MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-14174-TDI/1091

UMA ABORDAGEM PARA A PERSISTÊNCIA DOS MODELOS DE OBJETOS DE SISTEMAS CONFIGURÁVEIS E ADAPTÁVEIS

Warley Rodrigues de Almeida

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Maurício Gonçalves Vieira Ferreira, aprovada em 1 de abril de 2005.

INPE
São José dos Campos
2006

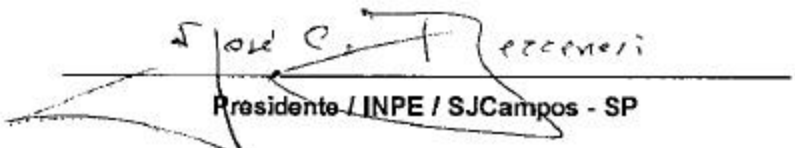
681.3.06

Almeida, W. R.
Uma abordagem para a persistência dos modelos de objetos de sistemas configuráveis e adaptáveis/ W. R. Almeida. – São José dos Campos: Instituto Nacional de Pesquisas Espaciais (INPE), 2005.
185 p.; - (INPE-14174-TDI/1091)

1. Metamodelo. 2. Padrão de Projeto. 3. Metadado. 4. Modelo de Objetos Adaptável. 5. Evolução de Software. I.Título.

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de Mestrado em Computação Aplicada

Dr. José Carlos Becceneri



Presidente / INPE / SJCampos - SP

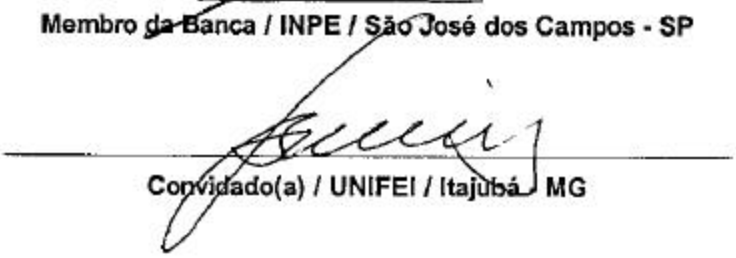
Dr. Mauricio Gonçalves Vieira Ferreira

Orientador(a) / INPE / São José dos Campos - SP

Nilson Sant'Anna

Membro da Banca / INPE / São José dos Campos - SP

Dr. Sidnei de Brito Alves



Convidado(a) / UNIFEI / Itajubá - MG

Aluno (a): Warley Rodrigues de Almeida

São José dos Campos, 01 de Abril de 2005

“O temor ao Senhor é o princípio de toda sabedoria”.

PROVÉRBIOS 1, 7

Dedico este trabalho a meus pais,
MÁRIO FERREIRA DE ALMEIDA e
MARIA DE LOURDES RODRIGUES DE ALMEIDA.

AGRADECIMENTOS

Agradeço em primeiro lugar ao meu Senhor Jesus Cristo.

Agradeço ao meu orientador, o professor Doutor Maurício Gonçalves Vieira Ferreira, pelo apoio, incentivo e orientação segura ao longo deste trabalho.

Aos professores Doutores: Nilson Sant'Anna, Airam Jonatas Preto e Stephan Stephany pelos ensinamentos nas disciplinas cursadas.

Aos membros da banca examinadora pela disposição em analisar este trabalho.

A minha futura esposa Danaise por todo amor, carinho, atenção, incentivo e principalmente paciência.

A meus pais pelos sacrifícios, ensinamentos e exemplos de vida. A meus irmãos e cunhados que sempre torceram pelo meu sucesso.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo auxílio financeiro, ao Instituto Nacional de Pesquisas Espaciais (INPE) pela oportunidade e apoio e a todos os professores e funcionários do Programa de Pós-Graduação em Computação Aplicada (CAP).

As minhas amigas gêmeas (Andreia e Adriana) pela amizade sincera, ajuda, convivência e caronas nestes dois anos.

A todos amigos do laboratório, em especial ao Élcio, Joanito, Ana Paula, Mariana e Isabela pela amizade, companheirismo e apoio nesta jornada.

Agradeço a todas as pessoas que me ajudaram a vencer mais esta etapa.

RESUMO

Alguns sistemas de informação possuem domínios que se alteram constantemente. Os desenvolvedores desses sistemas têm o problema de lidar com mudanças nos requisitos do domínio após a implementação do sistema. O Modelo de Objetos Adaptável (*Adaptive Object Model* - AOM) propõe uma solução para esse problema armazenando a estrutura e o comportamento dos objetos desses sistemas em um banco de dados, ao invés de codificá-los, permitindo realizar mudanças nesses objetos através de alterações nesse banco de dados, sem a necessidade de recodificação. Esses sistemas são adaptáveis e configuráveis em tempo de execução, mas o seu desenvolvimento apresenta algumas dificuldades para seus desenvolvedores. Duas dessas dificuldades são armazenar e recuperar o modelo de objetos em um banco de dados. Em sistemas distribuídos desenvolvidos baseados na arquitetura AOM, a tarefa de persistir o modelo de objetos em um banco de dados, se apresenta como um desafio e motivou esta dissertação que tem por objetivos: (1) estabelecer um mapeamento do modelo de objetos de uma aplicação para três sistemas de gerenciamento de dados; (2) como estudo de caso desenvolver três protótipos, baseados nos AOMs, que armazenem o modelo de objetos de um sistema para o controle de satélites e realizar uma comparação qualitativa entre os sistemas de gerenciamento de dados utilizados; (3) criar mecanismos para a instanciação do modelo de objetos a partir dos seus valores armazenados em um sistema de gerenciamento de dados.

AN APPROACH TO PERSISTENCE OF MODELS OF ADAPTIVE AND CONFIGURABLE SYSTEMS

ABSTRACT

Some information system domains change constantly. The developers of this kind of system have to address the problem of frequent changes in the domain requirements when the system is already coded. The Adaptive Object Model (AOM) proposes to solve this problem by storing structure and behavior of the objects in a database instead of codifying them. This allows the user to make changes in the object model without having to change the code. These are difficulties to be faced in this kind of system: how to store and how to retrieve the object model in a database. In distributed systems developed based on AOM architecture the task of persisting the object model in a database becomes a challenge. The aims of this work are: (1) to establish the mapping of an object model of a distributed, adaptive and configurable system to three data management systems; (2) to conduct a case study that consists of developing three prototypes based on AOM to store the object model of satellites control system and to compare the advantages and disadvantages of each data management system used; and (3) to create mechanisms to instantiate the object model using values stored in a data management system.

SUMÁRIO

Pág.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

AOM - *Adaptive Object Model*

API - *Application Programming Interface*

ARO - *Army Research Office*

BLOB - *Binary Large Objects*

CBERS - *China Brazil Earth Research Satellite*

CWM - *Common Warehouse Metamodel*

DARPA - *Defense Advanced Research Projects Agency*

ESL - *Electromagnetic Systems Laboratories*

IDE - *Integrated Development Environment*

INPE - Instituto Nacional de Pesquisas Espaciais

J2EE - *Java 2 Enterprise Edition*

JDBC - *Java Database Connectivity*

JNDI - *Java Naming and Directory Interface*

MMP - *Multi-Mission Platform*

MOF - *Meta Objects Facility*

NSF - *National Science Foundation*

OMG - *Object Management Group*

SCD - Sistema de coleta de dados

SGBD - Sistema Gerenciador de Banco de Dados

SICSDA - Sistema de Controle de Satélites Distribuído e Adaptável

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

W3C - *World Wide Web Consortium*

XMI - *XML Metadata Interchange*

XML - *eXtensible Markup Language*

CAPÍTULO 1

INTRODUÇÃO

A grande extensão do país e a existência de imensas áreas com baixa densidade populacional, especialmente na região Amazônica, são características predominantes para justificar o uso de satélites como instrumentos de integração do território nacional, através de suas redes de comunicação, serviços de previsão de tempo e acompanhamento dos processos de uso do solo. Essas condições fisiográficas, aliadas à prática adquirida no estudo e na utilização de técnicas espaciais, foram motivos suficientes para se iniciar um programa de desenvolvimento da tecnologia espacial no Brasil.

O Instituto Nacional de Pesquisas Espaciais (INPE), tem como uma de suas principais atividades o domínio da tecnologia espacial através do desenvolvimento e operação de satélites. Faz parte do programa espacial brasileiro o lançamento de quatro satélites. Dois deles para a coleta de dados: o Sistema de Coleta de Dados 1 (SCD1), lançado em 1993, e o SCD2, lançado em 1998, ambos ainda em operação. Os outros dois satélites foram lançados para realizar sensoriamento remoto: o *China Brazil Earth Research Satellite 1* (CBERS-1) lançado em 1999 com sua vida útil finalizada em 2003, e o CBERS-2 lançado no final de 2003.

Para gerir as peculiaridades inerentes ao controle de um satélite, o INPE criou uma infra-estrutura robusta, apoiada por duas estações de rastreamento e aplicativos de *software* para o controle de satélites.

Cada satélite tem características próprias, o que significa que um satélite sempre difere de outro, ainda que se trate de uma diferença sutil. Para cada um dos quatro satélites lançados até agora foi destinado uma máquina ou um conjunto máquinas específicas, para a execução de um aplicativo desenvolvido para esse determinado satélite, auxiliando no recebimento de seus dados e monitoramento de seu estado interno.

Dessa forma, para cada novo satélite lançado deve-se desenvolver um aplicativo especificamente para esse satélite e devem-se destinar máquinas à execução desse *software*. Isso gera um custo de desenvolvimento adicional, a cada novo lançamento, tanto em termos de *hardware*, quanto em termos de *software*.

As características do domínio do controle de satélites, aliadas ao desejo crescente dos usuários de obter aplicações que se adaptem à modificações de seus domínios, fez com que se pensasse em construir sistemas para o controle de satélites mais configuráveis, flexíveis e adaptáveis. Permitindo assim, que esses sistemas possam se adaptar, de uma maneira mais rápida, mais eficiente e a um custo menor, às novas necessidades do domínio (Thomé, 2004).

Uma maneira de se conseguir aplicações mais configuráveis é armazenar certos aspectos do sistema, como as regras do negócio por exemplo, em um banco de dados. Isso permite realizar alterações nessas regras do negócio sem a necessidade de se modificar o código que as implementam. O sistema resultante pode se adaptar mais facilmente aos novos requisitos do domínio através de modificações nos valores armazenados em um banco de dados, ao invés de alterações no código, evitando o tempo gasto para recompilar e liberar uma nova versão desse sistema.

A arquitetura dos modelos de objetos adaptáveis (*Adaptive Object Model Architecture*) é um tipo particular de arquitetura reflexiva[1] que abrange sistemas orientados a objetos que gerenciam elementos de determinados tipos e que podem ser estendidos para adicionar novos tipos de elementos (adaptável) ou modificar tipos de elementos existentes (configurável). Sistemas que possuem essa arquitetura recebem também os nomes de Modelos de Objetos Ativos (*Active Object Models*) ou Modelos de Objetos Dinâmicos (*Dynamic Object Models*) (Yoder et al., 2001).

A utilização dos modelos de objetos adaptáveis no desenvolvimento de sistemas pode amenizar algumas das dificuldades encontradas atualmente pelos desenvolvedores de *software*, principalmente em relação à flexibilidade, evolução e manutenção dos

[1] Arquiteturas que fornecem aos usuários informações sobre sua própria estrutura são chamadas de arquiteturas reflexivas (Yoder e Johnson, 2002).

requisitos do sistema desenvolvido, permitindo uma redução do custo total tanto no desenvolvimento quanto na manutenção desses sistemas (Ledeczi et al., 2000).

Por exemplo, a partir do momento em que se consiga implementar um sistema adaptável para o controle de satélites capaz de: (**1**) armazenar o modelo de objetos[2] do sistema em um banco de dados, (**2**) permitir ao usuário monitorar qualquer um dos satélites em um determinado instante, (**3**) realizar a manutenção do modelo de objetos e (**4**) adaptar o modelo de objetos a um novo satélite, esse sistema ajudará o sistema espacial brasileiro na operação de satélites, pois terá mais flexibilidade para atender novas missões.

Com a implementação desse sistema para o controle de satélites adaptável, o esforço necessário para se disponibilizar *softwares* para o controle de novos satélites tende a diminuir. Isso se deve ao fato de que adaptar esse sistema à um novo satélite pode ser realizado em menos tempo e a um custo mais baixo do que desenvolver um novo sistema para o controle desse satélite com todas as etapas de análise, projeto, implementação e testes.

Um sistema baseado nos AOMs fornece informações a seus usuários, em tempo de execução, sobre o seu próprio modelo, permitindo assim que seus usuários possam alterar o modelo do sistema de acordo com suas necessidades através da modificação dessas informações. Essa característica torna o sistema mais configurável e adaptável. Entretanto armazenar o modelo de objetos de um sistema baseado nesse tipo de arquitetura se torna uma dificuldade.

Um sistema desenvolvido baseado nos AOMs, armazena além dos valores dos atributos de um objeto (por exemplo, se um objeto possui um atributo Nome do tipo *string* com o valor ‘José da Silva’ apenas o valor ‘José da Silva’ é armazenado) armazena também as estruturas desses atributos. No exemplo anterior armazenaria o nome do atributo NomeAtributo=‘Nome’, o tipo do atributo TipoAtributo=‘String’ e o valor do atributo

[2] O modelo de objetos descreve a estrutura dos objetos de um sistema: sua identidade, suas associações com outros objetos, seus atributos e seus métodos. Ele é composto de um conjunto de definições sobre as entidades computacionais de um determinado domínio (Rumbaugh et al., 1994).

ValorAtributo='José da Silva'. Um sistema desenvolvido baseado nos AOMs armazena ainda os métodos e associações desse objeto.

Projetar a persistência[3] dos objetos de um sistema é uma tarefa difícil e que consome tempo. O projeto da persistência dos objetos de um sistema baseado nos AOMs acrescenta algumas dificuldades, principalmente devido ao fato desses sistemas armazenarem além dos valores dos atributos dos objetos, armazenarem os próprios atributos, métodos e associações desses objetos. Outra dificuldade vem do fato de existirem poucos sistemas desenvolvidos baseados nos AOMs, pois os AOMs são relativamente novos. Assim a maioria dos desenvolvedores não possui experiências que poderiam ser repetidas no desenvolvimento de novos sistemas.

Essas dificuldades foram uma motivação para este trabalho, cujo objetivo principal é estabelecer uma abordagem e definir um mapeamento para a persistência dos modelos de objetos de sistemas desenvolvidos baseados nos AOMs em três sistemas de gerenciamento de dados: Sistema Gerenciador de Banco de Dados Orientado a Objetos (SGBDOO), SGBD Relacionais (SGBDR) e *Native XML Databases* (NXD).

Outros objetivos são: (1) fazer uma comparação qualitativa entre os sistemas de gerenciamento de dados; (2) implementar um protótipo para cada sistema de gerenciamento de dados, que utilize o mapeamento definido para persistir seu modelo de objetos e (3) implementar nesses protótipos editores que facilitem a alteração dos modelos de objetos.

Este trabalho teve como origem a proposta de Thomé (2004) para uma arquitetura distribuída, configurável e adaptável aplicada ao controle de várias missões de satélites. Esta dissertação é organizada da seguinte maneira:

- O Capítulo dois apresenta algumas das fundamentações teóricas utilizadas como base para este trabalho de pesquisa: os conceitos de persistência de objetos, de sistemas distribuídos e da arquitetura *Java 2 Enterprise Edition*

[3] Persistência é a habilidade de um objeto de sobreviver ao término do processo onde o mesmo reside, permitindo sua utilização em um outro processo (Keller, 2004).

(J2EE), além da *Application Programming Interface* (API) de reflexão da linguagem Java. Esse Capítulo também faz uma introdução à arquitetura de metamodelagem da *Object Management Group* (OMG) com conceitos importantes para um melhor entendimento dos Modelos de Objetos Adaptáveis.

- O Capítulo 3 concentra-se nos Modelos de Objetos Adaptáveis, em sua arquitetura, nos padrões de projeto utilizados em sua arquitetura e em seus aspectos de implementação. Também apresenta a arquitetura proposta por Thomé (2004) para sistemas distribuídos, configuráveis e adaptáveis aplicados ao controle de satélites (arquitetura SICSDA), pois essa arquitetura foi a origem deste trabalho.

- O Capítulo 4 mostra a construção do metamodelo, a partir de um modelo inicial, de um sistema, baseado nos AOMs, aplicado ao controle de satélites.

- O Capítulo 5 mostra o mapeamento definido para a persistência dos modelos adaptáveis nos três sistemas de gerenciamento de dados escolhidos. Utilizando como estudo de caso o metamodelo construído no Capítulo 4.

- O Capítulo 6 mostra alguns aspectos importantes dos três protótipos desenvolvidos para o estudo de caso. Além de fazer uma comparação qualitativa entre os três sistemas de gerenciamento de dados utilizados para persistência dos modelos de objetos adaptáveis.

- No final, o Capítulo 7 apresenta as principais conclusões e propostas de trabalhos futuros.

CAPÍTULO 2

FUNDAMENTOS E CONCEITOS

Este Capítulo introduz alguns conceitos e tecnologias que foram utilizados no desenvolvimento dos Capítulos seguintes. Cada Seção introduz de uma forma concisa, determinado aspecto que se tornou relevante à dissertação, de forma que um conhecimento inicial possa ser oferecido ao leitor. Este Capítulo pode ser divido em duas partes, a primeira parte, compreendida pelas Seções 2.1 a 2.3, apresenta conceitos utilizados nos Capítulos 5 e 6 deste trabalho e a segunda parte, compreendida pelas Seções 2.4 e 2.5, apresenta a arquitetura de metamodelagem da OMG. Essa arquitetura deu suporte ao surgimento dos modelos de objetos adaptáveis apresentados no Capítulo seguinte.

A Seção 2.1 apresenta a linguagem XML. Na Seção seguinte, este Capítulo apresenta conceitos sobre a persistência de objetos. A Seção 2.3 apresenta os sistemas distribuídos, a linguagem Java e a arquitetura para sistemas distribuídos J2EE. Na Seção 2.4 apresenta-se os conceitos utilizados na metamodelagem da OMG. A última Seção deste Capítulo apresenta a arquitetura de metamodelagem da OMG.

2.1 XML

A linguagem *eXtensible Markup Language* (XML) é utilizada para construir documentos estruturados e autodescritivos que satisfazem um conjunto de regras criados para cada tipo de documento. Foi criada por um grupo de empresas e organizações, chamado *World Wide Web Consortium* (W3C), no final da década de 90. A linguagem XML surgiu da necessidade de um formato de dados portável (Bond et al., 2003).

Características de flexibilidade e portabilidade vêm fazendo com que, nos últimos anos, a XML seja aceita como um padrão para representação, intercâmbio e manipulação de dados em aplicações para as mais diversas áreas de negócios (Gabrick e Meiss, 2002). Representação de dados em aplicações de gerenciamento de conteúdo, aplicações de

transações bancárias e de publicação de conteúdo em intranets são alguns exemplos de uso da XML.

Um arquivo XML constitui-se de texto e tags (ou rótulos). Os tags fornecem estrutura ao texto representado no documento. Cada linguagem XML define sua própria gramática, um conjunto de regras específicas para a construção de documentos, que governam o conteúdo e a estrutura dos documentos escritos nessa linguagem. Um *parser* XML é capaz de validar a estrutura de um documento XML, através de sua gramática.

A XML vem sendo considerada um padrão da indústria para manipulação, representação e transporte de dados. Assim, ela pode ser utilizada em sistemas sem se preocupar com o sistema operacional ou com o *hardware* utilizado. Na área de transporte de dados, pode-se exemplificar a troca de mensagens entre sistemas semelhantes que são executados em plataformas incompatíveis e a troca de dados entre organizações.

Em adição a sua utilização no transporte de dados, a XML pode também ser usada como formato de armazenamento de dados em algumas situações (Gabrick e Meiss, 2002), essa foi a utilização da linguagem XML neste trabalho.

Os documentos XML consistem em elementos e declarações, além de comentários opcionais. Um elemento tem a seguinte sintaxe (Bond et al., 2003):

- <tag_inicio atributos> corpo <tag_fim>

As tags não são predefinidas. O autor de um documento XML pode criar as tags que forem apropriadas para os dados que estiver descrevendo. Um elemento pode conter dados, atributos e outros elementos aninhados. O corpo de um elemento é todo o texto, incluindo as tags aninhadas, englobado pelas tags inicio e fim.

As tags fornecem(Bond et al., 2003):

- Informações sobre os significados dos dados.

- Os relacionamentos entre diferentes partes dos dados.

A Figura 2.1 ilustra um arquivo XML criado para estruturar os dados de um catálogo de produtos específico, o arquivo foi numerado para facilitar o entendimento de suas partes, mas a numeração não faz parte do arquivo XML. Pode-se observar na linha [1] que um documento XML sempre começa com uma declaração para informar que ele é um documento em conformidade com a especificação 1.0 da linguagem XML. As linhas restantes representam o elemento <catalogo-produtos>. Um produto é representado pelo elemento formado pelo tag <produto>, compreendido entre as linhas [3] e [8]. Os tags das linhas [4] e [5] descrevem o produto nos EUA e no México respectivamente e as linhas [6] e [7] indicam o preço do produto nos EUA e no México. O arquivo XML ilustrado na Figura 2.1 representa apenas um produto, mas na prática o elemento <catalogo-produtos> pode conter vários elementos <produtos> descrevendo os produtos existentes em um catálogo.

```
[1]<?xml version="1.0"?>
[2]<catalogo-produtos>
[3]  <produto codigo="123456" nome="The Product">
[4]       <descricao local="en_US"> An excellent product.</descricao>
[5]       <descricao local="es_MX">Un producto excellente.</descricao>
[6]       <preco local="en_US" unidade="USD">99.95</preco>
[7]       <preco local="es_MX" unidade="MXP">9999.95</preco>
[8]  </produto>
[9]</catalogo-produtos >
```

FIGURA 2.1 – Exemplo de um arquivo XML.

FONTE: adaptada de Gabrick e Meiss (2002).

Como arquivos XML contêm dados, existem algumas tecnologias em desenvolvimento para o armazenamento de dados no formato XML nativo. A variedade de tecnologias e produtos existentes é razoável e ainda não se sabe quais dessas tecnologias se tornarão líderes (Gabrick e Meiss, 2002).

2.2 Persistência de Objetos

Os sistemas computacionais operam sobre dados. Quando há necessidade desses dados persistirem o sistema utiliza um depósito permanente de dados. Um objeto persistente pode então ser compartilhado por diversas aplicações e pode ser utilizado por diversas execuções de uma mesma aplicação (Rumbaugh et al., 1994).

Um objeto não persistente é chamado de objeto transiente e sua existência é restrita a uma única execução da aplicação. Quando termina a execução do sistema todos os seus objetos transientes são destruídos (Rumbaugh et al., 1994).

Diferentes sistemas podem ser empregados para fornecer às aplicações a persistência de objetos, sendo esses sistemas chamados de sistemas de gerenciamento de dados.

Os SGBDRs são utilizados por sua simplicidade e seus fundamentos matemáticos na álgebra relacional (Elmasri e Navathe, 2002). Mais recentemente, sistemas de armazenamento de arquivos XML em seu formato nativo, os NXDs, aparecem como alternativa para a persistência de dados e para o compartilhamento de dados entre aplicações (Nambiar et al., 2002). Para realizar a persistência dos objetos de uma aplicação nesses dois sistemas de gerenciamento de dados deve-se fazer um mapeamento das classes da aplicação para formato dos dados nos bancos de dados relacionais ou nos NXDs.

Os SGBDOOs, outro sistema de gerenciamento de dados utilizado neste trabalho, compartilham uma série de propriedades das linguagens de programação orientadas a objeto, mas ao contrário dessas, persistem os objetos além do tempo de vida das aplicações que os criaram. Ao contrário dos dois sistemas de gerenciamento de dados anteriores, os SGBDOOs podem realizar a persistência dos objetos de uma aplicação diretamente sem a necessidade de um mapeamento das classes da aplicação para seu formato de dados (Elmasri e Navathe, 2002).

2.3 Sistemas Distribuídos, Java e J2EE

2.3.1 Sistemas Distribuídos

Um sistema computacional distribuído pode ser definido como uma coleção de processos em computadores, independentes ou não, que comunicam entre si através do intercâmbio de mensagens (Gabrick e Meiss, 2002). Com o surgimento da orientação a objeto houve sua junção com a tecnologia de sistemas distribuídos dando origem à tecnologia de distribuição de objetos (Ferreira, 2001).

A distribuição de objetos permite, entre outras coisas, que (Ferreira, 2001):

- Objetos de uma aplicação possam residir em qualquer lugar da rede. Os objetos são instanciados em uma rede de computadores independentes e heterogêneos.
- Serviços de persistência possam armazenar e recuperar objetos eficientemente.
- Serviços de pesquisa possam localizar objetos apropriadamente, fornecendo a transparência de localização e possibilitando um cliente invocar um objeto sem conhecer sua localização.

Um sistema distribuído deve ter como um dos objetivos tornar sua natureza distribuída transparente, tanto para seus usuários quanto para os desenvolvedores de aplicações dentro de um sistema. Em um sistema distribuído existem vários tipos de transparência, resumidos na Tabela 2.1 (Gabrick e Meiss, 2002). Para conseguir esses tipos de transparência deve haver um meio que permita a comunicação entre objetos diferentes, de forma que um objeto não necessite conhecer detalhes de localização e implementação dos demais objetos (Ferreira, 2001).

Além disso, os objetos podem estar distribuídos em computadores diferentes e cada computador com seu sistema operacional e com um conjunto de *drivers* de dispositivos específicos. Na prática, seria muito difícil tentar descobrir todos os possíveis ambientes computacionais (um ambiente computacional é formado pelo sistema operacional mais

o *hardware* de um computador) onde um objeto seria executado. Surgiu então a necessidade de um mecanismo que servisse de interface entre objeto e seu ambiente computacional, tirando dos desenvolvedores de sistemas a preocupação das características de um ambiente computacional específico. Essa necessidade deu origem a uma nova classe de produtos de *software*, chamada *middleware* (Gabrick e Meiss, 2002).

TABELA 2.1 – Tipos de transparência em um sistema distribuído.

Tipo de Transparência	Descrição
Transparência de rede	Todos os recursos são acessados da mesma maneira, independente de sua real localização na rede.
Transparência de localização	Os recursos de *hardware* e *software* dedicados a uma atividade podem ser aumentados sem afetar os clientes. Essa característica aumenta a escalabilidade do sistema.
Transparência de falha	Através de técnicas para lidar com falhas o sistema permite seus clientes completarem suas tarefas mesmo quando ocorra falhas de *hardware* ou de *software*.
Transparência de mobilidade	O recursos no sistema podem ser reorganizados sem afetar os usuários.

FONTE: adaptada de Gabrick e Meiss (2002).

Middleware é uma camada adicional de *software* de conectividade, que consiste em um conjunto de serviços, que permitem a interação, através da rede, de muitos processos em execução em uma ou mais máquinas. Esse *software* de conectividade se localiza entre a aplicação e o sistema operacional. A Figura 2.2 mostra estas três camadas de *software*.

O principal propósito do *middleware* é ajudar na obtenção da transparência e na resolução dos problemas de conectividade e interoperabilidade de aplicações em um sistema distribuído; mas cabe ao desenvolvedor a tarefa de decidir quais funcionalidades serão colocadas no lado do cliente e quais estarão do lado do servidor da aplicação distribuída. Assim, é importante entender o problema que será resolvido pela aplicação e o valor dos serviços *middleware* que permitirão a distribuição dessa aplicação (Ferreira, 2001).

De um modo geral, os sistemas distribuídos devem levar em conta também a interoperabilidade, visto que um sistema distribuído deve ser capaz de expandir tanto geograficamente quanto em plataformas computacionais diferentes. O *middleware* é uma forma utilizada para os sistemas distribuídos encapsularem as diferenças entre os ambientes computacionais (Gabrick e Meiss, 2002).

FIGURA 2.2 – Camadas em sistemas distribuídos.
FONTE: adaptada de Gabrick e Meiss (2002).

2.3.2 A Arquitetura J2EE

Atualmente, uma das arquiteturas utilizadas para o desenvolvimento de sistemas distribuídos é a arquitetura J2EE. A arquitetura J2EE permite aos desenvolvedores a criação de aplicações em três camadas (também chamada de N camadas), através do fornecimento de serviços da camada *middleware* para comunicar com uma variedade de clientes e sistemas de gerenciamento de dados (Kassem et al., 2002).

Um sistema desenvolvido nesta arquitetura tem sua funcionalidade logicamente agrupada em três camadas: apresentação, lógica do negócio e acesso a dados. Em cada uma delas se encontram objetos para realizar as funções designadas para a sua camada (Bond et al., 2003).

A camada de apresentação contém os objetos responsáveis pela interface do sistema com os usuários. A camada de lógica dos negócios contém os objetos responsáveis pela implementação das funcionalidades do sistema. A camada dos dados da aplicação se refere aos objetos que gerenciam os dados do sistema. Esses dados ficam normalmente sobre o controle de um SGBD, como por exemplo o Caché, Oracle ou DB2 (Gabrick e Meiss, 2002).

Uma aplicação em três camadas possui vantagens sobre um modelo cliente servidor tradicional de duas camadas. No modelo de duas camadas a aplicação cliente deve ter conhecimento de como acessar os sistemas de gerenciamento de dados. Além disso, a aplicação cliente deve implementar algumas regras do negócio para manipular os dados recuperados dos sistemas de gerenciamento de dados. Em uma aplicação em três camadas a lógica do negócio e o acesso aos dados são feitos na camada de lógica do sistema, permitindo assim o desenvolvimento de aplicações clientes que sejam responsáveis apenas pela interface com o usuário (Kassem et al., 2002). Esses clientes requerem poucos recursos de *hardware* para sua execução, significando redução de custos para utilização do sistema.

Um dos aspectos mais importantes da arquitetura J2EE é a variedade de serviços *middleware* que ela fornece aos desenvolvedores. Para este trabalho dois serviços importantes que fazem parte da especificação J2EE são as APIs (Gabrick e Meiss, 2002):

- *Java Naming and Directory Interface* (JNDI): fornece um mecanismo padrão para localização de recursos, incluindo objetos remotos, propriedades do ambiente e serviços de diretório.

- *Java DataBase Connectivity* (JDBC): fornece acesso independente de fabricante aos SGBDRs.

O J2EE fornece diferentes tipos de componentes para diferentes propósitos. Este trabalho utilizou os componentes *Enterprise Java Beans* (EJBs). Em uma típica aplicação J2EE os EJBs contêm a lógica do negócio do aplicativo e os dados do

domínio do sistema. Os EJBs ficam na segunda camada da aplicação, a camada da lógica dos negócios (Bond et al., 2003).

Existem três tipos de EJB convenientes para diferentes propósitos, mas este trabalho utilizou apenas os dois primeiros (Bond et al., 2003):

- EJB de entidade: um EJB de entidade mapeia uma combinação de dados e suas funcionalidades. Os EJBs de entidade normalmente são baseados em um sistema de gerenciamento de dados e são criados com base nos dados existentes nesse sistema de gerenciamento de dados. Por exemplo, uma tupla de uma relação em um SGBDR, um objeto em um SGBDOO ou um elemento de um arquivo XML em um NXD.

- EJB de sessão: um EJB de sessão é útil para mapear a lógica do negócio. Os EJBs de sessão normalmente representam as regras do negócio da aplicação. Eles permitem que a funcionalidade do negócio seja desenvolvida e depois implantada independentemente da camada de apresentação. Por exemplo, se fosse necessário criar um aplicativo de recursos humanos para uma empresa esse aplicativo teria uma determinada interface com o usuário. Através dessa interface o usuário poderia ter acesso a alguma funcionalidade do negócio para os funcionários dessa empresa, através dos EJBs de sessão. Quando fosse necessário mudar a interface da aplicação, bastaria substituir a interface existente por uma outra sem a necessidade de desenvolver novamente a aplicação inteira.

- EJB dirigido por mensagens: um EJB dirigido por mensagens é conceitualmente muito parecido com um EJB de sessão, mas é ativado apenas para o recebimento de mensagens entre objetos.

Os protótipos deste trabalho foram desenvolvidos utilizando-se a plataforma J2EE, pelos seguintes motivos: (1) o J2EE oferece neutralidade de plataforma de sistema operacional, de *hardware* e de ambiente de execução, fornecendo um certo grau de portabilidade e (2) pode-se escolher entre uma série de ferramentas, a maioria delas de

software livre, que seguem a arquitetura J2EE, dando um certo grau de liberdade para utilizar a ferramenta que seja mais adequada tanto em termos de custos, quanto em termos de facilidades oferecidas e de disponibilidade para o uso.

2.3.3 Reflexão em Java

Java pode ser considerada muito mais do que apenas uma linguagem de programação. Java consiste também de um ambiente de desenvolvimento e execução de sistemas, com características marcantes como interoperabilidade, independência de plataforma e orientação a objetos. A linguagem Java oferece uma série de APIs, que são interfaces entre a aplicação e a plataforma que processa essa aplicação. Uma das APIs oferecidas pela linguagem Java, utilizada principalmente para o desenvolvimento de aplicações dinâmicas é a API *Reflection* (Green, 2004). Por essas razões, pode-se usar o termo plataforma Java, ao invés de simplesmente linguagem Java.

A reflexão habilita uma aplicação descobrir informações sobre os atributos, métodos e construtores de objetos em tempo de execução. A reflexão permite também executar os métodos e construtores de um determinado objeto cujo nome não se conheça em tempo de compilação. A API de reflexão em Java pode ser utilizada no desenvolvimento de ferramentas como depuradores, *browsers* de classes e construtores de *Graphical User Interface* (GUIs). Com essa API em tempo de execução consegue-se (Green, 2004):

- Determinar a classe de um objeto.
- Obter informações sobre superclasses, construtores, métodos e atributos de um objeto.
- Descobrir quais constantes de declarações de métodos pertencem a uma interface.
- Criar uma instância[4] de uma classe, cujo nome se torne conhecido apenas em tempo de execução.

[4] Instância de uma classe e objeto têm o mesmo significado (Rumbaugh et al., 1994).

- Obter e alterar o valor de um atributo de um objeto, mesmo se o nome do atributo for conhecido apenas em tempo de execução.

- Invocar um método de um objeto, mesmo se o nome do método só for conhecido em tempo de execução.

A maioria dessas funções são executadas pela classe *Class* ou precisam da classe *Class* como parâmetro. A seguir apresenta-se como obter um objeto *Class* de uma classe Java.

2.3.3.1 Recuperando um Objeto Class

Existem várias formas para se recuperar um objeto *Class*, que podem ser consultadas em Green (2004). Para este trabalho é importante a forma de recuperar um objeto *Class*, quando o nome da classe é conhecido apenas em tempo de execução. Nesse caso pode-se utilizar o método *forName*(*String* className) da classe java.lang.Class, que pertence a API *Reflection* da linguagem Java.

Esse método retorna o objeto *Class* associado ao valor da *string* passada pelo parâmetro className. No exemplo a seguir, a classe c obterá o valor do objeto *Class* java. awt.button:

- Class c = Class.forName("java.awt.Button");

2.3.3.2 Obtendo Informações de um Método

Para descobrir quais métodos pertencem a uma classe, deve-se utilizar o método *getMethods* da classe *Class*. O método *getMethods* retorna um *array* que contém objetos da classe java.lang.reflect.Method. Com um objeto *Method*, pode-se descobrir informações de um método, tais como o nome, o tipo de retorno, os tipos de parâmetros e o conjunto de exceções levantadas por esse método. Além disso, a classe *Method* possui o método *invoke* que executa o método.

Suponha que seja necessário implementar um depurador de programas que permita ao usuário selecionar e executar métodos durante uma depuração. Como em tempo de compilação, ainda não se sabe quais métodos o usuário invocará, os nomes dos métodos

não podem ser codificados. Ao invés disso, deve-se criar uma forma de o depurador executar os métodos cujos nomes só se tornem conhecidos em tempo de execução. Isso pode ser feito através dos seguintes passos (Green, 2004):

a) Criar um objeto *Class* que corresponda ao objeto cujos métodos deseja-se invocar.

b) Criar um objeto da classe *Method*, invocando o método *getMethod* do objeto *Class*. O método *getMethod* possui dois argumentos: uma *string* contendo o nome do método e um *array* de objetos *Class*. Cada elemento do *array*, corresponde a um parâmetro do método que deseja-se invocar.

c) Invoque o método chamando *invoke* do objeto da classe *Method*. O método *invoke* tem dois parâmetros: um *array* com os valores dos parâmetros do método a ser invocado e um objeto cuja classe declara ou herda o método. O método *invoke* executa o método desse objeto com os parâmetros passados pelo *array*.

2.4 Conceitos Utilizados na Metamodelagem

Esta Seção e a seguinte introduzem alguns conceitos que deram suporte ao surgimento dos AOMs, que são apresentados no próximo Capítulo.

Pode-se definir um modelo como uma descrição abstrata e uma representação simplificada de um sistema ou de um processo ou de apenas uma parte de um sistema ou de um processo, cujo propósito é auxiliar o seu entendimento. O modelo é o primeiro nível de abstração[5] utilizado pelos modeladores e como um modelo omite os detalhes não essenciais, seu entendimento e manipulação são mais fáceis que o sistema original (Rumbaugh et al., 1994).

A Figura 2.3 ilustra uma parte de um modelo orientado a objetos para um sistema de folha de pagamento, onde uma instância da classe Funcionario representa um

[5] Abstração é o exame seletivo de determinados aspectos de um problema. O objetivo da abstração é isolar os aspectos importantes para algum propósito e suprimir os que não forem (Rumbaugh et al., 1994).

funcionário da empresa. A classe Funcionario tem atributos com os dados de um funcionário, como matrícula, nome, salário, etc. Os métodos dessa classe são operações relacionadas com um funcionário da empresa como calcularINSS, calcularIRRF, calcularFerias, etc. A classe Funcionario se associa com a classe Sindicato e com a classe Departamento (Uma associação é representada por uma linha ligando duas classes). A multiplicidade[6] dessas associações é "um para muitos" (1:N), representado pelo numero 1 próximo às classes Sindicato e Departamento e pelo * próximo à classe Funcionario. Isso significa que uma instância da classe Sindicato pode se associar a muitas instâncias da classe Funcionario, mas uma instância da classe Funcionario só pode se associar a uma instância da classe Sindicato (Eriksson et al., 1998).

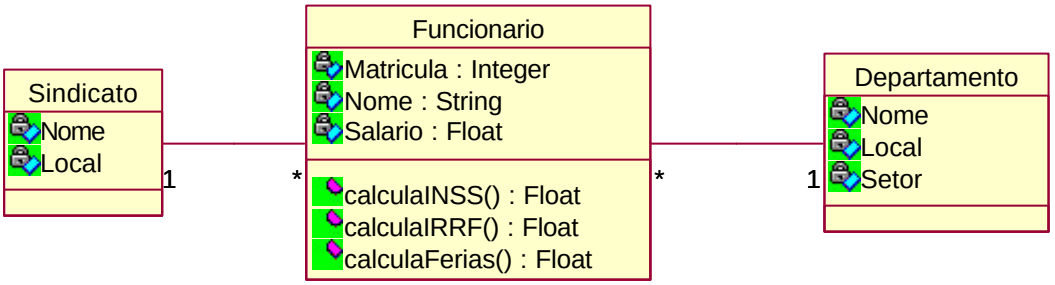


FIGURA 2.3 – Exemplo de um modelo.

Os metamodelos são o segundo nível de abstração utilizado pelos modeladores (Nordstrom et al., 1999) e descrevem formalmente os elementos de um modelo, a sintaxe e a semântica das notações que permitem suas manipulações (MOF, 2004). Um metamodelo é uma definição precisa das regras e construções necessárias para a criação de modelos (Estrella et al., 2001), (Yoder e Johnson, 2002), tais como definir que uma classe é composta de um nome, um conjunto de atributos e um conjunto de métodos, que todo atributo tem um tipo, que os métodos possuem um tipo de retorno, etc.

O Meta-metamodelo é o terceiro nível de abstração utilizado pelos modeladores (Nordstrom et al., 1999) e consiste de uma linguagem que define e expressa um metamodelo (Estrella et al.,2001).

[6] Multiplicidade especifica quantas instâncias de uma classe se relaciona a uma instância de uma classe associada (Rumbaugh et al., 1994)

Um metadado é um dado que descreve outro dado. Um metadado pode descrever a estrutura ou o significado desse outro dado. Por exemplo, considere a tabela Funcionario de um sistema de folha de pagamento, armazenada em banco de dados relacional. A descrição dessa tabela em um SGBDR informando que ela possui colunas como MATRICULA (tipo Inteiro) e NOME (tipo *string*), SALARIO (tipo real) é um exemplo de metadado (Riehle, 1999). No contexto deste trabalho os modelos são persistidos como metadados e estes dois termos podem ser utilizados com o mesmo significado (Poole, 2001).

2.5 Arquitetura de Metamodelagem da OMG

A arquitetura de metamodelagem da OMG é compreendida pelos seguintes padrões:

- *Unified Modeling Language* (UML): uma linguagem formal para determinar a estrutura e semântica de modelos orientados a objeto (Eriksson et al., 1998). A UML é um exemplo de metamodelo.

- *XML Metadata Interchange* (XMI): um mecanismo de intercâmbio de modelos e metamodelos definidos em UML usando XML (XMI, 2004).

- *Meta Objects Facility* (MOF): define interfaces e semânticas comuns, além de uma linguagem para a definição de metamodelos, facilitando a interoperabilidade dos mesmos. MOF é um exemplo de meta-metamodelo (MOF, 2004), (Auth et al., 2002).

- *Common Warehouse Metamodel* (CWM): define um metamodelo. O CWM é usado como base para o intercâmbio de modelos entre sistemas de *software* heterogêneos. O CWM é um metamodelo também (CWM, 2004).

A arquitetura de metamodelagem da OMG tem quatro níveis de abstração, chamados M0, M1, M2 e M3. A Tabela 2.2 resume estes níveis (Riehle et al., 2001).

- Nível M0: o nível dos objetos do usuário, também chamados de objetos do domínio, onde se encontram os objetos reais do sistema em tempo de

execução. Nesse nível, as informações consistem das propriedades das entidades de um determinado domínio.

- Nível M1: o nível dos modelos do usuário, onde residem por exemplo, as instâncias dos modelos UML desenvolvidos por modeladores UML. É nesse nível que ocorre a modelagem de sistemas. Um modelo do usuário descreve um domínio. Esse modelo é normalmente uma instância do metamodelo UML descrevendo a estrutura do nível M0.

- Nível M2: o nível dos metamodelos, o nível onde por exemplo o metamodelo UML é definido. Nesse nível, são definidos os conceitos utilizados por um modelador UML, além de regras para a construção de modelos.

- Nível M3: o nível dos meta-metamodelos, o nível mais alto de abstração em modelagem e uma instância dele próprio. Nesse nível se encontra a maioria dos elementos básicos nos quais a UML é baseada e esse nível define uma linguagem para a especificação de metamodelos.

TABELA 2.2 – Hierarquia de modelos de quatro níveis.

Nível Meta	Nível de Modelagem	Descrição	Exemplos
M3	Meta-Metamodelo/ Meta-Metametadado	Define a linguagem para a especificação de metamodelos.	Classe MOF, Atributo, Associação, Pacote, Operação
M2	Metamodelo / Meta-metadado	Uma instância de um meta-metamodelo. Define uma linguagem para a especificação de um modelo.	Classe UML, Atributo, Tabela CWM, Coluna
M1	Modelo/Metadado	Uma instância de um metamodelo. Define uma linguagem para descrever um domínio.	Produto : Tabela TipoProduto : Coluna
M0	Objeto/Dado	Uma instância de um modelo. Define um domínio específico.	"Torradeira" "Televisão" "Aparelho de Som"

FONTE: adaptada de Poole (2000).

Uma instância em um certo nível da arquitetura é sempre uma instância de algo definido em um nível superior, conforme ilustrado na Figura 2.4. Um objeto real em tempo de execução em M0 é uma instância de uma classe definida em M1. As classes definidas em modelos no nível M1 são instâncias do conceito de classes definidas em M2. O próprio metamodelo UML, definido em M2, é uma instância do MOF definido em M3. Outros metamodelos que definem outras linguagens de modelagem são também instâncias de M3 (Tan et al., 2001 e 2003).

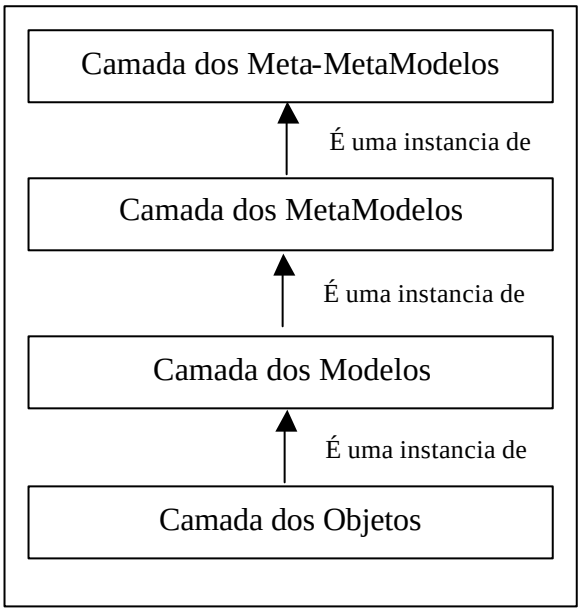


FIGURA 2.4 – Arquitetura de Modelagem da OMG.

FONTE: adaptada de Estrella(2001).

O Capítulo seguinte apresenta os Modelos de Objetos Adaptáveis e sua arquitetura. Esses modelos utilizam esses níveis de abstração resumidos na Tabela 2.2 (Yoder e Razavi, 2000). Sistemas desenvolvidos baseados nos Modelos de Objetos Adaptáveis permitem que seus usuários realizem, em tempo de execução, modificações no nível M1 também e não somente no nível M0 como ocorre na maioria dos sistemas(Poole, 2000).

Os Modelos de Objetos adaptáveis trabalham em um nível de abstração superior ao nível de abstração da maioria dos modelos. Usualmente, o desenvolvedor de sistemas

desenvolve um modelo para o domínio no nível M1 e fornece ao usuário condições de realizar alterações, em tempo de execução, no nível M0. Já em um sistema desenvolvido baseado nos AOMs o desenvolvedor de sistemas desenvolve um metamodelo para o domínio, no nível M2, e fornece ao usuário condições de realizar alterações em modelos criados a partir desse metamodelo, em tempo de execução, no nível M1. Pode-se verificar essas características dos AOMs nos Capítulos 5 e 6 deste trabalho.

CAPÍTULO 3

MODELOS DE OBJETOS ADAPTÁVEIS (AOM)

3.1 Introdução

Alguns fatores influenciam as metodologias utilizadas no desenvolvimento de *software*. Um desses fatores é a necessidade de se desenvolver programas os mais reutilizáveis possíveis. Outro fator importante é permitir que usuários possam tomar decisões de configuração do sistema em tempo de execução, o que se pode fazer deixando armazenado em uma base de dados algumas configurações do sistema (Foote e Yoder, 1998) (Nordstrom et al., 1999).

Usuários requerem sistemas que se adaptem a novos requisitos, gerados por mudanças do negócio após a implementação do sistema, de uma maneira mais rápida e a um custo menor. Usuários freqüentemente desejam mudar as regras do seu negócio sem a necessidade de alterar o código do sistema. Isso gera condições para a criação de novas formas de desenvolvimento de sistemas. Essas novas formas de desenvolvimento, podem ser utilizadas como uma alternativa à forma de desenvolvimento de sistemas onde as regras do negócio são embutidas no código (Yoder et al., 2001).

Atualmente muitos sistemas de informação necessitam ser adaptáveis e configuráveis. Precisam ser capazes de se adaptar a novas necessidades do seu domínio. Pode-se conseguir isso, movendo certos aspectos do sistema, tais como as regras do negócio, para um banco de dados facilitando a sua modificação. O modelo resultante permite a um sistema se adaptar a mudanças no negócio, através da mudança de valores no banco de dados ao invés de alterações no código.

A construção de sistemas adaptáveis e configuráveis encoraja o desenvolvimento de ferramentas que permitam aos usuários especialistas no domínio[7] realizarem mudanças no modelo do seu negócio, em tempo de execução, sem alterar o código do sistema. Dessa maneira pode-se por exemplo, reduzir o tempo necessário para se adaptar um sistema a uma nova idéia, um novo produto ou um novo serviço oferecido pelo negócio. Outra vantagem é dar aos usuários especialistas no domínio a capacidade de adaptar o sistema às necessidades do domínio (Yoder e Johnson, 2002).

Um sistema desenvolvido baseado nos AOMs, apresenta informações sobre seu modelo (suas classes, seus atributos e suas associações) como metadados. A construção do modelo de objetos do sistema é baseado nas instâncias das classes ao invés das classes. Quando ocorre alguma mudança no domínio do sistema, usuários especialistas no domínio modificam o metadado (modelo de objetos) para refletir essas mudanças, sem a necessidade de alterar o código que implementa o sistema. Essas mudanças modificam a estrutura (atributos de um objeto) e o comportamento (métodos e associações de um objeto) do sistema. Em outras palavras, o AOM armazena seu modelo de objetos em um banco de dados e em tempo de execução o interpreta. Conseqüentemente, o modelo de objetos é adaptável, pois quando o usuário especialista no domínio altera o modelo de objetos no banco de dados o sistema se adapta imediatamente refletindo a mudança realizada (Yoder et al., 2001).

A seguir apresenta-se alguns detalhes do estilo arquitetural AOM. Descreve-se os padrões de projeto mais comumente utilizados na arquitetura AOM e alguns detalhes do projeto de sistemas baseados nessa arquitetura.

3.2 Arquitetura dos AOMs

Os AOMs fornecem uma alternativa à maioria dos projetos orientados a objeto. Usualmente, nos projetos orientados a objeto cria-se uma classe diferente para representar cada tipo de entidade do negócio. Para cada classe criada define-se atributos,

[7] Usuários especialistas no domínio são usuários com amplo conhecimento do negócio e também com conhecimento de modelagem de sistemas ou apoiado por um profissional (ou equipe) de desenvolvimento de sistemas. Neste documento, quando for citado que o usuário pode realizar alterações no sistema em tempo de execução, refere-se ao usuário especialista no domínio e não ao usuário comum.

métodos e associações. As classes modelam o negócio, então qualquer mudança no negócio que ocorra após a implementação do sistema, por exemplo o surgimento de um novo tipo de entidade, resulta na criação ou no mínimo na alteração de uma classe, o que significa recodificar o sistema e liberar uma nova versão da aplicação (Yoder et al., 2001).

Um AOM não modela as entidades do negócio como classes. Em um AOM as entidades são modeladas por descrições (metadados) que são interpretados em tempo de execução. Assim, sempre que ocorrer uma mudança no negócio, pode-se modificar estas descrições refletindo imediatamente na aplicação.

A arquitetura de um AOM é uma composição de padrões de projeto menores (Riehle et al., 2000):

- Tipo Objeto (*Type Object*) (Johnson e Woolf, 1998).
- Propriedade (*Property*) (Foote e Yoder, 1998).
- Estratégia (*Strategy*) (Gamma et al.,1995).
- Regra Objeto (*Rule Object*) (Arsanjani, 2000 e 2001).
- Responsabilidade (*Accountability*) (Fowler, 1997).
- Composição (*Composite*) (Gamma et al.,1995).
- Intérprete (*Interpreter*) (Gamma et al.,1995).
- Construtor (*Builder*) (Gamma et al.,1995).

3.2.1 O Padrão de Projeto Tipo Objeto

As linguagens orientadas a objeto implementam um programa através de um conjunto de classes, onde uma classe define a estrutura e o comportamento de um tipo de objeto do programa.

Durante o desenvolvimento de um sistema pode ocorrer o fato de uma classe, que represente algum objeto do negócio, possuir um número de subclasses desconhecido ou muito grande. Isso se torna uma dificuldade para os desenvolvedores, pois embora uma classe possa criar novas instâncias de acordo com a necessidade, em tempo de execução, normalmente não se pode criar novas classes (ou subclasses) dessa classe sem a necessidade de reescrever o código e recompilar a aplicação. O padrão de projeto Tipo Objeto oferece uma solução para esse problema, substituindo uma classe com um desconhecido número de subclasses por uma classe com um desconhecido número de instâncias (Johnson e Woolf, 1998).

O padrão de projeto Tipo Objeto faz com que subclasses ainda desconhecidas (subclasses que podem surgir após a implementação do sistema) ou em número bastante elevado de uma classe, se tornem simples instâncias de uma classe genérica, como ilustrado na Figura 3.1. Dessa forma, pode-se criar novas classes em tempo de execução através da instanciação de uma classe genérica. O uso do padrão de projeto Tipo Objeto é muito eficiente no desenvolvimento de sistemas onde os tipos dos objetos possuem um comportamento semelhante e as diferenças entre um tipo de objeto e outro ocorrem primariamente na quantidade e nos tipos de seus atributos (Yoder et al., 2001).

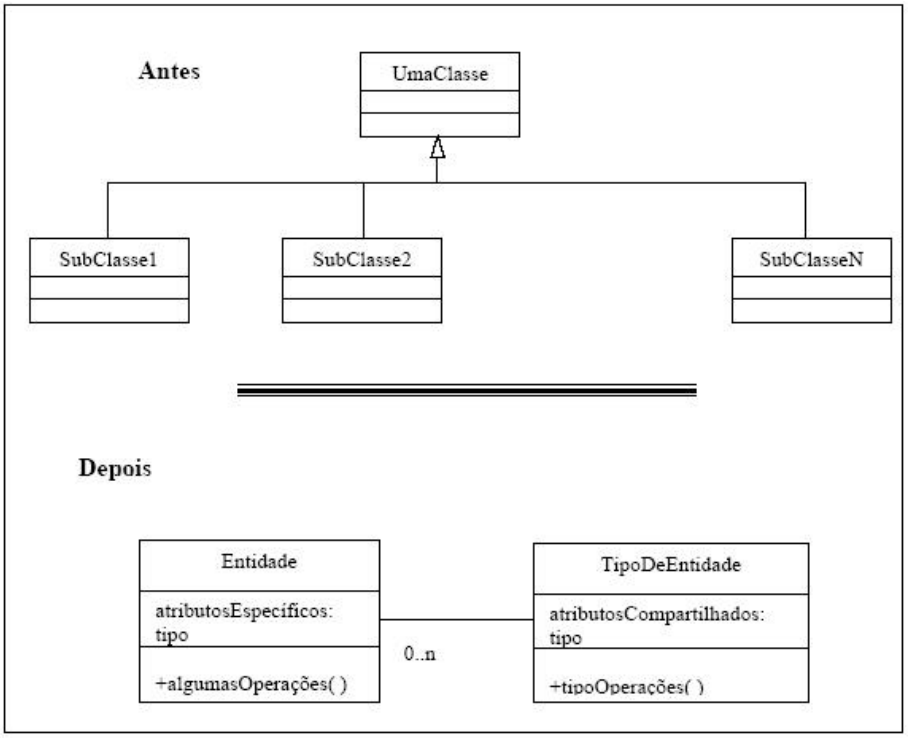


FIGURA 3.1 – O padrão de projeto Tipo Objeto.

FONTE: adaptada de Yoder et al. (2001).

3.2.2 O Padrão de Projeto Propriedade

Os atributos de uma classe são comumente implementados por variáveis de instância dessa classe. Normalmente, essas variáveis de instância são definidas internamente em uma classe. Porém, com a utilização do padrão de projeto Tipo Objeto, objetos de tipos diferentes, com atributos diferentes, pertencerão todos à mesma classe (são instâncias da mesma classe). Isso gera a necessidade de se adotar uma forma alternativa para implementar os atributos de uma classe: o padrão de projeto Propriedade (Foote e Yoder, 1998).

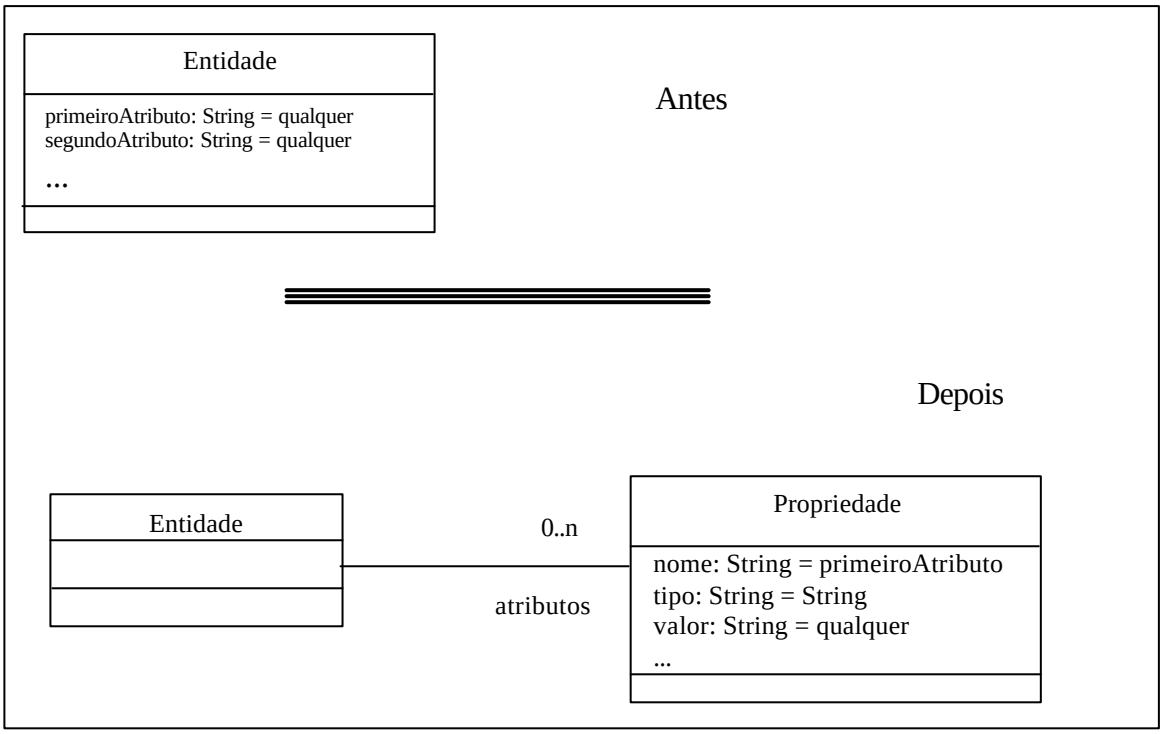


FIGURA 3.2 – O padrão de projeto Propriedade.
FONTE: adaptada de Yoder et al. (2001).

Na maioria das arquiteturas de modelos de objetos adaptáveis o uso do padrão de projeto Propriedade implica em uma segunda aplicação do padrão de projeto Tipo Objeto. Em sua primeira aplicação o padrão de projeto Tipo Objeto divide o sistema em entidades e tipos de entidades. As entidades possuem atributos que podem ser definidos usando-se o padrão de projeto Propriedade. Como cada propriedade[8] possui um tipo, aplica-se o padrão de projeto Tipo Objeto uma segunda vez para especificar os tipos dos atributos para suas entidades, dividindo a classe Propriedade em Propriedade e TipoPropriedade.

Pode-se implementar os atributos de uma classe da seguinte forma: ao invés de usar uma variável de instância diferente para cada atributo deve-se fazer com que uma

[8] Atributos e propriedades de uma classe possuem o mesmo significado. Este trabalho utilizou o termo atributo de uma classe com seu significado tradicional, mas utilizou o termo propriedade de uma classe para significar propriedades criadas em tempo de execução pelo padrão de projeto Propriedade. O mesmo ocorre com método e regra. Um método é a implementação de uma função ou transformação em uma classe (Rumbaugh et al.1994) e neste trabalho o termo regra foi utilizado para significar métodos criados ou associados em tempo de execução.

variável de instância englobe uma coleção de atributos, como ilustrado na Figura 3.2. Nesta figura, a classe Propriedade armazena o nome do atributo, seu tipo e seu valor corrente (Yoder et al., 2001).

A Figura 3.3 representa a arquitetura *TypeSquare* que é resultante da aplicação desses dois padrões de projeto. Um modelo de objetos baseado nessa arquitetura pode ser utilizado na implementação de sistemas onde diferentes tipos de objetos diferem apenas em seus atributos, mas possuem o mesmo comportamento (Yoder et al., 2001).

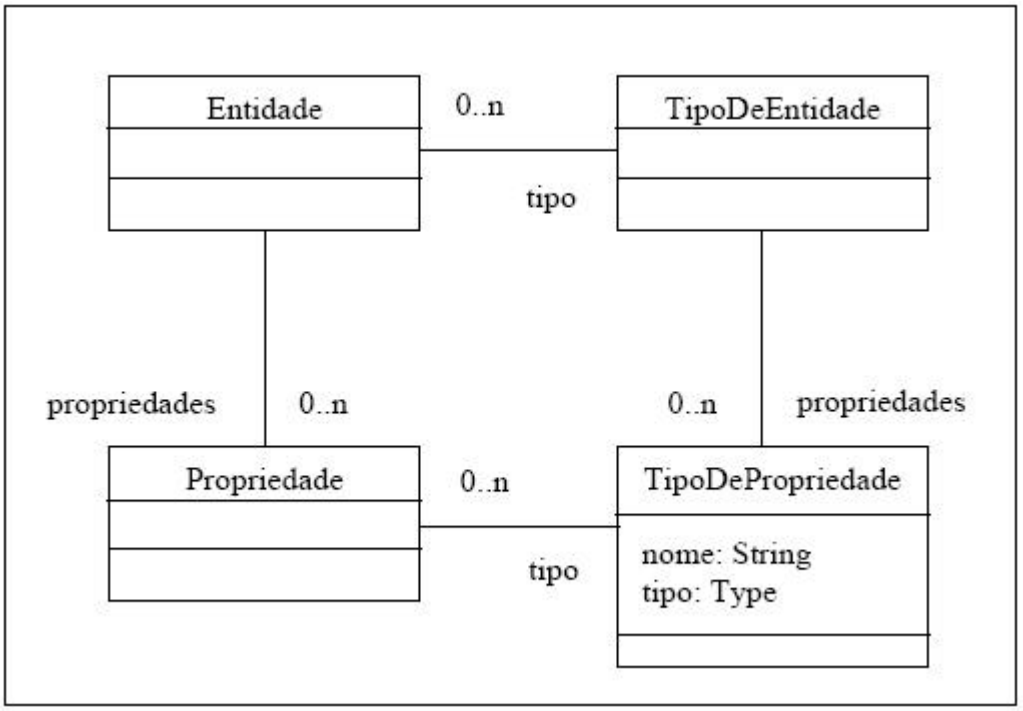


FIGURA 3.3 – A arquitetura *TypeSquare.*

FONTE: adaptada de Yoder et al. (2001).

Por exemplo, no projeto de um sistema com a função de apenas ler e escrever dados em todas as tabelas de um banco de dados relacional, onde cada tabela possui uma estrutura diferente (com diferentes colunas). Esse sistema pode usar uma instância da classe Entidade associada com um conjunto de instâncias da classe Propriedade para representar uma linha da tabela. Pode também usar uma instância da classe TipoDeEntidade associada com um conjunto de instâncias da classe TipoDePropriedade para representar uma tabela. Dessa forma a classe Entidade pode manipular os dados de uma linha de qualquer uma das tabelas do banco de dados.

Objetos de tipos diferentes normalmente também possuem comportamentos diferentes. Voltando ao exemplo anterior, talvez exista a necessidade de checar a consistência dos valores das propriedades das Entidades antes de realizar o armazenamento das mesmas em uma tabela do banco de dados. Pode-se ter checagens diferentes de uma tabela para a outra, além de checagens com diferentes graus de complexidade.

O comportamento de um objeto comumente é implementado por seus métodos. Assim, se dois objetos possuem comportamentos diferentes, significa que seus comportamentos são implementados por métodos diferentes. Na maioria dos projetos orientados a objeto esses objetos são implementados por classes diferentes, cada uma com seus próprios métodos. Assim, a utilização apenas dos padrões de projeto Tipo Objeto e Propriedade não é suficiente para eliminar a necessidade de subclasses. Um AOM utiliza outros padrões de projeto para descrever e permitir a alteração do comportamento dos objetos (Yoder et al., 2001).

3.2.3 O Padrão de Projeto Estratégia

Uma estratégia é um objeto que representa um algoritmo. O padrão de projeto Estratégia define uma interface padrão para um conjunto de algoritmos para que objetos possam trabalhar com qualquer um desses algoritmos. Se o comportamento de um objeto é definido por uma ou mais estratégias então se consegue modificar o comportamento desse objeto através da escolha de um dos algoritmos representados por essas estratégias.

Cada aplicação do padrão de projeto Estratégia leva a uma interface diferente e assim a uma diferente hierarquia de classes de Estratégias. Voltando ao exemplo do sistema de banco de dados da Seção anterior, pode-se associar Estratégias com cada propriedade e usar as Estratégias para validar as propriedades. As Estratégias então teriam uma operação pública, por exemplo validar(). Freqüentemente, associam-se as Estratégias com as entidades modeladas do domínio, onde as Estratégias implementam as operações nos métodos das classes que representam essas entidades.

Essas Estratégias podem se desenvolver para implementar regras do negócio mais complexas que são construídas ou interpretadas em tempo de execução. Essas regras podem ser primitivas ou combinação de regras do negócio implementadas através da aplicação do padrão de projeto Composição.

A Figura 3.4 é um diagrama UML da arquitetura *TypeSquare* com a adição dos padrões de projeto Estratégias ou Regras Objeto para representar o comportamento do sistema (Yoder et al., 2001).

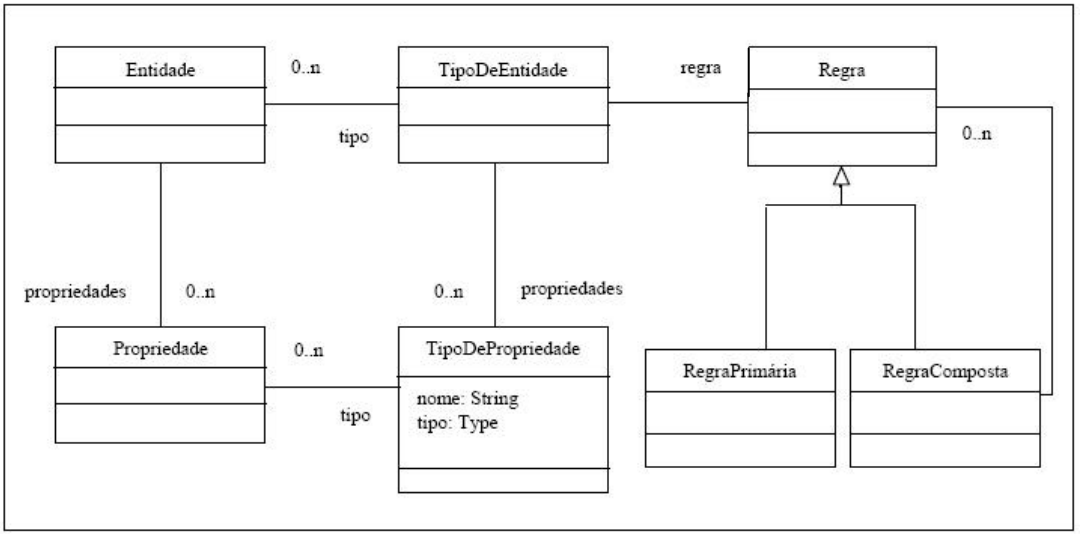


FIGURA 3.4 – A arquitetura *TypeSquare* com regras.
FONTE: adaptada de Yoder et al. (2001).

3.2.4 Padrão de Projeto Responsabilidade

Normalmente, objetos de tipos diferentes possuem tipos de associações diferentes. Assim, um modelo de objetos adaptável precisa, de alguma maneira, fornecer a seus usuários uma descrição dessas associações e permitir que os usuários alterem uma associação entre os objetos do seu modelo através da modificação dessas descrições. Ou seja, a aplicação necessita expor as associações entre os elementos do domínio (entidades), pois cada associação estabelece papéis aos participantes. Os papéis podem se alterar e o sistema precisa saber quais são os papéis utilizados em um determinado momento (Yoder,2003).

Em um sistema de informação, atributos são propriedades de uma entidade do negócio que comumente se referem a tipos de dados primitivos como números, *strings* ou datas. Entidades normalmente têm associações unidirecionais com seus atributos. Pode-se considerar as associações porém, como propriedades que se referem a outras entidades. As associações são normalmente bidirecionais (Por exemplo, se a classe A se associa com a classe B, então a classe B também se associa com a classe A).

Um modo de separar atributos de associações é usar o padrão de projeto Propriedade duas vezes, uma vez para atributos e uma vez para associações. A representação dessas associações como objetos permite a manipulação das mesmas em tempo de execução, além de possibilitar a sua persistência. Dessa maneira, um usuário especialista no domínio pode, devido à mudanças nos requisitos do negócio, adaptar as associações entre entidades do sistema em tempo de execução (Yoder et al., 2001).

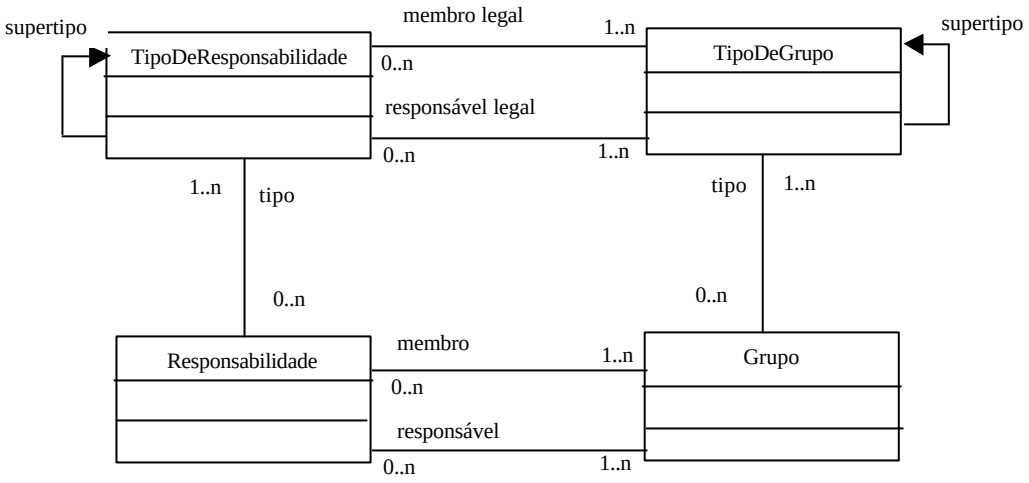


FIGURA 3.5 – O padrão de projeto Responsabilidade.
FONTE: adaptada de Yoder(2003).

Como ilustra a Figura 3.5, através do padrão de projeto Responsabilidade pode-se manipular muitas associações entre grupos. Responsabilidades podem ser organizadas em tipos, de forma que cada tipo de responsabilidade conheça os tipos de grupos associados a ela e onde cada tipo de grupo, por sua vez, conheça os membros de seu grupo (Fowler, 1997).

3.2.5 Interpretadores dos Metadados

Em um sistema desenvolvido baseado nos AOMs, seu modelo de objetos é armazenado em algum sistema de gerenciamento de dados como metadados. Na utilização do sistema, esses metadados devem ser interpretados em dois momentos: (1) quando o usuário inicia a aplicação, logo precisa-se instanciar o modelo de objetos e (2) durante a execução do sistema sempre que os metadados que representam o modelo de objetos for alterado por um usuário especialista no domínio (Yoder et al., 2001).

Isso significa que, independentemente de como as informações sobre os tipos de entidade, propriedades, associações e comportamento do domínio, como metadados, forem armazenadas, essas informações precisam ser interpretadas para a execução do sistema. Para que se possa, em um primeiro momento, montar o modelo de objetos adaptável que representa o modelo do domínio.

Além disso, sempre que o usuário especialista no domínio criar um novo tipo de objeto, uma nova propriedade, uma nova associação ou um novo método ou ainda quando alterar os já existentes, deve-se interpretar os metadados novamente para refletir as mudanças realizadas na estrutura e nas regras do negócio no sistema.

3.3 Aplicação dos AOMs

Deve-se considerar os modelos de objetos adaptáveis como uma boa alternativa para se construir um sistema quando (Riehle et al., 2000):

- Existem muitos tipos de objetos no domínio do sistema e esses tipos de objetos se diferenciam somente em alguns atributos e se deseja reduzir o número de classes do sistema.

- No domínio do sistema existe a necessidade da criação de novos tipos de objetos com atributos diferentes em tempo de execução.

- Deseja-se construir aplicações que permitam que os usuários finais configurem muitos tipos de objetos.

- Ocorre modificações freqüentes no domínio e o usuário requer evolução rápida do seu sistema de informação.

- Necessita-se de um modelo explícito da estrutura dos objetos, para que se possa acessar esse modelo em tempo de execução.

- Deseja-se fornecer validação de tipos em tempo de execução em linguagens tipadas dinamicamente.

- Deseja-se uma linguagem de modelagem específica para o domínio.

3.4 Projeto de AOMs

Além das etapas de definição e estudo de viabilidade, análise de requisitos, análise, projeto, implementação e teste de sistemas, o desenvolvimento de um sistema baseado na arquitetura AOM acrescenta três atividades: (1) definir as entidades, regras e associações do domínio; (2) desenvolver mecanismos para instanciação e manipulação dessas entidades de acordo com suas regras e associações; e (3) desenvolver ferramentas que permitam aos usuários especialistas no domínio modificarem a descrição dessas entidades, regras e associações de acordo com as necessidades do domínio (Yoder et al., 2001).

Essas atividades são alcançadas pela aplicação de um ou mais dos padrões de projeto previamente mencionados em conjunção com outros padrões de projeto tais como Composição, Intérprete, Construtor (Gamma et al., 1995) e Regra Objeto (Arsanjani, 2000 e 2001).

3.5 Aspectos de Implementação

Os aspectos primários de implementação que precisam ser abordados no desenvolvimento de modelos de objetos adaptáveis são: (1) como armazenar o modelo de objetos no banco de dados, (2) como apresentar o modelo de objetos para o usuário, (3) como dar ao usuário condições de realizar alterações no modelo de objetos em tempo de execução e (4) como lidar com o histórico das alterações realizadas pelo

usuário no modelo de objetos (Yoder e Johnson, 2002). A seguir, aborda-se cada um desses aspectos mais detalhadamente.

3.5.1 Tornando o Modelo de Objetos Persistente

Os AOMs armazenam o modelo de objetos como metadados. Para se manejar a persistência dos metadados, pode-se utilizar bancos de dados orientados a objeto ou banco de dados relacionais. Pode-se também armazenar os metadados em arquivos XML ou XMI.

Independente do sistema de gerenciamento de dados escolhido para o armazenamento dos metadados o sistema deve ter a capacidade de ler os metadados e instanciar o modelo de objetos adaptável com a configuração correta de suas instâncias. Isso pode ser conseguido usando-se construtores (utilizando o padrão de projeto Construtor, por exemplo) e interpretadores dos metadados para a criação e configuração das entidades, atributos, associações e comportamentos a partir dos metadados.

A Figura 3.6 ilustra que o modelo de objetos adaptável pode ficar armazenado tanto em um banco de dados, quanto em arquivos XML. Independentemente da forma como o modelo de objetos for armazenado ele deve ser instanciado e ficar disponível no repositório de metadados para que possa ser utilizado pela aplicação. No caso de se utilizar um banco de dados para armazenar o modelo de objetos o mecanismo de persistência e o interpretador de metadados realizam essas tarefas; e no caso do modelo de objetos ficar armazenado em arquivos XML, o XML *Parser* e o interpretador de metadados realizam essas tarefas. Se houver a necessidade de persistir os objetos do domínio da aplicação, eles podem ser armazenados em um banco de dados através do mecanismo de persistência.

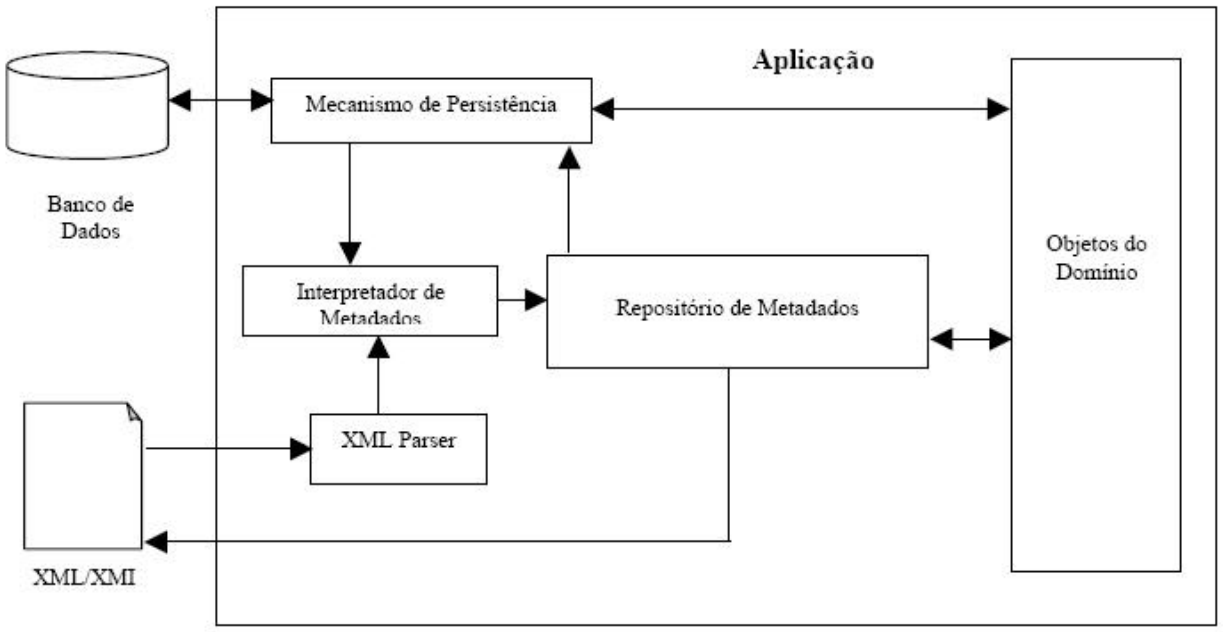


FIGURA 3.6 – Armazenando e recuperando metadados.
FONTE: adaptada de Yoder et al. (2001).

Alterações realizadas no modelo de objetos refletem diretamente no comportamento do *software*, resultando em uma extensão completa em tempo de execução. Por exemplo, alterar uma classe do sistema e armazenar a nova versão dessa classe faz com que todos os sistemas de *software* no ambiente automaticamente utilizem a nova versão dessa classe. Assim, pode-se atualizar os metadados durante a execução do sistema e as mudanças resultantes no comportamento e estrutura do sistema têm efeito assim que o modelo de objetos é interpretado (Poole, 2001).

Para armazenar o modelo de objetos, como metadados, em um banco de dados ou em arquivos XML, os desenvolvedores têm que definir como representar o modelo, bem como a semântica para descrever as regras do negócio. Deve-se estabelecer um mapeamento do modelo de objetos para um banco de dados, ou para arquivos XML. Além disso, deve-se desenvolver editores de metadados e ferramentas para auxiliar os usuários especialistas no domínio nas tarefas de criação e manutenção do modelo de objetos (Yoder e Johnson, 2002).

3.5.2 Apresentando o Modelo de Objetos para os Usuários

Permitir que usuários especialistas no domínio adaptem a estrutura e o comportamento do sistema ao surgimento de novos requisitos no domínio, sem a necessidade de modificar o código do sistema é uma das principais razões para se desenvolver um sistema baseado nos AOMs. Pode-se, em tempo de execução, adaptar o sistema definindo novas entidades, associações e regras ou alterando as já existentes.

Para os usuários especialistas no domínio realizarem adaptações com uma menor probabilidade de introdução de erros no sistema, e de uma maneira mais intuitiva, é preciso oferecer ferramentas adequadas a esses usuários para que eles possam realizar essas tarefas de uma maneira mais eficiente. Conseqüentemente, é essencial para um sistema adaptável possuir ferramentas para a definição de novas entidades, associações e regras, de forma interativa, e ferramentas para apresentar o modelo de objetos a esses usuários, permitindo a modificação de entidades, associações e regras existentes (Yoder et al., 2001).

3.5.3 Histórico do Modelo de Objetos

Um aspecto de implementação importante é como lidar com o histórico das alterações realizadas no modelo de objetos pelos usuários especialistas no domínio, tanto na estrutura (atributos dos objetos), quanto no comportamento (métodos dos objetos). Um outro aspecto de implementação encontrado é como fazer um controle de versões dos metadados do modelo de objetos.

Uma forma de se lidar com o histórico de alterações na estrutura do modelo de objetos é armazenar todas as alterações realizadas ao longo do tempo. Além disso, deve-se criar alguma maneira de identificar uma versão do modelo de objetos e fazer com que o interpretador de metadados use sempre a versão correta quando estiver lendo o modelo de objetos. As regras podem ser tratadas similarmente, incluindo-se a informação do histórico de cada regra e fazendo com que o interpretador de metadados sempre use a versão correta das regras. Armazenar o histórico das alterações realizadas no modelo de objetos é especialmente importante, quando se tem a necessidade de retornar o modelo

de objetos a uma condição anterior. A habilidade de voltar o modelo de objetos a uma condição anterior pode aumentar a segurança do sistema e facilitar a correção da introdução de erros no sistema por mudanças indevidas no modelo de objetos, feitas pelos usuários especialistas no domínio (Yoder e Johnson, 2002).

Um dos objetivos deste trabalho é lidar com dois aspectos de implementação de modelos de objetos adaptáveis apresentados nesta Seção 3.4: como tornar o modelo de objetos persistente e como apresentá-lo ao usuário. Apesar da computação genérica e dos modelos reflexivos serem utilizados há algum tempo, os modelos de objetos adaptáveis ainda são pouco conhecidos e apresentam portanto, um grande potencial para ser explorado, especialmente quando aliados ao conceito de distribuição. Exemplos de aplicações já desenvolvidas usando-se os AOMs podem ser encontrados em (Yoder et al., 2001), (Manolescu, 2001), (Yoder e Johnson, 2002) e (Souza, 2004).

Além disso, as referências citadas no parágrafo anterior não tratam diretamente a questão da persistência dos modelos de objetos adaptáveis criados por usuários especialistas do domínio. Não citam também qual sistema de gerenciamento de dados utilizar e como representar o modelo de objetos como metadados nesses sistemas de gerenciamento de dados. Nessas referências os modelos de objetos são tratados na memória e não são persistidos, o que permitiria manter as configurações do modelo de objetos. Este trabalho espera dar um passo na direção de facilitar a persistência dos modelos de objetos adaptáveis.

Esta dissertação estabelece uma abordagem para persistir os modelos de objetos adaptáveis em três sistemas de gerenciamentos de dados: um SGBD Orientado a Objetos, um SGBD Relacional e um NXD. Além de facilitar a manutenção desses modelos de objetos através da criação de protótipos para a alteração desses modelos. A origem desta dissertação foi a proposta de Thomé(2004) para uma arquitetura distribuída configurável e adaptável aplicada ao controle de satélites. Sua arquitetura foi utilizada como estudo de caso para este trabalho. A Seção seguinte apresenta uma breve descrição dessa arquitetura, além de fornecer informações sobre o domínio do problema dos três protótipos desenvolvidos.

3.6 Proposta de Thomé

3.6.1 Introdução

Thomé(2004) propõe uma arquitetura distribuída e reflexiva baseada nos modelos de objetos adaptáveis (SICSDA) para o Sistema de Rastreio e Controle de Satélites aplicado a múltiplas missões de satélites.

O *software* para o controle de satélites utilizado pelo INPE desempenha as funções listadas a seguir:

- Receber, em tempo real, das estações terrenas, os dados de telemetria[9], contendo informações sobre a orientação do satélite no espaço (atitude), temperaturas e parâmetros funcionais dos equipamentos de bordo do satélite, processá-los e arquivá-los. Essa função permite às equipes de controle de solo monitorar continuamente a atitude e o estado de funcionamento do satélite.

- Receber das estações e arquivar os dados de localização do satélite (medidas de distância ou angulares).

- Gerar e transmitir às estações terrenas telecomandos[10] que quando irradiados pelas estações ao satélite são recebidos e executados por seus sistemas de bordo. Isso permite que se atue a partir do solo no satélite para a reconfiguração do estado de funcionamento de seus equipamentos, ou execução de manobras de controle de atitude ou órbita.

- Monitorar o estado de funcionamento dos equipamentos residentes nas estações terrenas.

[9] A telemetria encapsula o estado interno do satélite, como por exemplo: a voltagem de um determinado circuito, o posicionamento de uma determinada chave (ligada ou desligada), etc.
[10] Os telecomandos são enviados das estações terrenas para os satélites.

As estações terrenas funcionam como um elo de ligação entre o Centro de Controle de Satélites (CCS) e o satélite em órbita. Suas atividades básicas são:

- Adquirir o sinal do satélite e segui-lo durante sua passagem sobre a estação.

- Extrair do sinal recebido os dados do estado dos equipamentos de bordo, datá-los e encaminhá-los ao CCS.

- Irradiar para o satélite os telecomandos recebidos do CCS no horário determinado.

- Efetuar as medidas de localização (distância do satélite até a estação e ângulos de azimute e de elevação do satélite em relação à estação), datá-las e encaminhá-las ao CCS para seu processamento, possibilitando assim, a determinação da órbita do satélite.

3.6.2 Arquitetura SICSDA

Na arquitetura SICSDA, os objetos do domínio do problema, como por exemplo: a telemetria, o telecomando e o *ranging*[11], ao invés de estarem localizados no código que implementa a aplicação, ficam armazenados em uma base de dados, como metadados, para que a aplicação possa instanciar esses objetos. Em tempo de execução é criado um modelo de objetos para cada satélite, adaptando cada modelo às características do satélite representado por esse modelo. Isso significa que a aplicação tem um interpretador de metadados capaz de instanciar os diferentes modelos de objetos dos vários satélites.

[11] *Ranging* encapsula as medidas de distância do satélite em relação à terra.

Outro fator importante considerado na arquitetura proposta diz respeito à organização dos objetos. Para esse tipo de aplicação é interessante que se utilize o conceito de objetos distribuídos, principalmente por questões de tolerância a falhas.

Então, propôs-se uma arquitetura distribuída onde as funcionalidades oferecidas pela aplicação – por exemplo, visualização de telemetria e emissão de telecomando – ficam distribuídas dentro de um domínio de rede predefinido. Portanto, pode-se instanciar os objetos da aplicação em máquinas diferentes na rede, o que ocasiona uma distribuição no código do sistema. A arquitetura SICSDA define a localização desses objetos informando em que máquina da rede eles estão disponíveis.

Assim, cada máquina da rede pode ter uma visão diferente do modelo de objetos. Entende-se por visão, nesse contexto, como a parte do modelo de objetos adaptável que é instanciado naquela máquina. Isso significa que uma determinada máquina da rede tem acesso apenas à parte do modelo de objetos adaptável que lhe for permitido.

A arquitetura possibilita também o uso de bancos de dados distribuídos através da replicação do banco de dados para um ou mais nós da rede. A distribuição torna necessário que cada nó contenha também um interpretador para os metadados, para que se possa instanciar o modelo de objetos adaptável e para que o sistema consiga refletir as mudanças realizadas no modelo de objetos em tempo de execução. Deve-se lembrar também da necessidade da existência de mecanismos que auxiliem o armazenamento do histórico de alterações nos metadados e o controle de versões.

Pode-se dizer que a arquitetura proposta é configurável em três aspectos: (1) primeiramente, porque os usuários especialistas do domínio e os desenvolvedores podem, em tempo de execução, reconfigurar os objetos do sistema, redefinindo por exemplo, tipos de atributos e até mesmo regras do negócio, sem a necessidade de alterar o código do sistema; (2) segundo, porque usuários especialistas do domínio e desenvolvedores podem reconfigurar o sistema para acomodar novas classes através da criação, em tempo de execução, dessas classes com seus atributos e métodos; e (3) porque pode-se, em tempo de execução, alternar entre os metadados dos modelos de

objetos de diversos satélites, permitindo que se instancie um modelo de objetos diferente cada vez o usuário deseje controlar outro satélite.

Finalmente, a arquitetura é adaptável porque possui a capacidade de acomodar possíveis mudanças no domínio do sistema, permitindo que se possa acompanhar a evolução dos requisitos do negócio, e adaptando o sistema às necessidades dos usuários. A Figura 3.7 ilustra a estrutura da arquitetura SICSDA.

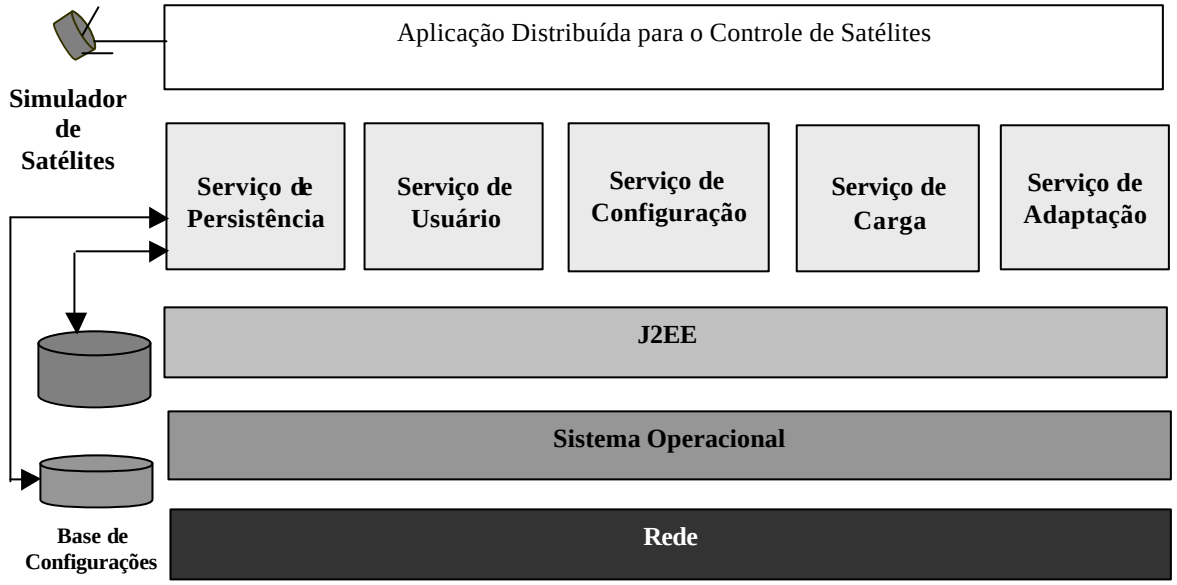


FIGURA 3.7 –Arquitetura SICSDA.

FONTE: Thomé (2004).

- **Aplicação Distribuída para o Controle de Satélites:** contém os objetos do *software* que realiza o controle dos satélites (telecomando, *ranging* e telemetria). O objeto Telemetria possui métodos, tais como processar telemetria, visualizar telemetria, etc. O objeto Telecomando possui métodos, tais como enviar telecomandos, visualizar telecomandos, etc. O objeto *Ranging* possui métodos responsáveis pelo cálculo da distância do satélite à Terra. A aplicação foi desenvolvida baseada nos AOMs e os metadados do modelo de objetos ficam armazenados em um banco de dados.

- **Serviço de Persistência**: responsável por armazenar e recuperar do banco de dados os modelos de objetos do sistema.

- **Serviço de Configuração**: responsável por manter os modelos de objetos e montar as visões dos objetos distribuídos.

- **Simulador de Satélite**: *software* que simula a interação com o satélite, simulando a chegada e envio de dados do e para o satélite.

- **Serviço de Carga**: responsável por carregar os modelos de objetos dos satélites, necessitando para isso saber qual satélite precisa ser monitorado em um determinado instante.

- **Serviço de Usuário**: responsável por oferecer uma interface apropriada ao usuário que utiliza a aplicação de controle de satélites.

- **Serviço de Conexão**: responsável por localizar um objeto dentro da rede.

Esta dissertação contribui também para a viabilizar a arquitetura proposta por Thomé(2004). No estudo de caso desta dissertação criou-se protótipos que realizam as funções que a arquitetura SICSDA designou ao elemento serviço de configuração, facilitando o desenvolvimento de um sistema baseado nessa arquitetura. A seguir apresenta-se algumas funções relacionadas ao elemento serviço de configuração, que foi

uma das motivações deste trabalho e algumas de suas funções foram desenvolvidas nos Capítulos seguintes.

3.6.3 Funções Relacionadas com o Serviço de Configuração

O serviço de carga do sistema é responsável por carregar o modelo de objetos do satélite que se deseja monitorar em um determinado instante. Para tal ele aciona o serviço de configuração, o responsável por montar as visões dos objetos distribuídos em cada nó para aquele determinado satélite.

Quando for necessário controlar um determinado satélite o serviço de configuração aciona o serviço de persistência, o responsável por recuperar os metadados do modelo de objetos desse satélite. Com os metadados, o serviço de configuração instancia o modelo de objetos do satélite em questão.

O Simulador de Satélites fornece os dados referentes ao satélite. Os objetos da aplicação ficam distribuídos e o serviço de conexão é o responsável por localizar um determinado objeto no sistema. Quando o usuário deseja monitorar outro satélite ele informa ao sistema e este aciona o serviço de carga, que imediatamente recupera os metadados, utilizando o serviço de persistência, e instancia o modelo de objetos desse satélite, utilizando o serviço de configuração.

Quando for necessária uma alteração no modelo de objetos para refletir alguma mudança no sistema, o usuário especialista no domínio pode realizar tais alterações através do serviço de configuração, que fornece a esse usuário uma interface apropriada para essa tarefa. Quando essas modificações forem realizadas o serviço de configuração deve ser acionado novamente, pois ele é responsável pela manutenção do modelo de objetos. Para fazer isso ele aciona o serviço de persistência, que imediatamente reflete na base de dados tais alterações.

Os protótipos implementados para cada um dos sistemas de gerenciamento de dados escolhidos: um SGBDOO, um SGBDR e um NXD, foram desenvolvidos com o

propósito de exercer as funções que a arquitetura SICSDA designou ao elemento serviço de configuração.

O Capítulo seguinte mostra um modelo para o controle de satélites e a aplicação dos padrões de projeto, que compõem os AOMs, nesse modelo até a criação de um metamodelo que foi utilizado como estudo de caso para esta dissertação.

CAPÍTULO 4

A CONSTRUÇÃO DO METAMODELO

O desenvolvimento de um sistema baseado na arquitetura dos AOMs envolve a criação de um metamodelo. Esse metamodelo define como os modelos específicos para o domínio podem ser criados em tempo de execução pelo usuário especialista no domínio. Resumidamente, esse metamodelo determina um padrão para a criação de modelos. Este Capítulo mostra a criação de um metamodelo para um sistema de controle de satélites, explicando a utilização dos padrões de projeto e as modificações realizadas desde o modelo inicial até se chegar ao metamodelo. Esse metamodelo foi utilizado como estudo de caso nos Capítulos seguintes.

O metamodelo gerado deve ser implementado através de uma linguagem de programação e funciona como uma máquina instanciadora de objetos, ou seja, ele é capaz de: instanciar os metadados armazenados no banco de dados e refletir mudanças feitas nos metadados, o que significa que se o domínio mudar as mudanças podem ser refletidas na aplicação através da alteração dos metadados. Com isso não se faz necessário realizar o processo de geração de código sempre que mudanças acontecerem no domínio, já que as alterações podem ser realizadas nos modelos armazenados no banco de dados através do metamodelo implementado.

A Figura 4.1 mostra o modelo inicial do domínio de um sistema para o controle de satélites. Partindo desse modelo, através da aplicação dos padrões de projeto que compõem a arquitetura AOM, apresentados no Capítulo anterior, chegou-se ao metamodelo, ilustrado na Figura 4.9, de um sistema para o controle de satélites baseado na arquitetura dos AOMs. O processo de transformar um modelo usando padrões de projeto é chamado *refactoring* baseado em padrões. Pode-se conseguir este processo através dos seguintes passos (France, 2003):

- Identificar os elementos do modelo que irão participar de uma determinada solução através de um determinado padrão de projeto. Modificá-los para que eles possam se comportar como o padrão de projeto especificado.

- Adicionar novos elementos ao modelo à medida que forem necessários para a realização do comportamento especificado no padrão de projeto.

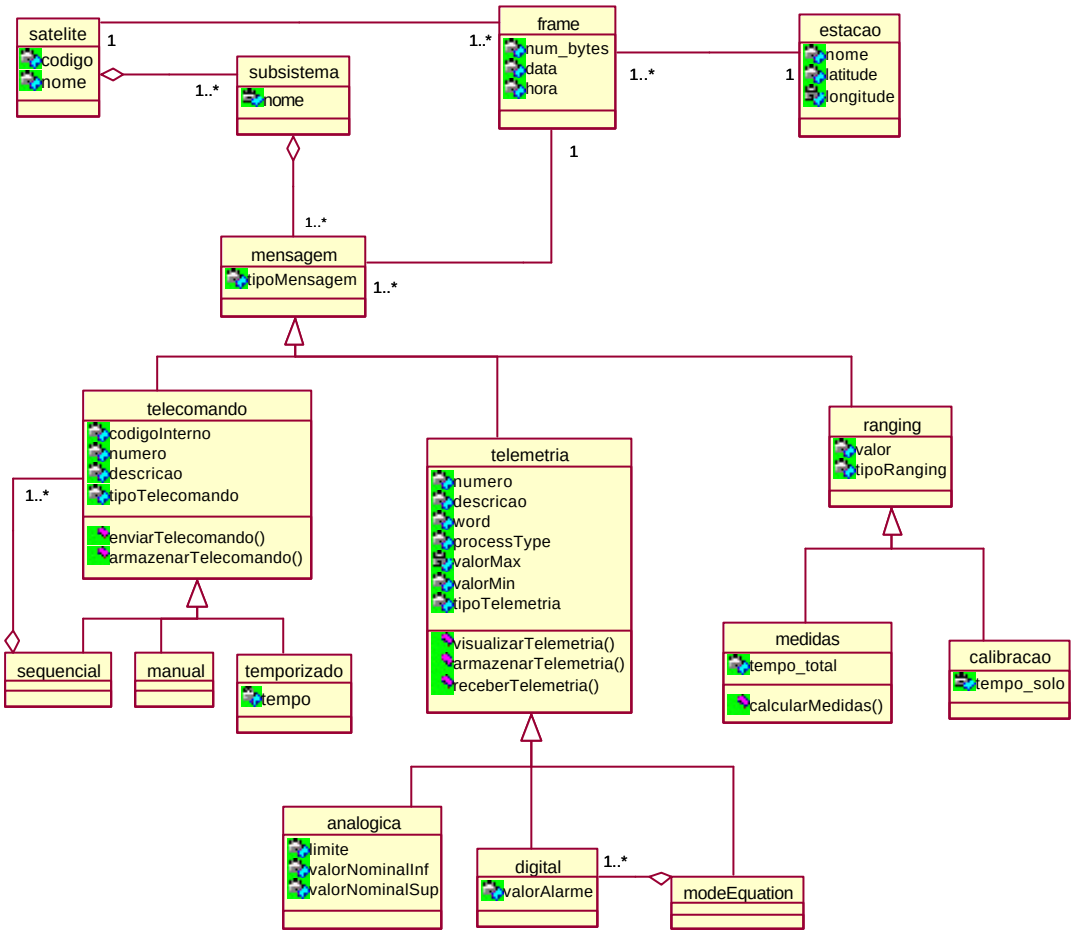


FIGURA 4.1 – Um modelo para o controle de satélites.

Dessa forma, o modelo ilustrado pela Figura 4.1 foi analisado sob o ponto de vista de cada padrão de projeto comumente encontrado em modelos de objetos adaptáveis, para que fosse obtida a construção do metamodelo ilustrado na Figura 4.9. O diagrama de atividades da Figura 4.2 resume as alterações realizadas pela aplicação dos padrões de projeto Tipo Objeto, Propriedade, Responsabilidade e Estratégia no modelo da Figura 4.1 até a obtenção do metamodelo da Figura 4.9. Essas atividades são detalhadas nas Seções seguintes.

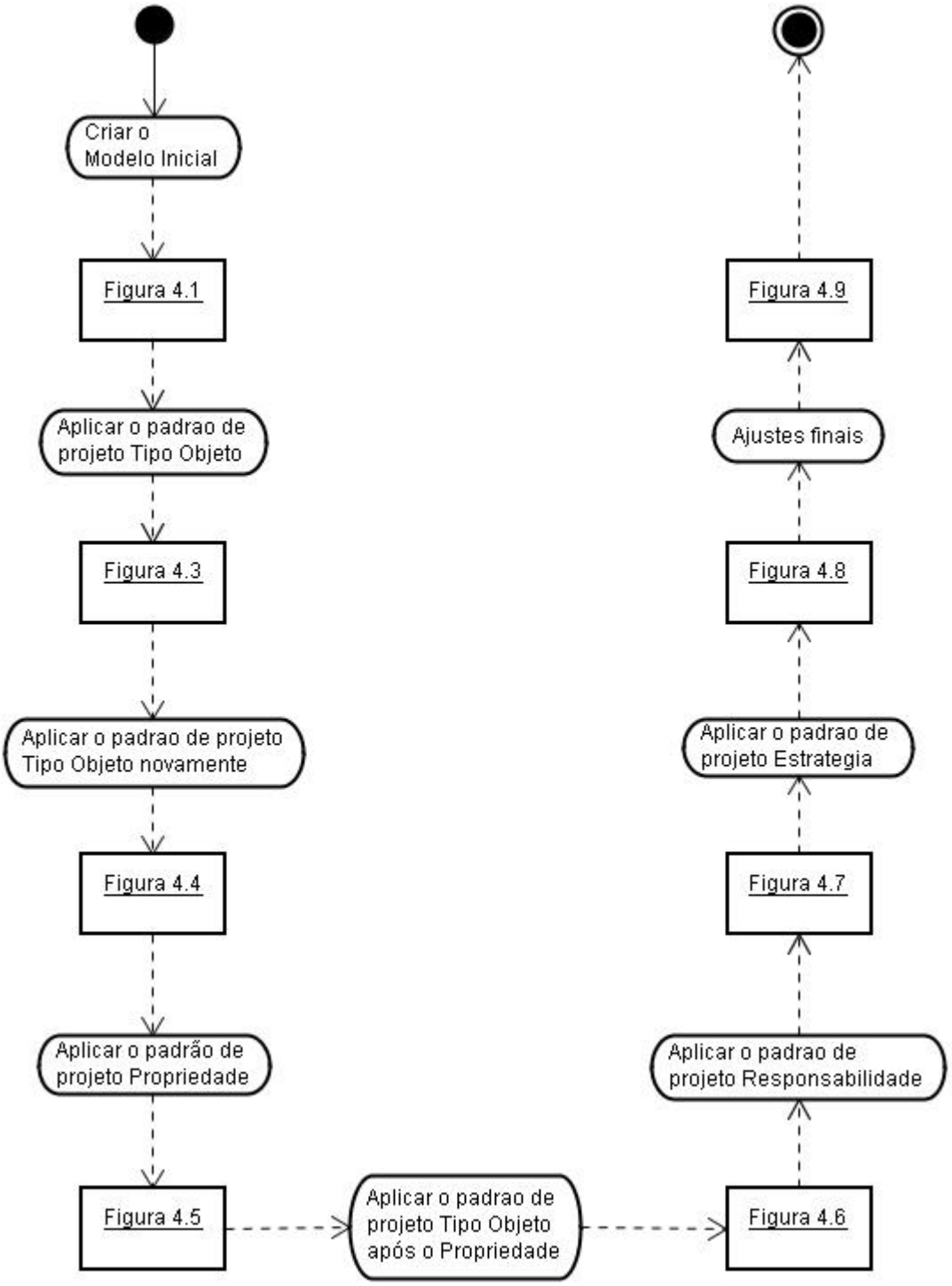


FIGURA 4.2 – Diagrama de atividades para obtenção do Metamodelo.

4.1 Aplicando o Padrão de Projeto Tipo Objeto

O primeiro padrão de projeto aplicado foi o Tipo Objeto. A Figura 4.3 mostra o modelo da Figura 4.1 após a aplicação desse padrão de projeto.

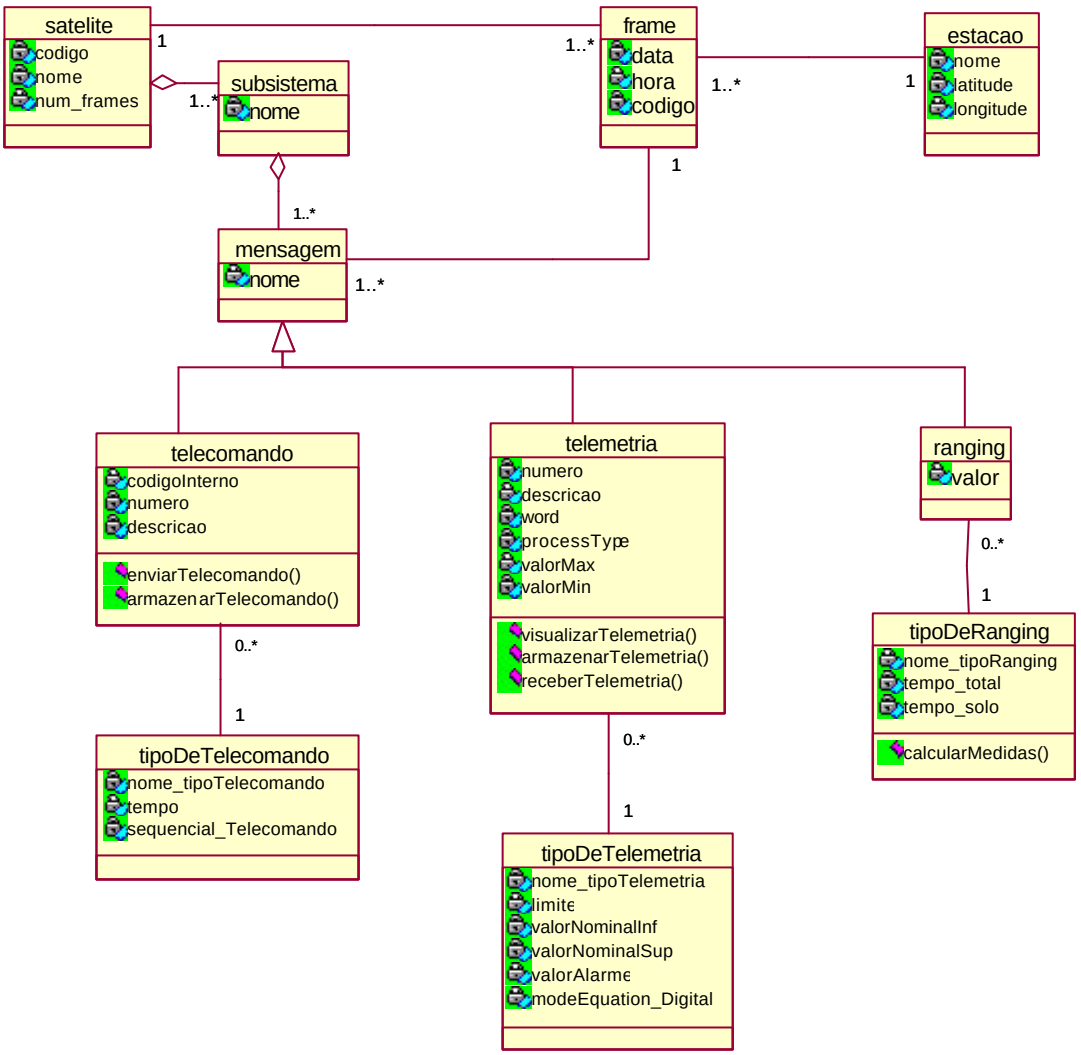


FIGURA 4.3 – Modelo após a aplicação do padrão de projeto Tipo Objeto.

Pode-se observar na Figura 4.3 que após a aplicação do padrão de projeto Tipo Objeto, as estruturas de herança existentes nas classes telecomando, telemetria e *ranging* foram eliminadas e foram acrescentadas as classes tipoDeTelecomando, tipoDeTelemetria e tipoDeRanging no diagrama de classes.

Os atributos e métodos comuns a todos os tipos de telemetria, telecomando e *ranging* permaneceram nas classes telemetria, telecomando e *ranging*, respectivamente. Os atributos e métodos específicos de cada tipo foram trazidos para as classes tipoDeTelemetria, tipoDeTelecomando e tipoDeRanging, respectivamente.

Por exemplo, um *ranging* pode ser do tipo medidas ou calibração. Se ele for do tipo medidas, terá como atributo específico: tempo_total. Se ele for calibração, terá como atributo específico: tempo_solo. Esses atributos específicos foram trazidos para a classe tipoDeRanging, sendo que os atributos comuns a todos os objetos ranging permaneceram na classe *ranging*.

Porém, como se pode notar na Figura 4.3, ainda existe uma estrutura de herança que pode ser eliminada. Por essa razão, o padrão de projeto Tipo Objeto foi aplicado novamente. O diagrama de classe resultante é apresentado na Figura 4.4.

Pode-se observar na Figura 4.4 que após a segunda aplicação do padrão de projeto Tipo Objeto, a estrutura de herança existente na classe mensagem foi eliminada e foram acrescentadas as classes tipoMensagem e tipoTipoMensagem no diagrama de classes.

Os atributos e métodos comuns a todos os tipos de mensagem permaneceram na classe mensagem e os atributos e métodos específicos de cada tipo foram trazidos para a classe tipoMensagem. Por exemplo, uma mensagem pode ser do tipo seqüencial, manual, temporizado, analógica, digital, modeEquation, medidas ou calibração. Se ela for do tipo temporizado, terá como atributo específico: tempo. Esses atributos específicos foram trazidos para a classe tipoMensagem, sendo que os atributos comuns a todos os objetos mensagem permaneceram na classe mensagem.

Adicionalmente, foi necessário criar a classe tipoTipoMensagem, já que cada mensagem também pode ser do tipo telemetria, telecomando ou *ranging*. A classe tipoTipoMensagem contém os atributos e métodos das classes telecomando, telemetria e *ranging*, apresentadas na Figura 4.3.

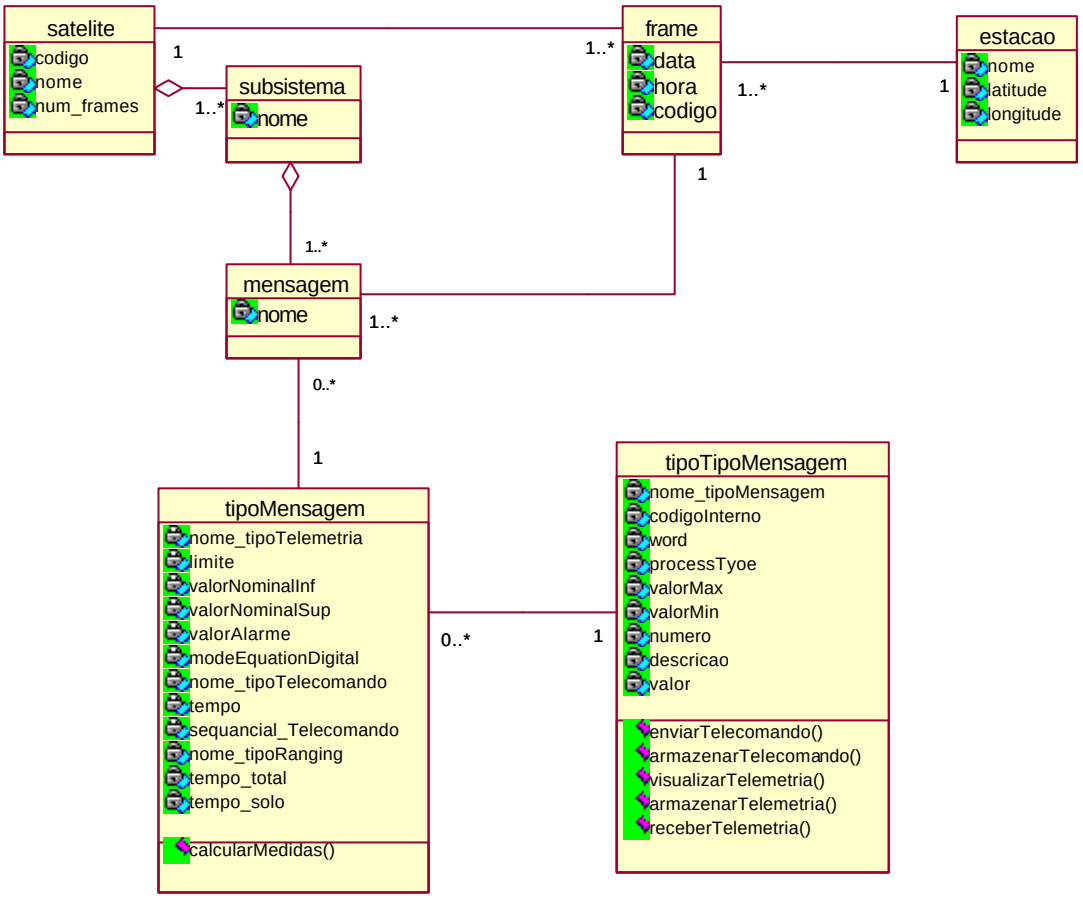


FIGURA 4.4 – Diagrama de classes após a segunda aplicação do Tipo Objeto.

4.2 Aplicando o Padrão de Projeto Propriedade

Como se pode observar na Figura 4.4, após a aplicação do padrão de projeto Tipo Objeto, objetos de tipos diferentes estarão todos em uma mesma classe TipoEntidade, o que significa que é necessário adotar uma forma alternativa para implementar seus atributos.

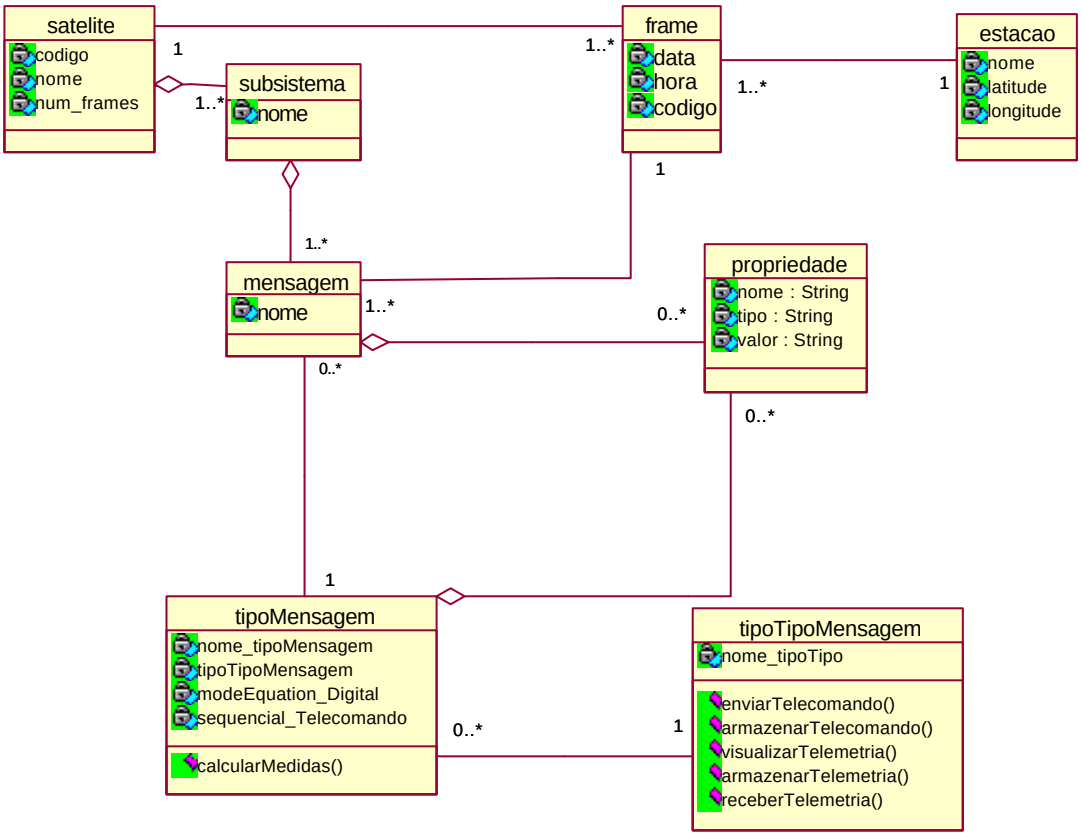


FIGURA 4.5 – Modelo após a aplicação do padrão de projeto Propriedade.

Uma maneira de resolver esta questão é aplicar o padrão de projeto Propriedade. A Figura 4.5 mostra o diagrama da Figura 4.4 após a aplicação desse padrão de projeto.

Como se pode observar na Figura 4.5, após a aplicação do padrão de projeto Propriedade, os atributos específicos de cada tipo representados na classes TipoEntidade foram eliminados e foi acrescentada a classe propriedade, que representa esses atributos. Por exemplo, na classe tipoTipoMensagem foram eliminados os atributos: word, processType, valorMax, valorMin, número, descrição, valor e valorAlarme.

4.3 Obtendo a Arquitetura *TypeSquare*

Com aplicação do padrão de projeto Propriedade resolveu-se o problema de se ter atributos específicos de diferentes tipos de entidades representados na mesma classe TipoEntidade. Porém, ainda assim não é possível identificar quais propriedades pertencem a quais tipos de entidades, uma vez que as propriedades estão associadas aos objetos da classe Entidade e não a objetos da classe TipoEntidade.

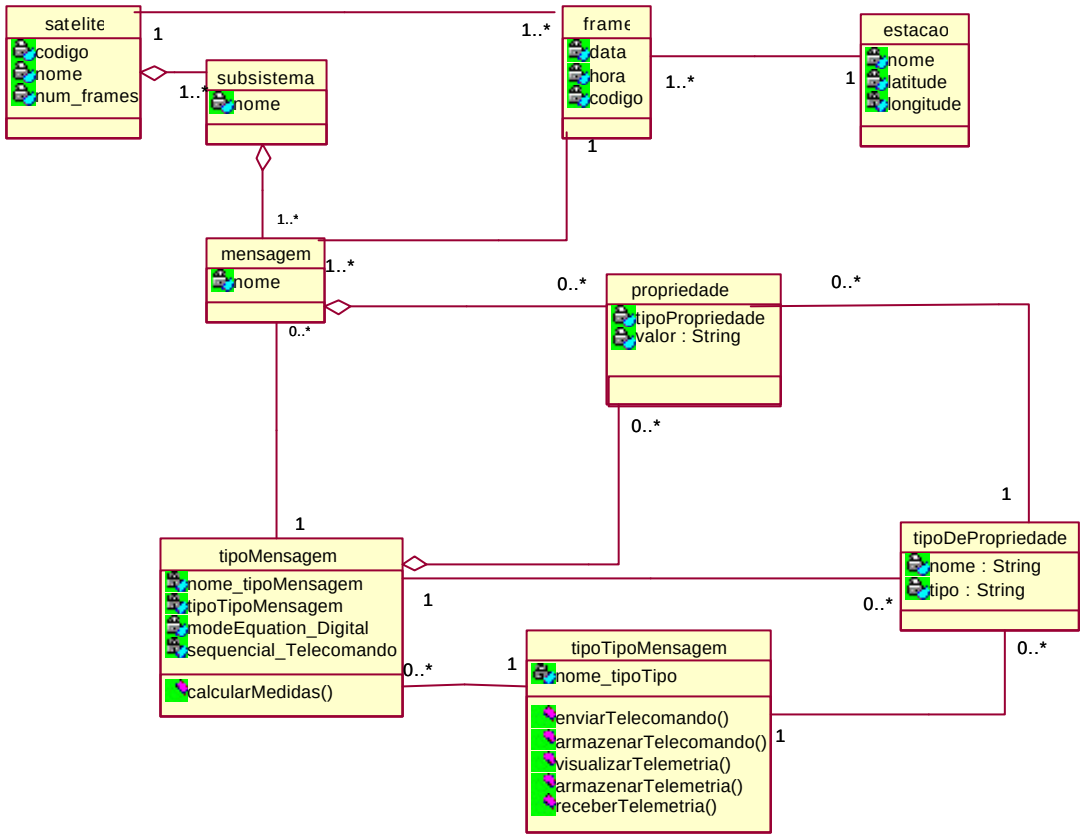


FIGURA 4.6 – A arquitetura *TypeSquare* para o domínio.

Para resolver esse problema, a maioria das arquiteturas de modelos de objetos adaptáveis aplica o padrão de projeto Tipo Objeto antes e depois de se usar o padrão de projeto Propriedade. Ou seja, o padrão de projeto Tipo Objeto divide o sistema em entidades e tipos de entidades. Entidades têm atributos que podem ser definidos usando-se o padrão de projeto Propriedade. Cada propriedade (atributo) tem um tipo, chamado

de “tipoDePropriedade”, e cada tipo de entidade pode então, especificar os tipos das propriedades para suas entidades.

Dessa forma, aplicou-se novamente o padrão de projeto Tipo Objeto para se obter a arquitetura *TypeSquare.* A Figura 4.6 a seguir mostra o diagrama da Figura 4.5 após a aplicação desse padrão de projeto.

Como se pode observar na Figura 4.6, após a terceira aplicação do padrão de projeto Tipo Objeto, pôde-se obter a arquitetura *TypeSquare*. Para tal, foi acrescentada ao modelo a classe tipoDePropriedade. Dessa forma, torna-se mais simples verificar quais tipos de entidades podem ter acesso a quais propriedades.

4.4 Aplicando o Padrão de Projeto Responsabilidade

Para representar as associações entre as classes do modelo, foi aplicado o padrão de projeto Responsabilidade no metamodelo. A Figura 4.7 a seguir mostra o diagrama da Figura 4.6 após a aplicação desse padrão de projeto.

Como se pode observar na Figura 4.7, após a aplicação do padrão de projeto Responsabilidade, foram criadas as classes responsabilidade e tipoDeResponsabilidade. Os atributos de associação modeEquation_Digital e seqüencial_Telecomando, que estavam na classe tipoMensagem foram eliminados e representados por uma responsabilidade.

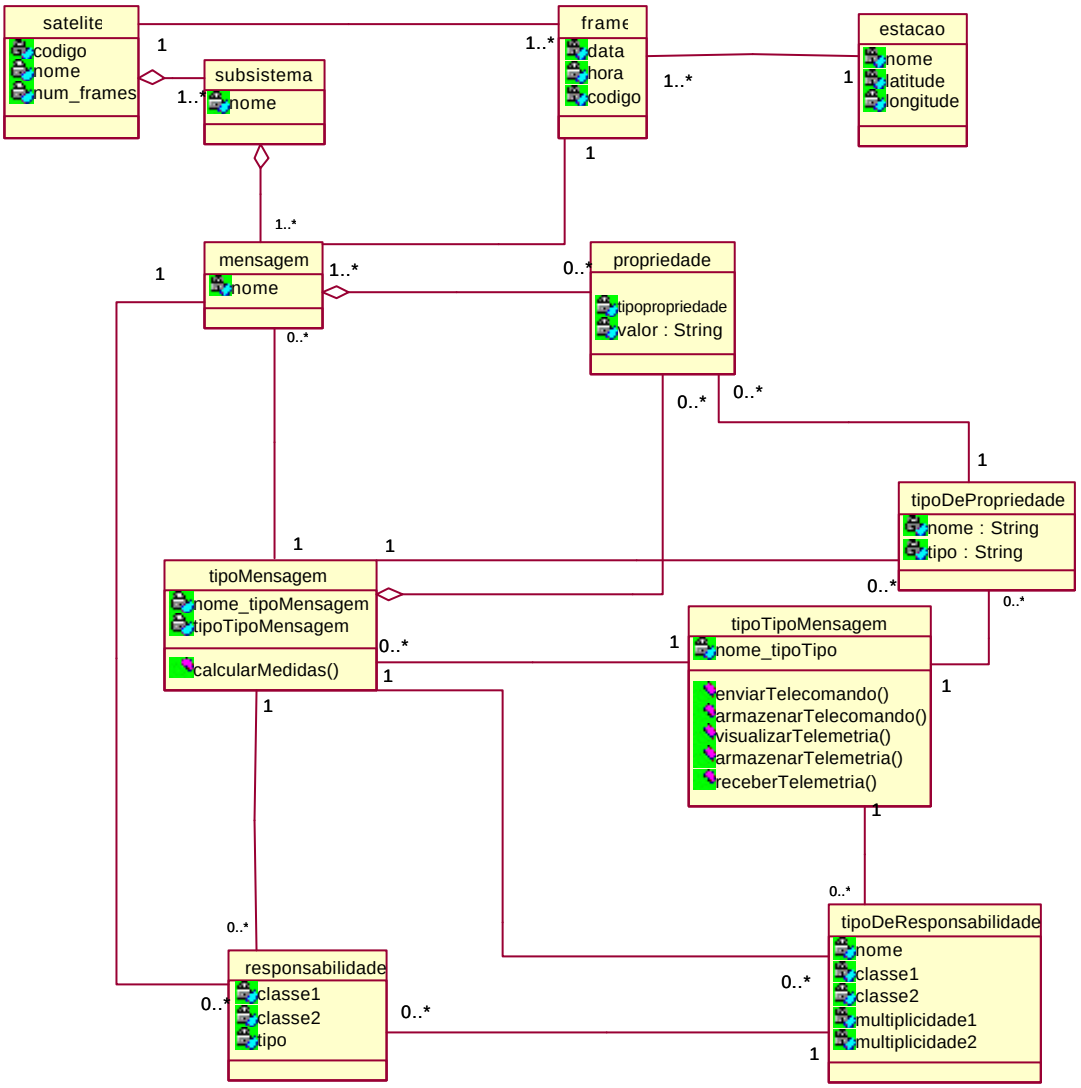


FIGURA 4.7 – *TypeSquare* após a aplicação do padrão de projeto Responsabilidade.

4.5 Aplicando o Padrão de Projeto Estratégia

Para representar as regras de negócio foi aplicado o padrão de projeto Estratégia. A Figura 4.8 a seguir mostra o diagrama da Figura 4.7 após a aplicação desse padrão de projeto.

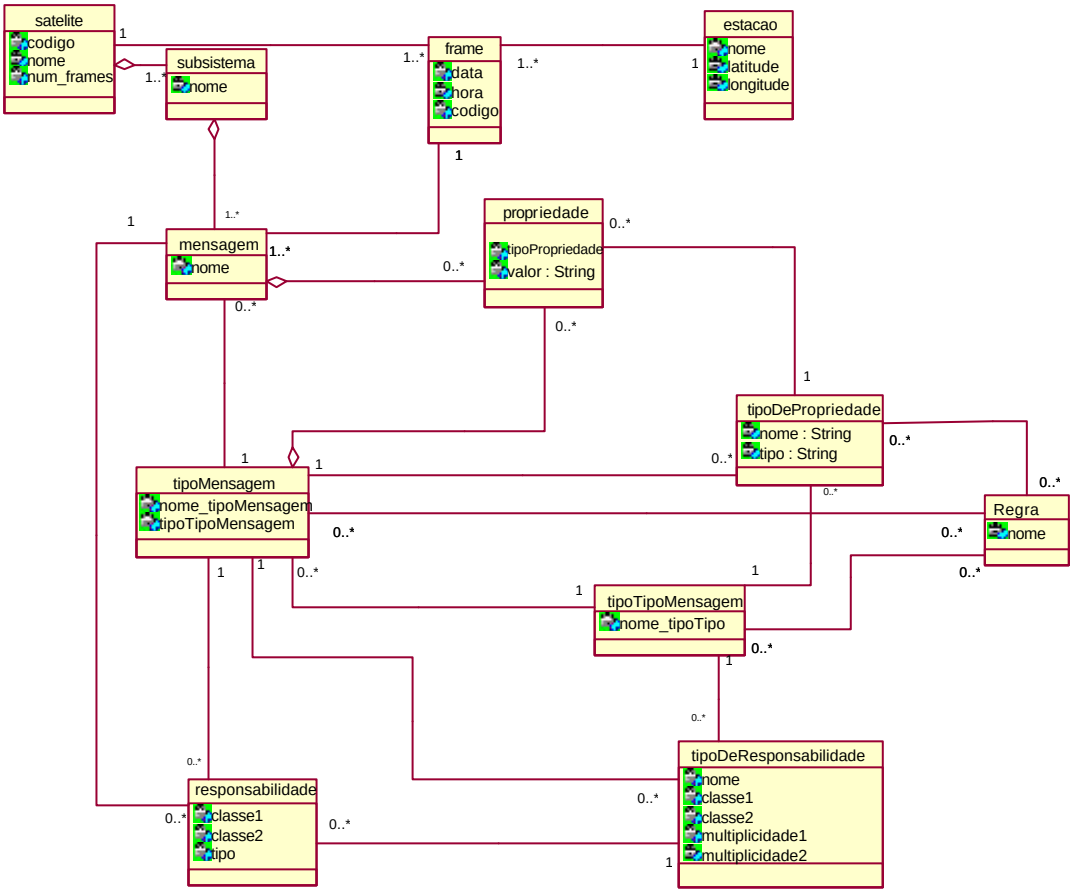


FIGURA 4.8 – *TypeSquare* após a aplicação do padrão de projeto Estratégia.

Como se pode observar na Figura 4.8, após a aplicação do padrão de projeto Estratégia, foi criada a classe Regra. Os métodos das classes tipoMensagem e tipoTipoMensagem foram eliminados, e foram representados através de uma estratégia.

Com o uso do padrão de projeto Estratégia, torna-se possível que novos métodos sejam associados a uma classe do domínio, em tempo de execução, fazendo com que o sistema possa ser configurável em termos das regras de negócio associadas às suas classes do domínio.

Pode-se observar também que as classes satélite, subsistema, estação e frame permaneceram intactas após a aplicação dos padrões de projeto desde a Figura 4.1 até a

Figura 4.8. Isso é perfeitamente possível, pois não necessariamente todas as classes do domínio têm que ser adaptáveis.

No desenvolvimento de um sistema baseado nos AOMs deve-se analisar, para a aplicação em questão, quais partes do sistema precisam ou não ser adaptáveis em função das necessidades do domínio. No caso desta aplicação específica, essas classes não foram consideradas como adaptáveis, uma vez que raramente sofrem mudanças em sua estrutura ao longo das várias missões.

Devido à características do domínio as classes representadas pela classe genérica tipoTipoMensagem não têm objetos diretos. Por isso constatou-se que as associações existente entre as classes genéricas propriedade e tipoMensagem; e tipoMensagem e responsabilidade podiam ser eliminadas.

Entende-se por objetos diretos nesse contexto o fato de que as classes do domínio telemetria, telecomando e *ranging* são classes abstratas, uma vez que só suas classes filhas têm objetos instanciados. Por exemplo, uma mensagem nunca é do tipo telecomando simplesmente, ela é sempre um telecomando seqüencial ou um telecomando manual ou um telecomando temporizado. O diagrama de classes apresentado na Figura 4.9 já apresenta essas alterações.

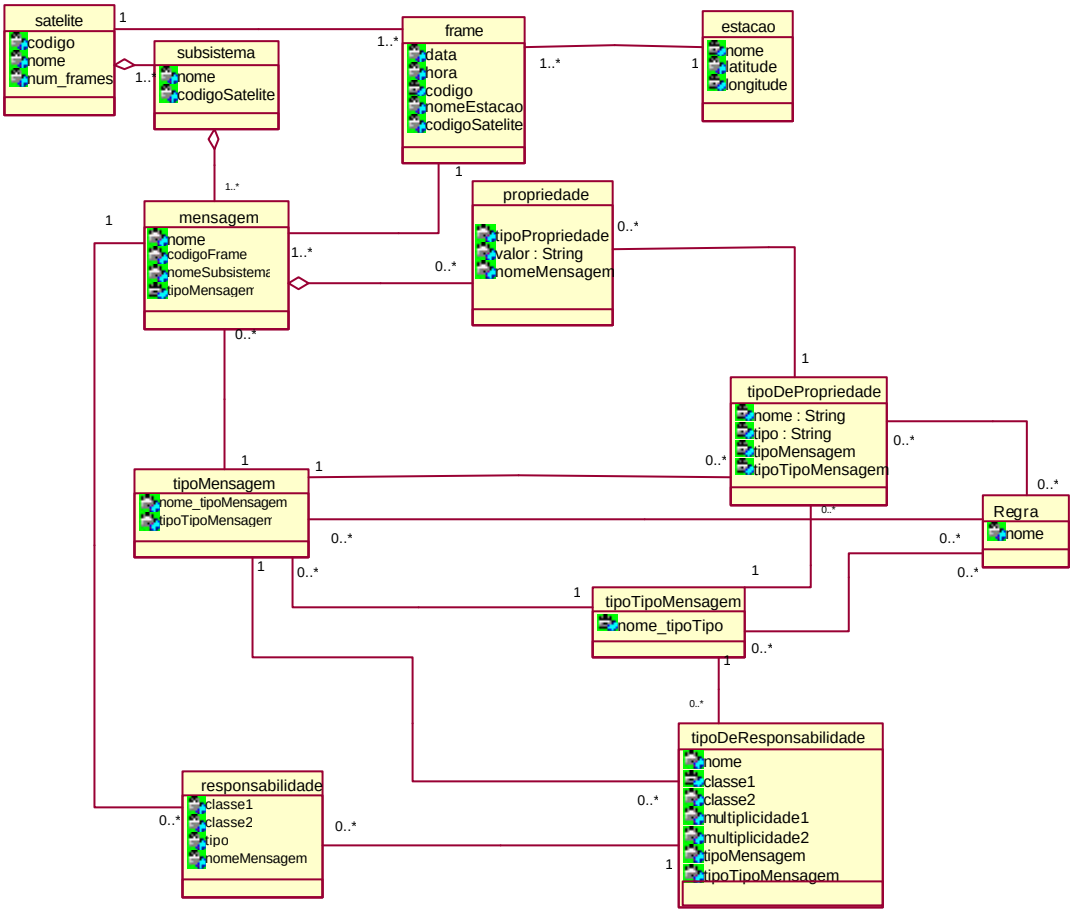


FIGURA 4.9 – O metamodelo da arquitetura SICSDA.

A arquitetura dos AOMs permite que, em tempo de execução, um usuário especialista no domínio baseado no metamodelo ilustrado na Figura 4.9, construa um modelo equivalente ao modelo ilustrado na Figura 4.1. Mas os AOMs vão além disso, pois sua arquitetura permite que o usuário especialista no domínio crie, em tempo de execução, modelos diferentes do modelo ilustrado na Figura 4.1. Por exemplo, ele poderia criar mais uma subclasse da classe mensagem, com diferentes atributos, métodos e associações. Da mesma forma ele poderia criar diversas subclasses da classe *Ranging*, da classe Telemetria ou da classe Telecomando. Assim, o usuário especialista no

domínio pode criar novos modelos, além de configurar ou adaptar os modelos existentes, em tempo de execução, sem a necessidade de alterar o código do sistema.

Persistir esses modelos de objetos adaptáveis criados em tempo de execução permite que os modelos mantenham a sua configuração definida pelos usuários especialistas no domínio, por diferentes execuções do sistema.

No Capítulo seguinte, mostra-se o mapeamento criado para a persistência de modelos de objetos de sistemas desenvolvidos baseados nos AOMs. Como estudo de caso e também como forma de facilitar o entendimento, foi utilizado o domínio de um sistema para o controle de satélites.

CAPÍTULO 5

A PERSISTÊNCIA DO MODELO DE OBJETOS

Este Capítulo trata da formalização do mapeamento para a persistência dos modelos de objetos de sistemas configuráveis e adaptáveis, desenvolvidos baseados na arquitetura dos AOMs em três sistemas de gerenciamento de dados:

- Um SGBD Orientado a Objetos.
- Um SGBD Relacional.
- Um *Native XML Database* (NXD).

A forma encontrada para facilitar a persistência dos modelos de objetos adaptáveis foi definir estratégias para fazer o mapeamento de seus metadados para os sistemas de gerenciamento de dados escolhidos. Para formalizar um mapeamento, pode-se utilizar mecanismos de extensão da UML, como por exemplo os estereótipos (Eriksson et al., 1998), na criação dos modelos de objetos. Assim, cria-se estereótipos específicos que devem ser inseridos no modelo adaptável, após a sua construção, para adicionar as informações necessárias para realizar a sua persistência (Witthawaskul e Johnson, 2003).

O metamodelo ilustrado na Figura 4.9 foi utilizado como nosso estudo de caso para demonstrar como realizar a persistência dos modelos criados, em tempo de execução, pelos usuários especialistas no domínio a partir desse metamodelo. Seguiu-se o mapeamento criado para a persistência desses modelos no desenvolvimento dos protótipos. Para facilitar um melhor entendimento da formalização criada, este Capítulo apresenta também alguns destes elementos:

- Trechos de códigos Java dos protótipos.
- Classes implementadas no SGBDOO Caché.

- Comandos *Structured Query Language* (SQL) do SGBDR PostgreSQL para a criação de tabelas.

- Arquivos XML.

- Metadados do modelo de objetos persistidos pelos protótipos.

Uma listagem mais completa dos três protótipos implementados, além dos metadados do modelo persistido, se encontra no apêndice A.

Conforme citado no Capítulo 3, os modelos de objetos criados a partir do metamodelo podem ser persistidos em banco de dados orientados a objeto, em banco de dados relacionais ou em arquivos XML. A seguir mostra-se uma abordagem para realizar a persistência dos modelos de objetos em cada um destes três sistemas de gerenciamentos de dados.

TABELA 5.1 – Estereótipos para mapeamento da persistência dos modelos.

Estereótipo	Descrição
«Entidade»	Representa as entidades do negócio.
«TipoEntidade»	Representa os tipos de entidades do negócio.
«Propriedade»	Representa as propriedades de uma entidade do negócio.
«TipoPropriedade»	Representa os tipos de propriedades que uma entidade do negócio pode conter.
«TipoResponsabilidade»	Representa os tipos de associações que podem existir entre os tipos de entidades do negócio.
«Responsabilidade»	Representa as associações entre as entidades do negócio.
«Regra»	Representa a classe que armazena as regras que podem ser associadas as entidades do negócio.

Para facilitar a formalização do mapeamento da persistência dos modelos de objetos adaptáveis criados em tempo de execução, definiu-se alguns estereótipos, que devem ser inseridos nas classes do metamodelo, após a sua criação. A Tabela 5.1 mostra estes estereótipos e suas descrições.

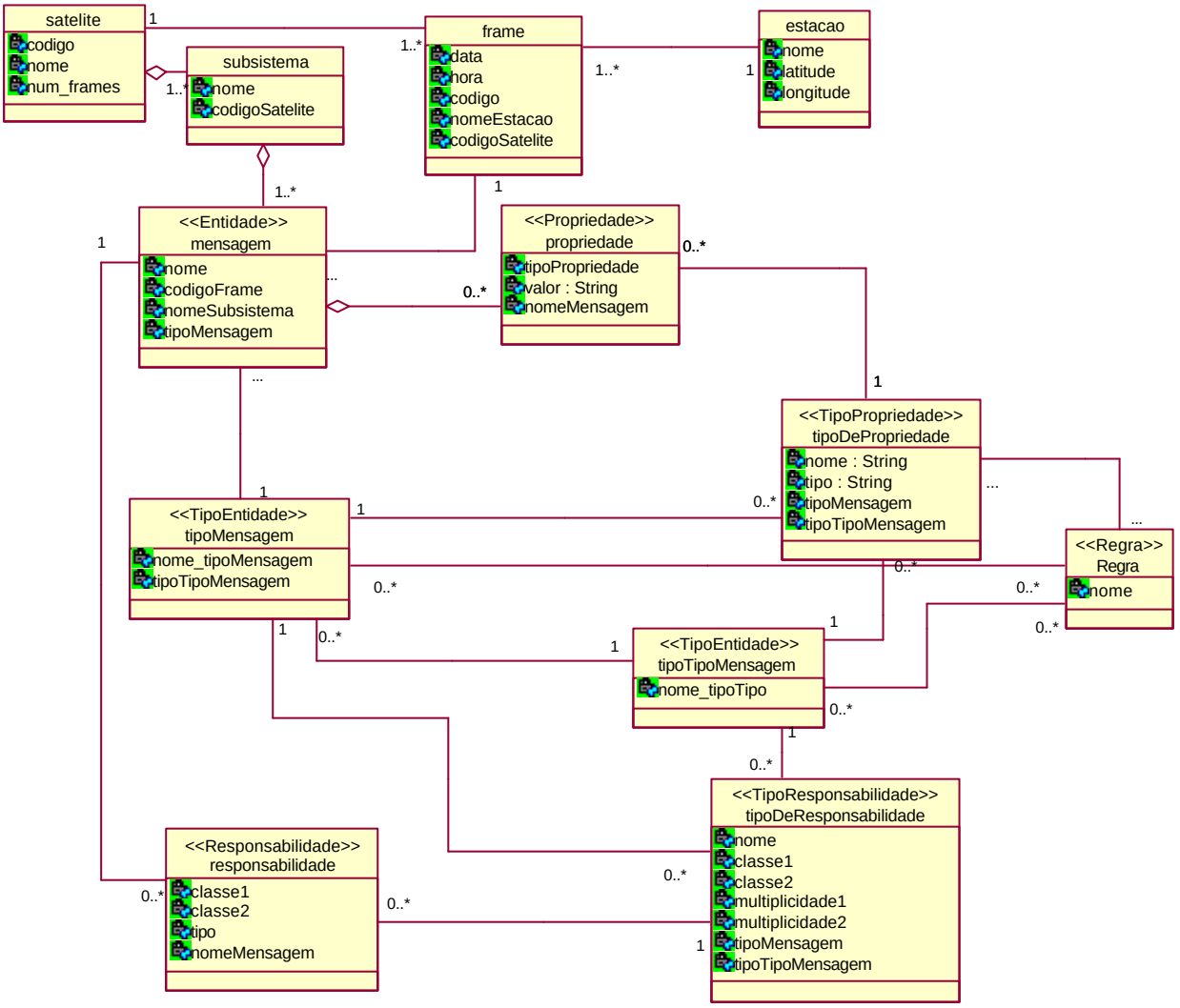


FIGURA 5.1 – O metamodelo da arquitetura SICSDA com os estereótipos.

Uma vez criado o metamodelo do sistema, deve-se adicionar ao metamodelo os estereótipos corretamente. A Figura 5.1 mostra o metamodelo da Figura 4.9 após a adição dos estereótipos. Em seguida, o desenvolvedor deve seguir as regras definidas no mapeamento de acordo com o sistema de gerenciamento de dados escolhido. Os estereótipos indicam qual o padrão de projeto foi utilizado na criação das classes, além do papel dessas classes nos padrões de projeto utilizados. Por exemplo, na Figura 5.1 a classe tipotipoMensagem possui o estereótipo «TipoEntidade». Isso significa que essa

classe foi criada pela utilização do padrão de projeto Tipo Objeto e que essa classe tem a função de definir os tipos de entidades do domínio.

A parte não adaptável do sistema, compreendida no metamodelo pelas classes sem estereótipos, pode ser persistida no banco seguindo as técnicas de projeto de banco de dados como encontrada em Elmasri e Navathe (2002). No estudo de caso essas classes são: satélite, frame, subsistema e estacao.

A formalização do mapeamento para a persistência foi dividida em três partes: As propriedades, as associações e as regras. Cada uma dessas partes foi dividida também em três partes, uma para cada sistema de gerenciamento de dados escolhido.

5.1 A persistência das Propriedades

Como exemplo das propriedades que foram persistidas criou-se, em tempo de execução, as propriedades da classe mensagem e suas classes descendentes mostradas na Figura 4.1. Por exemplo: para a classe *Ranging* criou-se as propriedades Valor e tipoRanging e para a classe digital criou-se a propriedade valorAlarme.

Durante a criação do metamodelo, após a aplicação dos padrões de projeto Tipo Objeto e Propriedade, o metamodelo continha as classes Propriedade e tipoDePropriedade. Na criação do banco de dados ou dos arquivos XML, pode-se adotar duas abordagens para a persistência das propriedades:

- Na primeira abordagem os objetos das classes marcadas com o estereótipo «TipoEntidade» podem se associar a um conjunto de objetos das classes marcadas com o estereótipo «TipoPropriedade». A Figura 5.2 ilustra essa abordagem.

- Na segunda abordagem um objeto da classe com o estereótipo «TipoPropriedade» vai estar associado a apenas um objeto de uma das classes marcadas com o estereótipo «TipoEntidade» no metamodelo. A Figura 5.3 ilustra essa abordagem.

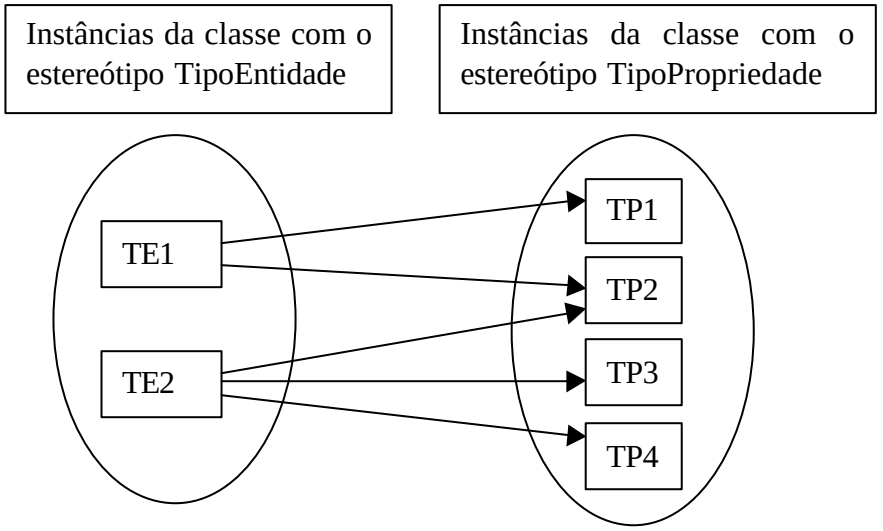


FIGURA 5.2 – Persistência de propriedades - primeira abordagem.

Apesar da primeira abordagem parecer mais condizente com os padrões de projeto utilizados, nos protótipos foi adotada a segunda abordagem, pois apresentou uma maior facilidade na implementação. Na primeira abordagem as propriedades criadas formam uma espécie de banco de propriedades que poderiam ser associadas aos tipos de objetos criados, em tempo de execução, pelos usuários especialistas no domínio. Essa abordagem evita a necessidade definir mais de uma vez um tipo de propriedade utilizado por mais de um tipo de objeto, o que ocorre na segunda abordagem. Essa vantagem pode ser verificada nas Figuras 5.2 e 5.3, onde na Figura 5.3 o tipo de propriedade TP2 foi criado duas vezes, uma para o TE1 e outra para o TE2.

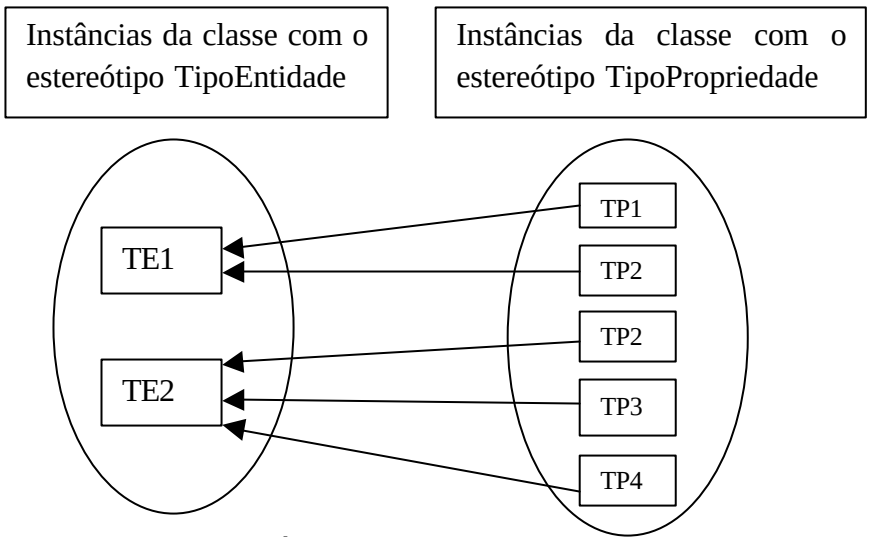


FIGURA 5.3 – Persistência de propriedades - segunda abordagem.

5.1.1 A persistência em um Banco de Dados Orientado a Objetos

Na segunda abordagem para persistir as propriedades, a classe tipoDePropriedade deve possuir além dos atributos Nome e Tipo, ambos do tipo *string*, uma referência para cada classe dos tipos de entidades no metamodelo, as classes marcadas com o estereótipo «TipoEntidade». No estudo de caso, pode-se observar na Figura 5.1 que a classe tipoDePropriedade se associa às duas classes com o estereótipo «TipoEntidade» (tipoMensagem e tipoTipoMensagem) assim a classe tipoDePropriedade deve conter uma referência para cada uma dessas classes.

Nessa abordagem, o atributo Nome não pode ser único pois duas classes diferentes podem possuir propriedades com o mesmo nome, conforme ilustra a Figura 5.3. A classe tipoDePropriedade implementada no Caché e utilizada no protótipo deste trabalho é mostrada a seguir.

```
Class User.tipoDePropriedade Extends %Persistent {
    Property Nome As %String;
    Property Tipo As %String;
    Property tipomensagem As TipoMensagem;
    Property tipotipomensagem As TipoTipoMensagem;
}
```

Para persistir as propriedades deve-se criar a classe propriedade. A classe propriedade possui o atributo Valor do tipo *string*, uma referência a classe tipoDePropriedade, além de uma referência para cada uma das classes marcadas com o estereótipo «Entidade» no metamodelo. A classe Propriedade utilizada no protótipo implementado é mostrada a seguir.

```
Class User.Propriedade Extends %Persistent {
    Property mensagem As Mensagem;
    Property tipo As tipoDePropriedade;
    Property Valor As %String;
}
```

5.1.2 A persistência em um Banco de Dados Relacional

Para realizar a persistência das propriedades em um banco de dados relacional, deve-se criar as tabelas de acordo com o metamodelo desenvolvido, com o apoio das técnicas de mapeamento objeto-relacional que podem ser consultadas em Elmasri e Navathe (2002).

Na segunda abordagem, a multiplicidade da associação entre a classe tipoDePropriedade e as classes com o estereótipo «TipoEntidade» é 1:N. Pelo mapeamento objeto-relacional verifica-se que não existe a necessidade da criação de uma tabela para cada associação existente, bastando criar na tabela TipoPropriedade uma chave estrangeira para cada uma das tabelas que representam as classes marcadas com o estereótipo «TipoEntidade» no metamodelo.

No estudo de caso, a tabela TipoPropriedade deve conter as colunas Codigo, Nome, Tipo, CodigoTipoTipoMensagem (Chave estrangeira para a tabela TipoTipoMensagem) e a coluna CodigoTipoMensagem (Chave estrangeira para a tabela TipoMensagem). O comando SQL que criou a tabela TipoPropriedade no PostgreSQL é mostrado a seguir.

```
CREATE TABLE TIPOPROPRIEDADE (
    CODIGO INT8 NOT NULL,
    NOME VARCHAR(80),
    TIPO VARCHAR(80),
    CODIGOTIPOMENSAGEM INT8,
    CODIGOTIPOTIPOMENSAGEM INT8,
    PRIMARY KEY(CODIGO));
```

Para persistir as propriedades deve-se também criar a tabela Propriedade. Essa tabela armazena informações do valor de uma propriedade de uma instância de uma classe marcada com o estereótipo «Entidade», possui uma chave estrangeira para a tabela TipoPropriedade, além de chaves estrangeiras para as tabelas que representam as classes marcadas com o estereótipo «Entidade» no metamodelo. No estudo de caso, criou-se a tabela propriedade com as colunas Codigo, Valor, uma chave estrangeira para a tabela TipoPropriedade, além de uma chave estrangeira para a tabela Mensagem (A classe marcada com o estereótipo «Entidade» no metamodelo). O Comando SQL que criou a tabela Propriedade no PostgreSQL é mostrado a seguir.

```
CREATE TABLE PROPRIEDADE (
    CODIGO INT8 NOT NULL,
    VALOR VARCHAR(80),
    CODIGOTIPOPROPRIEDADE INT8 NOT NULL,
    CODIGOMENSAGEM INT8,
    PRIMARY KEY(CODIGO));
```

A coluna CODIGOTIPOPROPRIEDADE não pode conter o valor NULL porque uma instância da classe propriedade deve sempre estar associada a uma instância da classe tipoDePropriedade e essa coluna é uma chave estrangeira para a tabela TipoPropriedade. A coluna CODIGOMENSAGEM é uma chave estrangeira para a tabela Mensagem e contém o código de a uma instância da classe Mensagem.

5.1.3 A persistência em Arquivos XML

Para realizar a persistência das propriedades em arquivos XML, de acordo com a segunda abordagem, deve-se criar o arquivo XML tipopropriedade.xml que contém todos os tipos de propriedade criados, em tempo de execução, pelo usuário especialista no domínio. Cada tipo de propriedade fica armazenado em um elemento tipopropriedade representado pelo tag <tipopropriedade>. Esse elemento deve conter os seguintes tags:

- <codigo> para armazenar o código do tipo da propriedade.
- <nome> para armazenar o nome do tipo da propriedade.
- <tipo> para armazenar o tipo da propriedade.

Além desses três tags, como a classe marcada com o estereótipo «Tipopropriedade» é associada com as classes marcadas com o estereótipo «TipoEntidade», o elemento tipopropriedade deve conter também um tag, para cada classe marcada com o estereótipo «TipoEntidade» no metamodelo. Cada tag contém o código de uma instância da classe marcada com o estereótipo «TipoEntidade».

No estudo de caso, foi criado para o elemento tipopropriedade o tag <tipotipomensagem> para armazenar o código de um objeto da classe tipoTipoMensagem, caso o objeto da classe tipoDePropriedade esteja associado a um objeto da classe tipoTipoMensagem.

Também foi criado para o elemento tipopropriedade o tag <tipomensagem> que armazena o código de um objeto da classe tipoMensagem, caso o objeto da classe tipoDePropriedade esteja associado a um objeto da classe tipoMensagem.

A Figura 5.4 mostra um exemplo do arquivo tipopropriedade.xml. Neste arquivo, o primeiro elemento tipopropriedade representa a propriedade numero da classe telecomando ilustrada na Figura 4.1. O Segundo elemento tipopropriedade representa a propriedade descricao da classe telecomando.

```
<?xml version="1.0"?>
<tipopropriedades>
 <tipopropriedade>
  <codigo> 1</codigo>
  <nome> Numero</nome>
  <tipo> java.lang.Integer</tipo>
  <tipotipomensagem> 1</tipotipomensagem>
  <tipomensagem>  </tipomensagem>
 </tipopropriedade>
 <tipopropriedade>
  <codigo> 2</codigo>
  <nome> Descrição </nome>
  <tipo> java.lang.String</tipo>
  <tipotipomensagem>  1</tipotipomensagem>
  <tipomensagem>  </tipomensagem>
 </tipopropriedade>
</tipopropriedades>
```

FIGURA 5.4 – Exemplo de um arquivo tipopropriedade.xml.

Os objetos da classe Propriedade ficam armazenados no arquivo propriedade.xml. Cada objeto da classe Propriedade é armazenado pelo elemento propriedade, representado pelo tag <propriedade>. O elemento propriedade deve conter os seguintes tags:

- <codigo> para armazenar o código do objeto.
- <valor> para armazenar o valor do objeto.
- <tipopropriedade> para armazenar o código do objeto da classe tipoDePropriedade.

O elemento propriedade deve conter ainda um tag para cada classe com o estereótipo «Entidade» no metamodelo. Esses tags armazenam o código do objeto da classe «Entidade» associado ao objeto da classe «Propriedade» representado por esse elemento. No estudo de caso foi criado o tag <mensagem> para armazenar o código do objeto da classe mensagem associado ao objeto representado pelo elemento propriedade.

A Figura 5.5 mostra um exemplo do arquivo propriedade.xml. Esse arquivo mostra a representação de dois objetos da classe Propriedade. O primeiro elemento representa a propriedade numero de uma instância da classe telecomando, ilustrada na Figura 4.1. O segundo elemento propriedade representa a propriedade descricao de uma instância da classe telecomando.

```
<?xml version="1.0"?>
<propriedades>
  <propriedade>
    <codigo> 1</codigo>
    <valor> 185</valor>
    <tipopropriedade> 1</tipopropriedade>
    <mensagem> 1</mensagem>
  </propriedade>
  <propriedade>
    <codigo> 2</codigo>
    <valor> Mensagem Telecomando1</valor>
    <tipopropriedade> 2</tipopropriedade>
    <mensagem> 1 </mensagem>
  </propriedade>
</propriedades>
```

FIGURA 5.5 – Exemplo do arquivo propriedade.xml.

5.1.4 Exemplos de Propriedades como Metadados

As Tabelas 5.2 e 5.3 ilustram a persistência das propriedades como metadados no Caché.

TABELA 5.2 – A classe tipoDePropriedade no Caché.

ID	Nome	Tipo	tipomensagem	tipotipomensagem
1	Tempo	java.lang.string		
2	CodigoInterno	java.lang.Integer		26
3	Numero	java.lang.Integer		26
4	Descricao	java.lang.String		26
5	Numero	java.lang.Integer		27
6	Descricao	java.lang.String		27
7	Word	java.lang.Integer		27
8	ProcessType	java.lang.Integer		27
9	ValorMax	java.lang.Integer		27
10	ValorMin	java.lang.Integer		27
11	Valor	java.lang.Integer		28
12	Tempo	java.lang.Integer	16	
13	Limite	java.lang.Integer	17	
14	ValorInf	java.lang.Integer	17	
15	ValorSup	java.lang.Integer	17	
16	ValorAlarme	java.lang.Integer	18	
17	Tempototal	java.lang.Integer	21	
18	TempoSolo	java.lang.Integer	22	
19	TipoListagem	java.lang.Integer		26
20	TipoListagem	java.lang.Integer		27

TABELA 5.3 – A classe propriedade no Caché.

ID	Tipo	Valor	mensagem
79	12	24	135
80	2	4	135
81	4	8	135
82	3	6	135
83	19	1	135
84	12	24	137
85	2	4	137
86	4	8	137
87	3	6	137
88	19	2	137
89	12	24	139
90	2	4	139
91	4	8	139
92	3	6	139
93	19	3	139
94	12	24	146
95	2	4	146
96	4	8	146
97	3	6	146
98	19	2	146

5.2 A Persistência das Associações

Como exemplo de persistência de associações pode-se citar a associação entre as classes digital e modeEquation e também a associação entre as classes telecomando e Sequencial ilustradas na Figura 4.1.

Como foi mostrado no Capítulo 3, as associações podem ser consideradas como atributos de uma classe cujo valor seja outra classe. Para formalizar o mapeamento para a persistência das associações, permitindo assim que os usuários especialistas no domínio possam criar e alterar associações entre as classes do sistema, em tempo de execução, criou-se dois estereótipos: «Responsabilidade» e «TipoResponsabilidade», mostrados na Tabela 5.1, para se adicionar ao metamodelo desenvolvido para o domínio do sistema.

Deve-se notar que um sistema desenvolvido baseado nos AOMs deve permitir que o usuário especialista do domínio crie associações entre tipos de entidade que ainda nem existem e são criados pelo usuário em tempo de execução. A classe tipoDeResponsabilidade criada pela aplicação do padrão de projeto Responsabilidade define os tipos de associações permitidas entre os tipos de entidades do negócio, além da multiplicidade dessas associações.

Uma associação é bidirecional, pois se a classe A se associa à classe B, então a classe B também se associa à classe A. Devido a essa característica, decidiu-se persistir uma associação como duas instâncias da classe tipoDeResponsabilidade, uma para cada sentido da associação.

A classe tipoDeResponsabilidade se associa a todas as classes marcadas com o estereótipo «TipoEntidade» e a classe responsabilidade se associa à todas as classes marcadas com o estereótipo «Entidade» no metamodelo.

5.2.1 Persistir as Associações em um Banco de Dados Orientado a Objetos

A classe com o estereótipo «TipoResponsabilidade» deve conter informações sobre a multiplicidade das associações em ambos os sentidos, para poder validar as associações criadas em tempo de execução. Isso pode ser feito pela criação de dois atributos: Multiplicidade1 e Multiplicidade2, esses atributos foram definidos como do tipo *string*, pois a multiplicidade muitos é representada por "1..*" (Elmasri e Navathe, 2002). Essa classe possui o atributo Nome do tipo *string* para armazenar o nome da associação.

Como foi definido na Seção 5.2, uma associação é representada por duas instâncias da classe marcada com o estereótipo «TipoResponsabilidade». Como a associação é bidirecional, são necessários dois atributos para armazenar a referência de cada classe no metamodelo marcada com o estereótipo «TipoEntidade», um para cada sentido da associação. Os nomes desses atributos podem ser o nome da classe marcada com o estereótipo «TipoEntidade» correspondente diferenciados pelos sufixos 1 ou 2. Uma instância é sempre um espelho da outra, pois uma instância armazena os valores em um sentido da associação e a outra instância os valores no outro sentido.

A Tabela 5.4 ilustra a persistência das duas associações encontradas no modelo da Figura 4.1. As instâncias da classe tipoDeResponsabilidade com o ID 28 e 29 armazenam a associação entre as classes sequencial (que é uma instância da classe tipoMensagem com ID 14) e telecomando (que é uma instância da classe tipoTipoMensagem com ID 26). A Instância da classe tipoDeResponsabilidade com o ID 28 armazena a associação no sentido da classe sequencial para a classe telecomando. A instância da classe tipoDeResponsabilidade com o ID 29 armazena a associação no sentido da classe telecomando para a classe sequencial.

No estudo de caso, a classe tipoDeResponsabilidade contém duas referências à classe tipoTipoMensagem e duas referências à classe tipoMensagem. A classe tipoDeResponsabilidade implementada no Caché e utilizada no protótipo deste trabalho é mostrada a seguir.

```
Class User.tipoDeResponsabilidade {
    Property Multiplicidade1 As %String;
    Property Multiplicidade2 As %String;
    Property Nome As %String;
    Property tipomensagem1 As TipoMensagem;
    Property tipomensagem2 As TipoMensagem;
    Property tipotipomensagem1 As TipoTipoMensagem;
    Property tipotipomensagem2 As TipoTipoMensagem
}
```

A Tabela 5.4 mostra os valores de instâncias da classe tipoDeResponsabilidade do estudo de caso. Estes valores representam as duas associações que podem ser verificadas no modelo do domínio, ilustrado na Figura 4.1. As instâncias da classe tipoDeResponsabilidade com o ID 26 e 27 armazenam a associação entre as classes Digital e ModeEquation.

TABELA 5.4 – Instâncias da classe tipoDeResponsabilidade do protótipo.

	ID	Multiplicidade1	Multiplicidade2	Nome	tipomensagem1	tipomensagem2	tipotipomensagem1	tipotipomensagem2
1	26	1	*	DigitalModeEquation2	20	18		
2	27	*	1	DigitalModeEquation	18	20		
3	28	*	1	Sequencial_Telecomando2		14	26	
4	29	1	*	Sequencial_Telecomando	14			26

A classe responsabilidade armazena as referências das instâncias das classes com o estereótipo «Entidade» que participam da associação, além de uma referência da classe tipoDeResponsabilidade a qual ela pertence. A classe responsabilidade implementada no Caché é mostrada a seguir.

```
Class User.Responsabilidade Extends %Persistent {
    Property mensagem1 As Mensagem;
    Property mensagem2 As Mensagem;
    Property tipoDeResponsabilidade As tipoDeResponsabilidade;
}
```

5.2.2 Persistir as Associações em um Banco de Dados Relacional

Pelo fato da multiplicidade das associações entre as classes com o estereótipo «TipoResponsabilidade» e «TipoEntidade» ser 1:1, não há necessidade da criação de tabelas extras para armazenar essas associações, de acordo com o mapeamento objeto-relacional. A tabela TipoResponsabilidade contém chaves estrangeiras para todas as

tabelas que representam as classes com o estereótipo «TipoEntidade» no metamodelo (No estudo de caso as tabelas TipoTipoMensagem e TipoMensagem).

A tabela TipoResponsabilidade contém as colunas Multiplicidade1, Multiplicidade2 para armazenar a multiplicidade de cada classe na associação e também a coluna Nome, contendo o nome da associação, todas do tipo *string*. Além disso contém duas chaves estrangeiras para cada tabela que represente uma classe com o estereótipo «TipoEntidade». Os nomes dessas chaves estrangeiras podem ser diferenciados pelos sufixos 1 e 2.

Assim como na persistência em um banco de dados orientado a objetos, uma associação é armazenada em duas linhas na tabela TipoResponsabilidade, uma linha para cada sentido da associação. Uma linha é como um espelho da outra linha que representa a mesma associação.

No estudo de caso, a tabela TipoResponsabilidade foi criada conforme mostrado a seguir. As colunas CODIGOTIPOMENSAGEM1 e CODIGOTIPOMENSAGEM2 são chaves estrangeiras para a tabela TipoMensagem e as colunas CODIGOTIPOTIPO-MENSAGEM1 e CODIGOTIPOTIPOMENSAGEM2 são chaves estrangeiras para a tabela TipoTipoMensagem.

```
CREATE TABLE TIPORESPONSABILIDADE (
    CODIGO INT8 NOT NULL,
    NOME VARCHAR(80),
    MULTIPLICIDADE1 VARCHAR(10),
    MULTIPLICIDADE2 VARCHAR(10),
    CODIGOTIPOMENSAGEM1 INT8,
    CODIGOTIPOMENSAGEM2 INT8,
    CODIGOTIPOTIPOMENSAGEM1 INT8,
     CODIGOTIPOTIPOMENSAGEM2 INT8,
    PRIMARY KEY(CODIGO));
```

Para persistir a classe responsabilidade deve-se criar uma tabela chamada Responsabilidade, esta tabela contém duas chaves estrangeiras para cada classe marcada com o estereótipo «Entidade», além da coluna CODIGO, que é a chave primária desta tabela, e uma chave estrangeira para a tabela TipoResponsabilidade.

No estudo de caso criou-se a tabela Responsabilidade, mostrada a seguir. A coluna MENSAGEM1 é uma chave estrangeira para a tabela Mensagem e armazena a chave do primeiro objeto na associação. A coluna MENSAGEM2 é uma chave estrangeira para a tabela Mensagem e armazena a chave do segundo objeto na associação.

```
CREATE TABLE RESPONSABILIDADE (
CODIGO INT8 NOT NULL,
CODIGOTIPORESPONSABILIDADE INT8,
CODIGOMENSAGEM1 INT8,
CODIGOMENSAGEM2 INT8,
PRIMARY KEY(CODIGO));
```

5.2.3 Persistir as Associações em Arquivos XML

Para realizar a persistência das associações em arquivos XML, deve-se criar um arquivo TipoResponsabilidade.xml para armazenar os objetos da classe marcada com o estereótipo «TipoResponsabilidade». Deve-se criar também o arquivo Responsabilidade.xml para armazenar os objetos da classe marcada com o estereótipo «Responsabilidade».

Como as associações são bidirecionais da mesma forma como foi feita nos outros mapeamentos, as associações são armazenadas em dois elementos do arquivo tipopropriedade.xml, um para cada sentido da associação.

No arquivo TipoResponsabilidade.xml, o elemento tipoResponsabilidade representa um objeto da classe marcada com o estereótipo «TipoResponsabilidade». O elemento tipoResponsabilidade possui dois tags <multiplicidade> um para cada classe da associação. O elemento tipoResponsabilidade possui também um tag para o código da associação e outro tag para o nome da associação.

O elemento tipoResponsabilidade deve conter dois tags para cada classe com o estereótipo «TipoEntidade» no metamodelo. Os nomes dos tags podem ser CODIGO, o nome da classe mais um sufixo 1 ou 2, para diferenciá-los. Esses tags armazenam os códigos dos objetos dessas classes que se relacionam pela associação representada pelo elemento tipoResponsabilidade.

No estudo de caso, o arquivo TipoResponsabilidade.xml tem o seguinte formato:

- <codigo> para armazenar o código da associação.
- <nome> para armazenar o nome da associação.
- <multiplicidade1> para armazenar a multiplicidade da primeira classe da associação.
- <multiplicidade2> para armazenar a multiplicidade da segunda classe da associação.
- <tipomensagem1> para armazenar o código do objeto da classe tipoMensagem, quando tipoMensagem for a primeira classe da associação.
- <tipomensagem2> para armazenar o código do objeto da classe tipoMensagem, se a segunda classe da associação for tipoMensagem.
- <tipotipomensagem1> para armazenar o código do objeto da classe tipoTipoMensagem, quando tipoTipoMensagem for a primeira classe da associação.
- <tipotipomensagem2> para armazenar o código do objeto da classe tipoTipoMensagem, se a segunda classe da associação for tipoTipoMensagem.

O arquivo Responsabilidade.xml, armazena os objetos das classes com o estereótipo «Responsabilidade». Esses objetos armazenam as referências dos objetos das classes com o estereótipo «Entidade» que participam da associação. Esses objetos são representados pelo elemento <responsabilidade>.

No estudo de caso, o elemento <responsabilidade>, contém os seguintes tags:

- <codigo> armazena o código do objeto
- <mensagem1> armazena o código do objeto da primeira classe que participa da associação.

- <mensagem2> armazena o código do objeto da segunda classe que participa da associação.

- <tipoResponsabilidade> armazena o código do objeto da classe tipoDeResponsabilidade associado a esse objeto da classe responsabilidade.

5.3 Persistência das Regras

As regras foram a parte mais complexa de se mapear a persistência e onde menos se avançou nesta dissertação. Um trabalho mais voltado para lidar com as regras de sistemas desenvolvidos baseados nos AOMs pode ser encontrado em Cardoso (2005). O desenvolvimento de um sistema baseado nos AOMs, especialmente os padrões de projeto utilizados, é influenciado pelas características do domínio do problema desse sistema.

Obter uma grande adaptabilidade e configurabilidade nas regras do negócio, em tempo de execução, pode significar a criação de uma linguagem de alto nível, o que implica a criação de gramáticas, *parsers* e compiladores para o funcionamento adequado dessa linguagem específica para o domínio. Isso seria de uma complexidade muito grande saindo do escopo deste trabalho. Se o domínio para o qual se desenvolve um sistema baseado nos AOMs não requisitar uma ampla adaptabilidade e configurabilidade das regras do negócio, certamente não compensará o esforço despendido para se obter tal característica.

Apenas a utilização do padrão de projeto Estratégia foi formalizada no mapeamento e implementada nos protótipos. A utilização do padrão de projeto Regra Objeto fica para um trabalho futuro. Em cada protótipo desenvolvido, baseado no padrão de projeto Estratégia, foi implementado uma série de variações para cada método disponível e foi fornecido ao usuário especialista no domínio formas de associar esses métodos a alguma classe marcada com o estereótipo «Entidade» no metamodelo. Os protótipos também permitem ao usuário escolher qual das variações do método essa classe utiliza.

Persistir as regras do modelo, consiste em persistir a classe marcada com o estereótipo «Regra» e suas associações com as classes marcadas com o estereótipo «TipoEntidade» e com o estereótipo «TipoPropriedade».

A classe Regra armazena todas as regras do negócio disponíveis e possui um único atributo: Nome. Os protótipos fornecem ao usuário especialista no domínio condições de associar essas regras às instâncias das classes com o estereótipo «TipoEntidade». A multiplicidade da associação entre a classe Regra e as classes com o estereótipo «TipoEntidade» no metamodelo é M:N.

Com o intuito de aumentar a configurabilidade na execução das regras criou-se a possibilidade de o usuário associar a uma regra um conjunto de atributos definidos para a classe marcada com o estereótipo «TipoEntidade». Pode-se utilizar os valores desses atributos tanto para indicar qual variação da regra deve ser utilizada, quanto para sua própria execução.

Dessa forma, pode-se por exemplo associar uma classe marcada com o estereótipo «TipoEntidade» à uma regra e em seguida determinar que um dos tipos de propriedades definidos para essa classe será utilizado na execução dessa regra.

Durante a execução de uma regra deve-se verificar se existe algum atributo associado a essa regra. Se a regra possui algum atributo associado a mesma deve-se buscar o seu valor na classe propriedade e fornecê-lo à regra antes de sua execução.

Para implementação das regras no protótipo foi utilizada a API de reflexão da linguagem Java, introduzida no Capítulo 2 deste trabalho. Esta API fornece condições de localizar classes e executar métodos cujos nomes só se tornem conhecidos em tempo de execução. Como acontece em sistemas desenvolvidos baseados nos AOMs.

Foi implementada uma classe chamada Regras.java para realizar a execução das regras. Nessa classe foram implementadas diversas variações para um mesmo método. O nome do método a ser executado fica persistido no banco de dados na classe Regra. Durante a execução de uma regra, verifica-se a existência de propriedades associadas à sua

execução. Se a regra possui propriedades associadas, busca-se seus valores antes de sua execução.

A seguir lista-se o método mais importante da classe Regras.java, chamado executarRegra. Esse método possui três parâmetros: (1) o nome da regra, (2) os parâmetros dessa regra e (3) os tipos dos parâmetros dessa regra. Esses parâmetros são as propriedades da classe com o estereótipo «TipoEntidade» que foram associadas à classe Regra pelo usuário especialista no domínio. A linha 1 recupera um objeto *Class* para a classe Regras.java, onde estão codificados todos os métodos. A segunda linha recupera o método desejado, a terceira linha cria uma instância da classe Regras.java, necessária como parâmetro na quarta linha. Finalmente a quarta linha executa o método.

```
public static void executarRegra(String nomeregra, Class partType[], Object
arglist[]) {
      try {
            Class cls = Class.forName("User.Regras");                  //Linha 1
            Method meth = cls.getMethod(nomeregra, partType);    //Linha 2
            Regras regra = new Regras();                               //Linha 3
            Object retobj = meth.invoke(regra, arglist);               //Linha 4
      } catch (Throwable e ) { System.err.println(e); }
}
```

O método executarRegra utiliza a API de reflexão da linguagem Java. Uma vez que os nomes das regras ficam armazenados em um banco de dados, com esse método pode-se conseguir que qualquer uma das instâncias das classes marcadas com o estereótipo «TipoEntidade» executem qualquer um dos métodos implementados na classe Regras.java. Consegue-se dessa forma uma grande configurabilidade das regras em tempo de execução.

5.3.1 Persistência das Regras em um Banco de Dados Orientado a Objetos

O único atributo da classe Regra é o atributo Nome, que contém o nome da regra, já que o código que implementa a regra foi codificado nos protótipos (na classe Regras.java) e não é persistido.

Como a multiplicidade da associação entre a classe Regra e as classes com o estereótipo «TipoEntidade» é M:N, para armazenar a associação entre a classe Regra e essas classes deve-se criar uma nova classe para cada classe com o estereótipo «TipoEntidade» no metamodelo. Essas classes devem conter as referências dos objetos da classe Regra e as referências dos objetos da classe com o estereótipo «TipoEntidade» associados.

Essas classes também possuem um atributo para armazenar uma coleção de propriedades definidas para a classe com o estereótipo «TipoEntidade» associada a classe Regra. Essas propriedades podem ser utilizadas durante a execução da regra. O nome desse atributo é PropriedadesAssociadas.

No estudo de caso criou-se duas novas classes: RegraTipoMensagem e RegraTipoTipoMensagem. A classe RegraTipoMensagem armazena as referências dos objetos da classe Regra e as referências dos objetos da classe tipoMensagem, que forem associados a alguma regra. A classe RegraTipoTipoMensagem armazena as referências dos objetos da classe Regra e as referências dos objetos da classe tipoTipoMensagem, que forem associados a alguma regra. A implementação da classe Regra no Caché é mostrada a seguir. A Tabela 5.5 mostra instâncias da classe Regra criadas no protótipo do estudo de caso.

```
Class User.Regra Extends %Persistent  {
Property NomeRegra As %String;
}
```

TABELA 5.5 – Instâncias da classe Regra do protótipo.

	ID	NomeRegra
1	1	ListarRelacionamento
2	2	VisualizarTelemetria
3	3	ArmazenarTelemetria
4	4	ReceberTelemetria
5	5	EnviarTelecomando
6	6	ArmazenarTelecomando
7	7	CalcularMedidas
8	8	REGRADINAMICA

A implementação da classe RegraTipoMensagem no Caché é mostrada a seguir. O atributo propriedadesassociadas é uma lista de objetos da classe tipoDePropriedade que podem ser utilizadas na execução da regra.

```
Class User.RegraTipoMensagem Extends %Persistent {
    Property regra As Regra;
    Property propriedadesassociadas As tipoDePropriedade [ Collection = list ];
    Property tipomensagem As TipoMensagem;
}
```

5.3.2 Persistência das Regras em um Banco de Dados Relacional

O primeiro passo para persistir as regras em um banco de dados relacional é criar a tabela Regra. A tabela Regra, que representa a classe Regra, contém apenas as colunas CODIGO, a chave primária da tabela, e NOME para armazenar o nome da regra.

Como a multiplicidade da associação entre a classe Regras e as classes com o estereótipo «TipoEntidade» no metamodelo é M:N, pelo mapeamento objeto-relacional, essas associações são mapeadas para uma tabela. Então deve-se criar uma tabela para cada associação entre a classe Regra e as classes com o estereótipo «TipoEntidade» no metamodelo. Os nomes dessas tabelas podem ser Regra seguido do nome das classes com o estereótipo «TipoEntidade».

No estudo de caso foi criado a tabela RegraTipoMensagem, para armazenar as associações entre a classe Regra e a classe TipoMensagem e foi criado também a tabela RegraTipoTipoMensagem, para armazenar as associações entre a classe Regra e a classe TipoTipoMensagem. Nos parágrafos seguintes essas tabelas são referenciadas como tabelas RegraTipoEntidade. Essas duas tabelas, listadas a seguir, possuem:

- uma coluna CODIGO do tipo Integer que é a chave primária dessas tabelas;
- uma chave estrangeira para a tabela Regras e
- uma chave estrangeira para a tabela que representa a classe com o estereótipo «TipoEntidade» da associação.

```
CREATE TABLE REGRATIPOTIPOMENSAGEM (
CODIGO INT8 NOT NULL,
CODIGOTIPOTIPOMENSAGEM INT8,
CODIGOREGRA INT8,
PRIMARY KEY (CODIGO));

CREATE TABLE REGRATIPOMENSAGEM (
CODIGO INT8 NOT NULL,
CODIGOTIPOMENSAGEM INT8,
CODIGOREGRA INT8,
PRIMARY KEY (CODIGO));
```

As regras podem se associar a um conjunto de tipos de propriedades. Pelo mapeamento objeto-relacional, isso significa que deve-se criar uma nova tabela para conter chaves estrangeiras das tabelas RegraTipoEntidade (no estudo de caso, as tabelas RegraTipoMensagem e RegraTipoTipoMensagem) e uma chave estrangeira para a tabela TipoPropriedade. Para o protótipo foram criadas duas tabelas, uma para cada tabela RegraTipoEntidade (As tabelas RegraTipoMensagemTipoPropriedade e RegraTipoTipoMensagemTipoPropriedade).

5.3.3 Persistência das Regras em Arquivos XML

Para persistir as regras em arquivos XML, primeiramente deve-se criar o arquivo regra.xml. Esse arquivo armazena as regras do sistema (a classe com o estereótipo «Regra» no metamodelo). O elemento regra representa um objeto da classe Regra e contém os tags <codigo> e <nome>. A Figura 5.6 ilustra um exemplo do arquivo regras.xml.

Para persistir as associações entre a classe com o estereótipo «Regra» e as classes com o estereótipo «TipoEntidade», deve-se criar um arquivo XML para cada uma dessas associações. Os nomes desses arquivos podem ser Regra seguido do nome da classe com o estereótipo «TipoEntidade» no metamodelo. Nos parágrafos seguintes, esses arquivos são referenciados como arquivos RegraTipoEntidade. Os elementos desses arquivos devem conter um tag <codigoregra> para armazenar o código da regra e um outro tag para conter o código da classe com o estereótipo «TipoEntidade» correspondente. No estudo de caso foram criados dois arquivos RegraTipoEntidade:

regratipotipomensagem.xml e regratipomensagem.xml. A Figura 5.7 ilustra um exemplo do arquivo regratipomensagem.xml.

```
<?xml version="1.0"?>
<regras>
  <regra>
    <codigo> 1</codigo>
    <nome> ListarRelacionamento</nome>
  </regra>
  <regra>
    <codigo> 2</codigo>
    <nome> ArmazenarTelemetria</nome>
  </regra>
  <regra>
    <codigo> 3</codigo>
    <nome> ReceberTelemetria</nome>
  </regra>
</regras>
```

FIGURA 5.6 – Exemplo do arquivo regras.xml.

```
<?xml version="1.0"?>
<regratipomensagens>
  <regratipomensagem >
    <codigo> 1</codigo>
    <codigoregra> 2</ codigoregra >
    <codigotipomensagem> 2</ codigotipomensagem >
  </regratipomensagem >
  < regratipomensagem >
    <codigo> 2</codigo>
    <codigoregra> 3</ codigoregra >
    <codigotipomensagem> 2</ codigotipomensagem >
  </ regratipomensagem >
</ regratipomensagens >
```

FIGURA 5.7 – Exemplo do arquivo regratipomensagem.xml.

Como as regras podem se associar a um conjunto de tipos de propriedades, então deve-se criar para cada arquivo RegraTipoEntidade gerado para o metamodelo um arquivo XML para armazenar os códigos dos objetos da classe com o estereótipo «Regra» e os códigos dos objetos da classe RegraTipoEntidade. No estudo de caso, foram criados

dois arquivos XML: o arquivo regratipotipomensagemtipopropriedade.xml e o arquivo regratipomensagemtipopropriedade.xml.

O Capítulo seguinte apresenta aspectos importantes dos três protótipos criados seguindo os mapeamentos formalizados neste Capítulo. Mostra também quais ferramentas foram utilizadas para o desenvolvimento desses protótipos, que seguiram a especificação J2EE para a construção de sistemas distribuídos.

CAPÍTULO 6

O PROTÓTIPO E A PERSISTÊNCIA NOS SISTEMAS DE GERENCIAMENTO DE DADOS

6.1 Desenvolvimento do Protótipo

Para se obter os benefícios da arquitetura J2EE, citados na Seção 2.3.2, de uma melhor forma, deve-se descobrir quais as ferramentas se adaptam mais facilmente e fornecem ao desenvolvedor o suporte necessário para obter resultados mais eficientes e de uma melhor qualidade (Gabrick e Meiss, 2002).

Dessa maneira, para o desenvolvimento do protótipo, primeiramente foi feito uma pesquisa sobre quais ferramentas seriam as mais apropriadas. Para este trabalho quatro fatores foram importantes para a escolha das ferramentas:

- Facilidade de instalação.
- Facilidade de configuração: criação e alteração e automática de variáveis de ambiente como *PATH* e *HOME*, poucos arquivos de configuração e principalmente documentação explicando quais alterações devem ser realizadas nessas variáveis de ambiente e nos arquivos de configuração.
- Facilidade de aprendizado da ferramenta: arquivos de ajuda, tutoriais na página do fabricante, apostilas, livros, etc...
- Integração com as demais ferramentas escolhidas: por exemplo, um SGBDOO que se integrasse com um servidor de aplicação e com um IDE Java.

Uma vez definido que os protótipos seriam desenvolvidos baseados na arquitetura J2EE conforme explicado na Seção 2.3.2, as ferramentas utilizadas foram:

- Ferramenta de desenvolvimento, também chamada de *Integrated Development Enviroment* (IDE) Java.

- Servidor de aplicação.
- Sistemas gerenciadores de dados: SGBDOO, SGBDR e NXD.

6.1.1 IDE Java

O JBuilder X foi a IDE escolhida, principalmente por já estar em uma versão estável. O JBuilder X possui uma versão de teste que foi utilizada para o desenvolvimento do protótipo. O JBuilder pode trabalhar com diversos servidores de aplicação, inclusive o JBoss, e também com diversos servidores de gerenciamento de dados, inclusive os três utilizados no desenvolvimento dos protótipos.

6.1.2 Servidor de Aplicação

O servidor de aplicação escolhido foi o JBoss, um dos servidores de aplicação de *software* livre mais utilizados no mercado. Pode ser configurado no JBuilder e também nos três sistemas de gerenciamento de dados escolhidos. A primeira versão do JBoss testada foi a 3.2.2, mas apresentou incompatibilidade com o Caché, em seguida testou se a versão 3.0.8 que foi compatível com o Caché.

6.1.3 Sistemas de Gerenciamento de Dados

O apêndice B fornece um histórico dos sistemas de gerenciamento de dados existentes, inclusive dos três escolhidos para este trabalho.

6.1.3.1 Sistema Gerenciador de Banco de Dados Orientado a Objetos (SGBDOO)

O SGBDOO escolhido foi o Caché versão 5.0.5. Na realidade o Caché é considerado um SGBD pós-relacional. A parte relacional do pós-relacional, refere ao fato de que o Caché possui todas as características de um SGBDR. Todos os dados dentro de um banco de dados Caché estão disponíveis como verdadeiras tabelas relacionais e podem ser consultados e modificados usando SQL padrão.

A parte pós do termo pós-relacional refere ao fato de que o Caché oferece uma variedade de características que vão além dos limites dos SGBDRs, mesmo que o Caché ainda suporte uma visão relacional dos dados. Estas características incluem (CACHÉ, 2004):

- A habilidade de modelar dados como objetos (embora cada objeto possua uma representação relacional automaticamente criada e sincronizada com o objeto).
- Tipos de dados definidos pelo usuário.
- A habilidade de tirar vantagens de métodos e herança, incluindo polimorfismo.
- Extensões objeto para SQL para manipular a identidade dos objetos e associações.
- A habilidade de mesclar acessos baseados em objetos e baseados na linguagem SQL dentro de uma única aplicação, utilizando cada um no momento em que for mais apropriado.

Para este trabalho, a principal desvantagem do uso do Caché foi o fato da versão de teste disponível gratuitamente possuir todas as funcionalidades da versão completa, menos suporte a distribuição. Isso impossibilitou os testes em um ambiente distribuído, que foi realizado apenas na implementação dos protótipos que utilizavam os outros dois sistemas de gerenciamento de dados.

6.1.3.2 Sistema Gerenciador de Banco de Dados Relacional (SGBDR)

O SGBDR escolhido foi o PostgreSQL por ser um *software* livre e foi lançado a mais de 15 anos, além de ser um SGBD em uma versão estável e confiável. Sua equipe de desenvolvimento sempre tem a precaução de manter compatibilidade com os padrões SQL92 e SQL99.

O PostgreSQL derivou do projeto POSTGRES da universidade de Berkley, que foi originalmente patrocinado pelo *Defense Advanced Research Projects Agency* (DARPA), *Army Research Office* (ARO), *National Science Foundation* (NSF) e *Electromagnetic Systems Laboratories* (ESL).

O PostgreSQL é um SGBD adequado para o estudo acadêmico do modelo relacional, além de ser uma opção para empresas implementarem seus sistemas sem altos custos de licenciamento. Se trata de um *software* aberto, o que torna o seu código fonte disponível e o seu uso livre para aplicações comerciais ou não. Alguns recursos presentes na versão mais recente (PostgreSQL, 2005):

- Integridade Referencial.
- Funções armazenadas (*Stored Procedures*), que podem ser escritas em várias linguagens de programação (PL/PgSQL, Perl, Python, Ruby e outras).
- Gatilhos (*Triggers*).
- Tipos definidos pelo usuário.
- Esquemas (*Schemas*).

6.1.3.3 *Native XML Database* (NXD)

Os *Native XML Databases*, que são mais detalhados no apêndice C, armazenam coleções de arquivos XML em seu formato original, como em um sistema de arquivos. Por ser uma tecnologia muito nova ainda não se tem ferramentas muito estáveis (Gabrick e Weiss, 2002).

O NXD utilizado foi o XIndice da Apache, que surgiu do NXD DBXML. Esse projeto foi doado para a Apache e deu origem ao projeto XIndice em dezembro de 2001 (Xindice, 2005). A página do Xindice na Internet já oferece uma série de guias para seu uso, sua administração e sua utilização em aplicações Java, o que facilitou o seu aprendizado. Além de ser um *software* livre sem custos de licenciamento.

6.1.4 Os Protótipos

Os protótipos desenvolvidos procuraram explorar algumas das características de sistemas desenvolvidos baseados na arquitetura dos AOMs: permitir ao usuário especialista no domínio realizar alterações no modelo de objetos, em tempo de execução, tais como criar propriedades, regras e associações. O objetivo desta Seção é fornecer uma idéia de como esses sistemas são executados na prática e como o usuário especialista no domínio pode realizar mudanças, em tempo de execução, nos modelos de objetos desse tipo de sistemas. Além de mostrar como essas mudanças refletem imediatamente na execução do sistema.

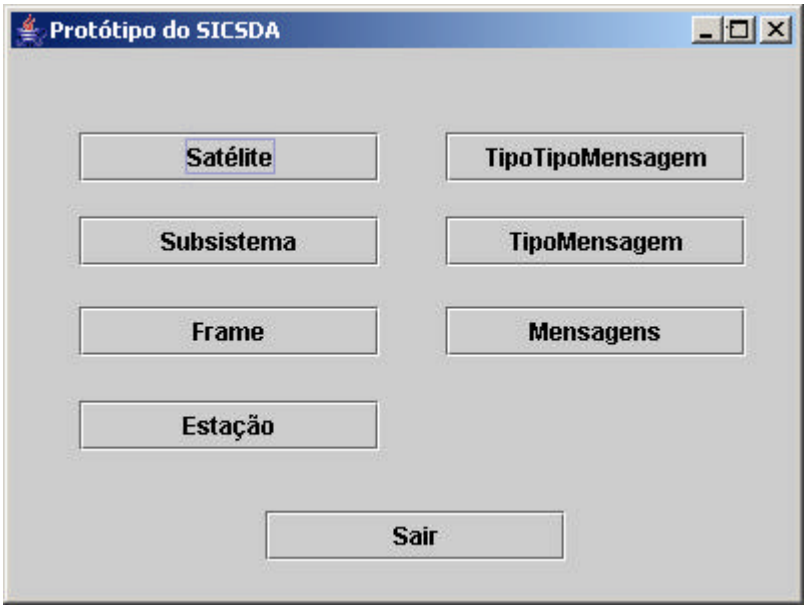


FIGURA 6.1 – Tela inicial.

A Figura 6.1 ilustra a tela inicial que foi dividida em duas partes, na primeira parte estão as chamadas para as telas que cuidam da parte do sistema que não pode ser alterada em tempo de execução (As classes do metamodelo, ilustrado na Figura 5.1, que não possuem nenhum dos estereótipos definidos na Tabela 5.1). Na segunda parte da tela se encontram as chamadas para as telas onde foram implementados os padrões de projeto que compõem os Modelos de Objetos Adaptáveis (As classes do metamodelo com algum dos estereótipos definidos na Tabela 5.1).

As telas que manipulam os satélites, os subsistemas, os frames e as estações não foram consideradas aqui e podem ser consultadas no apêndice A, que fornece uma listagem mais completa dos protótipos desenvolvidos.

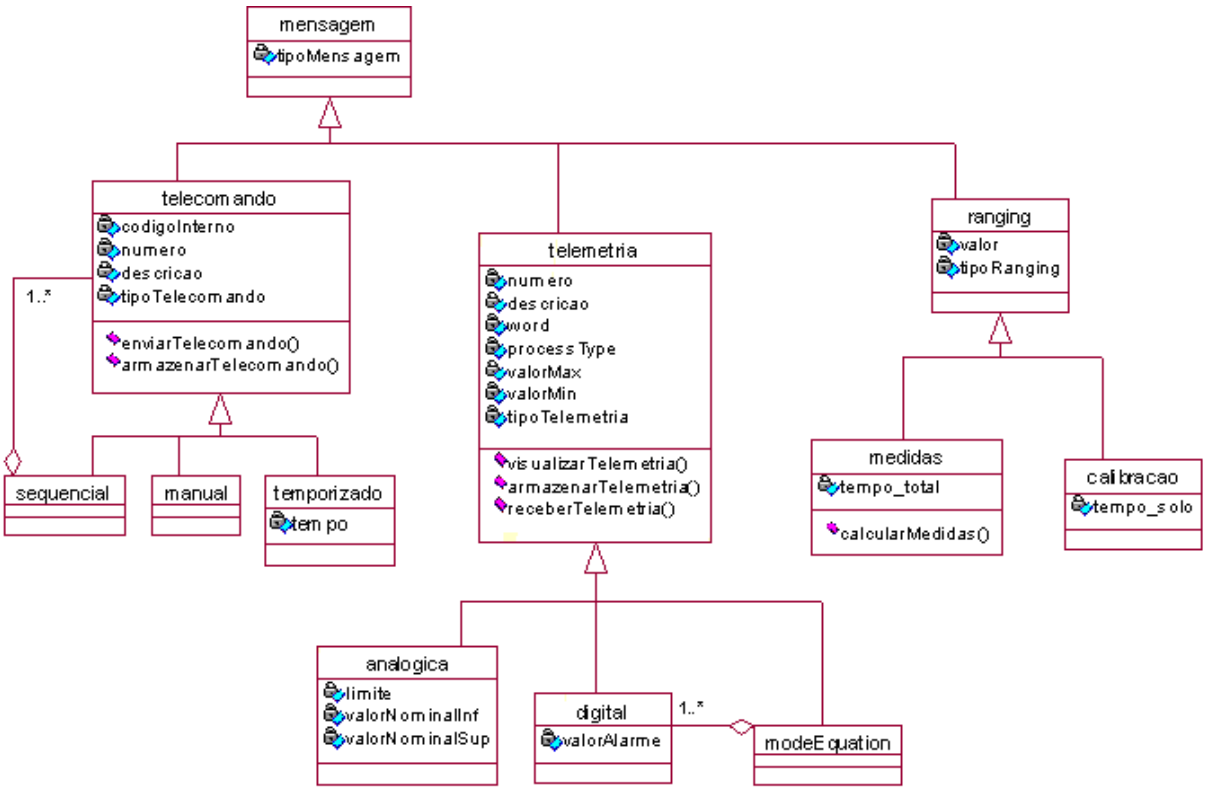


FIGURA 6.2 – Parte do modelo para o controle de satélites.

Esta Seção mostra como o usuário especialista no domínio pode criar, em tempo de execução, utilizando os protótipos desenvolvidos, um modelo correspondente ao modelo ilustrado na Figura 4.1. A Figura 6.2 ilustra a parte equivalente aos modelos que foram criados nos protótipos em tempo de execução. A seqüência de passos seguida foi:

- Criar as subclasses da classe mensagem: telecomando, telemetria e *ranging*.
- Criar as propriedades destas classes.
- Criar as subclasses das classes: telecomando, telemetria e *ranging*.
- Criar as associações entre essas classes.

- Associar regras a essas classes.

Criar instâncias da classe mensagem.O primeiro passo que o usuário especialista no domínio deve fazer é criar as três subclasses da classe mensagem: telecomando, telemetria e *ranging* que no metamodelo implementado nos protótipos e ilustrado na Figura 4.8 correspondem à instâncias da classe tipoTipoMensagem. A Figura 6.3 ilustra o diagrama de seqüência para criar novos tipos dos tipos das mensagens.

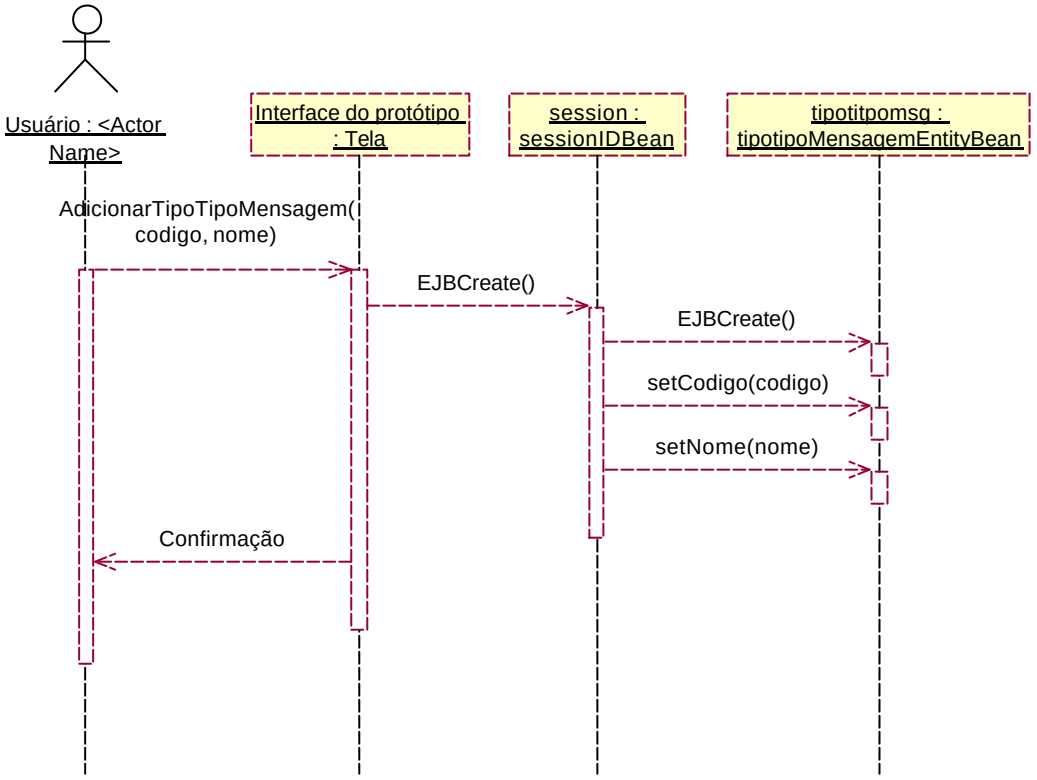


FIGURA 6.3 – Diagrama de seqüência para criar tipos de tipos de mensagens.

O diagrama de seqüência ilustrado pela Figura 6.3 foi implementado pela tela ilustrada na Figura 6.4, que ilustra a criação do tipo do tipo de mensagem Telecomando, que corresponde à classe telecomando ilustrada na Figura 6.2. Para obter o modelo da Figura 6.2 o usuário especialista do domínio deve ainda criar as classes Telemetria e *Ranging*.

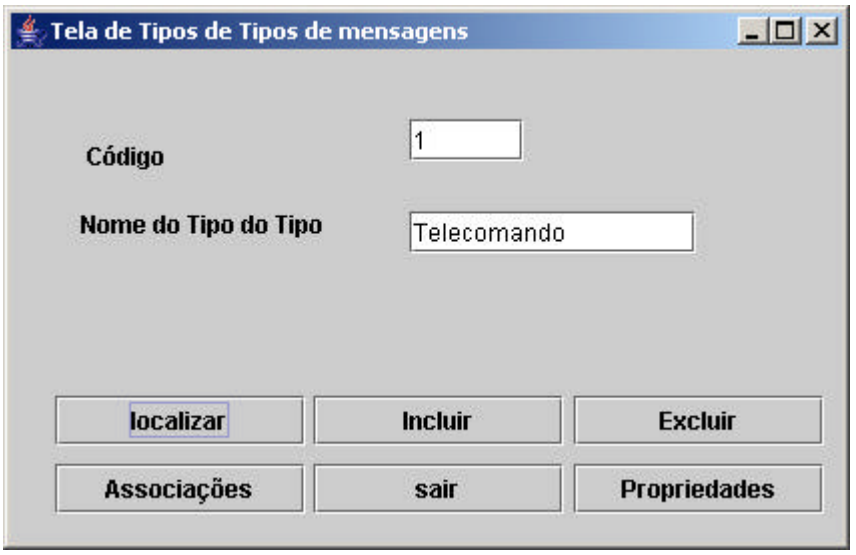


FIGURA 6.4 – Tela dos tipos de tipos de mensagens.

O próximo passo é criar as propriedades dessas classes. A Figura 6.5 ilustra o diagrama de seqüências para criar propriedades para os tipos de tipos de mensagens.

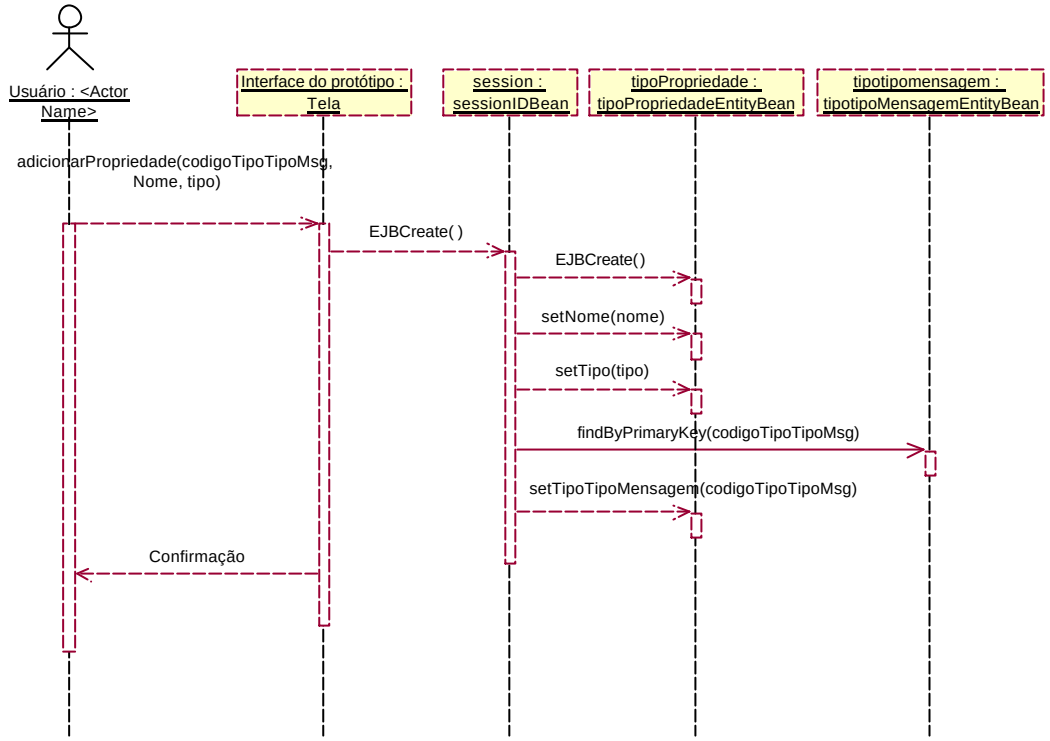


FIGURA 6.5 – Diagrama de seqüência para criar propriedades.

A Figura 6.6 ilustra tela onde esse diagrama de seqüência foi implementado, a tela mostra as propriedades criadas para a classe Telecomando conforme ilustrado na Figura 6.2. Essa tela é um editor de propriedades e também um editor de metadados. Ela tem a função de adicionar e remover propriedades para as mensagens criadas com um determinado tipo de tipo de mensagem. Utilizando essa tela, o usuário especialista no domínio pode definir por exemplo as propriedades codigoInterno, numero, descricao e tipoTelecomando, para a classe Telecomando, do modelo ilustrado na Figura 6.2.

O botão “Confirmar alterações”, da tela ilustrada na Figura 6.6, cria uma instância da classe tipoDePropriedade, para cada propriedade criada pelo usuário especialista no domínio, e associa esses tipos de propriedade ao tipo de tipo de mensagem que estiver sendo editado. O botão “Confirmar alterações”, também remove as instâncias dos tipos de propriedades das propriedades que o usuário especialista no domínio excluiu com o botão Remover.

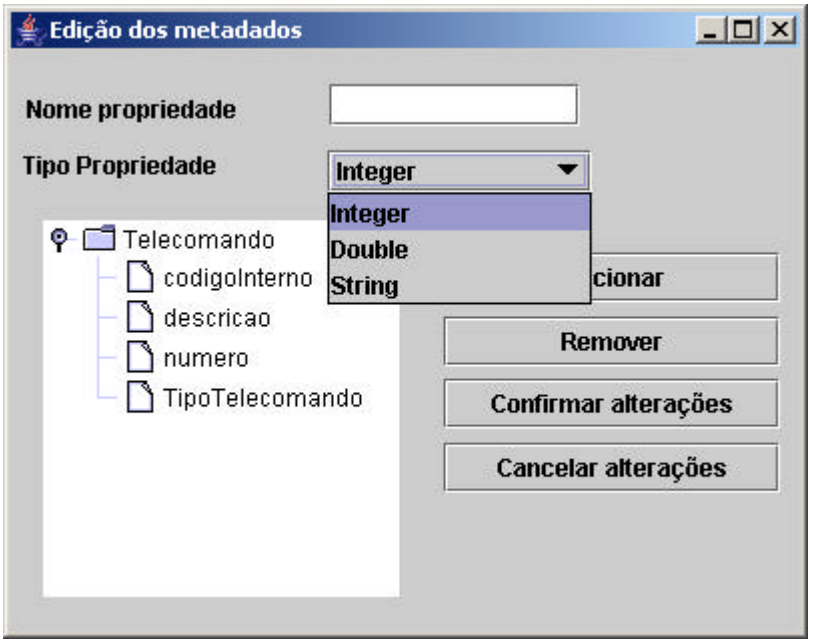


FIGURA 6.6 – Editor de propriedades.

O terceiro passo é criar as subclasses das classes Telecomando, Telemetria e *Ranging*. Essas classes correspondem a instâncias da classe TipoMensagem do metamodelo ilustrado na Figura 4.8. A Figura 6.7 ilustra o diagrama de seqüência para criar tipos de mensagens.

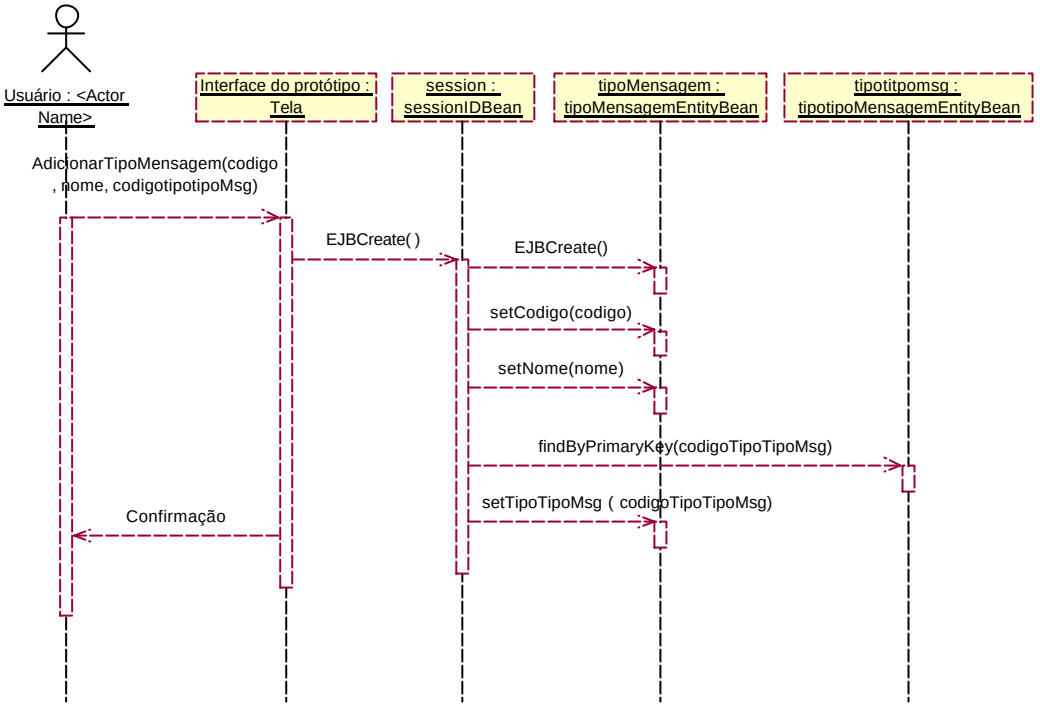


FIGURA 6.7 – Diagrama de seqüência para criar tipos de mensagens.

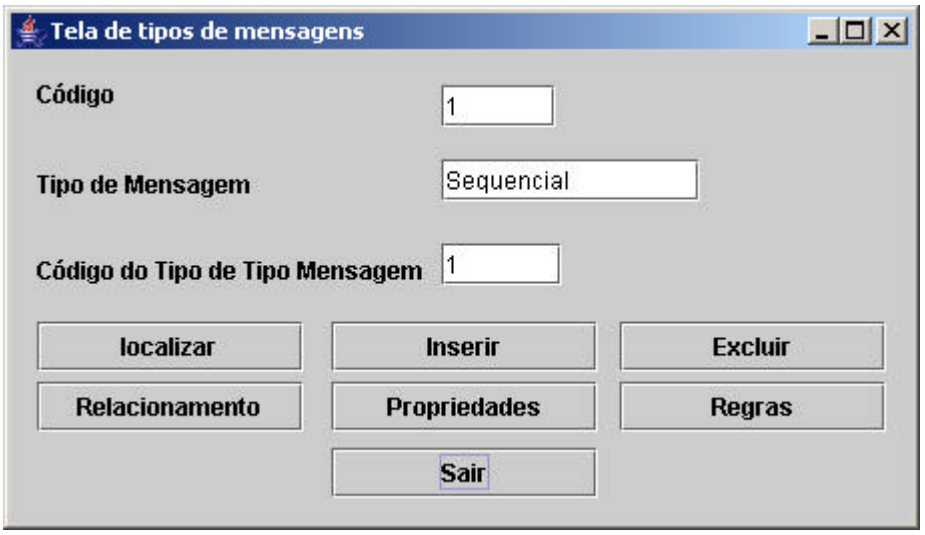


FIGURA 6.8 – Tela dos tipos de mensagens.

Esse diagrama de seqüência foi tratado na tela ilustrada pela Figura 6.8. Nessa tela o usuário especialista no domínio pode criar as classes: sequencial, manual e temporizado associadas à classe telecomando; analógica, digital e modeEquation associadas à classe telemetria e medidas e calibração associadas à classe *ranging*.

O próximo passo é criar as associações entre as classes. No modelo da Figura 6.2 existem apenas duas associações. Uma entre as classes telecomando e sequencial e a outra entre as classes digital e modeEquation. A Figura 6.9 ilustra o diagrama de seqüência para criar associações.

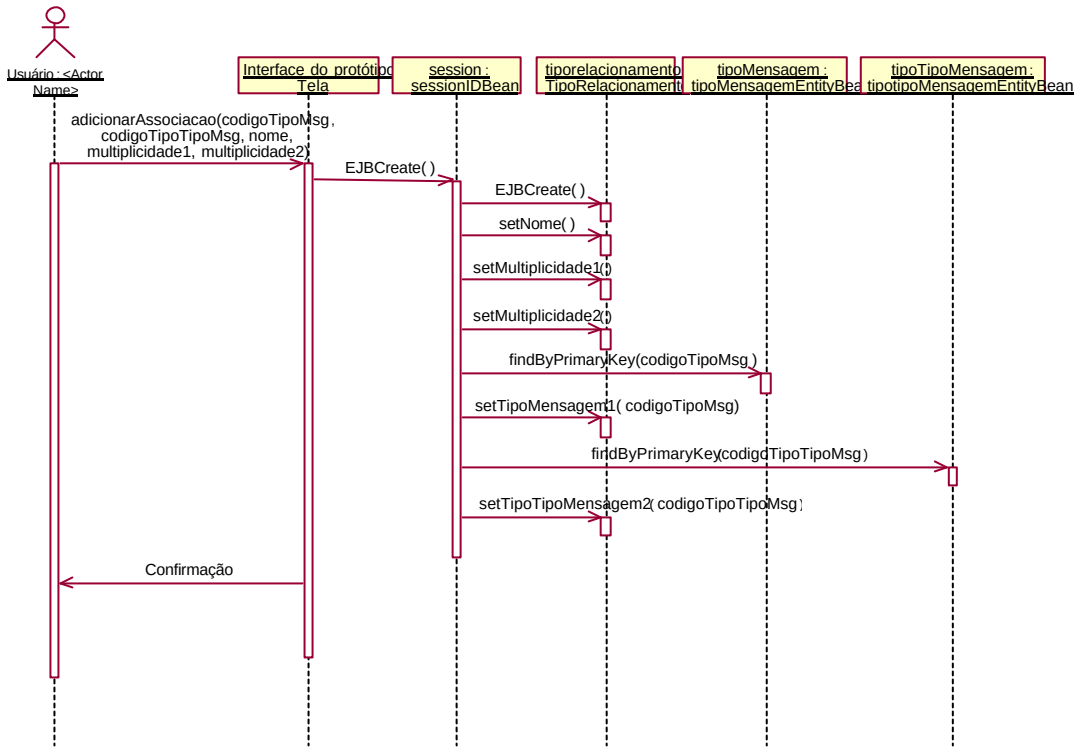


FIGURA 6.9 – Diagrama de seqüência para criar associações.

Esse diagrama de seqüência foi implementado na tela ilustrada na Figura 6.10. Essa tela é um editor de associações e outro editor de metadados. Nessa tela a primeira informação inserida se trata do nome da associação. Em seguida o usuário especialista

no domínio deve informar as multiplicidades das classes na associação: as multiplicidades 1 e 2.

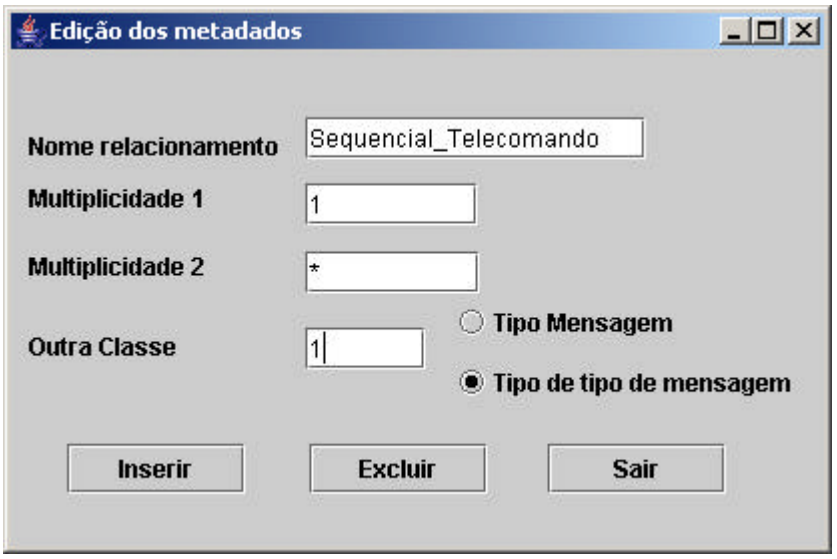


FIGURA 6.10 – Editor de associações.

Por último, o usuário especialista no domínio, deve informar o nome da outra classe na associação. Após digitar o nome da outra classe na associação o usuário deve informar a qual classe marcada com o estereótipo «TipoEntidade» ela pertence (a definição dos estereótipos se encontra na Tabela 5.1). Neste domínio existem apenas duas classes com esse estereótipo, mas em outros domínios de informação o número de classes com esse estereótipo pode ser maior. Então toda ferramenta para a edição de associações entre as classes deve fornecer meios para que os usuários especialistas no domínio configurem com qual classe marcada com o estereótipo «TipoEntidade» a classe editada se relaciona.

A forma adotada nos protótipos foi listar as duas classes com o estereótipo «TipoEntidade» e permitir que o usuário especialista no domínio selecione uma das duas para participar da associação. Isso permite que se crie associações entre classes diferentes, marcadas com o estereótipo «TipoEntidade».

A Figura 6.10 ilustra a criação da associação entre as classes telecomando e sequencial do modelo mostrado na Figura 6.2. O usuário especialista no domínio, primeiramente criou o tipo de tipo de mensagem telecomando e o tipo de mensagem sequencial. Em seguida ele editou o tipo de mensagem sequencial, preencheu os valores da associação no sentido da classe sequencial para a classe telecomando. Digitou o código da classe telecomando na caixa de texto “Outra Classe”, e informou que a classe telecomando é um tipo de tipo de mensagem, marcando o botão rádio “Tipo de tipo de mensagem”.

O passo seguinte é associar regras aos tipos de mensagens criados. A Figura 6.11 ilustra o diagrama de seqüência para associar regras.

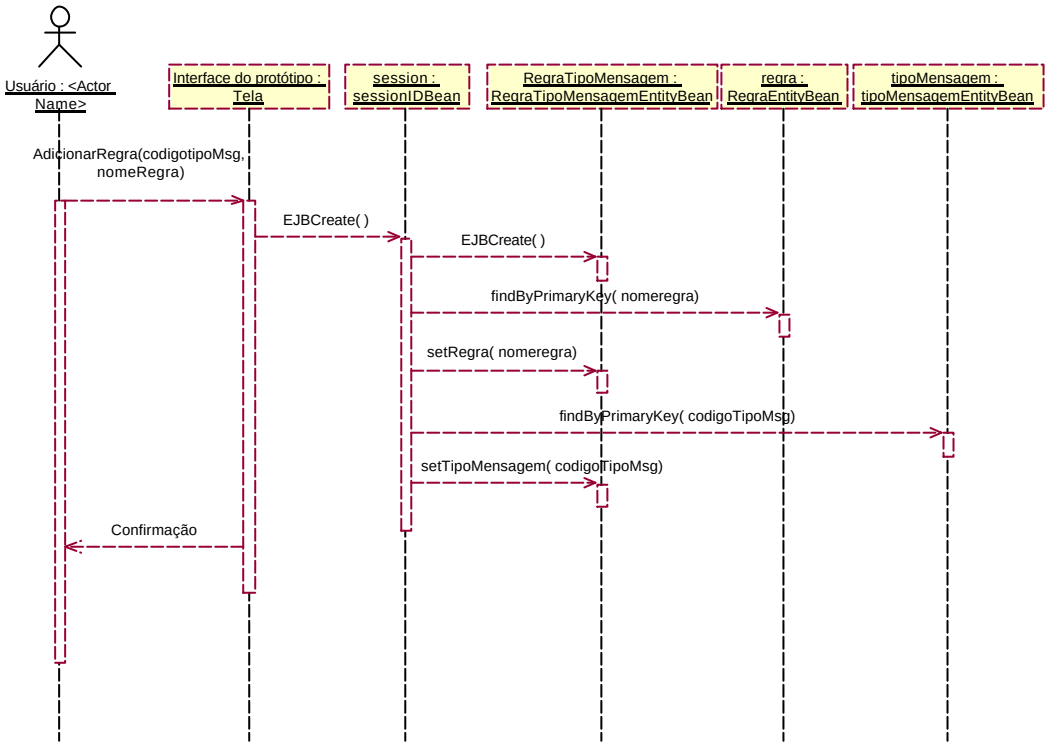


FIGURA 6.11 – Diagrama de seqüência para associar regras.

A Figura 6.12 ilustra a tela que implementou esse diagrama de classes. Nela o usuário especialista deve informar o nome da regra e indicar quais propriedades da classe são

utilizadas pela regra (Quando a regra possuir parâmetros). Essa tela ilustra a associação da regra enviarTelecomando a classe telecomando.

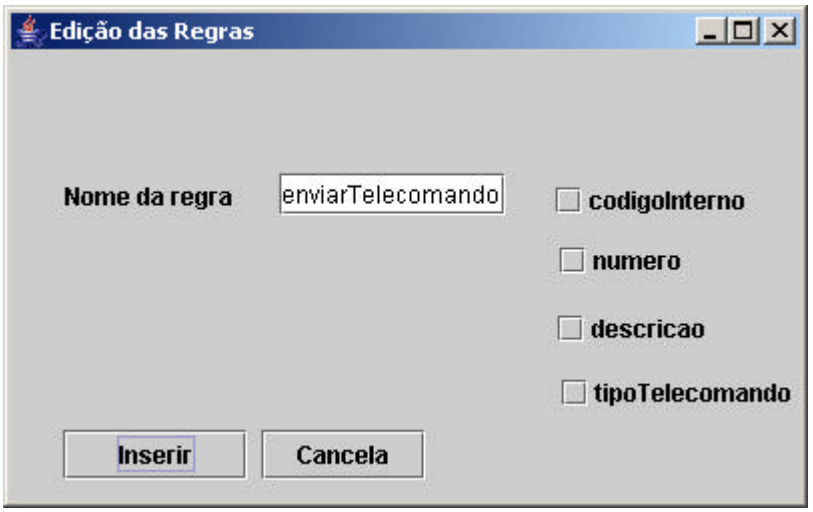


FIGURA 6.12 – Editor de associação de regras.

O último passo é o usuário inserir mensagens. A Figura 6.13 ilustra o diagrama de seqüência para inserir mensagens. Esse diagrama de seqüência foi implementado no protótipo pela tela ilustrada, em duas situações, na Figura 6.14. Nessa tela pode-se visualizar todas as configurações feitas pelo usuário especialista no domínio nas telas apresentadas anteriormente. Nessa tela o usuário pode associar as mensagens aos tipos criados e utilizar as propriedades criadas, fazer as associações criadas e executar as regras associadas à esses tipos. A Figura 6.14 mostra a tela que manipula as mensagens em duas situações:

- A primeira não se insere ou edita alguma mensagem.
- A segunda situação ilustra a edição de uma mensagem associada ao tipo de mensagem sequencial. Pode-se ver na tela as propriedades definidas para o tipo de mensagem sequencial e para o tipo de tipo de mensagem telecomando.

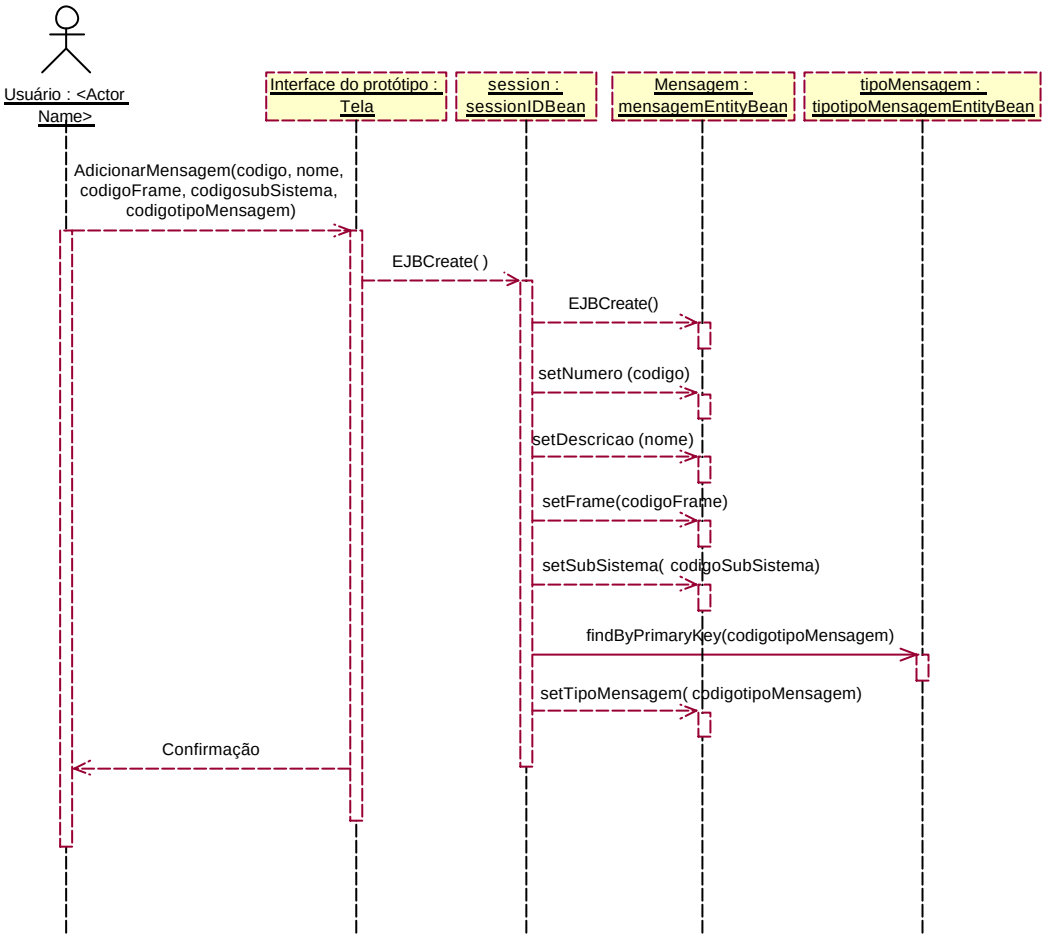


FIGURA 6.13 – Diagrama de seqüência para criar mensagens.

Quando o usuário precisar associar duas mensagens, através dos tipos de associações criados pelo usuário especialista no domínio para o tipo de mensagem dessa mensagem, ele deve selecionar uma das mensagens e em seguida clicar no botão Relacionamento.

O botão Relacionamento lista todos tipos de associações definidas para essa mensagem, baseado em seu tipo de mensagem. Quando o usuário especialista no domínio, selecionar um tipo de associação, o protótipo lista todas as mensagens que podem ser associadas a essa mensagem, baseado no tipo de associação escolhido. A Figura 6.15 ilustra a tela onde o usuário pode associar mensagens.

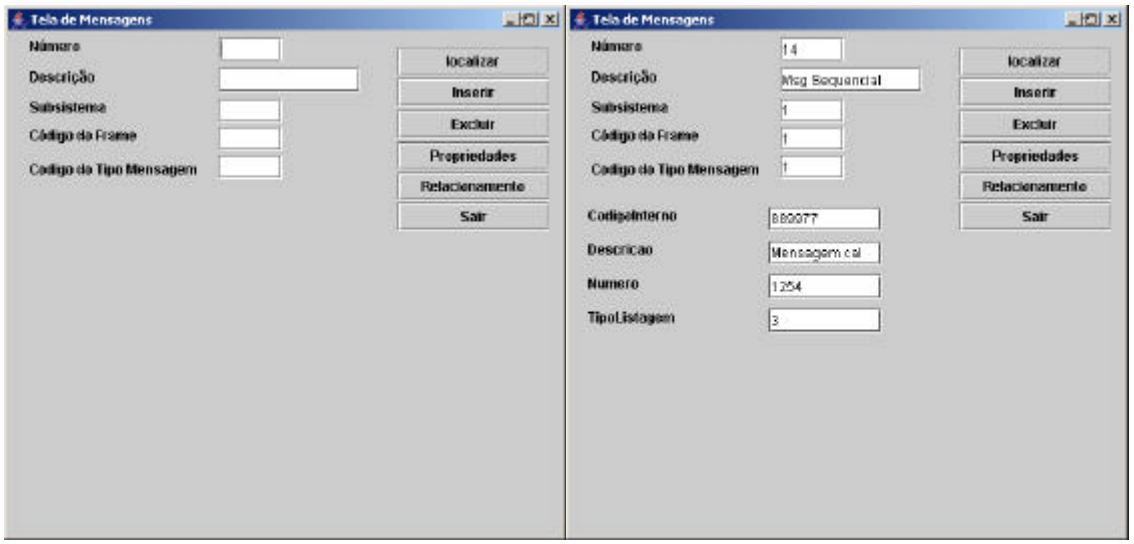


FIGURA 6.14 – Telas de manipulação de mensagens.

O usuário pode selecionar uma (ou mais de uma dependendo da multiplicidade da associação) das mensagens disponíveis para serem associadas a mensagem e confirmar clicando no botão Confirma.

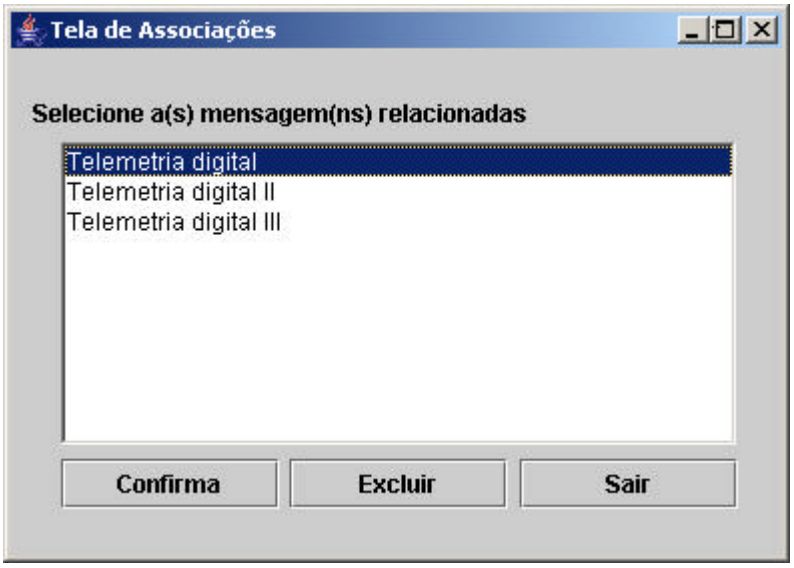


FIGURA 6.15 – Tela de associações de mensagens.

As partes do protótipo mostradas nesta Seção, podem dar uma idéia do funcionamento de um sistema desenvolvido baseado nos AOMs, sua capacidade de criar e configurar um modelo em tempo de execução, além de mostrar a criação de editores para a criação e edição do modelo de objetos pelo usuário especialista do domínio, em tempo de execução, como os editores de propriedades ilustrados na Figura 6.6, os editores de associações ilustrados na Figura 6.10 e os editores de regras ilustrados na Figura 6.12.

A interface dos três protótipos desenvolvidos é a mesma, a diferença se encontra apenas no sistema de gerenciamento de dados utilizado por cada um deles. Os três protótipos foram desenvolvidos utilizando os mapeamentos dos modelos de objetos adaptáveis para os sistemas de gerenciamento de dados, definidos no Capítulo anterior

6.2 Comparação Qualitativa entre os Sistemas de Gerenciamento de Dados

Como já foi dito anteriormente, para este trabalho desenvolveu-se um protótipo para cada um dos três sistemas gerenciadores de dados escolhidos. O primeiro realiza a persistência dos modelos adaptáveis no Caché. No desenvolvimento do segundo protótipo, que persiste os modelos de objetos adaptáveis no PostgreSQL, foi realizada pequenas alterações em relação ao primeiro protótipo, as modificações foram basicamente na parte de acesso aos dados. Já no desenvolvimento do terceiro protótipo, que persiste os modelos adaptáveis em um NXD, as mudanças foram maiores, principalmente por se tratar de uma nova tecnologia que ainda não oferece as facilidades encontradas nos SGBDR e SGBDOO.

A comparação realizada neste trabalho concentrou-se nos seguintes pontos: (1) armazenar os modelos de objetos; (2) recuperar o modelo de objetos e (3) facilidades encontradas na codificação dos protótipos. A seguir, mostra-se as conclusões dessas questões sobre cada um dos sistemas de gerenciamento de dados utilizados.

6.2.1 Armazenar os Modelos de Objetos

Armazenar os modelos de objetos adaptáveis em um SGBDOO tem as vantagens oferecidas por esse sistema de gerenciamento de dados:

- As classes do metamodelo são criadas no banco de dados da mesma maneira que no metamodelo.
- As heranças de classes no metamodelo são criadas no banco da mesma forma que no metamodelo.
- Em um SGBDOO você pode persistir diretamente uma classe com um atributo que possua uma coleção de objetos de outra classe. No desenvolvimento do protótipo, por exemplo, a classe RegraTipoMensagem e a classe RegraTipoTipoMensagem possuem o atributo propriedadesassociadas que é uma coleção de objetos da classe tipoDePropriedade (No Caché definido da seguinte forma: Property propriedadesassociadas as tipoDePropriedade [Collection=list]).
- Oferece a possibilidade da definição de métodos nas próprias classes.

Em contrapartida, sobre esse sistema de gerenciamento de dados pode-se citar as seguintes desvantagens:

- Devido a larga utilização e conhecimento dos SGBDs Relacionais, a utilização de um SGBDOO ainda encontra certas resistências por parte dos desenvolvedores de sistemas.
- Normalmente, os SGBDOO são vinculados à alguma linguagem específica, isso significa que uma aplicação desenvolvida em outra linguagem terá dificuldade para aproveitar as vantagens oferecidas por um SGBDOO. O Caché, por exemplo, oferece o recurso de exportar as classes definidas na linguagem do Caché para as linguagens Java e C++.

- Ainda existem poucas ferramentas disponíveis no mercado.

Armazenar os modelos de objetos adaptáveis em um SGBDR mostrou as seguintes vantagens:

- O SGBDR é o sistema de gerenciamento de dados mais utilizado e conhecido pelos desenvolvedores.
- Possui diversas ferramentas disponíveis no mercado, inclusive várias ferramentas gratuitas, como o PostgreSQL que foi utilizado.
- Completamente independente da linguagem de programação da aplicação.

Entretanto foi verificado os seguintes pontos negativos oferecidos por esse sistema de gerenciamento de dados:

- Impedância do mapeamento objeto-relacional, pois as classes e os relacionamentos devem ser aplanados (*flattened*) em relações (Manolescu, 2001).
- Maior dificuldade para armazenar atributos multivalorados, especialmente os atributos das classes que são coleções de instâncias de outras classes.

Na utilização de um NXD para armazenar os modelos de objetos adaptáveis, observou-se as seguintes vantagens:

- A linguagem XML é um formato de dados portável.
- A linguagem XML é independente de fabricante e oferece alto nível de interoperabilidade e pode ser utilizada sem preocupação com a dependência do fabricante.
- Os modelos de objetos, desde que não possua tipos de dados complexos, podem ser representados e armazenados em arquivos XML.

Por outro lado, esse sistema gerenciador de dados possui os seguintes pontos negativos:

- Como se trata de uma tecnologia muito nova, ainda não se pode garantir que venha a se tornar um padrão para gerenciamento de dados como os dois anteriores.

- A maioria das ferramentas disponíveis ainda estão em sua primeira versão e essas ferramentas ainda não oferecem muita estabilidade.

- Poucos tipos de dados podem ser armazenados em um arquivo XML e conseqüentemente em um NXD. Um arquivo XML não pode armazenar, por exemplo, arquivos de imagens bitmap e *strings* de texto longo, também conhecidos como grandes objetos binários ou *Binary Large OBjects* (BLOBs). Essa característica pode tornar inviável a sua utilização para o desenvolvimento de sistemas baseados nos AOMs para determinados domínios de informação.

- Deve ser utilizado um mapeamento dos objetos para arquivos XML e vice-versa, o que causa uma impedância, assim como nos SGBDR.

6.2.2 Recuperar o Modelo de Objetos

A recuperação do modelo de objetos em um SGBDOO apresentou alguma dificuldade especialmente nas associações entre as classes. Como não é possível fazer a junção de classes em consultas de um SGBDOO, no protótipo as consultas foram resolvidas graças a uma característica do Caché de fornecer suporte a linguagem SQL.

A recuperação do modelo de objetos em um SGBDR foi a principal vantagem desse sistema de gerenciamento de dados. A linguagem SQL se mostra eficiente em consultas, além de ser um padrão conhecido e utilizado pela maioria dos desenvolvedores, tanto que até mesmo o SGBDOO utilizado oferece suporte a essa linguagem. Como desvantagem tem o fato de não recuperar um objeto como acontece no SGBDOO, recupera apenas os atributos desse objeto e ele deve ser instanciado pela aplicação.

A recuperação do modelo de objetos em um NXD foi complicada pelo fato de não se conseguir utilizar a linguagem de consulta do NXD. A linguagem de consulta XPath[1], que consta no manual do Xindice, não funcionava como recomendado. Então o protótipo que utilizava o NXD recuperava completamente o modelo para a memória e as consultas eram realizadas pelo protótipo.

6.2.3 Desenvolvendo o Protótipo

Esta Seção descreve as vantagens e desvantagens encontradas no desenvolvimento de cada um dos três protótipos, utilizando as mesmas ferramentas escolhidas para o desenvolvimento, que foram citadas na Seção 6.1.

No desenvolvimento do primeiro protótipo, o Caché ofereceu várias facilidades para o desenvolvimento de uma aplicação J2EE. Uma vez criadas as classes do metamodelo no banco, o Caché gerou um EJB de entidade para cada classe do metamodelo para o servidor de Aplicação JBoss. O Caché criou também todos os arquivos necessários para a publicação dos EJBs de entidade no JBoss.

Os EJBs de sessão necessários foram criados no JBuilder e o código que faz a conexão com o banco de dados pode ser simplesmente copiado dos EJBs de entidade gerados automaticamente pelo Caché. Isso significou um considerável ganho de tempo, pois todas a conexões entre o servidor de aplicação (JBoss) o IDE (JBuilder) e o servidor de dados (Caché) pode ser entendido pelos códigos gerados pelo Caché ao exportar suas classes como EJBs de entidade para o servidor de aplicação JBoss.

Nesse protótipo os modelos de objetos ficam armazenados no banco e quando o usuário solicita um objeto, esse objeto é buscado do banco e disponibilizado para o usuário. Em algumas situações, onde o usuário solicita associações entre classes é utilizado a linguagem SQL para recuperar essas associações.

[1] O Xpath é um padrão do W3C que descreve uma sintaxe para selecionar partes de um documento XML (Bond et al., 2003).

Através da listagem dos objetos armazenados nas classes no Caché, o usuário especialista no domínio consegue ter uma visualização dos modelos de objetos persistidos, pois as classes ficam armazenadas como elas foram definidas no metamodelo. As Tabelas 5.4 e 5.5 que exemplificam a persistência de metadados foram tiradas do Caché.

No desenvolvimento do segundo protótipo, o PostgreSQL não ofereceu nenhuma facilidade quanto a geração automática dos EJBs de entidade, sendo todos criados na IDE do JBuilder pelo desenvolvedor, o que se tornou uma desvantagem. Os EJBs de sessão utilizados também foram criados da mesma forma no IDE do JBuilder.

Uma grande vantagem no desenvolvimento desse protótipo foi a possibilidade de testar a distribuição dos EJBs de sessão e de entidade, já que o PostgreSQL é uma ferramenta livre e suporta o acesso distribuído dos dados, possibilitando assim, a distribuição dos EJBs em diversas máquinas.

No terceiro protótipo a forma de desenvolvimento teve de ser alterada em relação aos dois primeiros protótipos. Nos dois primeiros protótipos quando o usuário requisita um objeto, o protótipo recupera do banco de dados aquele determinado objeto e o disponibiliza para o usuário. Isto não foi possível no NXD utilizado, o XIndice, pois não foi possível utilizar as linguagens de consulta (XPath) e atualização (XUpdate) dessa ferramenta.

Esse protótipo busca o arquivo XML que armazena uma classe e instancia todos os objetos dessa classe em um vetor. Toda instância dessa classe que o usuário especialista no domínio cria, altera ou excluí é refletido no vetor. Quando o usuário especialista no domínio não utiliza mais essa classe, o protótipo gera um arquivo XML com as instâncias armazenadas e substitui o arquivo XML correspondente no NXD.

Esse sistema gerenciador de dados apresentou mais dificuldades no desenvolvimento do protótipo, principalmente por ser uma tecnologia nova, onde o desenvolvedor possuía menos conhecimento, e encontrou mais dificuldades para realizar a sua configuração com o JBuilder e o JBoss. Mas pode ser uma boa opção para o futuro e deve ser

destacado que: persistir o modelo de objetos de um aplicativo desenvolvido em Java baseado na arquitetura AOM, que não siga a especificação J2EE, pode ser facilitado pelas APIs Java para manipulação de arquivos XML, como as classes XMLEncoder e XMLDecoder.

A classe XMLEncoder armazena um objeto em um arquivo XML e a classe XMLDecoder lê esse arquivo XML e instancia o objeto que foi armazenado pelo XMLEncoder (Witthawaskul e Johnson, 2003).

Observou-se que os três sistemas de gerenciamento de dados podem ser utilizados para a persistência dos modelos de objetos adaptáveis. Cada um com suas vantagens e desvantagens. Este trabalho não tem a intenção de determinar qual desses três sistemas de gerenciamento de dados é o mais indicado para o desenvolvimento de todo sistema baseado nos AOMs, mas pelas características do domínio dos protótipos desenvolvidos conclui-se que para esse domínio o SGBDOO seria o mais indicado.

No início do desenvolvimento de um sistema baseado nos AOMs o processo de escolha do sistema de gerenciamento de dados utilizado vai depender de uma série de fatores, tais como experiências da equipe de desenvolvimento, ambiente computacional, licenciamento de *software*, desempenho do *software* entre outros. As vantagens e desvantagens de cada um encontradas neste trabalho deve ser apenas mais um fator considerado no processo de escolha. As Tabelas 6.1 e 6.2 mostram respectivamente as principais vantagens e desvantagens de cada sistema de gerenciamento de dados.

TABELA 6.1 – Principais vantagens de cada sistema de gerenciamento de dados.

Item	SGBDOO	SGBDR	NXD
Armazenar o modelo	Não necessita de mapeamento dos objetos implementados para os objetos no banco de dados.	Independente de linguagem de programação.	Portabilidade e independência de ambiente computacional.
Recuperar o modelo	Recupera o objeto com seus atributos e métodos.	A linguagem SQL.	Não observado.
Implementação do sistema	Os objetos persistidos são os mesmos da implementação.	Grande número de ferramentas e APIs disponíveis como a JDBC.	Facilita o transporte de dados e a implementação de sistemas baseados na Internet.

TABELA 6.2 – Principais desvantagens de cada sistema de gerenciamento de dados.

Item	SGBDOO	SGBDR	NXD
Armazenar o modelo	Vinculado a linguagens de programação específicas e poucas ferramentas disponíveis.	Impedância do mapeamento objeto-relacional, não armazena diretamente atributos multivalorados.	Poucas ferramentas disponíveis e nem sempre estáveis ainda. Armazena poucos tipos de dados e também possui a impedância de um mapeamento para os objetos.
Recuperar o modelo	Dificuldade para recuperar classes associadas.	Recupera apenas os atributos de um objeto que precisa ser instânciado pela aplicação.	A linguagem de consulta não funcionou como especificado.
Implementação do sistema	As dificuldades foram descobrir ferramentas compatíveis e não encontrou-se um SGBDOO de software livre.	Sem muita integração com o IDE e com o servidor de aplicação, falta de ferramentas gráficas para a criação dos EJBs. Poucos passos realizados automaticamente.	Sem muita integração com o IDE e com o servidor de aplicação, falta de ferramentas gráficas para a criação dos EJBs. Falta de documentação e exemplos disponíveis, dificultando o aprendizado.

O Capítulo a seguir, mostra a conclusão deste trabalho e suas contribuições. Além de indicar possíveis trabalhos futuros.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Sistemas desenvolvidos baseados nos AOMs representam seus modelos como metadados, assim pode se realizar a persistência desses modelos. Esta dissertação conseguiu estabelecer um mapeamento para a persistência dos modelos de objetos de sistemas distribuídos, configuráveis e adaptáveis desenvolvidos baseados na arquitetura dos AOMs.

Os modelos de objetos podem ser persistidos em um SGBDOO, um SGBDR e ou em arquivos XML. Então desenvolveu-se um mapeamento do modelo de objetos do sistema para um banco de dados orientados a objeto, outro mapeamento do modelo de objetos do sistema para um banco de dados relacional e por último um mapeamento do modelo de objetos para arquivos XML, armazenados em um NXD. Como estudo de caso, os mapeamentos foram aplicados em três protótipos que persistiam seus modelos de objetos nesses sistemas de gerenciamento de dados.

Os três protótipos implementados exemplificaram o funcionamento de um sistema desenvolvido baseado nos AOMs e a sua capacidade de se adaptar às mudanças no domínio, em tempo de execução, através de alterações feitas por um usuário especialista no domínio nos metadados do modelo armazenado. Essa capacidade evita que se tenha que alterar o código que implementa a aplicação quando houver a necessidade realizar mudanças no modelo de objetos.

Os protótipos tiveram sua origem na proposta de Thomé(2004) de uma arquitetura distribuída, configurável e adaptável para o controle de várias missões de satélites. As funções implementadas nos protótipos foram as que Thomé destinou ao serviço de configuração de sua arquitetura.

7.1 Contribuições

Esta dissertação busca auxiliar o desenvolvimento de sistemas baseados nos modelos de objetos adaptáveis. Este trabalho obteve as seguintes contribuições:

- Mostrou ser possível mover o modelo de objetos de um sistema do código da aplicação para um banco de dados, permitindo que alterações no modelo do sistema sejam feitas através de modificações nesse banco de dados, evitando a necessidade de alterar o código da aplicação para satisfazer mudanças nos requisitos do domínio.

- Os protótipos desenvolvidos mostraram que é possível instanciar um modelo de objetos a partir de seus metadados persistidos em um banco de dados.

- Por meio dos protótipos mostrou-se a importância da criação de ferramentas para a edição dos metadados que descrevem os modelos de objetos adaptáveis, ou seja, um usuário especialista no domínio precisa de interfaces robustas e intuitivas para fazer a manutenção dos metadados em um banco de dados.

Esta dissertação fez também uma comparação qualitativa entre os três sistemas de gerenciamento de dados utilizados para persistência dos modelos de objetos adaptáveis, mostrando as vantagens e desvantagens de cada um:

- O SGBDOO não requer um mapeamento dos objetos da aplicação para os objetos armazenados no banco, além de recuperar o objeto com seus atributos e métodos. Em contrapartida, um SGBDOO é vinculado a alguma linguagem de programação específica, existem ainda poucas ferramentas disponíveis e apresenta dificuldade para trabalhar com associações entre classes.

- O SGBDR é independente da linguagem de programação que implementa o sistema, possui a linguagem SQL para consulta e atualizações no banco, se apresenta como o sistema de gerenciamento de dados mais conhecido pelos desenvolvedores e possui um grande número de ferramentas e APIs disponíveis. Como desvantagens existe a necessidade de um mapeamento do

modelo orientado a objetos para o modelo relacional, não armazena diretamente atributos multivalorados, recupera apenas as propriedades de um objeto e o mesmo deve ser instanciado pela aplicação.

- O XML apresenta uma grande portabilidade e é independente de ambiente computacional, além de facilitar o transporte de dados e a implementação de sistemas baseados na Internet. Por outro lado, ainda existem poucas ferramentas disponíveis e estáveis, o XML armazena poucos tipos de dados e possui a impedância do mapeamento dos objetos para um arquivo XML.

Durante a definição do mapeamento dos modelos de objetos adaptáveis para os sistemas de gerenciamento de dados, verificou-se que:

- As propriedades podem ser persistidas sem muita dificuldade e podem ser validadas em tempo de execução pela API de reflexão da linguagem Java.

- Persistir as associações é uma tarefa mais complexa devido a características como por exemplo o envolvimento de duas classes, a multiplicidade das associações e a tarefa de disponibilizar para o usuário instâncias da outra classe na associação.

- A persistência das regras se relaciona diretamente com a capacidade do sistema de recuperar uma regra do banco e executá-la. Nos protótipos foi possível associar regras, que já estavam implementadas, às classes do domínio e apenas o nome dessas regras eram persistidos. Em tempo de execução essas regras eram executadas utilizando a API de reflexão da linguagem Java.

Este trabalho ainda contribuiu para viabilizar o desenvolvimento de sistemas configuráveis e adaptáveis para o controle de satélites. Os protótipos utilizaram um metamodelo (ilustrado na Figura 4.8) para o controle de satélites.

Esses sistemas são um passo importante em relação a economicidade, uma vez que futuras missões de satélites podem reutilizar grande parte da infra-estrutura de *hardware* e *software* já desenvolvida para o sistema de controle de satélites em outras missões.

Isso vem diretamente de encontro aos objetivos do INPE, que propõe a plataforma multi-missão (MMP -*Multi-Mission Platform*) para a construção de satélites (INPE, 2001). A MMP visa à concepção de arquiteturas de *hardware* para a construção de satélites mais flexíveis e configuráveis. Assim, esses satélites necessitam de um sistema de *software* para seu controle igualmente configurável e adaptável.

7.2 Trabalhos Futuros

Como o AOM é um novo paradigma para o desenvolvimento de sistemas, pode-se destacar diversos trabalhos futuros. Como foi dito no final do Capítulo 2, os sistemas desenvolvidos baseados nos AOMs trabalham em um nível de abstração superior à maioria dos sistemas (o nível dos metamodelos). Caso esse tipo de arquitetura venha a ser amplamente utilizado no desenvolvimento de sistemas, pode-se estudar os impactos causados por seu nível de abstração nos processos de engenharia de *software* para o desenvolvimento de um sistema.

Para aumentar a adaptabilidade dos métodos, em tempo de execução, pode-se estudar a construção de uma linguagem específica para o domínio que se destina um sistema desenvolvido baseado nos AOMs.

Em relação à persistência, pode-se trabalhar a utilização do Hibernate da linguagem Java, que torna a aplicação independente do SGBD utilizado. O Hibernate abstrai o código SQL da aplicação e permite escolher o SGBD utilizado, enquanto o programa está executando, permitindo mudar sua base de dados sem alterar o seu código Java. Assim, o sistema pode trocar de SGBDR, por exemplo do PostgreSQL para o Oracle ou o DB2 sem a necessidade de reescrita de código (Hibernate, 2004). Pode-se também estudar a aplicação do padrão de projeto *Model View Controler* (MVC) no desenvolvimento de um sistema baseado nos AOMs. Uma vez que o usuário especialista no domínio está criando um modelo de objetos, o padrão de projeto MVC poderia cuidar da apresentação desse modelo aos usuários.

Uma outra questão a se considerar é o controle de versões para tratar as alterações no modelo de objetos realizadas por usuários especialistas no domínio. Um controle de versões poderia aumentar a segurança desse tipo de sistema, pois o sistema poderia voltar a uma condição anterior. Um controle de versões contornaria, por exemplo, o problema de um usuário especialista no domínio introduzir um erro no sistema devido a uma alteração indevida no modelo de objetos.

Pode-se também estudar a criação de ferramentas que criem automaticamente um metamodelo a partir de um modelo inicial, evitando assim todos os passos descritos no Capítulo 4 deste trabalho.

Este trabalho espera, por meio dos resultados obtidos, facilitar o desenvolvimento de sistemas baseados nos AOMs, facilitando também o desenvolvimento de sistemas distribuídos, configuráveis e adaptáveis para o controle de satélites que consigam atender às necessidades da proposta do INPE da plataforma multi-missão (MMP). Pretende-se dessa forma diminuir o tempo e o esforço necessário para atender os requisitos de *software* das novas missões espaciais.

As idéias desta dissertação já foram publicadas, na forma de artigos, até o momento em: Almeida e Ferreira(a)(2004), Almeida e Ferreira (b)(2004) e Almeida e Ferreira (c)(2004). Também foi submetido um artigo para a Revista de Informática Teórica e Aplicada (RITA) em março de 2005.

REFERÊNCIAS BIBLIOGRÁFICAS

Almeida, W. R.; Ferreira, M.G.V.(a). Especificar a persistência dos modelos de objetos de sistemas distribuídos e adaptáveis aplicados ao controle de satélites. In: Workshop dos Cursos de Computação Aplicada do INPE, 4., 2004, São José dos Campos, SP. **Anais ...** São José dos Campos: INPE. Out. 2004

Almeida, W. R.; Ferreira, M.G.V.(b). Persistir os Metadados dos Modelos de Objetos de Sistemas Distribuídos e Adaptáveis. In: Simpósio de Informática do CEFET-PI, 2., 2004, Teresina, PI. **Anais ...** Teresina: CEFET. Nov 2004.

Almeida, W. R.; Ferreira, M.G.V.(c). Uma abordagem para a persistência dos modelos de objetos de sistemas distribuídos e adaptáveis. In: Simpósio de Desenvolvimento de Manutenção de *Software*s da Marinha, 4., 2004, Rio de Janeiro, RJ. **Anais ...** Rio de Janeiro: Marinha do Brasil. (ISBN: 85-98947-01-6). Dez. 2004. 1 CD ROM.

Arsanjani, A. Rule Object: a pattern language for adaptive and scalable business rule construction. In: Conference on Patterns Languages of Programs (PloP'2000) Washingtom. **Proceedings ...** Washington: University Departament of Computer Science, 2000.

Arsanjani, A. Alpigini, J. Using Grammar-Oriented Object Design to seamlessly map business models to component-based *software* architectures. In: Proceedings of The International Association of Science and Technology for Development, Pittsburgh, PA, 2001. **Anais ...** Pittsburgh: IASTD, 2001.

Auth, G.; Von Maur, E.; Helfert, M. A Model-based Software Architecture for Metadata Management in Data Warehouse Systems. In: Proceedings of Fifth International Conference on Business Information Systems (BIS '02), Poznan Poland, 2002, **Anais ...** Poznan, Polan, 2002. p. 34-40.

Bond, M. et al. **APRENDA J2EE EM 21 DIAS.** São Paulo: Makron Books, 2003.

Cardoso, P. E. **Modelo de objetos dinâmico aplicado ao processamento de telemetrias de satélites**. 2005. Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais. São José dos Campos.

Caché, post-relational Database. Disponível em: <http://www.intersystems.com/cache/index.html>. Acesso em: 12 dezembro 2004.

Chang, D. T. **Common Warehouse Metamodel (CWM), UML and XML**. Arlington, VA, 2000. Presentation to the Metadata Conference/DAMA Symposium, Arlington, VA in 03/22/2000.

Elmasri, R.; Navathe, S. B. **Sistemas de Bancos de Dados – Fundamentos e Aplicações.** Rio de Janeiro: LTC, 2002. 3ª Edição. 837p.

Eriksson, H.; Penker, M. **UML Toolkit**. New York: John Wiley & Sons, 1998.

Estrella, F. et al. Meta-data Objects as the Basis for System Evolution. In: Second International Conference on Advances in Web-Age Information Management, 2001. **Proceedings** ... p: 390 – 399. (ISBN:3-540-42298-6).

Ferreira, M. G. V. **Uma arquitetura flexível e dinâmica para objetos distribuídos aplicada ao software de controle de satélites.** 2001. Tese (Doutorado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos. 2001.

Foote, B.;Yoder, J.W. Metadata and Active Object Models. In: Technical Report#WUCS-97-34 (PLoP'98),1998. **Anais ...** Washington: Dept. of Computer Cience, Washington University, 1998.

Fowler, M. **Analisys Patterns: reusable object models**. USA: Addison-Wesley, 1997. 357p.

France, R.; et al. A metamodeling approach to pattern-based model refactoring. **IEEE Software**. p. 52-58. 2003.

Gabrick, A. K.; Weiss, D. B. **J2EE and XML development.** Greenwich: Manning publications company, 2002.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. **Elements of Reusable Object-Oriented Software**. USA: Addison-Wesley, 1995.

Green D. The ReflectionAPI**. Sun Microsystems**. Disponível em: <http://java.sun.com/docs/books/tutorial/reflect >. Acesso em: 10 agosto 2004.

Hibernate 2. **Reference Documentation**. Disponível em: http://hibernate.bluemars.net. Acesso em: 04 novembro 2004.

INPE, Instituto Nacional de Pesquisas Espaciais. **Multi-Mission Platform Specification**. Number A822000-PRR-01/03. August of 2001.

Johnson, R.; Woolf, B. Type Object. In: **Pattern Languages of Program Design 3**. USA: Addison-Wesley, 1998. Cap. 8, p 47-66.

Kassem, N.; et al. Designing Enterprise Applications with the J2EE^{TM} Platform, Second Edition, **Sun Microsystems.** Disponível em: <http://java.sun.com>. Acesso em: 20 junho 2004.

Ledeczi, A.; Karsai, G.; Bapty, T. Synthesis of self-adaptive software. **IEEE Aerospace Conference Proceedings**. USA. V 4, pg. 501-507. 2000.

Manolescu, D.A. **Micro-Workflow: a Workflow architecture supporting compositional object oriented software development**. 2001. PhD Thesis – University of Illinois at Urbana-Champaign, Illinois, 2001.

Nambiar, U.; Lacroix, Z; Bressan, S; Lee, M. L; Li, Y. Current approaches to XML management. **IEEE Internet Computer**, pp. 43-51, julho-agosto, 2002.

Nordstrom, G.; Sztipanovits, J.; Karsai, G.; Ledeczi, A. Metamodeling-Rapid design and evolution of domain-specific modeling environments. In: 6th Symposium on

Engineering of Computer-Based Systems (ECBS '99), 7-12 March 1999, Nashville, TN, USA. **IEEE Computer Society**, 1999. (ISBN 0-7695-0028-5). p 68-74.

Object Management Group. **Common Warehouse Metamodel™ (CWM™) Specification, v1.1**. Disponível em: <http://www.omg.org/>. Acesso em: 02 out. 2004.

Object Management Group. **Meta-Object Facility (MOF™), version 1.4.** Disponível em: <http://www.omg.org/>. Acesso em: 02 out. 2004.

Object Management Group. **XML Metadata Interchange (XMI).** Disponível em: <http://www.omg.org/>. Acesso em: 02 out. 2004.

Poole, J. D. The Common Warehouse Metamodel as a Foundation for Active Object Models in the Data Warehouse Environment. In: ECOOP 2000 workshop on Metadata and Active Object-Model Pattern, 2000. **Anais ...** Cannes, France: Mining, 2000.

_____. Model-Driven Architecture: Vision, Standards And Emerging Technologies. In: ECOOP´2001 Workshop on Metamodeling and Adaptive Object Models, 2001. **Anais ...** London, April 2001.

Postgresql. **Relational Open Source Database**. Disponível em: <http://www.postgresql.org>. Acesso em: 12 dezembro 2004.

Riehle, D. What is metadata. In: Position paper for OOPSLA '99 Workshop on Metadata and Active Object Models, 1999. **Anais ...** Denver, Colorado, USA, novembro, 1999. Disponível em: <http://www.riehle.org//computer-science/research/1999/oopsla-1999-ws-21-pp.html>. Acesso em: 30 outubro 2004.

Riehle, D.; Tilman, M.; Johnson, R. Dynamic Object Model. In: Conference on Patterns Languages of Programs (PloP'2000), 2000. **Proceedings** ... Washington: Washington University Departament of Computer Science. October of 2000.

Riehle, D.; Fraleigh, S.; Bucka-Lassen, D.; Omorogbe, N. The Architecture of a UML Virtual Machine. In: Conference on Object-Oriented Programming Systems,

Languages, and Applications (OOPSLA '01), 2001. **Proceedings ...** ACM Press, 2001. Pages 327-341.

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen W. **Modelagem e projetos baseados em objetos.** Rio de Janeiro: Campus, 1994. 672p.

Souza, A. D. D. **Structuring adaptive applications using AspectJ**. 2004. Dissertação (mestrado em Ciência da Computação) – Universidade Federal de Pernambuco, Recife. 2004.

Souza, A. D. D.; Yoder, J. W.; Borba, P.; Johnson, R. Using Aspects to Make Adaptive Object-Models Adaptable. Workshop Position Paper; European Conference on Object Oriented Programming (ECOOP '04), 2004. **Anais ...** Oslo, Norway, June 2004.

Tan, J.; Zaslavsky, A.; Ewald, C. A.; Bond, A. A metadata management scheme for middleware - based multidatabase management. In: International Symposium on Distributed Objects and Applications Short Papers, Rome, Italy, 17-20 September 2001. **Proceedings ...** Roma: Universita Degli Studi Di Roma, (ISBN: 8-88665-811-7). p 45-52. (Copyright 2001 IEEE).

_____. Domain-Specific Metamodels for Heterogeneous Information Systems**.** In: Hawaii International Conference on System Sciences (HICSS36), 36.,2003, Big Island, Hawaii, 6-9 Jan. 2003. (Copyright 2003 IEEE).

Thomé, A. C. **Uma Arquitetura de Software Reflexiva Baseada em Modelos de Objetos Adaptáveis**. 2004. Tese (Doutorado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais. São Jose dos Campos.

Witthawaskul W.; Johnson, R. Specifying persistence in platform independent models. In: Workshop in *Software* Model Engineering at The Sixth International Conference on the Unified Modeling Language, UML 2003, San Francisco, California, USA. **Anais ...** Califórnia, October, 20-24, 2003.

Yoder, J.W.; Balaguer F.; Johnson R. Architecture and design of Adaptive Object Models. **ACM Sigplan Notices**. Vol 36, Fasc. 12, pg. 50-60, December 2001.

Yoder, J.W.; Johnson R. The Adaptive Object-Model Architectural Style.In: **IEEE/IFIP Conference on *Software* Architecture,** 2002 (WICSA3'02) Montreal, Quebec, Canada. August of 2002.

_____. **Architecture and Design of Adaptive Object-Models**. São Paulo, 2003. Cursos e Palestras sobre Desenvolvimento de *Software* Orientado a Objetos. IME/USP em 23 e 24 de agosto de 2003.

Yoder, J.W.; Razavi, R. Adaptive Object-Models. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000). **Proceedings ...** ACM Press, 2000.

Xindice. **A Native XML Database**. Disponível em: <http://xml.apache.org/xindice>. Acesso em: 15 outubro 2004.

APÊNDICE A

OS PROTÓTIPOS DESENVOLVIDOS

Este apêndice tem o objetivo de mostrar uma listagem mais detalhada dos três protótipos desenvolvidos para esta dissertação. Esta listagem é mais completa que os fragmentos de código do protótipo, das classes criadas no Caché e tabelas criadas no PostgreSQL e das telas mostradas nos capítulos 4 e 5 deste trabalho.

Primeiramente lista-se as classes criadas no banco Caché e as tabelas criadas no banco PostGreSQL. Em seguida lista-se os metadados dos modelos armazenados no Caché. Por último, lista-se os códigos Java de alguns dos EJBs de sessão e de entidade implementados e alguns arquivos Java dos protótipos. O final deste apêndice apresenta detalhes para o acesso ao PostGreSQL em um ambiente distribuído.

A.1 Esquemas dos Bancos

Os esquemas dos bancos criados para o metamodelo da figura 4.2, no Caché e no PostGreSQL são listados a seguir.

A.1.1 Caché

Classe Estacao:

```
Class User.Estacao {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME = "C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User", ROOTDIR = "C:\SICSDA");
Property Codigo As %Integer;
Property Latitude As %Float;
Property Longitude As %Float;
Property Nome As %String;
Index CodigoIndex On Codigo [ Unique ];
}
```

A linha com a palavra *Projection*, define uma projeção da classe Estacao para um EJB de entidade. Com base nesta linha, o Caché gera automaticamente os arquivos necessários para publicar essa classe como um EJB de entidade em um servidor de aplicação. O atributo APPSERVERHOME = "C:\JBOSS-3.0.8" indica que o EJB será

projetado para o servidor de aplicação JBoss. O atributo JAVAHOME = "C:\j2sdk1.4.2_03" indica onde o Java está instalado no computador onde foi criado a classe Estacao. O atributo ROOTDIR = "C:\SICSDA" indica o diretório onde o Caché colocará os arquivos que compõem o EJB de entidade gerado.

Classe Frame:

```
Class User.Frame {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Codigo As %Integer;
Property Data As %Date;
Property estacao As Estacao;
Property Hora As %Time;
Property NumBytes As %Integer;
Property satelite As Satelite;
Index CodigoIndex On Codigo [ Unique ];
}
```

Classe Satelite:

```
Class User.Satelite {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property CODIGO As %Integer;
Property NOME As %String;
Index CODIGOIndex On CODIGO [ Unique ];
}
```

Classe Subsistema:

```
Class User.SubSistema {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Codigo As %Integer;
Property Nome As %String;
Property satelite As Satelite;
Index CodigoIndex On Codigo [ Unique ];
}
```

Classe TipoPropriedade:

```
Class User.TipoPropriedade {
```

```
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Nome As %String;
Property Tipo As %String;
Property tipomensagem As TipoMensagem;
Property tipotipomensagem As TipoTipoMensagem;
}
```

Classe Propriedade:

```
Class User.Propriedade {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property mensagem As Mensagem;
Property Tipo As TipoPropriedade;
Property Valor As %String;
}
```

Classe TipoTipoMensagem:

```
Class User.TipoTipoMensagem {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Codigo As %Integer;
Property Nome As %String;
Index CodigoIndex On Codigo [ Unique ];
}
```

Classe TipoMensagem:

```
Class User.TipoMensagem {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Codigo As %Integer;
Property Nome As %String;
Index CodigoIndex On Codigo [ Unique ];
Property tipotipomensagem As TipoTipoMensagem;
}
```

Classe Mensagem:

```
Class User.Mensagem {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
```

```
Property Descricao As %String;
Property frame As Frame;
Property Numero As %Integer;
Property subsistema As SubSistema;
Property tipomensagem As TipoMensagem;
Index NumeroIndex On Numero [ Unique ];
}
```

Classe Regra

```
Class User.Regra {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Nome As %String;
}
```

Classe RegraTipoMensagem

```
Class User.RegraTipoMensagem {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property regra As Regra;
Property propriedadesassociadas As TipoPropriedade [ Collection = list ];
Property tipomensagem As TipoMensagem;
}
```

Classe TipoRelacionamento

```
Class User.TipoRelacionamento {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
Property Multiplicidade1 As %String;
Property Multiplicidade2 As %String;
Property Nome As %String;
Property tipomensagem1 As TipoMensagem;
Property tipomensagem2 As TipoMensagem;
Property tipotipomensagem1 As TipoTipoMensagem;
Property tipotipomensagem2 As TipoTipoMensagem;
}
```

Classe Relacionamento

```
Class User.Relacionamento {
Projection ProjEJB As %Projection.EJBJBoss(APPSERVERHOME =
"C:\JBOSS-3.0.8", JAVAHOME = "C:\j2sdk1.4.2_03", PACKAGE = "User",
ROOTDIR = "C:\SICSDA");
```

```
Property mensagem1 As Mensagem;
Property mensagem2 As Mensagem;
Property tiporelacionamento As TipoRelacionamento;
}
```

A.1.2 PostgreSQL

Os comandos DDL (Data Definition Language) SQL que criaram as tabelas utilizadas para persistir o modelo de objetos no PostgreSQL são listadas a seguir.

Tabela Estacao:

```
CREATE TABLE ESTACAO(
CODIGO INT8 NOT NULL,
LATITUDE FLOAT4,
LONGITUDE FLOAT4,
NOME VARCHAR(80),
PRIMARY KEY(CODIGO));
```

Tabela Frame:

```
CREATE TABLE FRAME(
CODIGO INT8 NOT NULL,
DATA DATE,
HORA TIME,
NUMBYTES INT8,
CODESTACAO INT8 NOT NULL,
CODSATELITE INT8 NOT NULL,
PRIMARY KEY(CODIGO));
```

Tabela Satelite:

```
CREATE TABLE SATELITE (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
PRIMARY KEY (CODIGO));
```

Tabela Subsistema:

```
CREATE TABLE SUBSISTEMA (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
CODSATELITE INT8 NOT NULL,
PRIMARY KEY(CODIGO));
```

Tabela TipoPropriedade:

```
CREATE TABLE TIPOPROPRIEDADE (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
TIPO VARCHAR(80),
CODTIPOMENSAGEM INT8,
CODTIPOTIPOMENSAGEM INT8,
PRIMARY KEY(CODIGO));
```

Tabela Propriedade:

```
CREATE TABLE PROPRIEDADE (
CODIGO INT8 NOT NULL,
VALOR VARCHAR(80),
CODTIPO INT8 NOT NULL,
CODMENSAGEM INT8 NOT NULL,
PRIMARY KEY(CODIGO));
```

Tabela TipoTipoMensagem:

```
CREATE TABLE TIPOTIPOMENSAGEM (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
PRIMARY KEY (CODIGO));
```

Tabela TipoMensagem:

```
CREATE TABLE TIPOMENSAGEM (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
CODTIPOTIPOMENSAGEM INT8 NOT NULL,
PRIMARY KEY (CODIGO));
```

Tabela Mensagem:

```
CREATE TABLE MENSAGEM (
CODIGO INT8 NOT NULL,
DESCRICAO VARCHAR(80),
NUMERO INT8,
CODTIPOMENSAGEM INT8 NOT NULL,
CODSUBSISTEMA INT8 NOT NULL,
CODFRAME INT8 NOT NULL,
PRIMARY KEY (CODIGO));
```

Tabela Regra:

```
CREATE TABLE REGRA (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
```

```
PRIMARY KEY (CODIGO));
```

Tabela RegraTipoMensagem:

```
CREATE TABLE REGRATIPOMENSAGEM (
CODIGO INT8 NOT NULL,
CODREGRA INT8 NOT NULL,
CODTIPOMENSAGEM INT8 NOT NULL,
PRIMARY KEY (CODIGO));
```

Tabela RegraTipoMensagemTipoPropriedade:

```
CREATE TABLE REGRATIPOMENSAGEMTIPOPROPRIEDADE (
CODIGO INT8 NOT NULL,
CODREGRATIPOMENSAGEM INT8 NOT NULL,
CODTIPOPROPRIEDADE INT8 NOT NULL,
PRIMARY KEY (CODIGO));
```

Tabela TipoRelacionamento:

```
CREATE TABLE TIPORELACIONAMENTO (
CODIGO INT8 NOT NULL,
NOME VARCHAR(80),
MULTIPLICIDADE1 VARCHAR(10),
MULTIPLICIDADE2 VARCHAR(10),
CODTIPOMENSAGEM1 INT8,
CODTIPOMENSAGEM2 INT8,
CODTIPOTIPOMENSAGEM1 INT8,
CODTIPOTIPOMENSAGEM2 INT8,
PRIMARY KEY (CODIGO));
```

Tabela Relacionamento:

```
CREATE TABLE TIPORELACIONAMENTO (
CODIGO INT8 NOT NULL,
CODTIPORELACIONAMENTO INT8 NOT NULL,
CODMENSAGEM1 INT8,
CODMENSAGEM2 INT8,
PRIMARY KEY (CODIGO));
```

A.2 Exemplos de Metadados no Caché

Classe TipoPropriedade:

ID	Nome	Tipo	tipomensagem	tipotipomensagem
1	Tempo	java.lang.string		
2	CodigoInterno	java.lang.Integer		26
3	Numero	java.lang.Integer		26
4	Descricao	java.lang.String		26
5	Numero	java.lang.Integer		27
6	Descricao	java.lang.String		27
7	Word	java.lang.Integer		27
8	ProcessType	java.lang.Integer		27
9	ValorMax	java.lang.Integer		27
10	ValorMin	java.lang.Integer		27
11	Valor	java.lang.Integer		28
12	Tempo	java.lang.Integer	16	
13	Limite	java.lang.Integer	17	
14	ValorInf	java.lang.Integer	17	
15	ValorSup	java.lang.Integer	17	
16	ValorAlarme	java.lang.Integer	18	
17	Tempototal	java.lang.Integer	21	
18	TempoSolo	java.lang.Integer	22	
19	TipoListagem	java.lang.Integer		26
20	TipoListagem	java.lang.Integer		27

Classe Propriedade:

ID	Tipo	Valor	mensagem
79	12	24	135
80	2	4	135
81	4	8	135
82	3	6	135
83	19	1	135
84	12	24	137
85	2	4	137
86	4	8	137
87	3	6	137
88	19	2	137
89	12	24	139
90	2	4	139
91	4	8	139
92	3	6	139
93	19	3	139
94	12	24	146
95	2	4	146
96	4	8	146
97	3	6	146
98	19	2	146

Classe TipoTipoMensagem:

ID	Codigo	Nome
26	1	Telecomando
27	2	Telemetria
28	3	Ranging

Classe TipoMensagem:

ID	Codigo	Nome	tipotipomensagem
14	1	Sequencial	26
15	2	Manual	26
16	3	Temporizado	26
17	4	Analogica	27
18	5	Digital	27
20	6	ModeEquation	27
21	7	Medidas	28
22	8	Calibracao	28

Classe Mensagem:

ID	Descricao	Numero	frame	subsistema	tipomensagem
32	Telemetria digital	9	15	2	18
33	Telemetria digital II	10	15	2	18
45	Telemetria modeequation	11	15	2	20
46	Telemetria modeequation II	12	15	2	20
48	Telemetria digital III	13	15	2	18
78	Mensagem Sequencial	14	15	2	14
100	Medida Calibração	15	15	2	21

Classe Regra:

ID	NOME
1	ListarRelacionamento
2	VisualizarTelemetria
3	ArmazenarTelemetria
4	ReceberTelemetria
5	EnviarTelecomando
6	ArmazenarTelecomando
7	CalcularMedidas
8	REGRADINAMICA

Classe RegraTipoMensagem:

ID	regra	propriedadesassociadas	tipomensagem
1	1		14
2	1		15
3	2		17
6	2	IIIIIIIIII	16
7	1		18
8	8		14
9	8		15
10	8		16

Classe TipoRelacionamento:

ID	Multiplicidade1	Multiplicidade2	Nome	tipomensagem1	tipomensagem2	tipotipomensagem1	tipotipomensagem2
26	1	*	DigitalModeEquation2	20	18		
27	*	1	DigitalModeEquation	18	20		
28	*	1	Sequencial_Telecomando2		14	26	
29	1	*	Sequencial_Telecomando	14			26

Classe Relacionamento:

ID	mensagem1	mensagem2	tiporelacionamento
13	33	46	27
14	48	45	27
15	46	32	26
16	46	32	26
17	45	33	26
18	7	78	28
19	8	78	28

A.3 Código Java dos Protótipos

No protótipo que persiste os modelos adaptáveis no Caché, todos os EJBs de entidade foram criados pelo próprio Caché através das projeções definidas para cada classe. Como criar uma projeção foi explicado após a listagem da classe Estacao na seção A.1.1. A seguir lista-se dois arquivos que compõem o sessionID, um EJB de sessão utilizado no protótipo que persiste os modelos de objetos no Caché.

A.3.1 EJB de sessão

```
package sessionbeansicsda;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;
import java.awt.List;
import javax.naming.NamingException;
import javax.ejb.EJBException;
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.util.Properties;
public class sessionIDsBean
   implements SessionBean {
  SessionContext sessionContext;
  Connection con = null;
  private Context jndiCntx;
  private DataSource ds;
  public void ejbCreate() throws CreateException {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext) {
   this.sessionContext = sessionContext;
```

```
  }
  protected Connection getConnection() throws SQLException {
    try {
      try {
System.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");

System.setProperty("java.naming.provider.url","jnp://150.163.21.51:1099");

       Properties pro = System.getProperties();

       if (jndiCntx == null) jndiCntx = new InitialContext(pro);
        if (ds == null) {
          String dataSource = "java:comp/env/USERDatabase";
          try {
            ds = (DataSource)jndiCntx.lookup(dataSource);
          }
          catch (Exception e) {
            throw new EJBException(e.getMessage());
          }
        }
        try {
          con = ds.getConnection();
          if (con == null) throw new EJBException("ds.getConnection()
unexpectedly returns NULL: ds="+ds);
         } catch (Exception ex) {
          throw new EJBException("ds.getConnection() falied: ds="+ds+"
:ex="+ex.getMessage());
         }
      } catch (NamingException ne) {
        throw new EJBException(ne);
      }
         return con;
    } catch (Exception ne) {
      java.io.StringWriter sw = new java.io.StringWriter();
      ne.printStackTrace( new java.io.PrintWriter( sw));
      throw new EJBException(sw.toString());
    }
  }

  public List getIDsPropertiesTipoTipoMsg(Integer IDEJB) {
    try {
      try {
        con = this.getConnection();
        java.sql.Statement st = con.createStatement();
        java.sql.ResultSet rs = st.executeQuery("SELECT ID FROM
TIPOPROPRIEDADE WHERE TIPOTIPOMENSAGEM = " + IDEJB );
        List IDs = new List();
        while (rs.next())
         IDs.add(rs.getString(1));
```

```
      return IDs;
     } catch (Exception ne) {
      java.io.StringWriter sw = new java.io.StringWriter();
      ne.printStackTrace( new java.io.PrintWriter( sw));
      throw new EJBException(sw.toString());
     }

   } finally {
    try {
     if (con != null) con.close();
     con = null;
    }catch (SQLException ce) {
     throw new EJBException (ce.getMessage());
    }
   }
  }

  public List getIDsPropriedadesTipoMsg(Integer IDEJB) {
   try {
    try {
     con = this.getConnection();
     java.sql.Statement st = con.createStatement();
     java.sql.ResultSet rs = st.executeQuery("SELECT ID FROM
PROPRIEDADE WHERE TIPOMENSAGEM = " + IDEJB);
     List IDs = new List();
     while (rs.next())
      IDs.add(rs.getString(1));
     return IDs;
    }
    catch (Exception ne) {
     java.io.StringWriter sw = new java.io.StringWriter();
     ne.printStackTrace(new java.io.PrintWriter(sw));
     throw new EJBException(sw.toString());
    }
   }
   finally {
      try {
         if (con != null) con.close();
         con = null;
      }
      catch (SQLException ce) {
         throw new EJBException (ce.getMessage());
      }
   }

  }

  public List getIDTipoPropriedadeTipomsg(Integer idEJB) {
   try {
    try {
```

```
        con = this.getConnection();
        java.sql.Statement st = con.createStatement();
        java.sql.ResultSet rs = st.executeQuery("SELECT ID FROM
TIPOPROPRIEDADE WHERE TIPOMENSAGEM = " + idEJB );
        List IDs = new List();
        while (rs.next())
         IDs.add(rs.getString(1));
        return IDs;
       } catch (Exception ne) {
        java.io.StringWriter sw = new java.io.StringWriter();
        ne.printStackTrace( new java.io.PrintWriter( sw));
        throw new EJBException(sw.toString());
       }

     } finally {
      try {
        if (con != null) con.close();
        con = null;
       }catch (SQLException ce) {
        throw new EJBException (ce.getMessage());
       }
     }
   }

   public List getIDsPropriedadeMsg(Integer idEJB) {
    try {
     try {
       con = this.getConnection();
       java.sql.Statement st = con.createStatement();
       java.sql.ResultSet rs = st.executeQuery("SELECT ID FROM
PROPRIEDADE WHERE MENSAGEM = " + idEJB);
       List IDs = new List();
       while (rs.next())
         IDs.add(rs.getString(1));
       return IDs;
      }
      catch (Exception ne) {
       java.io.StringWriter sw = new java.io.StringWriter();
       ne.printStackTrace(new java.io.PrintWriter(sw));
       throw new EJBException(sw.toString());
      }
    }
    finally {
       try {
           if (con != null) con.close();
           con = null;
       }
       catch (SQLException ce) {
          throw new EJBException (ce.getMessage());
       }
```

```
   }
  }

  public List getIDsTipoRelacionamento(Integer idEJBTipo,
                                       Integer idEJBTipoTipo) {
   try {
    try {
      con = this.getConnection();
      java.sql.Statement st = con.createStatement();
      java.sql.ResultSet rs = st.executeQuery("SELECT ID FROM
TIPORELACIONAMENTO WHERE TIPOMENSAGEM1 = " +
         idEJBTipo + " OR TIPOTIPOMENSAGEM1 ="+ idEJBTipoTipo);
      List IDs = new List();
      while (rs.next())
       IDs.add(rs.getString(1));
      return IDs;
    }
    catch (Exception ne) {
      java.io.StringWriter sw = new java.io.StringWriter();
      ne.printStackTrace(new java.io.PrintWriter(sw));
      throw new EJBException(sw.toString());
    }
   }
   finally {
      try {
         if (con != null) con.close();
         con = null;
      }
      catch (SQLException ce) {
         throw new EJBException (ce.getMessage());
      }
   }

  }

  public List getIDsMensagemTipoRelacionamento(Integer idEJB) {
try {
  try {
    con = this.getConnection();
    java.sql.Statement st = con.createStatement();
    java.sql.ResultSet rs=st.executeQuery("SELECT ID FROM MENSAGEM "+
    " where  TIPOMENSAGEM = (SELECT TIPOMENSAGEM2 FROM
TIPORELACIONAMENTO WHERE ID = "+idEJB+ ")  union "+
     " select ID from mensagem join tipomensagem on mensagem.tipomensagem
= tipomensagem.id where tipomensagem.tipotipomensagem = (SELECT
TIPOTIPOMENSAGEM2 FROM TIPORELACIONAMENTO" +
    " WHERE ID = "+idEJB+")");
    List IDs = new List();
    while (rs.next())
      IDs.add(rs.getString(1));
```

```
    return IDs;
   }
   catch (Exception ne) {
    java.io.StringWriter sw = new java.io.StringWriter();
    ne.printStackTrace(new java.io.PrintWriter(sw));
    throw new EJBException(sw.toString());
   }
  }
  finally {
     try {
         if (con != null) con.close();
         con = null;
     }
     catch (SQLException ce) {
         throw new EJBException (ce.getMessage());
     }
  }

  }

   public List getIDsRegras(Integer idTipoMsg) {
     try {
       try {
         con = this.getConnection();
         java.sql.Statement st = con.createStatement();
         java.sql.ResultSet rs = st.executeQuery( " SELECT DISTINCT
NOMEREGRA FROM REGRA JOIN REGRATIPOMENSAGEM ON
REGRA.ID = REGRATIPOMENSAGEM.REGRA WHERE
REGRATIPOMENSAGEM.TIPOMENSAGEM = "+idTipoMsg);
         List IDs = new List();
         while (rs.next())
           IDs.add(rs.getString(1));
         return IDs;
       }
       catch (Exception ne) {
         java.io.StringWriter sw = new java.io.StringWriter();
         ne.printStackTrace(new java.io.PrintWriter(sw));
         throw new EJBException(sw.toString());
       }
     }
     finally {
         try {
             if (con != null) con.close();
             con = null;
         }
         catch (SQLException ce) {
             throw new EJBException (ce.getMessage());
         }
     }
   }
```

```
  public List getIDsMsgRelacionadas(Integer idEJBMsg, Integer idEJBTipoRel)
{
   try {
    try {
      con = this.getConnection();
      java.sql.Statement st = con.createStatement();
       java.sql.ResultSet rs = st.executeQuery(" SELECT MENSAGEM2 FROM
RELACIONAMENTO WHERE MENSAGEM1 = "+idEJBMsg+ " AND
TIPORELACIONAMENTO = "+idEJBTipoRel);
      List IDs = new List();
      while (rs.next())
        IDs.add(rs.getString(1));
      return IDs;
    }
    catch (Exception ne) {
      java.io.StringWriter sw = new java.io.StringWriter();
      ne.printStackTrace(new java.io.PrintWriter(sw));
      throw new EJBException(sw.toString());
    }
   }
   finally {
      try {
         if (con != null) con.close();
         con = null;
      }
      catch (SQLException ce) {
         throw new EJBException (ce.getMessage());
      }
   }

  }

  public String getValorProp(Integer idEJBMSG, Integer idEJBTipoProp) {

   try {
    try {
      con = this.getConnection();
      java.sql.Statement st = con.createStatement();
      java.sql.ResultSet rs = st.executeQuery(" SELECT VALOR FROM
PROPRIEDADE WHERE MENSAGEM = "+idEJBMSG+ " AND TIPO
="+idEJBTipoProp);
      rs.next();
      return rs.getString(1);
    }
    catch (Exception ne) {
      java.io.StringWriter sw = new java.io.StringWriter();
      ne.printStackTrace(new java.io.PrintWriter(sw));
      throw new EJBException(sw.toString());
    }
```

```
    }
    finally {
       try {
          if (con != null) con.close();
          con = null;
       }
       catch (SQLException ce) {
          throw new EJBException (ce.getMessage());
       }
    }
  }
}
```

A.3.2 Interface do EJB de sessão

```
package sessionbeansicsda;
import javax.ejb.EJBObject;
import java.awt.List;
import java.rmi.RemoteException;
public interface sessionIDs  extends EJBObject {
  public List getIDsPropertiesTipoTipoMsg(Integer IDEJB) throws
RemoteException;
  public List getIDsPropriedadesTipoMsg(Integer IDEJB) throws
RemoteException;
  public List getIDTipoPropriedadeTipomsg(Integer idEJB) throws
RemoteException;
  public List getIDsPropriedadeMsg(Integer idEJB) throws RemoteException;
  public List getIDsTipoRelacionamento(Integer idEJBTipo, Integer
idEJBTipoTipo) throws RemoteException;
  public List getIDsMensagemTipoRelacionamento(Integer idEJB) throws
RemoteException;
  public List getIDsRegras(Integer idTipoMsg) throws RemoteException;
  public List getIDsMsgRelacionadas(Integer idEJBMsg, Integer idEJBTipoRel)
throws RemoteException;
  public String getValorProp(Integer idEJBMSG, Integer idEJBTipoProp)
throws RemoteException;
}
```

A.3.3 EJB de entidade

A seguir lista-se o código do EJB de entidade Estacao, do protótipo que persiste os modelos de objetos adaptáveis no PostGreSQL.

```
package User;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.CreateException;
```

```
import javax.ejb.RemoveException;
public abstract class EstacaoBean
    implements EntityBean {
  EntityContext entityContext;
  public Long ejbCreate(Long codigo) throws CreateException {
    setCodigo(codigo);
    return null;
  }
  public void ejbPostCreate(Long codigo) throws CreateException {
  }
  public void ejbRemove() throws RemoveException {
  }
  public abstract void setCodigo(Long codigo);
  public abstract void setLatitude(Double latitude);
  public abstract void setLongitude(Double longitude);
  public abstract void setNome(String nome);
  public abstract Long getCodigo();
  public abstract Double getLatitude();
  public abstract Double getLongitude();
  public abstract String getNome();
  public void ejbLoad() {
  }
  public void ejbStore() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setEntityContext(EntityContext entityContext) {
    this.entityContext = entityContext;
  }
  public void unsetEntityContext() {
    this.entityContext = null;
  }
}
```

A.3.4 Interface do EJB de entidade

```
package User;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface EstacaoRemote
    extends EJBObject {
  public Long getCodigo() throws RemoteException;

  public void setLatitude(Double latitude) throws RemoteException;
```

```
  public Double getLatitude() throws RemoteException;

  public void setLongitude(Double longitude) throws RemoteException;

  public Double getLongitude() throws RemoteException;

  public void setNome(String nome) throws RemoteException;

  public String getNome() throws RemoteException;
}
```

A.3.5 Conexão com o PostgreSQL

A conexão com o banco PostgreSQL é definida no arquivo postgres-service.xml. Esse arquivo deve ser colocado no diretório de publicação do JBoss. O protótipo foi publicado na partição *default* do JBoss (diretório C:\jboss-3.0.8\server\default\deploy).

A.3.6 EJB de sessão para o XIndice

A seguir lista-se o código de um EJB de sessão utilizado no protótipo que persistia os modelos de objetos adaptáveis em arquivos XML armazenados no NXD XIndice. Este EJB de sessão mantém um vetor com todos os objetos da classe Estacao existentes no sistema.

```
package testedbxml;
import java.util.*;
import javax.ejb.*;
import org.jdom.Element;
import org.jdom.input.*;
import org.w3c.dom.*;
import org.xmldb.api.base.Collection;
import org.xmldb.api.modules.*;
public class SessionEstacaoBean implements javax.ejb.SessionBean {
  SessionContext sessionContext;
  private Estacao estacao = null;
  protected Vector vetor = new Vector();
  protected Collection colecao = null;
  protected String nomeClasse = "estacao";
  protected void instanciaEstacoes(Iterator iter) {
    String cod, nome, lat, longi;
    try {
      while(iter.hasNext()) {
        Element e = (Element)iter.next();
        cod   = e.getChild("codigo").getText().trim();
```

```
      nome  = e.getChild("nome").getText().trim();
      lat   = e.getChild("latitude").getText().trim();
      longi = e.getChild("longitude").getText().trim();
      vetor.add(new Estacao(new Integer(cod), nome, new Double(lat), new
Double(longi)));
     }
    } catch (Exception ex) {
     System.out.println(ex.getMessage());
     ex.printStackTrace();
    }
  }

  public void ejbCreate() throws CreateException {
   try {
    colecao = Global.getColecao();
    XMLResource document = (XMLResource)
colecao.getResource(nomeClasse);
    Node noderoot = document.getContentAsDOM();
    DOMBuilder builder = new DOMBuilder();
    org.jdom.Document jdomDocument =
builder.build(noderoot.getOwnerDocument());
    Element root = jdomDocument.getRootElement();
    List list = root.getChildren(nomeClasse);
    Iterator iter = list.iterator();
    instanciaEstacoes (iter);
    colecao.close();
   }catch (Exception ex) {
    System.out.println(ex.getMessage());
   }
  }

  public void ejbRemove() {
  }

  protected String getXMLUpdated() {
    String res = new String("<?xml version=\"1.0\" encoding=\"UTF-8\"?>
\n<estacoes>");
     for (int i = 0; i < vetor.size(); i++) {
      estacao  = (Estacao)vetor.get(i);
      res += "\n  <estacao>\n    <codigo> "+estacao.getCodigo()+"</codigo>"+
                  "\n    <nome> "+ estacao.getNome() + "</nome>"+
                  "\n    <latitude> "+estacao.getLatitude() +"</latitude>"+
                  "\n    <longitude> "+estacao.getLongitude() +"</longitude>"+
                  "\n  </estacao>";
     }
     res += "\n</estacoes>";
   return res;
  }
```

```
  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void setSessionContext(SessionContext sessionContext) {
   this.sessionContext = sessionContext;
  }

  public void insere(Integer cod, String nome, Double lat, Double lg) {
   estacao = new Estacao(cod, nome, lat, lg);
   vetor.add(estacao);
  }

  public String listarEstacoes() {
   String res = "Estacoes --> " ;
   for (int i = 0; i < vetor.size(); i++) {
    estacao  = (Estacao)vetor.get(i);
    res += estacao.getNome();
   }
   return res;
  }

  public void removeEstacao(Integer cod) {
   for (int i = 0; i < vetor.size();i++) {
    estacao = (Estacao) vetor.get(i);
    if (cod.equals(estacao.getCodigo())) {
     vetor.remove(i);
     break;
    }
   }
  }

  public Boolean localizarEstacao(Integer cod) {
   estacao = null;
   for (int i = 0; i < vetor.size();i++) {
    estacao = (Estacao) vetor.get(i);
    if (cod.equals(estacao.getCodigo())) {
     return new Boolean(true);
    }
   }
   return new Boolean(false);
  }

  public void atualizaBanco() {
   try {
    colecao = Global.getColecao();
    XMLResource document = (XMLResource)
colecao.getResource(nomeClasse);
```

```
      document.setContent(getXMLUpdated());
      colecao.storeResource(document);
      colecao.close();
     } catch (Exception ex) {
      System.out.println (ex.getMessage());
     }
    }

    public Estacao getEstacao() {
     return estacao;
    }
   }
```

A.3.7 Interface do EJB de sessão

Os métodos disponíveis neste EJB de sessão são listados a seguir.

```
package testedbxml;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface SessionEstacao extends EJBObject {
  public void insere(Integer cod, String nome, Double lat, Double lg) throws
     RemoteException;

  public String listarEstacoes() throws RemoteException;

  public void removeEstacao(Integer cod) throws RemoteException;

  public Boolean localizarEstacao(Integer cod) throws RemoteException;

  public void atualizaBanco() throws RemoteException;

  public Estacao getEstacao() throws RemoteException;
}
```

A.3.8 Arquivos dos protótipos

A seguir lista-se um arquivo fonte de cada um dos três protótipos implementados. O primeiro arquivo se chama frmTipoTipoMensagem.java e implementa a classe ilustrada na figura 6.4 no protótipo que persiste os modelos adaptáveis no NXD XIndice.

```
package User;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.naming.Context;
import EJBsxml.*;

public class FrmTipoTipoMensagem extends JFrame {
  JTextField edNome = new JTextField();
  JLabel jLabel1 = new JLabel();
  JButton btnIncluir = new JButton();
  JButton btnSair = new JButton();
  JLabel jLabel2 = new JLabel();
  JTextField edCodigo = new JTextField();
  JButton btnLocalizar = new JButton();
  JButton btnExcluir1 = new JButton();
  JButton btnProp = new JButton();
  JButton jButton1 = new JButton();

  private SessionTipoTipoMensagem tipotipomsg = null;
  private SessionTipoTipoMensagemHome tipotipomsg_home = null;

  public FrmTipoTipoMensagem() {
    try {
      jbInit();
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void jbInit() throws Exception {
    edNome.setText("");
    edNome.setBounds(new Rectangle(199, 80, 143, 21));
    this.getContentPane().setLayout(null);
    jLabel1.setText("Nome do Tipo do Tipo");
    jLabel1.setBounds(new Rectangle(35, 79, 166, 15));
    btnIncluir.setBounds(new Rectangle(147, 172, 93, 25));
    btnIncluir.setText("Incluir");
    btnIncluir.addMouseListener(new
FrmTipoTipoMensagem_btnIncluir_mouseAdapter(this));
    btnSair.setBounds(new Rectangle(147, 206, 93, 25));
    btnSair.setText("sair");
    btnSair.addMouseListener(new
FrmTipoTipoMensagem_btnSair_mouseAdapter(this));
    jLabel2.setText("Código");
    jLabel2.setBounds(new Rectangle(38, 45, 61, 15));
    edCodigo.setText("");
    edCodigo.setBounds(new Rectangle(199, 34, 57, 21));
    btnLocalizar.setBounds(new Rectangle(36, 172, 93, 25));
    btnLocalizar.setText("localizar");
    btnLocalizar.addMouseListener(new
FrmTipoTipoMensagem_btnLocalizar_mouseAdapter(this));
    btnExcluir1.setText("Excluir");
```

```
    btnExcluir1.addMouseListener(new
FrmTipoTipoMensagem_btnExcluir1_mouseAdapter(this));
    btnExcluir1.setBounds(new Rectangle(267, 172, 93, 25));
    btnProp.setBounds(new Rectangle(267, 206, 93, 25));
    btnProp.setText("Propriedades");
    btnProp.addMouseListener(new
FrmTipoTipoMensagem_btnProp_mouseAdapter(this));
    this.setTitle("Tela de Tipos de Tipos de mensagens");
    jButton1.setBounds(new Rectangle(36, 206, 93, 25));
    jButton1.setText("Relacionamento");
    jButton1.addMouseListener(new
FrmTipoTipoMensagem_jButton1_mouseAdapter(this));
    this.getContentPane().add(jLabel1, null);
    this.getContentPane().add(edNome, null);
    this.getContentPane().add(jLabel2, null);
    this.getContentPane().add(edCodigo, null);
    this.getContentPane().add(btnExcluir1, null);
    this.getContentPane().add(btnSair, null);
    this.getContentPane().add(btnIncluir, null);
    this.getContentPane().add(btnLocalizar, null);
    this.getContentPane().add(btnProp, null);
    this.getContentPane().add(jButton1, null);

    try {
      Context jndiContext = Globais.getInitialContext();
      Object ref = jndiContext.lookup(new
String("SessionTipoTipoMensagem"));
      tipotipomsg_home = (SessionTipoTipoMensagemHome)
javax.rmi.PortableRemoteObject.narrow(
          ref, SessionTipoTipoMensagemHome.class);
      tipotipomsg = tipotipomsg_home.create();
    }
    catch (Exception re) {
      String msgin = re.getMessage();
      System.out.println("RemoteException re " + msgin);
    }
  }

  void btnSair_mousePressed(MouseEvent e) {
    this.dispose();
  }

  void btnIncluir_mousePressed(MouseEvent e) {
    try {
      tipotipomsg.insere(new Integer(edCodigo.getText()), edNome.getText());
    } catch (Exception ex) { System.out.println(ex.getMessage());
    }
  }

  void btnLocalizar_mousePressed(MouseEvent e) {
```

```
    try {
      if (tipotipomsg.localizarTipoTipoMsg(new
Integer(edCodigo.getText())).booleanValue()) {
        edNome.setText(tipotipomsg.getObjTipoTipoMsg().getNome());
      } else edNome.setText("");
    } catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
  }

  void btnExcluir1_mousePressed(MouseEvent e) {
    try {
      tipotipomsg.removeTipoTipoMsg(new Integer(edCodigo.getText()));
    } catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
  }

  void btnProp_mousePressed(MouseEvent e) {
    EdicaoMetadado edicaometadado = new EdicaoMetadado(new
Integer(edCodigo.getText()), edNome.getText(), 0);
    edicaometadado.setSize(600, 400);
    edicaometadado.show();
  }

  void jButton1_mousePressed(MouseEvent e) {
    try {
      FrmTipoRelacionamento frmTipoRelacionamento = new
FrmTipoRelacionamento(tipotipomsg.get_ID(), 0);
      frmTipoRelacionamento.setSize(600, 400);
      frmTipoRelacionamento.show();
  } catch (Exception ex) {
    System.out.println(ex.getMessage());
  }
  }
}

class FrmTipoTipoMensagem_btnSair_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoTipoMensagem adaptee;

  FrmTipoTipoMensagem_btnSair_mouseAdapter(FrmTipoTipoMensagem
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnSair_mousePressed(e);
  }
}
```

```
class FrmTipoTipoMensagem_btnIncluir_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoTipoMensagem adaptee;

  FrmTipoTipoMensagem_btnIncluir_mouseAdapter(FrmTipoTipoMensagem
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnIncluir_mousePressed(e);
  }
}

class FrmTipoTipoMensagem_btnLocalizar_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoTipoMensagem adaptee;


FrmTipoTipoMensagem_btnLocalizar_mouseAdapter(FrmTipoTipoMensagem
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnLocalizar_mousePressed(e);
  }
}

class FrmTipoTipoMensagem_btnExcluir1_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoTipoMensagem adaptee;

  FrmTipoTipoMensagem_btnExcluir1_mouseAdapter(FrmTipoTipoMensagem
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnExcluir1_mousePressed(e);
  }
}

class FrmTipoTipoMensagem_btnProp_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoTipoMensagem adaptee;

  FrmTipoTipoMensagem_btnProp_mouseAdapter(FrmTipoTipoMensagem
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnProp_mousePressed(e);
```

```
  }
}

class FrmTipoTipoMensagem_jButton1_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoTipoMensagem adaptee;

  FrmTipoTipoMensagem_jButton1_mouseAdapter(FrmTipoTipoMensagem
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.jButton1_mousePressed(e);
  }
}
```

O segundo arquivo manipula os satélites no protótipo que persiste os modelos de objetos adaptáveis no PostgreSQL.

```
package User;
import javax.swing.*;
import java.awt.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.awt.event.*;

public class frmSatelite extends JFrame {
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JTextField edCodigo = new JTextField();
  JTextField edNome = new JTextField();
  JButton btnLocalizar = new JButton();
  JButton btnInserir = new JButton();
  JButton btnExcluir = new JButton();
  JButton btnSair = new JButton();
  private SateliteRemoteHome satelite_home = null;
  private SateliteRemote satelite = null;

  public frmSatelite() {
    try {
      jbInit();
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void jbInit() throws Exception {
    jLabel1.setText("Código");
```

```
    jLabel1.setBounds(new Rectangle(35, 46, 52, 15));
    this.setLocale(java.util.Locale.getDefault());
    this.setTitle("Frame dos satélites");
    this.getContentPane().setLayout(null);
    jLabel2.setText("Nome");
    jLabel2.setBounds(new Rectangle(33, 82, 51, 15));
    edCodigo.setText("");
    edCodigo.setBounds(new Rectangle(104, 44, 65, 21));
    edNome.setText("");
    edNome.setBounds(new Rectangle(102, 83, 194, 21));
    btnLocalizar.setBounds(new Rectangle(21, 181, 93, 25));
    btnLocalizar.setText("localizar");
    btnLocalizar.addMouseListener(new
frmSatelite_btnLocalizar_mouseAdapter(this));
    btnInserir.setBounds(new Rectangle(134, 181, 89, 25));
    btnInserir.setText("Inserir");
    btnInserir.addMouseListener(new
frmSatelite_btnInserir_mouseAdapter(this));
    btnExcluir.setBounds(new Rectangle(242, 181, 98, 25));
    btnExcluir.setText("Excluir");
    btnSair.setBounds(new Rectangle(131, 239, 73, 25));
    btnSair.setText("Sair");
    btnSair.addMouseListener(new frmSatelite_btnSair_mo useAdapter(this));
    this.getContentPane().add(jLabel1, null);
    this.getContentPane().add(jLabel2, null);
    this.getContentPane().add(edCodigo, null);
    this.getContentPane().add(edNome, null);
    this.getContentPane().add(btnSair, null);
    this.getContentPane().add(btnLocalizar, null);
    this.getContentPane().add(btnInserir, null);
    this.getContentPane().add(btnExcluir, null);

    try {
      Context jndiContext = Globais.getInitialContext();
      Object ref = jndiContext.lookup(new String(
"User_Satelite_User_EJBSatelite"));
      satelite_home = (SateliteRemoteHome)
javax.rmi.PortableRemoteObject.narrow(ref, SateliteRemoteHome.class);
    }    catch (Exception re) {
      String msgin = re.getMessage();
      System.out.println("RemoteException re " + msgin);
    }
    btnExcluir.addMouseListener(new
frmSatelite_btnExcluir_mouseAdapter(this));
  }

  void btnInserir_mousePressed(MouseEvent e) {
    try {
      satelite = satelite_home.create(new Long(edCodigo.getText()));
      satelite.setNome(edNome.getText());
```

```
    } catch (Exception ex) { System.out.println(ex.getMessage());
    }

  }

  void btnExcluir_mousePressed(MouseEvent e) {
    try {
      if (!edCodigo.getText().equals("")) {
        satelite = satelite_home.findByPrimaryKey(new
Long(edCodigo.getText()));
        satelite.remove();
      }
  } catch (Exception ex) {
    System.out.println(ex.getMessage());
  }
  }

  void btnLocalizar_mousePressed(MouseEvent e) {
    try {
      if (!edCodigo.getText().equals("")) {
        satelite = satelite_home.findByPrimaryKey(new
Long(edCodigo.getText()));
        edCodigo.setText(satelite.getCodigo().toString());
        edNome.setText(satelite.getNome());
      }
    } catch (Exception ex) {
      System.out.println(ex.getMessage());
    }

  }

  void btnSair_mousePressed(MouseEvent e) {
    this.dispose();
  }
}

class frmSatelite_btnInserir_mouseAdapter extends
java.awt.event.MouseAdapter {
  frmSatelite adaptee;

  frmSatelite_btnInserir_mouseAdapter(frmSatelite adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnInserir_mousePressed(e);
  }
}

class frmSatelite_btnExcluir_mouseAdapter extends
java.awt.event.MouseAdapter {
```

```
  frmSatelite adaptee;

  frmSatelite_btnExcluir_mouseAdapter(frmSatelite adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnExcluir_mousePressed(e);
  }
}

class frmSatelite_btnLocalizar_mouseAdapter extends
java.awt.event.MouseAdapter {
  frmSatelite adaptee;

  frmSatelite_btnLocalizar_mouseAdapter(frmSatelite adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnLocalizar_mousePressed(e);
  }
}

class frmSatelite_btnSair_mouseAdapter extends java.awt.event.MouseAdapter
{
  frmSatelite adaptee;

  frmSatelite_btnSair_mouseAdapter(frmSatelite adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnSair_mousePressed(e);
  }
}
```

O terceiro e último arquivo mostrado neste apêndice, é o arquivo que manipula a criação dos tipos de associações. Este arquivo é do protótipo que persiste os modelos de objetos adaptáveis no NXD XIndice. A figura 6.10 ilustra a tela deste arquivo.

```
package User;
import javax.swing.*;
import java.awt.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.awt.event.*;
public class FrmTipoRelacionamento extends JFrame {
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JLabel jLabel3 = new JLabel();
```

```
JTextField edNome = new JTextField();
JTextField edMultiplicidade1 = new JTextField();
JTextField edOClasse = new JTextField();
private EJBTipoRelacionamento_Home tiporel_home = null;
private EJBTipoRelacionamento tiporel = null;
private EJBTipoTipoMensagem_Home tipotipomsg_home = null;
private EJBTipoTipoMensagem tipotipomsg = null;
private Integer idEJB;
private EJBTipoMensagem_Home tipomsg_home = null;
private EJBTipoMensagem tipomsg = null;
private EJBMensagem_Home msg_home = null;
private EJBMensagem msg = null;
JButton btnExcluir = new JButton();
JButton btnSair = new JButton();
JButton btnInserir = new JButton();
int tiporelacionamento = 0;
ButtonGroup buttonGroup1 = new ButtonGroup();
JLabel jLabel4 = new JLabel();
JTextField edMultiplicidade2 = new JTextField();
JRadioButton rbtnTipo = new JRadioButton();
JRadioButton rbtnTipoTipo = new JRadioButton();
ButtonGroup buttonGroup2 = new ButtonGroup();
public FrmTipoRelacionamento(Integer id, int type) {
 try {
  idEJB = id;
  tiporelacionamento = type;
  jbInit();
 }
 catch(Exception ex) {
  ex.printStackTrace();
 }
}

void jbInit() throws Exception {
 jLabel1.setText("Nome relacionamento");
 jLabel1.setBounds(new Rectangle(9, 40, 132, 15));
 this.getContentPane().setLayout(null);
 jLabel2.setText("Multiplicidade 1");
 jLabel2.setBounds(new Rectangle(9, 66, 132, 15));
 jLabel3.setText("Outra Classe");
 jLabel3.setBounds(new Rectangle(9, 140, 132, 15));
 edNome.setText("");
 edNome.setBounds(new Rectangle(142, 37, 111, 21));
 edMultiplicidade1.setText("");
 edMultiplicidade1.setBounds(new Rectangle(142, 81, 111, 21));
 edOClasse.setText("");
 edOClasse.setBounds(new Rectangle(142, 135, 111, 21));
 edNome.setText("");
 edNome.setBounds(new Rectangle(147, 33, 170, 21));
 edMultiplicidade1.setText("");
```

```
    edMultiplicidade1.setBounds(new Rectangle(147, 66, 86, 21));
    edOClasse.setText("");
    edOClasse.setBounds(new Rectangle(147, 138, 60, 21));
    btnExcluir.setBounds(new Rectangle(149, 196, 89, 25));
    btnExcluir.setText("Excluir");
    btnExcluir.addMouseListener(new
FrmTipoRelacionamento_btnExcluir_mouseAdapter(this));
    btnSair.setBounds(new Rectangle(268, 196, 89, 25));
    btnSair.setText("Sair");
    btnSair.addMouseListener(new
FrmTipoRelacionamento_btnSair_mouseAdapter(this));
    btnInserir.setBounds(new Rectangle(28, 196, 89, 25));
    btnInserir.setText("Inserir");
    btnInserir.addMouseListener(new
FrmTipoRelacionamento_btnInserir_mouseAdapter(this));
    jLabel4.setText("Multiplicidade 2");
    jLabel4.setBounds(new Rectangle(9, 100, 132, 15));
    edMultiplicidade2.setText("");
    edMultiplicidade2.setBounds(new Rectangle(147, 100, 87, 21));
    rbtnTipo.setSelected(true);
    rbtnTipo.setText("Tipo Mensagem");
    rbtnTipo.setBounds(new Rectangle(220, 124, 166, 23));
    rbtnTipoTipo.setText("Tipo de tipo de mensagem");
    rbtnTipoTipo.setBounds(new Rectangle(220, 154, 187, 23));
    this.setTitle("Edição dos metadados");
    this.getContentPane().add(edNome, null);
    this.getContentPane().add(edMultiplicidade2, null);
    this.getContentPane().add(edMultiplicidade1, null);
    this.getContentPane().add(edOClasse, null);
    this.getContentPane().add(rbtnTipo, null);
    this.getContentPane().add(rbtnTipoTipo, null);
    this.getContentPane().add(jLabel3, null);
    this.getContentPane().add(jLabel1, null);
    this.getContentPane().add(jLabel2, null);
    this.getContentPane().add(jLabel4, null);
    this.getContentPane().add(btnExcluir, null);
    this.getContentPane().add(btnInserir, null);
    this.getContentPane().add(btnSair, null);
    this.getContentPane().add(edNome, null);
    this.getContentPane().add(edOClasse, null);
    this.getContentPane().add(edMultiplicidade1, null);

    try {
      Context jndiContext = new InitialContext();
      Object ref = jndiContext.lookup(new
String("User_TipoRelacionamento_User_EJBTipoRelacionamento"));
      tiporel_home = (EJBTipoRelacionamento_Home)
javax.rmi.PortableRemoteObject.narrow(
          ref, EJBTipoRelacionamento_Home.class);
```

```
    ref = jndiContext.lookup(new
String("User_TipoTipoMensagem_User_EJBTipoTipoMensagem"));
    tipotipomsg_home = (EJBTipoTipoMensagem_Home)
javax.rmi.PortableRemoteObject.narrow(
        ref, EJBTipoTipoMensagem_Home.class);

    ref = jndiContext.lookup(new
String("User_TipoMensagem_User_EJBTipoMensagem"));
         tipomsg_home = (EJBTipoMensagem_Home)
javax.rmi.PortableRemoteObject.narrow(
             ref, EJBTipoMensagem_Home.class);

              ref = jndiContext.lookup(new
String("User_Mensagem_User_EJBMensagem"));
              msg_home = (EJBMensagem_Home)
javax.rmi.PortableRemoteObject.narrow(
                  ref, EJBMensagem_Home.class);

    if (tiporelacionamento == 0)
     tipotipomsg = tipotipomsg_home.findById(idEJB);
   else
    tipomsg = tipomsg_home.findById(idEJB);
   }
   catch (Exception re) {
    String msgin = re.getMessage();
    System.out.println("RemoteException re " + msgin);
   }
   buttonGroup2.add(rbtnTipo);
   buttonGroup2.add(rbtnTipoTipo);
  }

  void btnInserir_mousePressed(MouseEvent e) {
   try {
    EJBTipoRelacionamento tiporelinverso = tiporel_home.create();
    tiporel = tiporel_home.create();
    tiporel.set_Multiplicidade1(edMultiplicidade1.getText());
    tiporelinverso.set_Multiplicidade1(edMultiplicidade2.getText());
    tiporelinverso.set_Multiplicidade2(edMultiplicidade1.getText());
    tiporel.set_Multiplicidade2(edMultiplicidade2.getText());
    tiporel.set_Nome(edNome.getText());
    tiporelinverso.set_Nome(edNome.getText()+"2");

    if (tiporelacionamento == 0) {
     tiporel.set_tipotipomensagem1(tipotipomsg);
     tiporelinverso.set_tipotipomensagem2(tipotipomsg);
    }
    else  {
     tiporel.set_tipomensagem1(tipomsg);
     tiporelinverso.set_tipomensagem2(tipomsg);
    }
```

```
    if (rbtnTipo.isSelected()) {
      EJBTipoMensagem tipomsgtemp =
tipomsg_home.findByCodigoIndex(new Integer(edOClasse.getText()));
      tiporel.set_tipomensagem2(tipomsgtemp);
      tiporelinverso.set_tipomensagem1(tipomsgtemp);
    }
    else {
      EJBTipoTipoMensagem tipotipomsgtemp =
tipotipomsg_home.findByCodigoIndex(new Integer(edOClasse.getText()));
      tiporel.set_tipotipomensagem2(tipotipomsgtemp);
      tiporelinverso.set_tipotipomensagem1(tipotipomsgtemp);
    }

   }catch (Exception ex) {
     System.out.println("btnInserir_mousePressed re " + ex.getMessage() );
   }

  }

  void btnExcluir_mousePressed(MouseEvent e) {
   try {
     if (tiporelacionamento == 0) {
       System.out.println(tiporel.get_tipotipomensagem1().get_Nome());
       }
     else {
       System.out.println(tiporel.get_tipomensagem1().get_Nome());
       }
   } catch (Exception ex) {
     System.out.println(ex.getMessage());
   }

  }

  void btnSair_mousePressed(MouseEvent e) {
   this.dispose();
  }
}

class FrmTipoRelacionamento_btnInserir_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoRelacionamento adaptee;

  FrmTipoRelacionamento_btnInserir_mouseAdapter(FrmTipoRelacionamento
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnInserir_mousePressed(e);
  }
```

```
}

class FrmTipoRelacionamento_btnExcluir_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoRelacionamento adaptee;

  FrmTipoRelacionamento_btnExcluir_mouseAdapter(FrmTipoRelacionamento
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnExcluir_mousePressed(e);
  }
}

class FrmTipoRelacionamento_btnSair_mouseAdapter extends
java.awt.event.MouseAdapter {
  FrmTipoRelacionamento adaptee;

  FrmTipoRelacionamento_btnSair_mouseAdapter(FrmTipoRelacionamento
adaptee) {
    this.adaptee = adaptee;
  }
  public void mousePressed(MouseEvent e) {
    adaptee.btnSair_mousePressed(e);
  }
}
```

A.4 Distibuição no PostgreSQL

Para você acessar um banco criado no PostgreSQL em um ambiente distribuído, você deve configurar o arquivo pg_hba.conf criado pelo PostgreSQL no diretório onde o banco foi criado. Este arquivo controla quais hosts são permitidos se conectar ao banco, além de quais usuários podem se conectar ao banco. Uma explicação mais detalhadas dos campos pode ser encontrada no próprio arquivo pg_hba.conf criado pelo PostgreSQL. A seguir lista-se este arquivo utilizado no protótipo.

# type	database	user	ip-address	ip-mask	method
local	all	all			trust
host	all	all	127.0.0.1	255.255.255.255	trust
host	all	all	150.163.21.51	255.255.255.255	trust
host	all	all	150.163.21.56	255.255.255.255	trust
host	all	all	150.163.21.52	255.255.255.255	trust
host	all	all	150.163.21.50	255.255.255.255	trust

Este arquivo foi criado de acordo com a rede do laboratório onde os protótipos foram implementados e testados. As máquinas que compõem a rede são mostradas na tabela A.1.

Tabela A.1 – As máquinas onde os protótipos foram implementados

Nome da máquina	IP
Lefkas	150.163.21.51
Anafi	150.163.21.52
Zitise	150.163.21.50
Skopelos	150.163.21.56

Os EJBs de sessão e de entidade desenvolvidos para o protótipo que utilizava o PostregreSQL foram distribuídos entre essas quatro máquinas. O arquivo Java que informava as máquinas onde o protótipo devia localizar os EJBs é listado abaixo.

```
public class Globais {
  public Globais() {
  }

  public static Context getInitialContextLefkas() throws NamingException {
    Hashtable environment = new Hashtable();
    environment.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
    environment.put(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
    environment.put(Context.PROVIDER_URL, "jnp://150.163.21.51:1099");
    return new InitialContext(environment);
  }

  public static Context getInitialContextAnafi() throws NamingException {
    Hashtable environment = new Hashtable();
    environment.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
    environment.put(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
    environment.put(Context.PROVIDER_URL, "jnp://150.163.21.52:1099");
    return new InitialContext(environment);
  }

  public static Context getInitialContextZitise() throws NamingException {
    Hashtable environment = new Hashtable();
    environment.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
    environment.put(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
    environment.put(Context.PROVIDER_URL, "jnp://150.163.21.50:1099");
    return new InitialContext(environment);
```

```
  }

  public static Context getInitialContextSkopelos() throws NamingException {
    Hashtable environment = new Hashtable();
    environment.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
    environment.put(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
    environment.put(Context.PROVIDER_URL, "jnp://150.163.21.56:1099");
    return new InitialContext(environment);
  }

}
```