelagem de requisitos tempo-dependentes. Um tópico para trabalhos futuros estaria relacionado ao desenvolvimento desta metodologia, e de ferramentas CASE capazes de guiar o desenvolvedor de forma gráfica não apenas na especificação de classes e papéis, mas sim no desenvolvimento de aplicações, suportando o ciclo completo de análise e projeto de sistemas.

Tal metodologia deve explorar os recursos para criação de hierarquias de classes TF-ORM e suas ferramentas devem, portanto, permitir a reutilização e a decorrente redefinição das classes.

Tópicos para pesquisas futuras residem, também, na implementação da linguagem de consulta TF-ORM e sua decorrente integração às aplicações construídas sobre especificações TF-ORM.

ANEXO 1 SINTAXE DA LINGUAGEM TF-ORM

A seguir é apresentada a sintaxe da linguagem de especificação TF-ORM (extraída de [EDE 94]), representada através de uma gramática livre do contexto, utilizando uma BNF ("Backus Naur Form"). A notação utilizada é a seguinte:

- As unidades sintáticas da gramática são definidas através de regras de produção;
- As metavariáveis são delimitadas pelos símbolos "<" e ">";
- Palavras-chaves são escritas em negrito;
- Símbolos especiais são delimitados por aspas;
- Alternativas apresentadas no lado direito das produções são separadas pelo símbolo "|";
- Parâmetros opcionais são delimitados pelos símbolos "[" e "]";
- Padrões que podem repetir-se zero ou mais vezes são delimitados pelos símbolos "{" e "}*";
- Padrões que podem repetir-se uma ou mais vezes são delimitados pelos símbolos "{" e "}+";
- Comentários são delimitados pelos símbolos "/*" e "*/".

<class declaration> ::=
<process class declaration>
| <resouce class declaration>
| <agent class declaration>
<process class declaration> ::= **process class** <process class definition>
<resource class declaration> ::= **resource class** <resource class definition>
<agent class declaration> ::= **agent class** <agent class definition>
<process class definition> ::=
"(" <class name> "," <base-role declaration>
{"," <process role declaration> }* ")"
| "(" <class name> <specialization declaration> ","
[<disabled roles declaration> ","] [<extended role declaration> ","]
<base-role declaration> { "," <process role declaration>}* ")"
<resource class definition> ::=
"(" <class name> "," <base-role declaration>
{ "," <resource role declaration> }* ")"
| "(" <class name> <specialization declaration> ","
[<inherited roles declarations> ","]
[<disabled roles declaration> ","] [<extended role declaration> ","]
<base-role declaration> { "," <resource role declaration> }* ")"
| "(" <class name> <component declaration> "," [<disabled roles declaration> ","]
< base-role declaration> { "," <resource role declaration> }* ")"
<agent class definition> ::=
"(" <class name> "," <base-role declaration>
{ "," <agent role declaration> }* ")"
| "(" <class name> <specialization declaration> ","
[<inherited roles declaration> ","]
[<disabled roles declaration> ","] [<extended role declaration> ","]
<base-role declaration> { "," <agent role declaration> }* ")"

```
<class name> ::= <identifier>
<identifier> ::= <letter> { <identifier character> }*
<identifier character> ::= <letter> | <digit> | "_"
<specialization declaration> ::=
        is_a <class name>
      | is_a "(" <class name> { "," <class name> }+ ")"
<inherited roles declaration> ::=
        inherits [<class name> "." ] <role name>
      | inherits  "(" [<class name> "." ] <role name>
              { "," [ <class name> "." ] <role name> }* ")"
<disabled roles declaration> ::=
        not_inherits [ <class name> "." ] <role name>
      | not_inherits "(" [ <class name> "." ] <role name>
              { "," [ <class name> "." ] <role name> }* ")"
<extended role declaration> ::=
        extends [ <class name> "." ] <role name>
      | extends "(" [ <class name> "." ] <role name>
              { "," [ <class name> "." ] <role name> }* ")"
<component declaration> ::=
        composed_of "{" <class name> { "," <class name> }* "}"
<role name> ::= <identifier>

<base-role declaration> ::=
        "<" base_role  [ "," <static properties declaration> ]
              [ "," <dynamic properties declaration> ] "," <rule declaration> ">"
<process role declaration> ::=
        "<" <process role name>
              [ "," <static properties declaration> ]
              [ "," <dynamic process properties declaration> ]
              < message declaration> "," <state declaration> "," <rule declaration> ">"
<resource role declaration> ::=
        "<" <resource role name>
              [ "," <static properties declaration> ]
              [ "," <dynamic properties declaration> ]
              < message declaration> "," <state declaration> "," <rule declaration> ">"
<agent role declaration> ::=
        "<" <agent role name>
              [ "," <static properties declaration> ]
              [ "," <dynamic properties declaration> ]
              [ "," <decision declaration> ]
              < message declaration> "," <state declaration> "," <agent rule declaration> ">"

<process role name> ::= <agent name> "." <activity name> | <activity name>
<agent name> ::= [ <class name> "." ] <agent role name>
<activity name> ::= <identifier>
<resource role name> ::= <identifier>
<agent role name> ::= <identifier>

<static properties declaration> ::= static properties "=" "{" [ <property list> ] "}"
<dynamic process properties declaration> ::=
        dynamic properties "=" "{"
              [ executing agent "," <class name> "," ]
              [ message senders "," "{" <sender list> "}" "," ]
```

```
            [ message receivers "," "{" <sender list> "}" "," ]
            [ <property list> ] "}"
<sender list> ::= <sender> | <sender> { "," <sender> }
<sender> ::= <class name> | <class name> "." <role name> | <role name>
<dynamic properties declaration> ::= dynamic properties "=" "{" <property list> "}"
<property list> ::= "(" <property name> "," <property domain> ")" [ "," <property list> ]
<property name> ::= <identifier>
<property domain> ::= <simple domain> | <complex domain>
<simple domain> ::=
        <class name> [ "." <role name> ] | <role name>
      | <predefined domain>
      | "{" <string list> "}"
<predefined domain> ::=
        integer | real | boolean | string | text | place | title | image
      | <temporal point type> | <interval> | <span> | <limit>
<temporal point type> ::=
        instant | date | time | year | month | day | hour | minute
      | week | semester | century | weekday
<interval> ::= interval "(" <interval type> "," <interval limits> ")"
<interval type> ::= closed | open | open_down | open_up | floating_down | floating_up
<interval limits> ::= instant | date | time | year | month | day | hour | minute
<span> ::= span "(" <span type> ")"
<span type> ::= year | month | day | hour | minute| week | semester | century
<limit> ::= after "(" <limit type> ")" | before "(" <limit type> ")"
<limit type> ::= instant | date | time | year | month | day | hour | minute
<string list> ::= <identifier> [ "," <string list> ]
<complex domain> ::=
      "{" <simple domain> "}" | "{" <property list> "}" | "(" <property list> ")" |
      set_of <simple domain> | list_of <simple domain>

<decision declaration> ::=
        decisions "=" "{" <decision definition> { "," <decision definition> }* "}"
<decision definition> ::= <decision>
<decision> ::= <decision name> [ "(" <decision parameters declaration> ")" ]
<decision name> ::= <identifier>
<decision parameters declaration> ::=
        <parameter declaration>{ "," <parameter declaration> }*
            [ "," valid_time ":" <date value> ]
<parameter declaration> ::= <parameter name> ":" <property domain> | <property name>
<parameter name>::= <identifier>

<message declaration > ::=
        messages "=" "{" <message definition> { "," <message definition> }* "}"
<message definition> ::=
        <message> to <roles> [ with <roles> ]
      | <message> to EXTERNAL_WORLD [ with <roles> ]
      | <message> to itself [ with <roles> ]
      | <message> from <roles>
      | <message> from EXTERNAL_WORLD
<message> ::= <message name> [ "(" <message parameters declaration> ")" ]
<message name> ::= <identifier>
<message parameters declaration> ::=
        <parameter declaration> { "," <parameter declaration> }*
            [ "," valid_time ":" <date value> ]
```

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

```
<roles> ::= <role> | "{" <role> { "," <role> }* "}"
<role> ::= [ <class name> "."] <role name>

<state declaration> ::= states "=" "{" [ <state> { "," <state> }* ] "}"
<state> ::= <state name> | "(" <state name> "," <state name> { "," <state name> }* ")"
<state name> ::= <identifier>

<rule declaration> ::=
        rules "=" "{" [ <rule name> ":" <rule> { "," <rule name> ":" <rule> }* ] "}"
<rule name> ::= <identifier>
<rule> := <state transition rule> | <integrity rule>
<agent rule declaration> ::=
        rules "=" "{" [ <rule name> ":" <agent rule> { "," <rule name> ":" <agent rule> }* ] "}"
<agent rule> := <agent state transition rule> | <integrity rule>
<state transition rule> ::=
        <left predicate> "⇒" <right predicate> [ <state transition condition> ]
<left predicate> ::=
        <state predicate> [ "," <message predicate in> ]
      | <message predicate in>
<agent state transition rule> ::=
        <agent left predicate> "⇒" <right predicate> [ <state transition condition> ]
<agent left predicate> ::=
        <state predicate> [ "," <message predicate in> ]
      | <message predicate in>
      | <state predicate> [ "," <decision predicate> ]
      | <decision predicate>
<right predicate> ::=
        <message predicate out> [ "," <right predicate> ]
      | <state predicate>
<state predicate> ::= <defined state predicate> | <predefined state predicate>
<defined state predicate> ::= state "(" [ <id variable> "," ] <state name> ")"
<predefined state predicate> ::= state "(" [ <oid variable> "," ] <predefined state> ")"
<predefined state> ::= active | suspended
<message predicate in> ::= msg "(" "←" <message in> ")"
<message in> ::=
        <message name> [ "(" <message parameters> ")" ]
      | <predefined message predicate>
<decision predicate> ::=
        decision "(" <decision name> [ "(" <message parameters> ")" ] ")"
<message parameters> ::= <parameter> { "," <parameter> }*
<parameter> ::= <parameter name>

<predefined message predicate> ::=
        create_object [ "(" <class name> "," < oid variable> ")" ]
      | suspend_object [ "(" <oid variable> ")" ]
      | resume_object [ "(" <oid variable> ")" ]
      | kill [ "(" <oid variable> ")" ]
      | kill "(" itself ")"
      | add_role [ "(" [ <oid variable> "," ] <role> [","<role variable> ] ")" ]
      | suspend_role [ "(" <role variable> ")" ]
      | resume_role [ "(" <role variable> ")" ]
      | terminate_role [ "(" <role variable> ")" ]
      | forbid_role [ "(" [ <role variable> "," ] <role> ")" ]
      | allow_role [ "(" [ <role variable> "," ] <role> ")" ]
```

| **forbid_op** "(" [<role variable> ","] <direction> <message name> ")"
| **allow_op** "(" <role variable> "," <direction> <message name> ")"
| **forget** "(" [<oid variable> ","] <rule name> ")"
| **recall** "(" [<oid variable> ","] <rule name> ")"
| **start** ["(" [<oid variable> ","] <role> ["," <role variable>] ")"]
| **stop** ["(" [<oid variable> ","] <role> ["," <role variable>] ")"]
| **in_class** "(" <class name> ")"
| **out_class** "(" <class name> ")"
<direction> ::= "←" | "→"

<message predicate out> ::= **msg** "(" →" <message out> ")"
<message out> ::=
<message name> ["(" <message parameters> ")"]
| <predefined message predicate> [**to** <receiver>]
<receiver> ::= [<role variable> ":"] <role> | **itself**
<oid variable> ::= <variable>
<role variable> ::= <variable>
<id variable> ::= <oid variable> | <role variable>
<state transition condition> ::= ";" "(" <logical expression> ")"

<integrity rule> ::= **constraint** "(" <integrity condition declaration> ")"
<integrity condition declaration> ::=
<simple integrity condition> | <instanciated integrity condition>
<simple integrity condition> ::= <logical expression> "⇒" <logical expression>
<instanciated integrity condition> ::=
[<temporal operator>] <quantifier> <variable> "(" <integrity condition declaration> ")"

<logical expression> ::=
<logical term> | <logical expression> <or operator> <logical term>
<logical term> ::=
<logical factor> | <logical term> <and operator> <logical factor>
<logical factor> ::= <logical element> | **not** <logical element>
<logical element> ::=
<predicate> | "(" <logical expression> ")"
| <logical element> <temporal logical operator> <predicate>
<or operator> ::= **or** | ";"
<and operator> ::= **and** | ","
<temporal logical operator> ::= **since** | **until** | **before** | **after**
<predicate> ::=
<predefined predicate>
| <predefined temporal predicate>
| <state predicate>
| <predefined state predicate>
| [<temporal operator>] <quantifier> <variable> "(" <predicate> ")"
| <arit expression> <comp operator> <arit expression>
| <temporal operator> <logical expression>
| **false**
| **true**
<predefined predicate> ::=
has_class_instance "(" <class name> "," <oid variable> ")"
| **has_role_instance** "(" <oid variable> "," <role name> "," <role variable> ")"
| **active_class** "(" <oid variable> ")"
| **active_role** "(" <role variable> ")"
| **active_class_at** "(" <oid variable> ","<temporal instant> ")"

```
        | active_role_at "(" <role variable> "," <temporal instant> ")"
        | is_valid "(" <id variable> "," <property name> ")"
        | is_valid_at "(" <id variable> "," <property name> "," <temporal instant> ")"
        | out_role "(" <oid variable> "," role name> ")"
        | role "(" <oid variable> "," <role variable> ")"
<predefined temporal predicate> ::=
        belongs "(" <function argument> "," <function name argument> ")"
        | contains "(" <function name argument> "," <function argument> ")"
        | before "(" <function argument> "," <function argument> ")"
        | equal "(" <function argument> "," <function argument> ")"
<function argument> ::=
        [ <id variable> "," ] <property name> | <temporal instant> | <variable>
<function name argument> ::=
        [ <id variable> "," ] <property name> | <variable>
<temporal instant> ::= <number> "/" <number> "/" <number> ","
<number> ":" <number>
<temporal operator> ::=
        sometime past | immediately past | always past | sometime future
        | immediately future | always future
<quantifier> ::= exists | forall
<comp operator> ::= "<" | ">" | "=" | "≤" | "≥" | "≠"
<arit expression> ::=
        <term> | "-" <arit expression> | <arit expression> "+" <term>
        | <arit expression> "-" <term>
<term> ::= <factor> | <term> "*" <factor> | <term> "/" <factor>
<factor> ::=
        <element> | <factor> "**" <element>
        | <element> union <element>
        | <element> intersection <element>
<element> ::=
        [ <id variable> "," ] <property name>
        | [ <id varible> ","] <predefined property name>
        | <function> | <value> | <variable> | "(" <arit expression> ")" | now
<predefined property name> ::=
        oId | object_instance | end_object
        | rId | role_instance | end_role
<function> ::=
        year "(" <function argument> ")"
        | month "(" <function argument> ")"
        | day "(" <function argument> ")"
        | hour "(" <function argument> ")"
        | minute "(" <function argument> ")"
        | weekday "(" <function argument> ")"
        | lower_bound "(" <function name argument> ")"
        | upper_bound "(" <function name argument> ")"
        | duration "(" <function name argument> ")
        | interval "(" <function name argument> "," <function name argument> ")"
        | to_minutes "(" <function argument> ")"
        | to_months "(" <function argument> ")"
        | to_days "(" <function argument> ")"
        | value "(" [ <id variable> "," ] <property name> ")"
        | past_value "(" [ <id variable> "," ] <property name> <temporal instant> ")"
        | valid_time "(" [ <id variable> "," ] <property name> ")"
        | transaction_time "(" [ <id variable> ","] <property name> ")"
```

| ***class_creation_time*** *"(" <oid variable> ")"*
| ***role_creation_time*** *"(" <role variable> ")"*
| ***class_end_time*** *"(" <oid variable> ")"*
| ***role_end****_time "(" <role variable> ")"*
| ***state*** *"(" <id variable> ")"*
| *state_at "(" <id variable> <temporal instant> ")"*

<value> ::= <integer number> | <string> | <temporal value> | ***null*** | ***nonull***
*<integer number> ::= <digit> { <digit> }**
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<string> ::= """ { <any character including blanck> }+ """
<temporal value> ::= <temporal instant> | <date value> | <hour value>
<date value> ::= <number> "/" <number> "/" <number>
<hour value> ::= <number> ":" <number>
<number> ::= <digit> <digit>
<variable> ::= <identifier>
*<identifier> ::= <letter> { <letter> | <digit> | "_" }**
<letter> ::=

A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b| c | d | e | f | g | h | i | j | k | l | m | n
| o | p | q | r | s | t | u | v | w | x | y | z

ANEXO 2 SINTAXE RESUMIDA DOS COMANDOS O_2

A seguir é apresentada uma sintaxe resumida dos principais comandos da linguagem de definição de dados (DDL) do modelo O_2 (extraída de [O2T 91a]). A representação da sintaxe segue a notação adotada no ANEXO 1 Sintaxe da Linguagem TF-ORM. A linguagem de manipulação de dados (DML) O_2C, juntamente com a linguagem de consulta O_2Query, não são apresentadas, no entanto, uma completa explanação sobre estas pode ser encontrada em [O2T 91b].

ANEXO 2.1 Definição de Classes

<class definition> ::=
 class *<class name>*
 [<specialization declaration>]
 [<type specification>]
 [<method declaration>]
 end

<specialization declaration> ::=
 inherit *<class name> { "," <class name> }* [<properties conflicts>]*
<properties conflicts> ::=
 rename *<property name>* **as** *<property name> { "," <property name>* **as** *<property name>}**

<type specification> ::= <simple type> | <complex type>
<simple type> ::= **integer** | **real** | **char** | **string** | **boolean** | **bits** | *<class name>*
<complex type> ::= <set type> | <list type> | <tuple type >
*<set type> ::=[***unique***]* **set** *"(" <type specification> ")"*
<list type> ::= **list** *"(" <type specification> ")"*
<tuple type> ::= **tuple** *"(" <attribute declaration> ")"*
<attribute declaration> ::=
 *<attribute name> ":" <type specification> { "," <attribute name> ":" <type specification> }**

<method declaration> ::= **method** *<method definition> { "," <method definition> }**
<method definition> ::=
 [**public** | **private** | **read** | **write** *] <method name>*
 ["(" < signature> ")"]
 [":" <type specification>]
<signature> ::= <attribute declaration>

ANEXO 2.2 Definição de *Named-Objects* e *Named-Values*

<named-item declaration> ::=
 [**constant** *]* **name** *<named-item name> [":" <type specification>]*

ANEXO 2.3 Importação de Classes

<import declaration> ::=
 import *[* **schema** *<schema name>]*
 [**class** *<class name> { "," <class name> }*]*
 [**name** *<named-item name] { "," <named-item name> }*]*

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

ANEXO 2.4 Definição de Aplicações, Programas, Transações e Funções

<application declaration> ::=
application *<application name>*
[<variable definition>]
[<program declaration>]
end

<variable definition> ::=
[**variable** *<variable name> ":" <type specification>*
*{ "," **variable** <variable name> ":" <type specification> }**

<program declaration> ::= **program** *<program definition> { "," <program definition> }**
<program definition> ::=
[**public** | **private** | **read** | **write** *] <program name> ["(" < signature> ")"]*

ANEXO 2.5 Definição de Código O_2C

<method body declaration> ::=
method body *<method name> [<signature>]*
[**in class** *<class name>]*
<O_2C-code>

<program body declaration> ::=
program body *<program name> [<signature>]*
[**in application** *<application name>]*
<O_2C-code>

<transaction body declaration> ::=
transaction body *<transaction name> [<signature>]*
[**in application** *<application name>]*
<O_2C-code>

<function body declaration> ::=
function body *<function name> [<signature>]*
<O_2C-code>

<O_2C-code> ::= / Código O_2C */*

ANEXO 2.6 Definições Gerais

*<class name> ::= <upper-case letter> { <letter> | <underscore> }**
<property name> ::= <identifier>
<attribute name> ::= <identifier>
<method name> ::= <identifier>
<named-item name> ::= <identifier>

*<identifier> ::= <lower-case letter> { <lower-case letter> | <underscore> }**
<letter> ::= <upper-case letter> | <lower-case letter>
<upper-case letter> ::=
A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<lower-case letter> ::= a | b| c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<underscore> ::= "_"

ANEXO 3 ESTUDO DE CASO: CÓDIGO TF-ORM

ANEXO 3.1 Classes Recurso

ANEXO 3.1.1 Classe Documento

```
resource class (
        Documento,
        <base_role,
                static properties = {(Número, integer), (Emissão, instant)},
                dynamic properties = {(Responsável, Funcionário)},
                rules = {Criação_do_Documento: msg( ←create_object) ⇒ state( active)} >,
        <Pedido_de_Cliente,
                static properties = {(Data_de_Confirmação, date), (Cliente, Pessoa.Cliente)},
                dynamic properties = {(Produto, Item_de_Estoque.Produto), (Quantidade, integer),
                        (Data_de_Entrega, date) },
                states = {Aguardando_Atendimento, Encerrado},
                messages = {Encerrado from Nota_Fiscal},
                rules = {
                        Aguardar_Atendimento: msg( ←add_role) ⇒ state(
                                Aguardando_Atendimento),
                        Encerramento: state( Aguardando_Atendimento), msg( ←Encerrado) ⇒ state(
                                Encerrado) } >,
        <Ordem_de_Serviço,
                dynamic properties = {(Estágio, integer), (Intervalo_de_Produção, interval(closed,
                        instant))},
                states = {Iniciando_Produção, Aguardando_Preparação, Aguardando_Montagem,
                        Aguardando_Teste, Encerrada},
                messages = {Modificar_Estágio from
                        Administração_de_Produção.Linha_de_Produção, Produção_Encerrada from
                        Administração_de_Produção.Linha_de_Produção},
                rules = {
                        Iniciar_Produção: msg( ←add_role) ⇒ state( Iniciando_Produção),
                        Iniciar_Preparação: state( Iniciando_Produção), msg( ←Modificar_Estágio)
                                ⇒ state( Aguardando_Preparação),
                        Iniciar_Montagem: state( Aguardando_Preparação), msg(
                                ←Modificar_Estágio) ⇒ state( Aguardando_Montagem),
                        Iniciar_Teste: state( Aguardando_Montagem), msg( ←Modificar_Estágio) ⇒
                                state( Aguardando_Teste),
                        Encerrar: state( Aguardando_Teste), msg( ←Produção_Encerrada) ⇒ state(
                                Encerrada),
                        constraint( is valid( Estágio) ⇒ immediately past Estágio < Estágio } >,
        <Ordem_de_Compra,
                static properties = {(Instante_de_Emissão, instant), (Intervalo_de_Espera,
                        interval(open_down, date)), (Insumo, Item_de_Estoque.Insumo)},
                dynamic properties = {(Fornecedor, Fornecedor), (Data_Prevista, date), (Quantidade,
                        integer), (Valor, real)},
                states = {Aguardando_Recebimento, Indicando_Recebimento, Encerrada},
                messages = {Emitir_Ordem_de_Compra to Fornecedor, Recebimento from
                        Fornecedor, Recebimento_Efetuado to
                        Administração_de_Compras.Controle_de_Compra, Registrar_Compra to
                        Item_de_Estoque.Insumo},
                rules = {
```

Emissão_de_Ordem_de_Compra: msg(←add_role) ⇒ state(Aguardando_ Recebimento),
Indicação_de_Recebimento: state(Aguardando_Recebimento), msg(←Recebimento) ⇒ msg(→Recebimento_ Efetuado), state(Indicando_ Recebimento),
Encerramento_da_Compra: state(Indicando_Recebimento) ⇒ msg(→Registrar_ Compra), state(Encerrada) } >,
<Nota_Fiscal,
static properties = {(Data_de_Emissão, date)},
dynamic properties = {(Valor, real)},
states = {Registrando_Venda, Encerrando, Emitida},
messages = {Registrar_Venda to Pessoa.Cliente, Encerrado to Pedido_de_Cliente, Emitir_Nota_Fiscal to itself},
rules = {
Registro_de_Venda: msg(←add_role) ⇒ msg(→Registrar_Venda), state(Registrando_Venda),
Encerramento_do_Pedido: state(Registrando_Venda) ⇒ msg(→Encerrado), state(Encerrando),
Emissão_de_Nota_Fiscal: state(Encerrando) ⇒ msg(→Emitir_Nota_Fiscal), state(Emitida)} >}

ANEXO 3.1.2 Classe Item de Estoque

resource class (
Item_de_Estoque,
<base_role,
static properties = {(Código_da_Peça, integer), (Descrição, text) },
dynamic properties = {(Valor, real), (Composição, set_of(Item_de_Estoque.Insumo), (Quantidade, integer)},
messages = { Cadastrar_Produto from external_world, Cadastrar_Insumo from external_ world, Alterar_Quantidade from Administração_de_Materiais.Controle_de_ Estoque, Verificar_Quantidade from Administração_de_Materiais.Controle_de_ Estoque, Quantidade_Suficiente to Administração_de_Materiais.Controle_de_ Estoque, Quantidade_Insuficiente to Administração_de_Materiais.Controle_de_ Estoque},
rules = {
Criação_do_Item_de_Estoque: msg(←create_object) ⇒ state(active),
Cadastro_de_Produto: state(active), msg (←Cadastrar_Produto) ⇒ msg(→add_role(Produto),
Cadastro_de_Insumo: state(active), msg (←Cadastrar_Insumo) ⇒ msg(→add_role(Insumo),
Alteração_de_Quantidade: state(active), msg(←Alterar_Quantidade) ⇒ state(active),
Quantidade_OK: state(active), msg(←Verificar_Quantidade(Q)) ⇒ msg(→Quantidade_Suficiente); Q <= Quantidade,
Quantidade_NOK: state(active), msg(←Verificar_Quantidade(Q)) ⇒ msg(→Quantidade_Insuficiente); Q >Quantidade } >,
<Produto,
dynamic properties = {(Nome_Fantasia, string), (Custo, real), (Ciclo_de_Produção, duration(hour)},
states = {Ativo},
rules = { Criação_do_Produto: msg(←add_role) ⇒ state(Ativo) } >,
<Insumo,
dynamic properties = {(Fornecedores, set_of(Fornecedor)), (Valor, Real), (Fornecedor, Fornecedor)},
states = {Ativo},

*rules = {**Criação_do_Produto**: msg(←add_role) ⇒ state(**Ativo**) >}*

ANEXO 3.2 Classes Processo

ANEXO 3.2.1 Classe Administração de Vendas

process class (
***Administração_de_Vendas**,*
<base_role,
*dynamic properties = {(**Responsável**, **Pessoa.Funcionário**), (**Data_de_Posse**, date)},*
*messages = {**Pedido_de_Cliente** from **Pessoa.Cliente**},*
rules = {
***Início_das_Vendas**: msg(←create_object) ⇒ state(active),*
***Criação_do_Pedido_de_Cliente**: state(active), msg(←**Pedido_de_Cliente**) ⇒ msg(→add_role(**Controle_de_Pedido**))} >*
*<**Controle_de_Pedido**,*
*static properties = {(**Documento**, **Documento**), (**Instante_de_Início**, instant)},*
*states = {**Criando_Documentação**, **Criando_Pedido**, **Checando_Pedido**, **Alterando_Estoque**, **Separando_Insumos**, **Aguardando_Produção**, **Encerrando**},*
*messages = {**Verificar_Estoque_de_Produto** to **Administração_de_Materiais.Controle_de_Estoque**, **Liberar_Pedido** from **Administração_de_Materiais.Controle_de_Estoque**, **Alterar_Estoque** to **Administração_de_Materiais.Controle_de_Estoque**, **Produzir_Pedido** from **Administração_de_Materiais.Controle_de_Estoque**, **Separar_Insumos** to **Administração_de_Materiais.Gerência_de_Almoxarifado**, **Separação_OK** from **Administração_de_Materiais.Gerência_de_Almoxarifado**, **Emitir_Ordem_de_Serviço** to **Administração_de_Produção**, **Produção_Encerrada** from **Administração_de_Produção.Linha_de_Produção**},*
rules = {
***Criação_da_Documentação**: msg(←add_role) ⇒ msg(→create_object(**Documento**)), state(**Criando_Documentação**),*
***Criação_do_Pedido_de_Cliente**: state(**Criando_Documentação**) ⇒ msg(→add_role(**Documento.Pedido_de_Cliente**)), state(**Criando_Pedido**),*
***Checagem_do_Pedido**: state(**Criando_Pedido**) ⇒ msg(→**Verificar_Estoque_de_Produto**), state(**Checando_Pedido**),*
***Alteração_de_Estoque**: state(**Checando_Pedido**), msg(←**Liberar_Pedido**) ⇒ msg(→**Alterar_Estoque**), state(**Alterando_Estoque**),*
***Encerramento_via_Estoque**: state(**Alterando_Estoque**) ⇒ msg(→add_role(**Documento.Nota_Fiscal**), state(**Encerrando**),*
***Separação_de_Insumos**: state(**Checando_Pedido**), msg(←**Produzir_Pedido**) ⇒ msg(→**Separar_Insumos**), state(**Separando_Insumos**),*
***Ativação_da_Produção**: state(**Separando_Insumos**), msg(←**Separação_OK**) ⇒ msg(→**Emitir_Ordem_de_Serviço**), state(**Aguardando_Produção**),*
***Encerramento_via_Produção**: state(**Aguardando_Produção**), msg(← **Produção_Encerrada**) ⇒ msg(→add_role(**Documento.Nota_Fiscal**), state(**Encerrando**),*
***Encerramento**: state(**Encerrando**) ⇒ msg(→terminate_role(itself))} > }*

ANEXO 3.2.2 Classe Administração de Produção

process class (
***Administração_de_Produção**,*
<base_role,
*dynamic properties = {(**Ocupação**, integer), (**Limite**, integer), (**Responsável**, **Pessoa.Funcionário**), (**Data_de_Posse**, date)},*

messages = {Emitir_Ordem_de_Serviço from Administração_de_Vendas.
Controle_de_Pedido, Aguardar_Desocupação to Pessoa.Funcionário},
rules = {
Início_da_Produção: msg(←create_object) ⇒ state(active),
Verificação_de_Ocupação: state(active), msg(←Emitir_Ordem_de_Serviço)
⇒ msg(→Aguardar_Desocupação);Ocupação >= Limite,
Criação_da_Linha_de_Produção: state(active) ⇒ msg(→add_role(
Linha_de_Produção);(immediately past Ocupação >= Limite) and
(Ocupação < Limite),
constraint(is_valid(Limite) ⇒ (Limite >= 0) and (Limite <= 100)} >
<Linha_de_Produção,
static properties = {(Documento, Documento)},
dynamic properties = {(Responsável, Pessoa.Funcionário)},
states = {Emitindo_Ordem_de_Serviço, Acompanhando_Produção, Encerrando_
Produção},
messages = {Alocar to Pessoa.Funcionário, Estágio_Encerrado from Pessoa.
Funcionário, Produção_Encerrada from Pessoa.Funcionário,
Modificar_Estágio to Documento.Ordem_de_Serviço, Produção_Encerrada to
Administração_de_ Vendas.Controle_de_Pedido},
rules = {
Emissão_de_Ordem_de_Serviço: msg(←add_role) ⇒ msg(→add_role(
Documento.Ordem_de_Serviço)), state(Emitindo_Ordem_de_Serviço),
Alocação_de_Funcionário: state(Emitindo_Ordem_de_Serviço) ⇒ msg(
→add_role(Alocar)), state(Acompanhando_Produção),
Mudança_de_Estágio: state(Acompanhando_Produção), msg(←Estágio_
Encerrado) ⇒ msg(→Modificar_Estágio),
Encerramento_Produção: state(Acompanhando_Produção), msg(
←Produção_Encerrada) ⇒ msg(→Produção_Encerrada), state(
Encerrando_Produção),
Final: state(Encerrando_Produção) ⇒ msg(→terminate_role(itself))} > }

ANEXO 3.2.3 Classe Administração de Materiais

process class (
Administração_de_Materiais,
<base_role,
dynamic properties = {(Responsável, Pessoa.Funcionário), (Data_de_Posse, date)},
rules = {
Criação_do_Almoxarifado: msg(←create_object) ⇒ state(active),
Início_do_Controle_de_Estoque: state(active) ⇒ msg(
←add_role(Controle_de_ Estoque)),
Início_da_Gerência_de_Materiais: state(active) ⇒ msg(
←add_role(Gerência_ de_Materiais)) } >,
<Gerência_de_Almoxarifado,
dynamic properties = {(Ocupação, integer)},
states = {Disponível, Verificando_Estoque, Aguardando_Compra, Separando},
messages = {Separar_Insumos from Administração_de_Vendas.Controle_de_Pedido,
Verificar_Estoque to Controle_de_Estoque, Liberar_Separação from Controle_
de_Estoque, Alterar_Estoque to Controle_de_Estoque, Emitir_Solicitação_de_
Compra from Controle_de_Estoque, Efetuar_Solicitação_de_Compra to
Administração_de_ Compras, Recebimento_Efetuado from Administração_
de_Compras.Controle de Compra, Separação_OK to
Administração_de_Vendas. Controle_de_Pedido},
rules = {
Criação_da_Gerência: msg(←add_role) ⇒ state(Disponível),
Separação_de_Insumos: state(Disponível), msg(←Separar_Insumos) ⇒ msg(
→Verificar_Estoque), state(Verificando_Estoque),

```
                Ordem_de_Separação: state( Verificando_Estoque), msg(
                        ←Liberar_Separação) ⇒ msg( →Alterar_Estoque), state( Separando),
                Emissão_de_Solicitação_de_Compra: state( Verificando_Estoque), msg(
                        ←Emitir_Solicitação_de_Compra) ⇒ msg(
                        →Efetuar_Solicitação_de_Compra), state( Aguardando_Compra),
                Separação_Após_Compra: state( Aguardando_Compra), msg(
                        ←Recebimento_Efetuado) ⇒ msg( →Alterar_Estoque),
                        state(Separando),
                Aviso_de_Separação: state( Separando) ⇒ msg( →Separação_OK),
                        state(Disponível),
                constraint( is_valid( Ocupação) ⇒ (Ocupação >= 0) and (Ocupação <= 100)}
                        >,
        <Controle_de_Estoque,
                dynamic properties = {(Data_de_Verificação, date)},
                states = {Disponível, Verificando_Estoque_para_Pedido, Verificando_Estoque_para_
                        Produção },
                messages = { Alterar_Estoque from Gerência_de_Almoxarifado, Alterar_Estoque
                        from Administração_de_Vendas.Controle_de_Pedido, Alterar_Quantidade to
                        Item_de_Estoque, Verificar_Estoque_de_Produto from Administração_de_
                        Vendas.Controle_de_Pedido, Verificar_Estoque_de_Insumo from Gerência_
                        de_Almoxarifado, Liberar_Pedido to Administração_de_Vendas.Controle_de_
                        Pedido, Liberar_Separação to Gerência_de_Almoxarifado, Produzir_Pedido to
                        Administração_de_Vendas.Controle_de_Pedido, Efetuar_Solicitação_de_
                        Compra to Gerência_de_Almoxarifado, Verificar_Quantidade to Item_de_
                        Estoque, Quantidade_Suficiente from Item_de_Estoque, Quantidade_
                        Insuficiente from Item_de_Estoque},
                rules = {
                        Início_do_Controle: msg( ←add_role) ⇒ state( Disponível),
                        Alteração_de_Estoque: state( Disponível), msg( ←Alterar_Estoque) ⇒ msg(
                                →Alterar_Quantidade),
                        Verificação_de_Estoque_Pedido: state( Disponível), msg(
                                ←Verificar_Estoque_de_Produto) ⇒ msg( →Verificar_Quantidade),
                                state( Verificando_Estoque_para_Pedido),
                        Verificação_de_Estoque_Produção: state( Disponível), msg(
                                ←Verificar_Estoque_de_Insumo) ⇒ msg( →Verificar_Quantidade),
                                state( Verificando_Estoque_para_Produção),
                        Liberação_de_Pedido: state( Verificando_Estoque_para_Pedido), msg(
                                ←Quantidade_Suficiente) ⇒ msg( →Liberar_Pedido), state(
                                Disponível),
                        Liberação_de_Separação: state( Verificando_Estoque_para_Produção), msg(
                                ←Quantidade_Suficiente) ⇒ msg( →Liberar_Separação), state(
                                Disponível),
                        Produção_de_Pedido: state( Verificando_Estoque_para_Pedido), msg(
                                ←Quantidade_Insuficiente) ⇒ msg( →Produzir_Pedido), state(
                                Disponível),
                        Solicitação_de_Compra: state( Verificando_Estoque_para_Produção), msg(
                                ←Quantidade_Insuficiente) ⇒ msg(
                                →Efetuar_Solicitação_de_Compra), state( Disponível)} > }
```

ANEXO 3.2.4 Classe Administração de Compras

```
process class (
        Administração_de_Compras,
        <base_role,
                dynamic properties = {(Responsável, Pessoa.Funcionário), (Data_de_Posse, date)},
                messages = {Efetuar_Solicitação_de_Compra from Administração_de_Materiais.
                        Gerência_de_Almoxarifado},
                rules = {
```

Criação_do_Setor_de_Compras: msg(←create_object) ⇒ state(active),
Acompanhamento_da_Compra: state(active), msg(←Efetuar_Solicitação_de_Compra) ⇒ msg(→add_role(Controle_de_Compra))} >,
<Controle_de_Compra,
static properties = {(Documento, Documento)},
dynamic properties = {(Responsável, Pessoa.Funcionário)},
states = {Aguardando_Recebimento, Encerrando_Compra},
messages = {Recebimento_Efetuado from Documento.Ordem_de_Compra, Recebimento_Efetuado to Administração_de_Materiais.Gerência_de_Almoxarifado},
rules = {
Criação_da_Ordem_de_Compra: msg(←add_role) ⇒ msg(→add_role(Documento.Ordem_de_Compra)), state(Aguardando_Recebimento),
Comunicação_de_Recebimento: state(Aguardando_Recebimento), msg(←Recebimento_Efetuado) ⇒ msg(→Recebimento_Efetuado), state(Encerrando_Compra),
Encerramento: state(Encerrando_Compra) ⇒ msg(→terminate_role(itself))} >
}

ANEXO 3.3 Classes Agente

ANEXO 3.3.1 Classe Pessoa

agent class (
Pessoa,
<base_role,
static properties = {(CPF, string), (Data_de_Nascimento, date)},
dynamic properties = {(Nome, string), (Endereço, string)},
messages = {Cadastrar_Cliente from external_world, Cadastrar_Funcionário from external_world},
rules = {
Criação_da_Pessoa: msg(←create_object) ⇒ state(active),
Cadastro_de_Cliente: state(active), msg(←Cadastrar_Cliente) ⇒ msg(→add_role(Cliente)),
Cadastro_de_Funcionário: state(active), msg(←Cadastrar_Funcionário) ⇒ msg(→add_role(Funcionário))} >,
<Cliente,
dynamic properties = {(Último_Documento, Documento)},
states = {Ativo, Aguardando_Atendimento_de_Pedido},
messages = {Pedido from external_world, Pedido_de_Cliente to Administração_de_Vendas, Registrar_Venda from Documento.Nota_Fiscal},
rules = {
Cadastro_do_Cliente: msg(←add_role) ⇒ state(Ativo),
Solicitação_do_Pedido: state(Ativo), msg(←Pedido) ⇒ state(Aguardando_Atendimento_de_Pedido),
Ativação_do_Pedido: state(Aguardando_Atendimento_de_Pedido) ⇒ msg(→Pedido_de_Cliente),
Atendimento_do_Pedido: state(Aguardando_Atendimento_de_Pedido), msg ←Registrar_Venda) ⇒ state(Ativo)} >,
<Funcionário,
dynamic properties = {(Setor, string), (Salário, real), (Número_de_Horas, Duration(hour)),(Atuação, boolean), (Estado, boolean)},
decisions = {Reativar_Fornecedor(fornecedor_suspenso: Fornecedor)},
states = {Aguardando, Produzindo, Encerrando},
messages = {Alocar from Administração_de_Produção, Término_de_Estágio from external_world, Término_da_Produção from external_world,

Estágio_Encerrado to **Administração_de_Produção.Linha_de_Produção**,
Produção_Encerrada to **Administração_de_Produção.Linha_de_Produção**,
Desalocar to itself, **Reativar** to **Fornecedor**},

rules = {

Criação_de_Funcionário: msg(←add_role) ⇒ state(**Aguardando**),
Início_de_Produção: state(**Aguardando**), msg(←**Alocar**) ⇒ state(
Produzindo); **Atuação** = **True**,
Mudança_de_Estágio: state(**Produzindo**), msg(←**Término_de_Estágio**) ⇒
msg(→**Estágio_Encerrado**), state(**Produzindo**),
Término_da_Produção: state(**Produzindo**), msg ←**Término_da_Produção**) ⇒
msg (→**Produção_Encerrada**), state(**Aguardando**)
Reativação_de_Fornecedor: state(**Produzindo**), decision(
Reativar_Fornecedor(fornecedor_suspenso: **Fornecedor**)) ⇒ msg(
→resume_object(**fornecedor_suspenso**: **Fornecedor**)),
constraint(is_valid(**Salário**) ⇒ immediately past **Salário** <= **Salário**} >}

ANEXO 3.3.2 Classe Fornecedor

agent class (
Fornecedor,
<base_role,
static properties = {(**CGC**, string)},
dynamic properties = {(**Nome**, string), (**Endereço**, string), (**Telefone**, string),
(**Índice_de_Qualidade**, integer)},
messages = {**Emitir_Ordem_de_Compra** from **Documento.Ordem_de_Compra**,
Ordem_de_Compra to external_world, **Recebimento** from external_world,
Recebimento to **Documento.Ordem_de_Compra**, **Reativar** from
Pessoa.Funcionário},

rules = {

Criação_do_Fornecedor: msg(←create_object) ⇒ state(active),
Emissão_da_Ordem_de_Compra: state(active), msg(
←**Emitir_Ordem_de_Compra**) ⇒ msg(→**Ordem_de_Compra**),
Comunicação_de_Recebimento: state(active), msg(←**Recebimento**) ⇒ msg(
→**Recebimento**),
Suspensão: state(active) ⇒ msg(→suspend_object(itself));
Índice_de_Qualidade < 100 } >}

ANEXO 4 ESTUDO DE CASO: CÓDIGO O_2

São apresentados exemplos de identificadores únicos e descritores e as estruturas das classes e papéis. Os identificadores de estado e o código das regras não são apresentados. A exemplo do estudo de caso, não é dada especial atenção aos parâmetros das mensagens.

ANEXO 4.1 Classes Recurso

ANEXO 4.1.1 Classe Documento

```
constant name Documento: integer;
run body {Documento = 0;}

class   Descriptor_Documento inherit Descriptor
        type tuple (   instances: unique set (Documento) )
        method
                       public init (name: integer): Descriptor_Documento,

                       public create_object: Documento,
                       public kill (old: Documento): boolean
end;

class   Documento  inherit Object_class
        type tuple (   número: Integer,
                       emissão: Instant,
                       Responsável: Dynamic_Funcionário)
        method
                       public init: Documento
end;

constant name Pedido_de_cliente: integer;
run body {Pedido_de_cliente = 1;}

class   Descriptor_Pedido_de_cliente inherit Descriptor
        method
                       public init (name: integer): Descriptor_Pedido_de_cliente
end;

class   Pedido_de_cliente  inherit Role_class
        type tuple (   data_de_confirmação: Date,
                       cliente: Cliente,
                       produto: Dynamic_Produto,
                       quantidade: Dynamic_Integer,
                       data_de_entrega: Dynamic_Date)
        method
                       public init: Pedido_de_cliente,

                       public encerrado
end;
...
class   Ordem_de_serviço  inherit Role_class
        type tuple (   estágio: Dynamic_Integer,
                       intervalo_de_produção: Dynamic_Instant_interval )
```

```
        method
                    public init: Ordem_de_serviço,

                    public modificar_estágio,
                    public produção_encerrada
end;
...
class   Ordem_de_compra inherit Role_class
        type tuple (  instante_de_emissão: Instant,
                      intervalo_de_espera: Date_interval,
                      insumo: Insumo,
                      fornecedor: Dynamic_Fornecedor,
                      data_prevista: Dynamic_Date,
                      quantidade: Dynamic_Integer,
                      valor: Dynamic_Real )
        method
                    public init: Ordem_de_compra,

                    public recebimento
end;
...
class   Nota_fiscal inherit Role_class
        type tuple (  data_de_emissão: Date,
                      valor: Dynamic_Real )
        method
                    public init: Ordem_de_compra,

                    public emitir_nota_fiscal
end;
```

ANEXO 4.1.2 Classe Item de Estoque

```
class   Item_de_estoque inherit Object_class
        type tuple (  código_da_peça: Integer,
                      descrição: Text,
                      valor: Dynamic_Real,
                      composição: set(Dynamic_Insumo),
                      quantidade: Dynamic_Integer)
        method
                    public init: Item_de_estoque

                    public cadastrar_produto,
                    public cadastrar_insumo,
                    public alterar_quantidade,
                    public verificar_quantidade
end;
...
class   Produto inherit Role_class
        type tuple (  nome_fantasia: Dynamic_String,
                      custo: Dynamic_Real,
                      ciclo_de_produção: Dynamic_Hour_duration)
        method
                    public init: Produto
end;
...
class   Insumo inherit Role_class
        type tuple (  fornecedores: set(Dynamic_Fornecedor),
                      valor: Dynamic_Real,
```

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

```
                    fornecedor: Dynamic_Fornecedor)
          method
                    public init: Insumo
end;
```

ANEXO 4.2 Classes Processo

ANEXO 4.2.1 Classe Administração de Vendas

```
class  Administração_de_vendas inherit Object_class
       type tuple (  responsável: Dynamic_Funcionário,
                     data_da_posse: Dynamic_Date)
       method
                     public init: Administração_de_vendas

                     public pedido_de_cliente
end;
...
class  Controle_de_pedido inherit Role_class
       type tuple (  documento: Documento,
                     instante_de_início: Instant)
       method
                     public init: Controle_de_pedido,

                     public liberar_pedido,
                     public produzir_pedido,
                     public separação_OK,
                     public produção_enecerrada
end;
```

ANEXO 4.2.2 Classe Administração de Produção

```
class  Administração_de_produção inherit Object_class
       type tuple (  limite: Dynamic_Integer,
                     ocupação: Dynamic_Integer,
                     responsável: Dynamic_Funcionário,
                     data_de_posse: Dynamic_Date)
       method
                     public init: Administração_de_produção

                     public emitir_ordem_de_serviço
end;
...
class  Linha_de_produção inherit Role_class
       type tuple (  documento: Documento,
                     responsável: Dynamic_Funcionário)
       method
                     public init: Linha_de_produção,

                     public estágio_encerrado,
                     public produção_encerrada
end;
```

ANEXO 4.2.3 Classe Administração de Materiais

```
class  Administração_de_materiais inherit Object_class
       type tuple (  responsável: Dynamic_Funcionário,
```

```
                    data_de_posse: Dynamic_Date)
        method
                    public init: Administração_de_materiais
end;
...
class   Gerência_de_almoxarifado inherit Role_class
        type tuple (  ocupação: Dynamic_Integer)
        method
                    public init: Gerência_de_almoxarifado,

                    public separar_insumos,
                    public liberar_separação,
                    public emitir_solicitação_de_compra,
                    public recebimento_efetuado
end;
...
class   Controle_de_estoque inherit Role_class
        type tuple (  data_de_verificação: Dynamic_Date)
        method
                    public init: Controle_de_estoque,

                    public alterar_estoque,
                    public verificar_estoque_de_produto,
                    public verificar_estoque_de_insumo,
                    public quantidade_suficiente,
                    public quantidade_insuficiente
end;
```

ANEXO 4.2.4 Classe Administração de Compras

```
class   Administração_de_compras inherit Object_class
        type tuple (  responsável: Dynamic_Funcionário,
                      data_de_posse: Dynamic_Date)
        method
                    public init: Administração_de_compras

                    public efetuar_solicitação_de_compra
end;
...
class   Controle_de_compra inherit Role_class
        type tuple (  documento: Documento)
        method
                    public init: Controle_de_compra,

                    public recebimento_efetuado
end;
```

ANEXO 4.3 Classes Agente

ANEXO 4.3.1 Classe Pessoa

```
class   Pessoa inherit Object_class
        type tuple (  cpf: String,
                      data_de_nascimentp: Date,
                      nome: Dynamic_String,
                      endereço: Dynamic_String)
        method
```

```
                    public init: Pessoa

                    public cadastrar_cliente,
                    public cadastrar_funcionário
end;
...
class  Cliente  inherit Role_class
       type tuple (  último_documento: Dynamic_Documento)
       method
                    public init: Cliente,

                    public pedido,
                    public registrar_venda
end;
```

ANEXO 4.3.2 Classe Fornecedor

```
class  Administração_de_produção  inherit Object_class
       type tuple (  cgc: String,
                     nome: Dynamic_String,
                     endereço: Dynamic_String,
                     telefone: Dynamic_String,
                     índice-de_qualidade: Dynamic_Integer)
       method
                    public init: Administração_de_produção

                    public emitir_ordem_de_compra,
                    public recebimento,
                    public reativar
end;
```

ANEXO 5 DESCRIÇÃO DE UMA FERRAMENTA DE TRADUÇÃO TF-ORM PARA O_2

O modelo de implementação TF-ORM sobre o SGBD O_2 requer a existência de uma ferramenta automatizada de tradução, para que a tarefa do desenvolvedor seja facilitada. Desta forma, este será capaz de criar uma especificação de dados TF-ORM sem ater-se a detalhes desta linguagem ou mesmo à sintaxe e às estruturas de implementação em O_2.

A ferramenta, denominada *TFORM2O₂*, foi desenvolvida em ambiente operacional de janelas Microsoft Windows, utilizando linguagem Visual Basic e o SGBD Access. Esta plataforma foi escolhida por ser de fácil utilização e possibilitar um desenvolvimento rápido de sistemas.

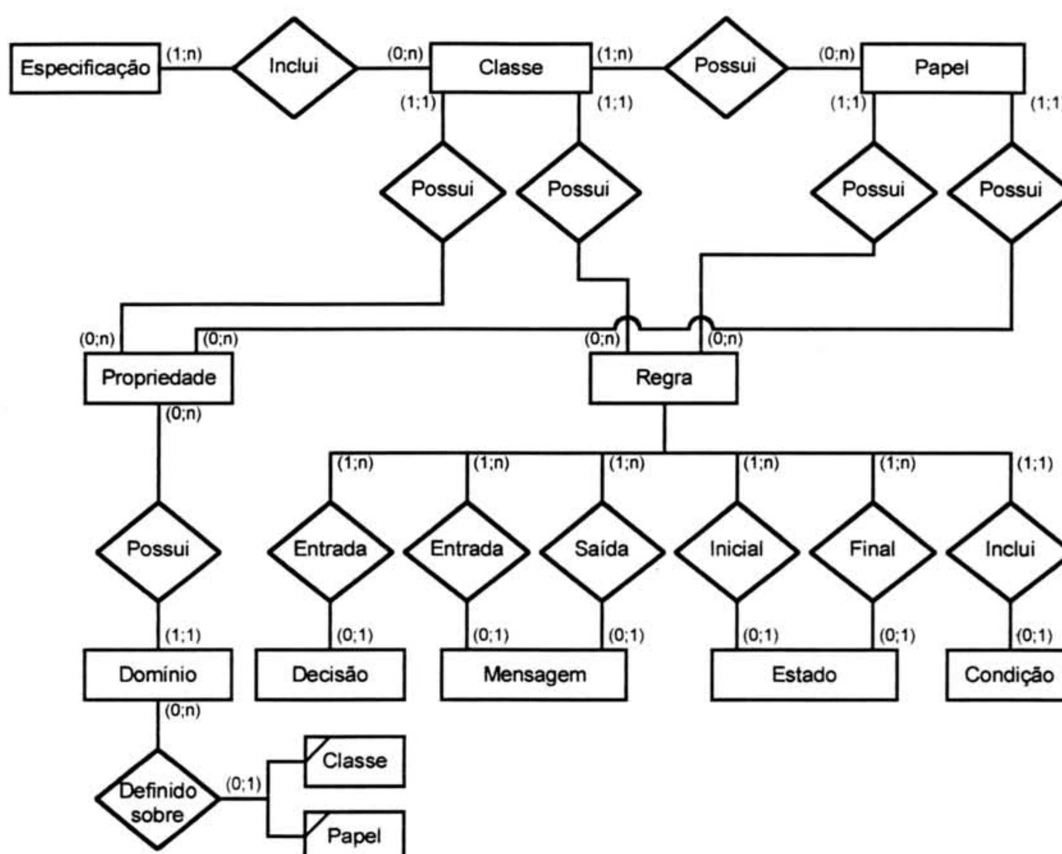


FIGURA A-5.1 - Esquema do Banco de Dados da Ferramenta *TFORM2O₂* (utilizando notação baseada em [YOU 92])

TFORM2O₂ está estruturada basicamente em três módulos: interface de especificação, banco de dados e gerador de código. A interface de especificação orienta o desenvolvedor através de uma estrutura de janelas, permitindo a definição de classes e papéis, e suas respectivas propriedades, regras, estados, mensagens e decisões. Durante a especificação são realizadas verificações simples, como a sintaxe da linguagem de lógica temporal. O banco de dados armazena as especificações já definidas, permitindo, desta forma, a reutilização de classes e papéis. O gerador de código é acionado após o término da especificação, percorrendo o banco de dados e traduzindo-a para código O_2, segundo a abordagem de implementação proposta neste

trabalho. O resultado da tradução é armazenado em um arquivo texto, o qual pode ser utilizado como entrada para o interpretador de comandos do SGBD O_2.

Está definido um esquema de dados O_2, denominado *TFORMkit*, no qual estão especificadas todas as classes pré-definidas pelo modelo de implementação. Este esquema é importado por todas as especificações TF-ORM.

O esquema do banco de dados da ferramenta está definido através do Diagrama Entidade-Relacionamento da Figura A-5.1.

BIBLIOGRAFIA

[ARR 92] ARRUDA, E. **Modelagem temporal em sistemas de gerenciamento de bancos de dados orientados a objetos**. Porto Alegre: Instituto de Informática da UFRGS, 1992. 138p. Trabalho de Conclusão.

[ARR 94] ARRUDA, E. **Um modelo de implementação da linguagem TF-ORM para o SGBD O_2**. Porto Alegre: CPGCC da UFRGS, 1994. 94p. (TI-408).

[ARR 95] ARRUDA, E.; EDELWEISS, N.; PALAZZO OLIVEIRA, J. Implementação de um modelo orientado a objetos com papéis. In: CONFERÊNCIA LATINO-AMERICANA DE INFORMÁTICA - CLEI PANEL, 20., 1994, Ciudad del México. **Proceedings...** Ciudad del México: [s.n.], 1994.

[ATK 89] ATKINSON, M.; BANCILHON, F.; DEWITT, D. et al. **The object-oriented database system manifesto**. Le Chesnay: INRIA, 1989. 18p. (Rapport Technique ALTAIR, v. 30-89)

[BAU 91] BAUZER MEDEIROS, C.; PFEFFER, P. Transformação de um banco de dados orientado a objetos em um BD ativo. In: SIMPÓSIO BRASILEIRO DE BANCOS DE DADOS, 6., 1991, Manaus. **Anais...** Manaus: Imprensa Universitária FUA, 1991. 320 p. p. 238-51.

[BEE 91] BEERI, C.; MILO, T. A model for active object oriented database. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., 1991, Barcelona. **Proceedings...** Barcelona: Morgan Kaufmann, 1991. 596p. p. 337-49.

[BEL 92] BELLINZONA, R.; FUGINI, M.; BRACCHI, G. **Scripting Reusable Requirements through RECAST**. Milão, Itália, 1992. 32p. Draft Version.

[BOO 91] BOOCH, G. **Object oriented design with applications**. Menlo Park, CA: Benjamin / Cummings, 1991.

[COA 92] COAD, P.; YOURDON, E. **Análise Baseada em Objetos**. 2. ed. Rio de Janeiro: CAMPUS, 1992.

[COM 90] COMMITEE FOR ADVANCED DBMS FUNCTION. **Third-generation data base system manifesto**. Berkeley: Electronics Research Laboratory, 1990. 28 p. (Memorandum No. UCB/ERL M90/28)

[DAT 91] DATE, C. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Campus, 1991.

[DAY 88] DAYAL, U.; BUCHMANN, A.; McCARTHY, D. Rules are objects too: a knowledge model for an active, object-oriented database system. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASES SYSTEMS, 2., 1988, Bad Münster.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Proceedings... Bad Münster: Springer-Verlag, 1988. 373p. p. 129-43.

[DEA 91] DeANTONELLIS, V.; PERNICI, B.; SAMARATI, P. F-ORM method: A F-ORM methodology for reusing specifications. In: ASSCHE, F.; MOULIN, B.; ROLLAND, C. **Object oriented aproach in information systems**. Amsterdan: North-Holland, 1991. p. 117-35.

[DEA 92] DeANTONELLIS, V.; CASTANO, S.; MASERATI, A. et al. **Ithaca object-oriented methodology manual** - version 2. Milano: ITHACA, 1992. (ITHACA Report ITHACA. POLIMI-UDUNIV. E.8.11)

[DÍA 91] DÍAZ, O.; PATON, N.; GRAY, P. Rule management in object oriented databases: a uniform approach. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., 1991, Barcelona. **Proceedings**... Barcelona: Morgan Kaufmann, 1991. 596p. p. 317-26.

[EDE 92] EDELWEISS, N.; PERNICI, B.; OLIVEIRA, J. et al. Um modelo temporal orientado a objetos. In: CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 12., 1992, Rio de Janeiro. **Anais**... Rio de janeiro: SBC, 1992. 390 p. p. 166-77.

[EDE 94] EDELWEISS, N. **Sistemas de informação de escritórios**: um modelo para especificações temporais. Porto Alegre: CPGCC da UFRGS, 1994. Tese de Doutorado.

[FRE 90] FREITAS, D. **Análise de Sistemas de Banco de Dados Orientados a Objeto**. Porto Alegre: CPGCC da UFRGS, 1990. 69 p. (TI-162).

[FUG 91] FUGINI, M.; GUGGINO, M.; PERNICI, B. Reusing Requirements through a Modelling and Composition Support Tool. In: INTERNATIONAL CONFERENCE CAISE, 3., 1991, Trondheim, Norway. **Proceedings**... Trondheim, Norway: [s.n.], 1991.

[GEH 91] GEHANI, N.; JAGADISH, H. Ode as an active database: constraints and triggers. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., Sept, 1991, Barcelona. **Proceedings**... Barcelona: Morgan Kaufmann, 1991. 596p. p. 327-36.

[GEH 92] GEHANI, N.; JAGADISH, H.; SHMUELI, O. **SIGMOD RECORD**, New York, v.21, 416p., p. 81-90, June 1992. Trabalho apresentado na ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2., June, 1992, San Diego.

[HAN 91] HANSON, E. Rule condition testing and action execution in Ariel. **SIGMOD RECORD**, New York, v.21, 416p., p. 49-58, June 1992. Trabalho apresentado na ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2., June, 1992, San Diego.

[HEV 92] HEVNER, A. Object-oriented system development methods. **Advances in Computers**, New York, v.35, p. 135-98, 1992.

[HSU 88] HSU, M.; CHEATHAM Jr., T. Rule execution in CPLEX: a persistent objectbase. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASES SYSTEMS, 2., Sept, 1988, Bad Münster. **Proceedings**... Bad Münster: Springer-Verlag, 1988. 373p. p. 150-5.

[HUL 91] HULL, R.; JACOBS, D. Language constructs for programming active databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., 1991, Barcelona. **Proceedings**... Barcelona: Morgan Kaufmann, 1991. 596p. p. 455-67.

[KOR 93] KORTH, H.; SILBERSCHATZ, A. **Sistemas de bancos de dados**. São Paulo: MAKRON, 1993.

[LEC 90] LECLUSE, C.; RICHARD, P.; VELEZ, F. O_2, an object-oriented data model. In: **Advances in database programming languages**. New York: ACM, 1990. p. 257-76.

[MAR 90] MARKOWITZ, V. Referential integrity revisited: an object-oriented perpective. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 16., 1990, Brisbane. **Proceedings**... Brisbane: Morgan Kaufmann, 1990. p. 578-89.

[O2T 91] O_2 TECHNOLOGY. **The O_2 application designer's manual**. Versailles: O_2 Technology, 1991.

[O2T 91a] O_2 TECHNOLOGY. **The O_2 user's guide**. Versailles: O_2 Technology, 1991.

[O2T 91b] O_2 TECHNOLOGY. **The O_2 programmer's manual**. Versailles: O_2 Technology, 1991.

[PAL 95] PALAZZO OLIVEIRA, J.; EDELWEISS, N.; ARRUDA, E. et al. Implementation of an object oriented temporal model. In: DEXXA CONFERENCE, 1995, London. **Proceedings...** London: [s.n.], 1995.

[PER 90] PERNICI, B. Objects with roles. **SIGBOIS Bulletin**, New York, v. 11, n. 2, p. 205-15, April 1990. Trabalho apresentado na CONFERENCE ON OFFICE INFORMATION SYSTEMS, 5., 1990, Cambridge.

[RUM 91] RUMBAUGH, J; BLAHA, M.; PREMERLANI, W. et al. **Object-Oriented Modeling and Design**. Englewoods Cliffs, NJ: Prentice Hall, 1991.

[SCH 91] SCHREIER, U.; PIRAHESH, H.; AGRAWAL, R. et al. Alert: an architecture for transforming a passive DBMS into an active DBMS. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., 1991, Barcelona. **Proceedings**... Barcelona: Morgan Kaufmann, 1991. 596p. p. 469-78.

[SIE 92] SIEBES, A.; VOORT, M. H. van der; KERSTEN, M. **Towards a design theory for database triggers**. Amsterdam, The Netherlands: [s.n.], 1992. 17p. (CWI Technical report).

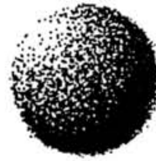
[TAK 90] TAKAHASHI, T.; LIESENBERG, H. **Programação Orientada a Objetos:** uma Visão Integrada do paradigma de Objetos. São Paulo: IME-USP, 1990. 340 p. Trabalho apresentado na VII ESCOLA DE COMPUTAÇÃO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, v.7, São Paulo, 1990.

[VOO 91] VOORT, M. H. van der; KERSTEN, M. **Facets of database triggers**. Amsterdam, The Netherlands: [s.n.], 1991. 35p. (CWI Technical report)

[VOO 91a] VOORT, L. van der; SIEBES, A. **Termination and confluence of rule execution**. Amsterdam, The Netherlands: [s.n.], 1991. 14p. (CWI Technical report)

[WIR 90] WIRFS-BROOK, R.; WILKERSON, B.; WIENER, L. **Designing object-oriented software**. Englewoods Cliffs, NJ: Prentice-Hall, 1990.

[YOU 92] YOURDON, E. **Análise estruturada moderna**. 3. ed. Rio de Janeiro: Campus, 1992.



CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Estudo para Implementação do Modelo TF-ORM

por

Eduardo Henrique Pereira de Arruda

Dissertação apresentada aos senhores:

Prof. Dr. Clesio Saraiva dos Santos

Prof. Dr. Alberto Henrique Frade Laender (DCC/UFMG)

Profa. Dra. Nina Edelweiss

Vista e permitida a impressão.
Porto Alegre, 14 / 11 / 97.

Prof. Dr. José Palazzo Moreira de Oliveira,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
[illegible] do Curso de
[illegible] Computação
[illegible] UFRGS