MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-13044-TDI/1020

UMA NOVA ARQUITETURA PARA A REPRESENTAÇÃO DAS REGRAS DE NEGÓCIO EM MODELOS DE OBJETOS DINÂMICOS

Paulo Eduardo Cardoso

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Maurício Gonçalves Vieira Ferreira, aprovada em 01 de março de 2005.

INPE
São José dos Campos
2005

681.3.06

CARDOSO, P. E.
Uma nova arquitetura para a representação das regras de negócio em modelos de objetos dinâmicos / P. E. Cardoso. – São José dos Campos: INPE, 2005.
133p. – (INPE-13044-TDI/1020).

1.Sistema adaptativo. 2.Reuso de software. 3.Meta computação. 4.Programação orientada a objetos. 5.Sistemas computacionais. I.Título.

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

Dr. Sólon Venâncio de Carvalho

Presidente / INPE / São José dos Campos - SP

Dr. Mauricio Gonçalves Vieira Ferreira

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

Dr. Nilson Sant'Anna

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

Dr. Henrique Jesus Quintino de Oliveira

Convidado(a) / UMC / Mogi das Cruzes - SP

Aluno (a): Paulo Eduardo Cardoso

São José dos Campos, 01 de março de 2005

“Vá até onde puder ver; quando lá chegar poderá ver ainda mais longe”.
GOETHE

A minha esposa,
LUCIANA SÊDA CARDOSO.

AGRADECIMENTOS

Agradeço a todas pessoas que me incentivaram e ajudaram a realizar este trabalho.

À Divisão de Desenvolvimento de Sistemas de Solo – DSS , e em especial a Edenilse F. E. Orlandi, pela oportunidade de estudos e utilização de suas instalações.

Ao meu orientador, Prof. Dr. Mauricio G. V. Ferreira, pela orientação e amizade.

À minha esposa, Luciana Sêda Cardoso, pelo apoio e paciência na revisão e organização deste trabalho.

A meus pais por sempre acreditarem na importância do estudo.

RESUMO

A Divisão de Desenvolvimento de Sistemas de Solo (DSS) sempre desenvolveu os sistemas de controle para os satélites do INPE com arquiteturas que permitissem que os mesmos fossem re-usados por futuros satélites com um conjunto reduzido de mudanças. O desafio que se apresenta é construir um sistema de controle adaptativo, usando a tecnologia de Objetos Dinâmicos, de modo que tais sistemas possam atender futuros requisitos sem necessidade de mudanças no código. De acordo com esta tecnologia as estruturas dos objetos e os seus comportamentos são armazenados em banco de dados de modo que os usuários finais possam modificá-los usando ferramentas de configuração e possivelmente uma linguagem especifica ao domínio do problema. Este trabalho propõe um sistema de Objetos Dinâmicos segundo uma arquitetura que facilite sua utilização pelos sistemas de controle de satélites. Este trabalho também apresenta uma nova abordagem para representação dos Objetos Dinâmicos, baseada em uma árvore de entidades e propriedades. Como resultados são apresentados um protótipo da ferramenta de configuração e a utilização da arquitetura proposta em um dos subsistemas do sistema de controle de satélites. Em função destes resultados chegou-se à conclusão que a abordagem de representação e arquitetura propostas são viáveis mas, estudos adicionais devem ser realizados para avaliar alguns pontos desfavoráveis que foram identificados. Outro resultado importante é que a arquitetura proposta é um *framework* que pode ser facilmente adaptado para outros domínios de problema.

AN NEW ARCHITECTURE FOR REPRESENTING BUSINESS RULES IN DYNAMIC OBJECT MODELS

ABSTRACT

The Ground System Development Division (DSS) at INPE has always developed satellite control systems with architectures that may be used as much as possible by future satellites. The present challenge is to build an Adaptive Satellite Control System, using the technology of the Dynamic Object Model, so as to comply with later requirements without having to make significant changes in the code. According to this technology, object structures and their behavior are mapped onto a database, so end-users can modify them by using configuration tools and, possibly, a domain specific language. This paper proposes a Dynamic Object System, using an architecture that facilitates its long term use by the satellite control systems. The paper also presents a new approach for representing Dynamic Objects, based on a tree of entities and properties. Results are presented from a configuration tool prototype and the use of the proposed architecture in one subsystem of the satellite control system. Based on these results, the proposed architecture and the representation approach were proved to be viable although some unfavorable results were identified. Further studies need to evaluate these problems. Another important result is that the proposed architecture is a framework that can easily be adapted to other problem domains.

SUMÁRIO

Pág.

LISTA DE FIGURAS

LISTA DE SIGLAS E ABREVIATURAS

AOM		*Adaptive Object Model*
API		*Application Programming Interface*
CBERS	-	*China Brazil Earth Resource Satellites*
CRC		Centro de Rastreio e Controle
DLL		*Dynamic Link Library*
DOM		*Dynamic Object Model*
DSS	-	Divisão de Desenvolvimento de Sistemas de Solo
EQUARS	-	*Equatorial Atmosphere Research Satellite*
INPE	-	Instituto Nacional de Pesquisas Espaciais
MECB	-	Missão Espacial Completa Brasileira
MIRAX	-	Monitor e Imageador de Raios-X
MMP	-	*Multi-Mission Platform*
MON	-	Subsistema de Processamento de Telemetria
SCD	-	Satélite de Coleta de Dados
SICS	-	Sistema de Controle de Satélites
SICSD	-	Sistema de Controle de Satélites Distribuído
SSR	-	Satélites de Sensoriamento Remoto
TMTC		Sistema de Telemetria e Telecomando
UML		*Unified Modeling Language*

CAPÍTULO 1

INTRODUÇÃO

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. Dada a extensão do território brasileiro, os satélites são uma importante ferramenta para várias aplicações de grande interesse para a população do país, como por exemplo: monitoração da vegetação, plantações, rios e clima; levantamento de recursos naturais; e comunicação.

No momento o INPE está operando três satélites: SCD1, SCD2 e CBERS-2. Os Satélites de Coleta de Dados (SCD1 e SCD2) fazem parte da Missão Espacial Completa Brasileira (MECB) e têm como objetivo receber os dados transmitidos por estações de coleta de dados, espalhadas pelo território brasileiro, processá-los e deixá-los disponíveis para a comunidade. O satélite SCD1 está em operação desde 1993 e o SCD2 desde 1998. O satélite CBERS-2 é o segundo satélite de sensoriamento remoto desenvolvido em conjunto com a China dentro do programa *China Brazil Earth Resource Satellites*. O CBERS-2 está em operação desde 2003. O primeiro satélite CBERS foi lançado em 1999 e foi controlado pelo INPE, em conjunto com a China, até o final da sua vida útil em 2003.

No momento estão em andamento: o programa de micro-satélites científicos, o qual prevê o lançamento dos satélites *Equatorial Atmosphere Research Satellite* (EQUARS) e Monitor e Imageador de Raios-X (MIRAX); e o programa de uma plataforma multi-missão (MMP) a ser utilizada no projeto dos satélites SSR-1 e SSR-2 (Satélites de Sensoriamento Remoto). Além disso, o acordo com a China está sendo renovado para o desenvolvimento de mais três satélites de sensoriamento remoto (CBERS-2B, CBERS-3 e CBERS-4).

Para a operação destes satélites, o INPE possui uma infra-estrutura composta de: um Centro de Controle, localizado em São José de Campos; as Estações Terrenas de Cuiabá, Alcântara e Natal; e uma rede de comunicação de dados que permite a troca de

dados entre estas unidades. Uma importante parte desta infra-estrutura é composta de sistemas de softwares que são responsáveis pelo controle e monitoração dos satélites e das estações terrenas. O desenvolvimento destes sistemas está sob a responsabilidade da Divisão de Desenvolvimento de Sistemas de Solo (DSS) do INPE.

Desde o inicio de suas atividades a DSS sempre buscou o desenvolvimento de sistemas cuja arquitetura permitisse o máximo de reuso para outros satélites.

O primeiro sistema desenvolvido por esta divisão foi o Sistema de Controle de Satélites (SICS) que tinha o objetivo de atender os satélites da MECB. Graças à sua concepção baseada em arquivos de configuração, este sistema foi inicialmente utilizado no controle do primeiro satélite brasileiro (SCD1) e foi reutilizado, com um mínimo de adaptação, para atender o satélite SCD2 (Yamaguti et al., 1990).

Embora a arquitetura deste sistema tenha atendido os requisitos dos satélites da MECB, ela não se mostrou suficientemente flexível para atender os requisitos de controle dos satélites de sensoriamento remoto do programa CBERS. Para atender os novos requisitos foi necessário desenvolver o Sistema de Telemetria e Telecomando (TMTC).

O TMTC foi desenvolvido, segundo uma arquitetura orientada a objetos, com o objetivo principal de atender os requisitos do CBERS e além disso permitir também o controle dos satélites da MECB. Este segundo objetivo tinha a finalidade de permitir a redução dos custos do Centro de Rastreio e Controle (CRC) do INPE. O sistema anterior, SICS, utilizava máquinas VAX/Alpha de alto custo de manutenção, enquanto que o TMTC utiliza microcomputadores.

Do SICS, o TMTC herdou a facilidade de configuração, com a diferença que a tecnologia de armazenamento de dados migrou de arquivos para banco de dados relacional. A novidade foi a utilização de uma modelagem baseada em objetos em contraponto à arquitetura baseada em procedimentos do SICS. O TMTC foi projetado para ser um *framework* de classes que permitisse uma fácil adaptação para atendimento tanto dos requisitos dos satélites CBERS quanto os da MECB. As classes do *framework* contêm as funções comuns às duas famílias de satélites e ao mesmo tempo permite que

as funções específicas sejam redefinidas através da especialização de algumas de suas classes (Cardoso et al., 1996 ; Cardoso, Gonçalves e Ambrósio., 1998; Gonçalves, Cardoso e Ambrósio, 1998).

Frameworks são projetos reusáveis, onde todas as partes do sistema são descritas por um conjunto de classes abstratas e pela forma com que as instâncias dessas classes colaboram entre si (Roberts e Johnson,1998).

Apesar de este sistema estar sendo utilizado com sucesso no controle dos satélites CBERS, SCD1 e SCD2, ele apresenta limitações, nos pontos de especialização de classes previstos no *framework,* para atender requisitos específicos de cada missão. Esta falta de flexibilidade ficou demonstrada diante dos requisitos de operação estabelecidos para a linha de micro-satélites científicos, uma vez que estes satélites possuem uma interface de comunicação solo-bordo bastante diferente das missões anteriores.

Com a finalidade de gerar um sistema com maior grau de reusabilidade, uma terceira geração de Sistemas de Controle de Satélites está sendo desenvolvida pela DSS, para atender os requisitos da missão de micro-satélites científicos, fazendo uso da experiência acumulada com os dois sistemas anteriores (arquivos de configuração e *framework*) e introduzindo uma maior utilização de *Design Patterns* (Gamma et al., 1995).

Os *design patterns* servem de diretrizes para os desenvolvedores, fazendo com que os modelos de computação que usam o mesmo *design pattern* sejam construídos de acordo com os mesmos princípios (Lee, 2001).

O processo de transformar um modelo usando um *design pattern* é chamado de *pattern-based refactoring*. Pode-se criar processos rigorosos de transformação usando *design patterns* através do desenvolvimento de metamodelos chamados "especificações das transformações" (France et al., 2003).

O que se percebe que a cada nova missão, se ganha mais experiência para definir um sistema de controle mais flexível e que possa ser reutilizado por outras missões com um

mínimo de modificações. Mas como as modificações são inevitáveis, já que a cada nova missão novos requisitos são definidos, o desafio deste trabalho é buscar um meio para que estes novos requisitos possam ser adicionados ao sistema sem que haja, ou pelo menos reduzindo, a necessidade de modificação do código.

Uma forma de se conseguir isto é mover certos aspectos do sistema, como regras de negócio (lógica do sistema), por exemplo, para o banco de dados, fazendo com que dessa forma, elas possam ser facilmente modificadas. O modelo resultante permite que o sistema possa se adaptar rapidamente às novas necessidades do domínio através de modificações nos valores armazenados no banco de dados, ao invés de modificações no código (Yoder, Balaguer e Johnson, 2001).

Esta facilidade de modificação encoraja o desenvolvimento de ferramentas que permitam que os próprios especialistas do domínio introduzam novos elementos ao *software* sem a necessidade de programação adicional, e que possam até mesmo fazer mudanças em seus modelos de domínio em tempo de execução, reduzindo significantemente o tempo para incorporação de novos requisitos ao *software*.

Arquiteturas que podem dinamicamente se adaptar em tempo de execução a novos requisitos de usuários são chamadas de “arquiteturas reflexivas” ou “meta-arquiteturas”. Esse tipo de arquitetura baseia-se na propriedade de “reflexão”, que é a propriedade de um sistema permitir que sua estrutura e operação possam ser controladas e atualizadas de fora dele. (Killijian e Fabre, 2000).

Assim, um sistema reflexivo tem uma representação de si mesmo que pode ser observada (auto-representação). Freqüentemente, essa auto-representação é expressa em termos de entidades abstratas que podem ser manipuladas para modificar o comportamento do sistema. Então, um sistema reflexivo promove a escrita de componentes genéricos e reusáveis que manipulam essa auto-representação (Nguyen-Tuong e Grimshaw, 1999).

O modelo reflexivo sugere um novo paradigma para o desenvolvimento de sistemas. Nesse novo paradigma, um sistema é decomposto em pelo menos dois níveis: **(1)** o

nível base, onde estão agrupadas as funções relacionadas exclusivamente ao domínio do problema e **(2)** o nível reflexivo que agrupa o código que supervisiona, adapta e retorna informações sobre o nível base (Stehling, 1999).

Uma "arquitetura de modelos de objetos adaptáveis" *(Adaptive Object_Model Architecture-)* é um tipo particular de arquitetura reflexiva que abrange sistemas orientados a objeto que gerenciam elementos de algum tipo, e que podem ser estendidos para adicionar novos elementos. Sistemas que têm este tipo de arquitetura podem ser chamados de "Modelos de Objetos Ativos" *(Active Object-Models)*, "Modelos de Objetos Dinâmicos" *(Dynamic Object Models - DOM)*, ou "Modelos de Objetos Adaptáveis" *(Adaptive Object-Models)* (Yoder, Balaguer e Johnson, 2001).

Usar a abordagem dos modelos de objetos adaptáveis no desenvolvimento de sistemas pode amenizar alguns dos problemas que vêm sendo encontrados pelos desenvolvedores de *software*, principalmente em relação à flexibilidade e evolução, permitindo que o custo total do desenvolvimento e manutenção possa ser reduzido (Ledeczi, Bapty e Karsai, 2000).

1.1 Objetivos do Trabalho de Pesquisa

Para reduzir os custos e o tempo de desenvolvimento dos sistemas de solo, responsáveis pelo controle de satélites, está se propondo uma arquitetura que permita que uma nova missão espacial possa ser acomodada sem a necessidade de se criar um sistema específico para cada satélite a ser controlado. Considerando que uma nova missão pode apresentar novos requisitos de operação, esta arquitetura deve também permitir que o esforço necessário para adaptar o sistema de controle para atender estes novos requisitos seja minimizado.

Além disso, devido à escassez de recursos humanos na área de desenvolvimento de sistemas de *software* de controle de satélites do INPE, deseja-se que os usuários especialistas do domínio (engenheiros e controladores de satélites) tenham cada vez mais a condição de realizar pequenas configurações no sistema de controle para atender

novos requisitos de menor complexidade que se apresentem. Neste caso, os desenvolvedores dos sistemas de software de controle poderiam ser então alocados para a análise de novos e mais complexos requisitos que obrigatoriamente exigissem o desenvolvimento de novos sistemas ou modificações mais profundas nos sistemas existentes.

O objetivo desta pesquisa é criar uma infra-estrutura baseada em um repositório de objetos adaptáveis que permitam que os próprios usuários possam fazer pequenas modificações em algumas regras de negócio dos seus sistemas sem depender das equipes de desenvolvimento de software. Para alcançar este objetivo é necessário: desenvolver uma ferramenta de edição das regras de negócio que leve em consideração a não familiaridade destes usuários com técnicas de programação e que portanto, seja a mais natural possível em relação ao domínio de conhecimento dos mesmos; em segundo lugar é necessário criar uma interface entre os sistemas destes usuários e o repositório onde os objetos estão armazenados.

Este trabalho tem seus fundamentos parcialmente baseados em uma tese de doutorado que visa o estabelecimento de uma arquitetura distribuída, configurável e adaptável para o *software* de controle de satélites, que está baseada em modelos de objetos adaptáveis (Thomé, 2004). A arquitetura proposta por Thomé apresenta-se como uma evolução da arquitetura SOFTBOARD, proposta em Cunha (1997), e da arquitetura SICSD, proposta em Ferreira (2001).

É importante salientar que esta pesquisa não tem a pretensão de permitir que qualquer pessoa possa fazer mudanças nos sistemas dos quais é usuário. Portanto, no restante deste trabalho sempre que for utilizado o termo “usuário” deve-se entender que o mesmo não se refere a um usuário comum, mas a pessoas com grande conhecimento em um domínio de problema e com capacidade e autorização para fazer mudanças no comportamento de seus sistemas.

1.2 Motivação do Trabalho de Pesquisa

Os modelos de objetos adaptáveis ainda são pouco conhecidos, e apresentam um grande potencial para ser explorado, principalmente no que refere à definição das regras de negócio configuráveis pelo usuário. Exemplos de aplicações desenvolvidas usando-se modelos de objetos adaptáveis podem ser encontrados em Johnson e Yoder (2002), Yoder, Balaguer e Johnson (2001) e Yoder e Johnson (2003).

Na maior parte dos trabalhos pesquisados as soluções dadas se concentram em permitir que os objetos possam ter seus atributos e associações configuráveis pelo usuário. Poucas referências, como por exemplo Arsanjani (2000) e Manolescu (2000), se concentram em soluções para prover os objetos de regras de negócio configuráveis pelo usuário.

A proposta de transferir aos usuários tarefas que normalmente são realizadas por analistas e programadores é bastante desafiadora e motivadora. Estes usuários têm conhecimento do domínio do problema, mas não necessariamente de programação. Portanto, não é viável oferecer a eles uma "interface de programação", para alterar as regras de negócio, que tenha como base uma linguagem de programação tradicional. Por outro lado, esta interface deve ter um grau mínimo de recursos para maximizar a possibilidade do próprio usuário fazer as modificações necessárias, sem depender de um especialista em desenvolvimento de *software*. Uma possível solução é oferecer uma ferramenta de edição das regras de negócio e uma linguagem que sejam as mais naturais e próximas possíveis do domínio de conhecimento destes usuários.

Outra motivação importante para o desenvolvimento deste trabalho vem do próprio INPE. Na verdade, visando aproveitar o esforço despendido no projeto e na construção dos equipamentos que compõe um satélite, o INPE lançou um projeto multi-missão que deverá entrar em vigor em 2007. Esse projeto propõe a construção de um barramento configurável, chamado de MMP que serve como base para a construção de vários satélites (INPE, 2001).

Ao propor a plataforma multi-missão MMP, o INPE demonstrou o seu interesse em reaproveitar esforços realizados, visando à concepção de arquiteturas de *hardware* mais flexíveis, e que possam ser facilmente configuráveis.

O trabalho aqui proposto, na realidade, vem ao encontro dos objetivos do projeto multi-missão, uma vez que o reaproveitamento do esforço despendido para gerir as missões pode ser melhorado ainda mais, se esta facilidade de configuração, além do *hardware*, for extendida ao *software* de controle.

1.3 Metodologia de Desenvolvimento do Trabalho de Pesquisa

Na primeira etapa deste trabalho, a pesquisa se concentrou no levantamento da bibliografia existente e no estudo das diversas metodologias, tecnologias, filosofias e arquiteturas que abrangem os aspectos que o motivaram. Assim, este estudo abrangeu as arquiteturas dos sistemas de controle de satélites desenvolvidos ou em desenvolvimento pelo INPE até o momento, aspectos sobre sistemas configuráveis, conceitos sobre computação reflexiva, e finalmente, conceitos e aplicações de modelos de objetos adaptáveis.

Em seguida, uma vez que na literatura pesquisada sobre modelos de objetos adaptáveis muito se encontrou sobre a configuração das propriedades dos objetos e muito pouco sobre como configurar suas regras de negócio, o trabalho se concentrou principalmente na adaptação do modelo para a representação destas.

Uma vez definido o modelo das regras de negócio foram elaborados o projeto e a implementação do protótipo de uma ferramenta de edição amigável para o usuário e que ao mesmo tempo pudesse ser usada para testes dos objetos criados através dela. Este projeto utilizou a *Unified Modeling Language* (UML) como notação para representar os diagramas de casos de uso, de classe, de seqüência e as entidades e relacionamentos da base de dados.

Finalmente, o trabalho de pesquisa realizado foi expresso nesta Dissertação, que tem a sua estrutura detalhada a seguir.

1.4 Esboço Geral

Este trabalho foi dividido em mais seis capítulos, descritos a seguir:

- *CAPÍTULO 2 – ARQUITETURAS DOS SISTEMAS DE CONTROLE DE SATÉLITES DO INPE*: neste capítulo é apresentada uma visão geral das arquiteturas dos vários sistemas de controle desenvolvidos ou em desenvolvimento pelo INPE, mostrando principalmente suas características de reusabilidade.

- *CAPÍTULO 3 – MODELOS DE OBJETOS ADAPTÁVEIS*: neste capítulo são abordados os conceitos de modelos de objetos adaptáveis, suas particularidades, e os *design patterns* que estão envolvidos na obtenção de um metamodelo ou de um modelo genérico para representar um sistema. São abordadas, ainda, algumas vantagens e desvantagens envolvidas no uso de modelos de objetos adaptáveis.

- *CAPÍTULO 4* – SOLUÇÃO PARA UTILIZAÇÃO DO MODELO ADAPTATIVO NOS SISTEMAS DE CONTROLE DE SATÉLITES: neste capítulo é apresentada uma proposta de utilização do modelo adaptativo nos sistemas de controle de satélites. Esta proposta inclui a abordagem para a implementação do modelo, as classes que compõem o *framework* gerado, a interface entre o sistema adaptativo e o sistema de controle de satélites, e os metadados, que foram definidos para a construção do protótipo da arquitetura. Também é apresentada uma visão da dinâmica do funcionamento das classes do *framework* e da ferramenta de edição.

- *CAPÍTULO 5* – IMPLEMENTAÇÃO DE UM PROTÓTIPO DA ARQUITETURA: este capítulo inclui a descrição da interface com o usuário da ferramenta que foi construída para facilitar a edição dos objetos dinâmicos e um exemplo de utilização do modelo DOM pelo sistema de controle de satélites.

- *CAPÍTULO 6 – CONCLUSÃO*: neste capítulo são apresentados as conclusões e resultados obtidos com o trabalho de pesquisa, bem como, algumas sugestões para a realização de trabalhos complementares.

CAPÍTULO 2
ARQUITETURAS DOS SISTEMAS DE CONTROLE DE SATÉLITES DO INPE

Neste Capítulo será apresentado um resumo da evolução dos sistemas de controle de satélites desenvolvidos pelo INPE com enfoque nas suas características de reusabilidade. Em função da complexidade e do volume de conceitos relativos ao domínio de controle de satélites e para facilitar o entendimento da evolução ocorrida, nas descrições a seguir foi selecionado apenas o subsistema de telemetria, uma vez que ele é representativo da arquitetura do sistema completo.

O subsistema de Telemetria é responsável pela monitoração remota dos equipamentos do satélite. Periodicamente, o computador de bordo do satélite coleta todas as informações necessárias para que os operadores do Centro de Controle possam monitorar o estados dos equipamentos a bordo e codifica tais informações em uma seqüência de bits. Estas seqüências de bits são transmitidas para o solo durante o período de visibilidade do satélite pelas estações terrenas. Para otimizar o link de comunicação entre o solo e o satélite, todos estes valores são codificados de forma compactada na seqüência de bits enviada ao solo. Por exemplo, o estado de um equipamento pode ser codificado através de um único bit (ligado ou desligado). A esta seqüência de bits transmitidos para o solo se dá o nome de mensagem de telemetria ou pacote de telemetria. As telemetrias podem ser digitais ou analógicas. As telemetrias digitais podem ser: o estado de um equipamento ou um contador de telecomandos recebidos pelo satélite. As telemetrias analógicas podem ser, entre outros, os valores de temperatura, de corrente e de tensão obtidos de sensores conectados nos equipamentos.

Na Figura 2.1 é apresentada a forma padrão de uma mensagem de telemetria. Ela é composta de um cabeçalho e de um campo de dados. O cabeçalho identifica e caracteriza o conteúdo do campo de dados. Tanto o cabeçalho quanto o campo de dados são compostos de parâmetros que correspondem às informações coletadas dos equipamentos a bordo.

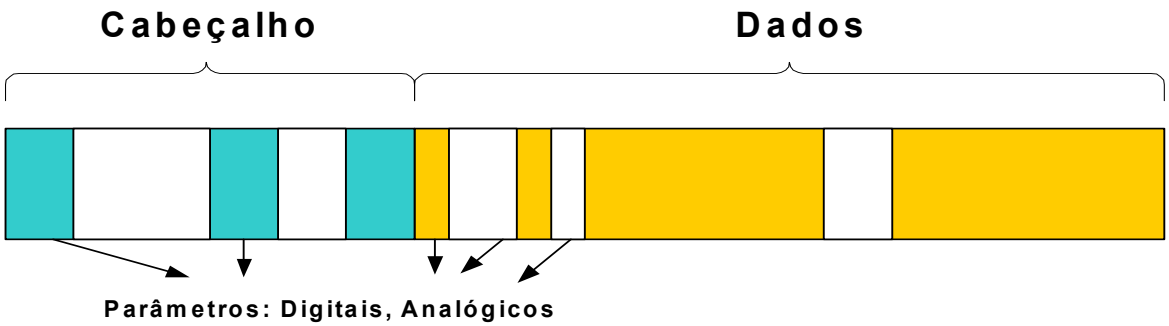

FIGURA 2.1 – Pacote de telemetria.

Os sistemas de solo responsáveis pela monitoração do satélite têm que analisar os parâmetros do cabeçalho para identificar o conteúdo do campo de dados e em seguida processar os parâmetros do campo de dados. Este processamento é composto de duas atividades principais: decodificação e monitoração. Na decodificação, os valores brutos dos parâmetros são extraídos da seqüência de bits e transformados para valores em unidade de engenharia. Nesta transformação são utilizadas funções de transferência de vários tipos. As duas principais são: curvas de calibração, que permitem obter valores em unidade de engenharia a partir da interpolação do valor bruto; e tabelas que associam rótulos a valores brutos. A atividade de monitoração visa auxiliar o operador na identificação de problemas nos equipamentos a bordo. Os dois principais tipos de monitoração são a verificação de valores limites e a mudança de estado. Por exemplo, o operador deve ser notificado se a temperatura de um sensor ultrapassar um valor predefinido ou se um equipamento mudou para um estado não esperado.

A abstração do processamento de telemetria descrita acima só foi obtida após vários anos de experiência acumulada no desenvolvimento do subsistema de telemetria para três diferentes famílias de satélites. Esta experiência está relatada de forma resumida nos sub-itens a seguir relativos aos sistemas desenvolvidos ou em desenvolvimento: SICS, TMTC e FBM.

2.1 Sistema de Controle de Satélites - SICS

O subsistema de telemetria do SICS, que foi o primeiro sistema de controle de satélites desenvolvido pelo INPE, provia a facilidade de uma base de dados (arquivos de configuração do sistema) para descrever os parâmetros que compunham o campo de dados da mensagem de telemetria, curvas de calibração e monitoração por valores limites e mudanças de estado. Provia também um recurso que permitia que o usuário pudesse, via os arquivos de configuração, preparar equações lógicas em função de parâmetros lógicos (0 ou 1), extraídos da mensagem de telemetria recebida do satélite, e de operadores lógicos (OU, E e NOT). Os valores calculados através destas equações podiam ser exibidos para o operador do satélite de modo a compor uma informação sobre o estado de um subsistema do satélite ou mesmo do satélite completo. Estes valores podiam também ser utilizados para gerar condições para o processamento ou não de outros parâmetros.

Apesar das facilidades oferecidas, a base de dados estava restrita aos requisitos de processamento de telemetria dos satélites da MECB: só suportava os tipos de parâmetros então conhecidos; não previa a existência de diferentes campos de dados; e o processamento dos parâmetros do cabeçalho das mensagens estava embutido no código e não era configurável. Outra restrição deste sistema era que o cálculo de parâmetros derivados mais complexos que aqueles suportados pelas equações lógicas estava embutido no próprio código. Com isto este subsistema só foi capaz de atender os requisitos de processamento de telemetria dos satélites SCD1 e SCD2. Para a missão CBERS foi necessário desenvolver um novo sistema (TMTC) que incluiu o desenvolvimento de um novo subsistema de telemetria.

2.2 Sistema de Telemetria e Telecomando - TMTC

No projeto do subsistema de telemetria do TMTC foi levado em conta o atendimento tanto dos requisitos dos satélites CBERS quanto os dos satélites da MECB. Para se atingir este objetivo foi projetada uma nova base de dados descritiva para suportar o processamento de telemetria e o subsistema foi modelado como um *framework*. O novo

projeto da base de dados incluiu os novos tipos de parâmetros do CBERS e os parâmetros do cabeçalho das mensagens de telemetria. Através do *framework* agrupou-se tudo que havia de comum no processamento de telemetria dos satélites das duas famílias em um conjunto de classes conforme mostrado nas Figuras 2.2 e 2.3.

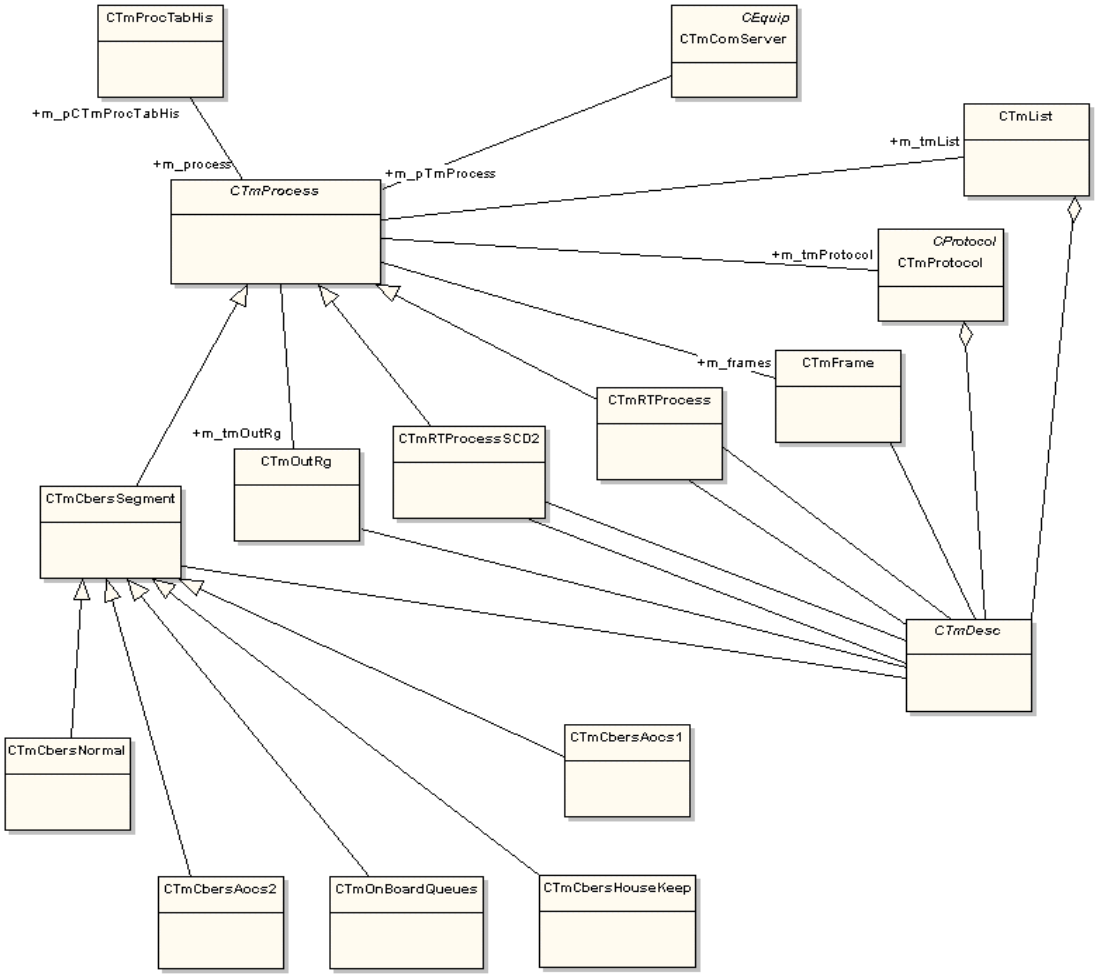

FIGURA 2.2 – Diagrama de classes do subsistema de telemetria.

Na Figura 2.2, a classe CTmProcess é responsável pelo processamento de telemetria comum às duas famílias de satélites. Esta classe é especializada para os satélites CBERS através das classes CTmRTProcess e CTmCbersSegment e para os satélites da MECB através da classe CTmRTProcessSCD2. Nestas especializações, basicamente foram redefinidos os serviços da classe base, responsáveis pela identificação do campo de dados das mensagens de telemetria e pela criação de alguns tipos especiais de telemetria. Toda a lógica restante de inicialização do sistema e processamento de mensagens de

telemetria está incluída na classe base CTmProcess. Durante a inicialização a classe base recupera da base de dados, a descrição de todos os parâmetros e as listas de parâmetros que fazem parte do cabeçalho e dos diferentes campos de dados. Em seguida, ela cria os objetos correspondentes aos parâmetros (derivados da classe base CTmDesc) e os armazena em uma lista de parâmetros (classe CTmList), cria o objeto da classe CTmProtocol que é responsável pela análise do cabeçalho e cria objetos da classe CTmFrame que é responsável pelo processamento de cada campo de dados. Na classe base também está definida a seqüência de ativação dos objetos para processar as mensagens de telemetria.

Na Figura 2.3 estão representadas as classes responsáveis pelo processamento de cada tipo de parâmetro que pode estar contido no cabeçalho ou campo de dados da mensagem de telemetria. A classe CTmDesc é uma classe base que descreve um parâmetro genérico e a partir dela são geradas derivadas para cada tipo específico de parâmetro. Nestas especializações são incluídas as funções de decodificação, função de transferência e monitoração associadas com cada tipo de parâmetro.

As funções de equações lógicas e parâmetros derivados, já presentes no sistema anterior, foram inovadas de modo a oferecer mais recursos e serem mais facilmente configuráveis. No caso das equações lógicas foram acrescentados novos operadores (MAIOR, MENOR, IGUAL) que podiam ser aplicados em parâmetros do tipo inteiro, extraídos da mensagem de telemetria, de modo a gerar valores lógicos. No caso dos parâmetros derivados era possível inserir novas funções através da configuração das mesmas, bem como dos parâmetros de telemetria utilizados por elas, na base de dados do sistema. Porém, o código destas funções tinha que ser definido através da utilização de uma biblioteca do tipo *Dynamic Link Library* (DLL). A vantagem da utilização da DLL é que novas funções podem ser incluídas sem alterar o código executável, como acontecia no sistema anterior.

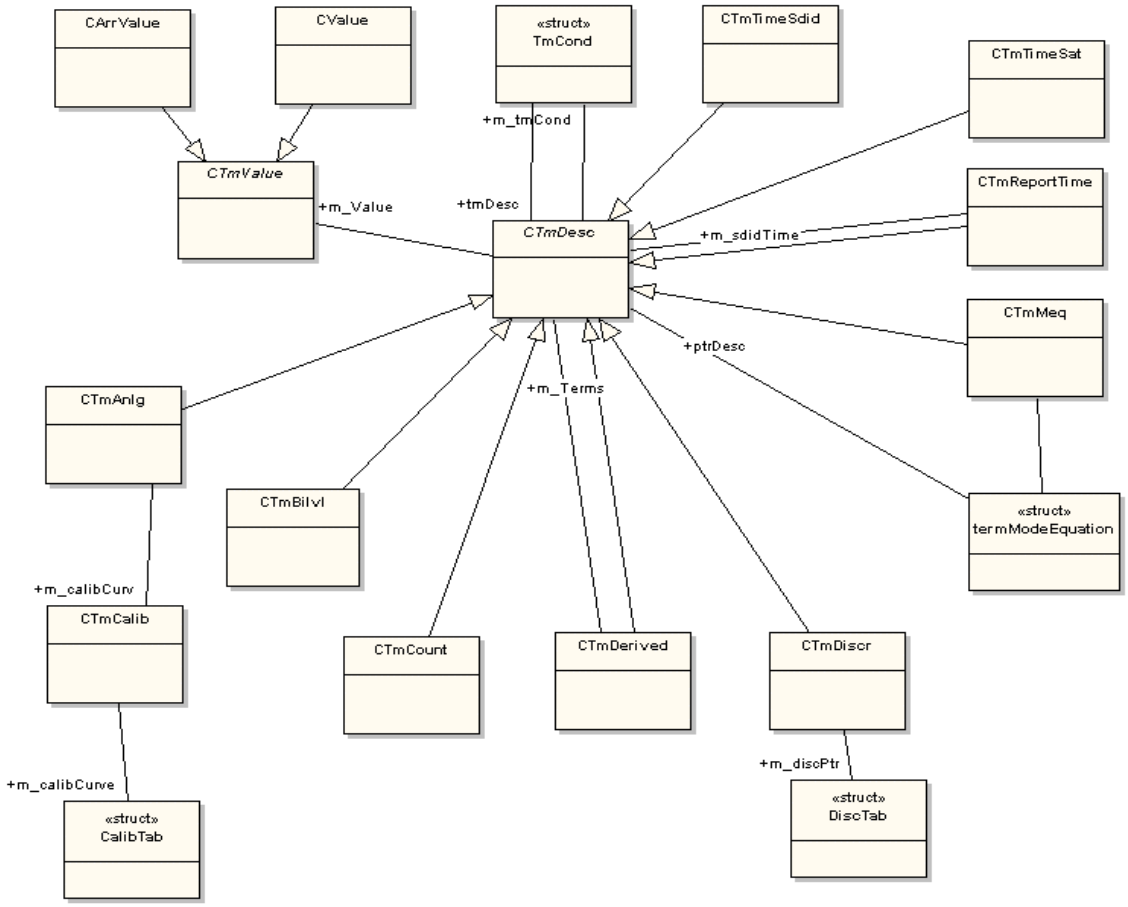

FIGURA 2.3 – Diagrama de classes dos tipos de parâmetros.

Como o *framework* de classes do TMTC apresenta limitações para atender os requisitos de processamento de telemetria das missões de satélites científicos, foi iniciado o projeto de uma terceira geração do sistema de controle, incluindo um novo subsistema de telemetria.

2.3 Micro-Satélites Científicos

Neste novo projeto buscou-se aumentar a flexibilidade e generalização da base de dados e do *framework* do TMTC de modo que o mesmo pudesse atender aos requisitos das três diferentes famílias de satélites com um reduzido número de alterações. Para atingir a meta de aumentar a flexibilidade do *framework* foi introduzido no projeto o uso de *Design Patterns* (Gamma et al.,1995).

A Figura 2.4 apresenta o diagrama de classes idealizado para o novo subsistema de Processamento de Telemetria (MON). Neste diagrama, os pacotes de telemetria são tratados como uma estrutura de árvore que tem como base a estrutura apresentada na Figura 2.1. Pacotes contêm cabeçalho e campo de dados; cabeçalhos contêm parâmetros; o campo de dados é composto de vários diferentes conjuntos de parâmetros e podem condicionalmente conter subestruturas; e subestruturas contêm parâmetros. Esta estrutura de árvore foi modelada com auxilio do padrão *Composite*: classes CMONTelemetry, CMONParameter e CMONTmComposite.

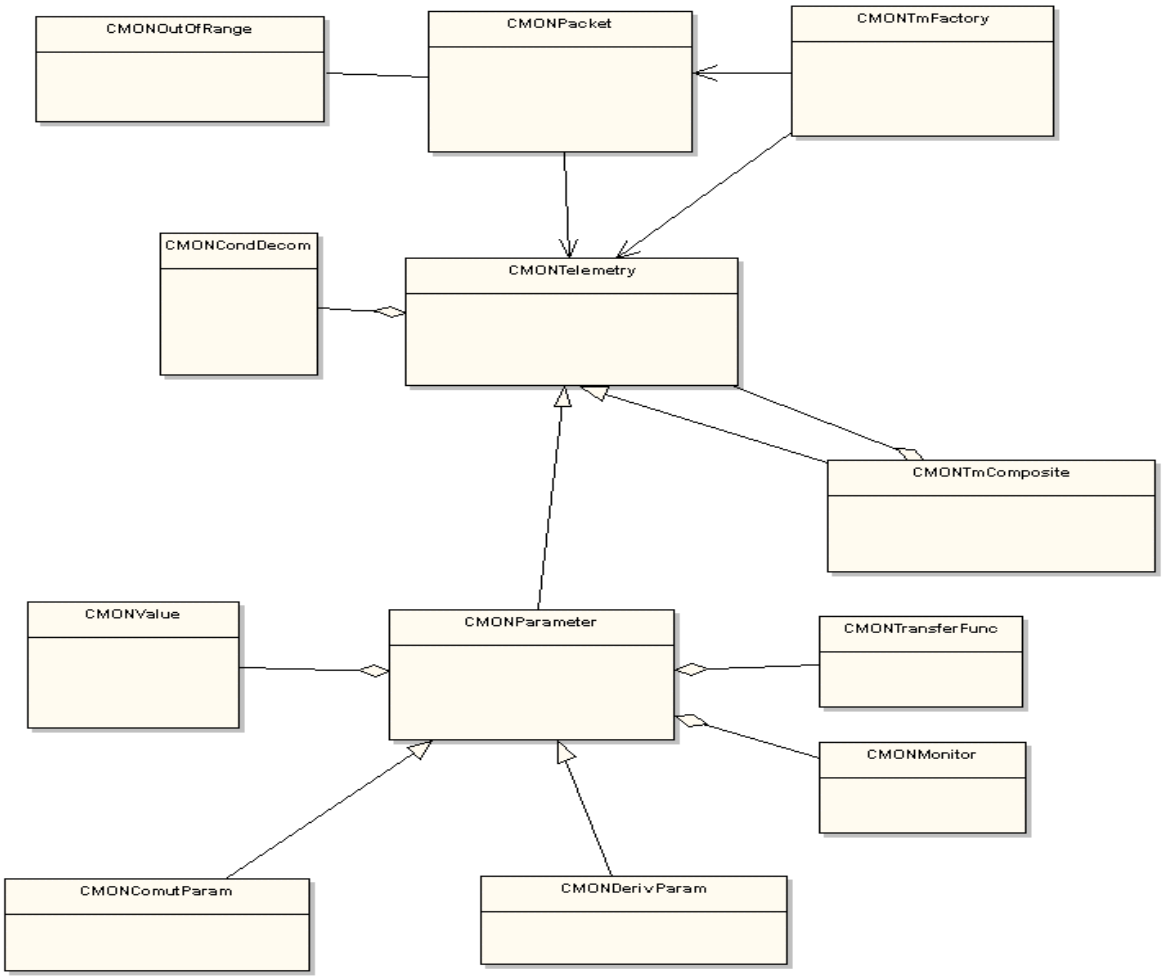

FIGURA 2.4 – Diagrama de classes MON.

Em vez de ser especializada em várias classes, como no TMTC (vide Figura 2.3), a classe base dos parâmetros (CMONParameter) tem apenas duas subclasses. Esta

redução no número de classes de especialização foi atingida através da delegação das funções de monitoração e função de transferência, que anteriormente estavam intrinsecamente ligadas a cada tipo de parâmetro, para objetos das classes CMONMonitor e CMONTransferFunc. Para recuperar os registros da base de dados e criar os objetos na memória de uma forma que facilitasse futuras adaptações foram utilizados os padrões *Builder* e *Factory* na composição dos elementos de um pacote de telemetria e na criação dos vários tipos de funções de transferência e monitoração. A classe CMONTmFactory é responsável pela criação da estrutura dos pacotes de telemetria a partir das informações contidas na base de dados. Embora não mostrado na figura, também se adotou uma adaptação do padrão *Strategy* no modelamento da classe CMONDerivParam. Esta classe é responsável por obter valores derivados através de expressões matemáticas e lógicas, contidas na base de dados, que utilizam os valores de outros parâmetros de telemetria. Também não está ilustrada na figura a utilização do padrão *Singleton* para garantir a existência de um único de objeto de uma classe.

A classe CMONDerivParam foi uma evolução muito importante, do ponto de vista de reusabilidade, em relação aos serviços de cálculos de parâmetros derivados disponíveis no sistema anterior. A maior parte das funções derivadas, que no sistema anterior exigiam codificação e inserção em uma DLL, passou a ser totalmente configurável, via base de dados, pelo próprio usuário especialista no domínio de controle de satélites. Porém como nem mesmo esta classe cobre algumas funções de processamento que são particulares de cada missão, o sistema em desenvolvimento ainda mantém a possibilidade de incluí-las através da utilização de uma DLL de modo a minimizar as alterações no código executável do sistema.

2.4 Evolução dos Sistemas de Controle de Satélites

Como pode ser visto ao longo deste Capítulo, em todos os sistemas desenvolvidos ou em desenvolvimento pelo INPE para o controle de satélites sempre existiu a preocupação de torná-los o mais reusáveis possível e que pudessem ser configurados pelo próprio usuário especialista no controle dos satélites. Todavia, o histórico apresentado mostrou que este objetivo depende de uma maior experiência e

conhecimento sobre o domínio do problema, e também do grau de inovação de requisitos de controle apresentado por uma nova missão espacial.

Acredita-se que através do sistema de controle que está sendo desenvolvido para os micro-satélites científicos se alcançou um *framework* robusto que contempla o núcleo de processamento comum aos satélites em operação e aos previstos para serem lançados nos próximos anos. Pensando nas particularidades de cada missão espacial, este sistema foi estruturado de modo a facilitar a inclusão de código para atender os requisitos específicos apresentados por cada satélite:

- Utilização dos padrões *Builder* e *Factory* para inserção de novos tipos de composição dos elementos de um pacote de telemetria, novos tipos de funções de transferência e novos tipos monitoração;

- Utilização de uma DLL para inclusão de funções de processamento especiais.

Mas por menores que sejam, estas mudanças devem ser realizadas pelos especialistas em desenvolvimento de *software* com conhecimento do *framework*. Para continuar o processo de busca de aplicações cada vez mais reutilizáveis, o próximo passo da evolução do sistema de controle que está sendo proposto, através deste trabalho, é a transferência da responsabilidade destas mudanças para usuários especialistas no domínio do controle de satélites. No próximo Capítulo está descrita a tecnologia de objetos adaptáveis que foi estudada com uma possível solução para se atingir esta evolução.

CAPÍTULO 3

MODELO DE OBJETOS ADAPTÁVEIS

Muitos sistemas de informações hoje necessitam ser dinâmicos e configuráveis de modo que possam ser rapidamente adaptados para novas necessidades de negócio. Isto normalmente é feito movendo-se certos aspectos do sistema, tais como as regras de negócio, para dentro de banco de dados de maneira que eles possam ser facilmente modificados. O modelo resultante permite que um sistema possa ser adaptado às novas regras de negócio através de uma simples mudança nos valores no banco de dados em vez de realizar uma mudança no código. Desta maneira é possível introduzir novos produtos sem programação e até mesmo fazer mudanças nos modelos de negócio em tempo de execução. Com esta nova abordagem, é possível se reduzir bastante o tempo de lançamento de um novo produto e o poder de modificar um sistema pode ser, dentro de certos limites, movido das mãos dos programadores para aqueles que têm o conhecimento do negócio.

A solução para implementação de um sistema com estas características é a utilização de arquiteturas que se adaptam dinamicamente em tempo de execução a novos requisitos do usuário. Estas arquiteturas são algumas vezes chamadas de arquiteturas reflexivas ou meta-arquitetura. Sistemas construídos com esse tipo de arquitetura são também chamados de Modelos de Objetos Dinâmicos (*DOM*) ou Modelos de Objetos Adaptativos (*AOM*).

Um sistema com base no modelo AOM é um sistema que representa classes, atributos e relacionamentos como metadados. Os usuários mudam os metadados para refletir mudanças no domínio. Estas mudanças modificam o comportamento do sistema. Em outras palavras, o modelo de objetos é armazenado em um banco de dados e interpretado em tempo de execução. (Yoder, Balaguer e Johnson, 2001)

Estes tipos de sistemas normalmente emergem de *frameworks* maduros de um domínio específico. À medida que um *framework* evolui, os desenvolvedores ganham um melhor entendimento do domínio do problema e conseguem identificar as partes mais

propensas a mudanças. Estas partes são então definidas como dados de configuração. O sistema é desenvolvido de modo a interpretar os dados de configuração em tempo de execução. Desta maneira futuras mudanças de comportamento não mais requerem mudanças no código.

Contudo, estas características têm o seu preço. Este estilo é complexo e estes sistemas são desenvolvidos normalmente por um pequeno grupo de desenvolvedores com grande experiência. Esta arquitetura normalmente demanda um longo tempo para desenvolver o *framework* e desenvolvedores com pouca experiência sentem muita dificuldade em depurar, manter e entender os sistemas produzidos devido ao alto grau de abstração do metamodelo gerado. Além disso, para permitir que o usuário possa modificar o sistema sem cometer erros, é necessário fornecer ferramentas de suporte adequadas (Manolescu e Johnson, 1999).

Sistemas deste tipo geralmente fazem uso de uma arquitetura comum baseada em um conjunto de *Design Patterns* fundamentais (Johnson e Yoder, 2002). Os objetos das classes que compõem esta arquitetura são construídos a partir de metadados os quais contém a descrição dos objetos do mundo real e seus relacionamentos.

A seguir serão apresentados mais detalhes sobre os padrões fundamentais (Item 3.1), a arquitetura comum (Item 3.2), os metadados (Item 3.3) e técnicas reflexivas (Item 3.4). Em função de sua importância para a arquitetura DOM, também serão apresentadas considerações sobre a construção dos objetos do modelo a partir dos metadados (Item 3.3.1), a interpretação das regras definidas através dos metadados (Item 3.3.2) e o controle de mudanças nos metadados (Item 3.3.3).

3.1 Padrões fundamentais do modelo DOM

O núcleo da arquitetura DOM é geralmente definido através da combinação de alguns padrões de modo a permitir que o usuário possa controlar o estado, o comportamento e os relacionamentos entre as entidades do modelo.

3.1.1 Padrão *Type-Object*

A maioria das linguagens orientadas a objeto estrutura um programa como sendo um conjunto de classes, onde uma classe define a estrutura e o comportamento dos seus objetos.

É comum em um sistema orientado a objeto uma determinada classe requerer, além de um número desconhecido de instâncias, também um número desconhecido de subclasses. Como sistemas orientados a objeto geralmente usam uma classe separada para cada tipo de objeto, introduzir um novo tipo de objeto requer que se construa uma nova classe, o que requer programação (Johnson e Wolf, 1998).

O padrão *TypeObject* faz com que as subclasses desconhecidas sejam substituídas por associações com instâncias de uma classe genérica. Assim, novas classes podem ser criadas dinamicamente em tempo de execução através da instanciação da classe genérica (Yoder, Balaguer e Johnson, 2001).

A Figura 3.1 ilustra a substituição de várias classes especializadas por uma única classe *Entity*. que se relaciona com a classe *EntityType*. Cada instância de *EntityType* armazena as informações e lógica que caracterizam as diferentes especializações da classe *Entity* original. Desta forma qualquer instância de uma das especializações da antiga classe *Entity* (*Entity i*) pode ser então substituída por uma instância da classe *Entity* associada com uma instância adequada da classe *EntityType*. Substituir uma hierarquia como essa é possível quando o comportamento entre as subclasses é bastante similar ou pode ser quebrado em objetos separados. Por essa razão, as diferenças primárias entre subclasses são seus atributos (Yoder, Balaguer e Johnsonr, 2001).

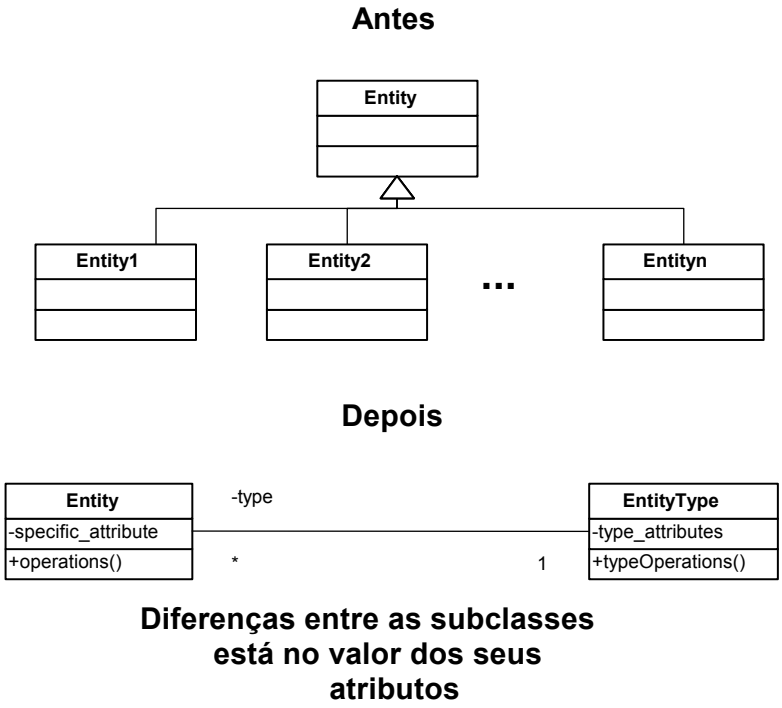

FIGURA 3.1 - Padrão *Type-Object.*
FONTE: adaptada de Yoder, Balaguer e Johnson(2001, p. 51).

3.1.2 Padrão *Property*

Os atributos de um objeto são usualmente implementados por suas variáveis de instância, sendo que essas variáveis são usualmente definidas em cada subclasse. Porém, com a utilização do padrão *TypeObject*, objetos de tipos diferentes estarão todos definidos através de uma mesma classe, o que significa que se deve adotar uma forma alternativa para implementar seus atributos.

Uma solução é utilizar o padrão *Property*, o qual em lugar de implementar cada atributo como uma variável, utiliza uma única variável para armazenar uma coleção de atributos. Cada atributo é associado a uma chave. Os usuários podem usar estas chaves para acessar, modificar ou remover atributos durante o tempo de execução (Manolescu e

Johnson, 1999). A Figura 3.2 mostra como uma classe pode ser modificada de modo a permitir uma configuração dinâmica de atributos.

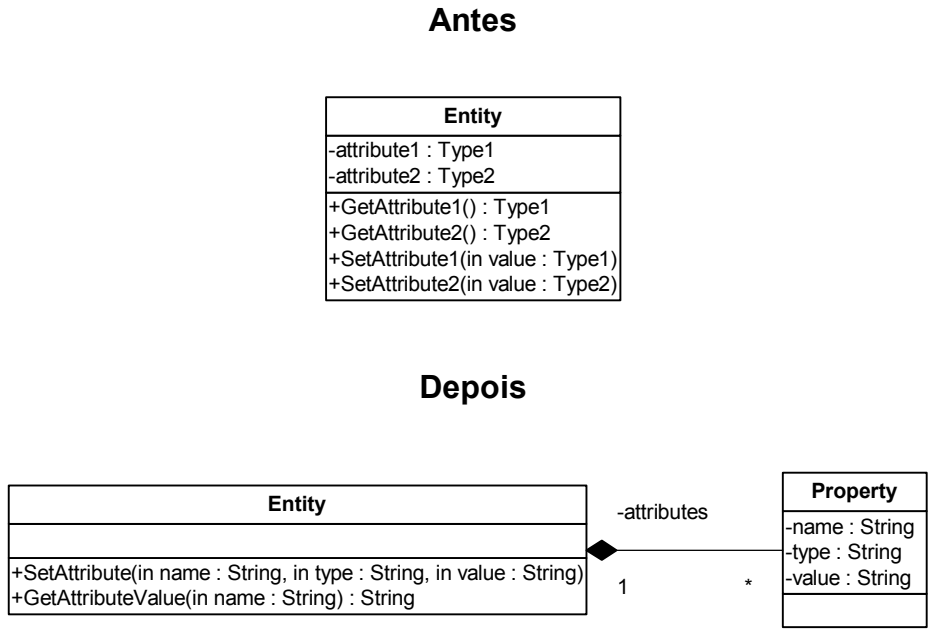

FIGURA 3.2 - Padrão *Property*.
FONTE: adaptada de Yoder, Balaguer e Johnson(2001, p. 51).

Na maioria das arquiteturas de modelos de objetos adaptáveis o padrão *TypeObject* é aplicado duas vezes: antes e depois de se usar o padrão *Property*; ou seja, o *TypeObject* divide o sistema em entidades e tipos de entidades. Entidades têm atributos que podem ser definidos usando-se o padrão *Property*. Cada propriedade (atributo) tem um tipo, chamado de TipoDePropriedade, e cada tipo de entidade pode então, especificar os tipos das propriedades para suas entidades. A Figura 3.3 representa a arquitetura resultante depois de se aplicar esses dois padrões, que pode ser chamada de *TypeSquare.* Ela freqüentemente guarda o nome da propriedade e se o valor da propriedade é um número, uma data, uma *string* etc.

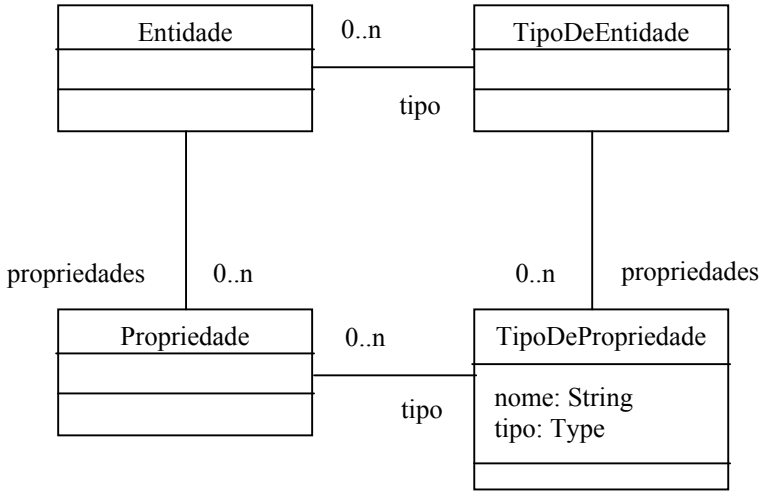

FIGURA 3.3 – Padrão *TypeSquare.*
FONTE: adaptada de Yoder, Balaguer e Johnson(2001, p. 52).

3.1.3 Padrão *Accountability*

Diferentes tipos de objetos usualmente têm diferentes tipos de relacionamentos e diferentes tipos de comportamentos. Um modelo de objetos adaptável precisa de uma maneira para descrever e alterar os relacionamentos e comportamento dos objetos. Ou seja, a aplicação necessita expor os relacionamentos entre os elementos do domínio (entidades), pois cada relacionamento estabelece papéis aos participantes. Papéis podem mudar e o sistema precisa saber quais são os papéis correntes que estão sendo praticados e validados (Yoder e Johnson, 2003).

Atributos são propriedades que se referem a tipos de dados primitivos como, por exemplo, números, *strings*, ou datas. Entidades usualmente têm uma associação unária *(one-way associations)* com seus respectivos atributos. Os relacionamentos, porém, são

propriedades que se referem a outras entidades, e são usualmente associações binárias *(two-way associations).*

Essa distinção, que vem acontecendo desde a modelagem entidade-relacionamento e que foi incorporada às notações mais modernas da modelagem orientada a objetos, faz, normalmente, parte de uma arquitetura de modelo de objetos adaptável. Ela freqüentemente nos leva a ter duas subclasses de propriedades, uma para atributos e uma para relacionamentos.

Assim sendo, uma maneira de separar os atributos dos relacionamentos é usar o padrão *Property* duas vezes, uma para atributos e uma para associações. Uma outra forma é construir duas subclasses de propriedade: atributo e relacionamento. Um relacionamento deve conhecer sua cardinalidade, como ilustra a Figura 3.4.

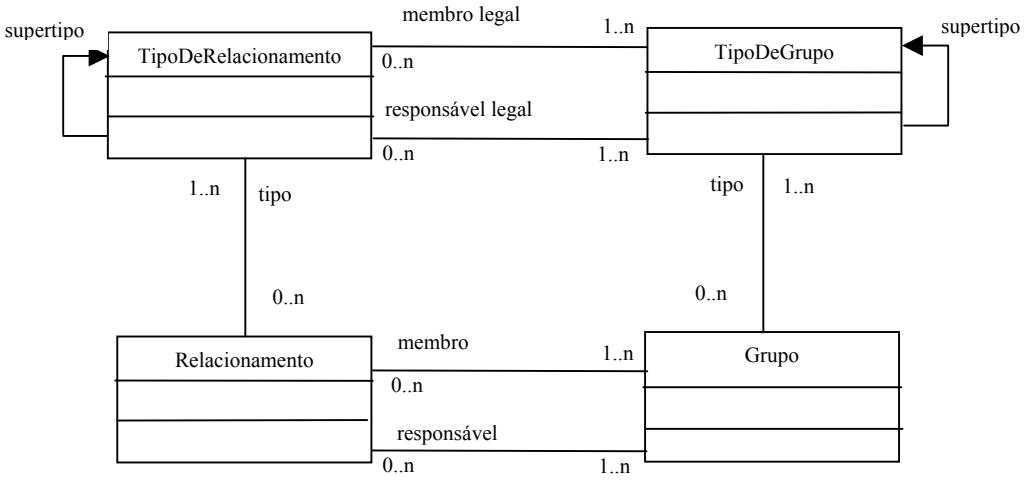

FIGURA 3.4 – Padrão *Accountability*.
FONTE: adaptada de Yoder e Johnson(2003, p. 28).

Através do padrão *Accountability* pode-se manipular muitos relacionamentos entre grupos. Responsabilidades podem ser organizadas em tipos, de forma que cada tipo de responsabilidade conheça os tipos de grupos associados a ela, onde cada tipo de grupo, por sua vez, conhece os membros de seu grupo (Fowler, 1997).

Uma terceira forma de separar atributos de relacionamentos é pelo valor da propriedade. Uma propriedade cujo valor é uma entidade representa um relacionamento, enquanto que propriedades cujo valor são tipos primitivos de dados são atributos (Johnson e Yoder, 2002).

Representar esses relacionamentos como objetos permitem manipulações em tempo de execução, permitindo a um usuário com certos poderes imediatamente adaptar essas entidades-relacionamentos às mudanças de requisitos do domínio (Yoder e Johnson, 2001).

3.1.4 Padrão *Strategy*

As regras de negócio tendem a mudar mais freqüentemente do que os outros objetos do domínio com os quais elas estão associadas. Essas regras são tipicamente implementadas nos métodos de um objeto do domínio. As regras também fazem referência a outros objetos, formando uma teia de dependências difíceis de serem mantidas. Dessa forma, mudar uma regra de negócio pode causar um impacto no conjunto de objetos que dependem daquela regra (Arsanjani, 2000).

Regras de negócio para sistemas orientados a objeto podem ser representadas de muitas formas. Algumas regras irão definir tipos de entidades em um sistema, juntamente com seus atributos. Outras regras podem definir subtipos legais, o que é usualmente feito através de subclasses. Outras regras irão definir os tipos legais dos relacionamentos entre entidades. Essas regras podem definir também restrições básicas como, por exemplo, cardinalidade de relacionamentos e se um certo atributo é requerido ou não. A maioria desses tipos de regras é pertinente à estrutura básica, e já foi mostrado, previamente neste Capítulo (Itens 3.1.1 a 3.1.3), como os modelos de objetos adaptáveis podem adaptar essas regras em tempo de execução.

Contudo, algumas regras não podem ser definidas desta forma. Elas têm uma natureza mais funcional ou procedural. Por exemplo, pode existir uma regra que descreva que tipos de valores legais um atributo pode ter. Ou, pode existir uma regra que especifique

que certas entidades-relacionamentos serão somente legais se as entidades tiverem certos valores, e assim, outras restrições são adicionadas. Essas regras de negócio são, por natureza, mais complexas, e os modelos de objetos adaptáveis usam *Strategies* e *RuleObjects* para tratá-las.

Modelos de objetos adaptáveis usualmente começam com estratégias *(Strategies)* simples que são, na verdade, as funções básicas necessárias para os tipos de entidades *(EntityTypes)*. Essas estratégias podem ser mapeadas para o tipo de entidade através de informações descritivas que são interpretadas em tempo de execução.

Uma estratégia é um objeto que representa um algoritmo. O padrão *Strategy* define uma interface comum para uma família de algoritmos de forma que clientes possam trabalhar com qualquer um deles. Se o comportamento de um objeto é definido por uma ou mais estratégias, então o seu comportamento é fácil de ser mudado (Johnson, 2002).

A intenção do padrão *Strategy* é, então, definir uma família de algoritmos, encapsulá-los, e torná-los intercambiáveis (Chung, Cooper e Yi, 2002).

A Figura 3.5 ilustra a estrutura do padrão *Strategy*. Ela nos mostra que o cliente invoca uma estratégia, que, dependendo do contexto, aciona a estratégia específica que deve ser usada diante daquele contexto. Isso permite que o comportamento dos objetos possa ser alterado em tempo de execução de acordo com o contexto em que ele se encontra.

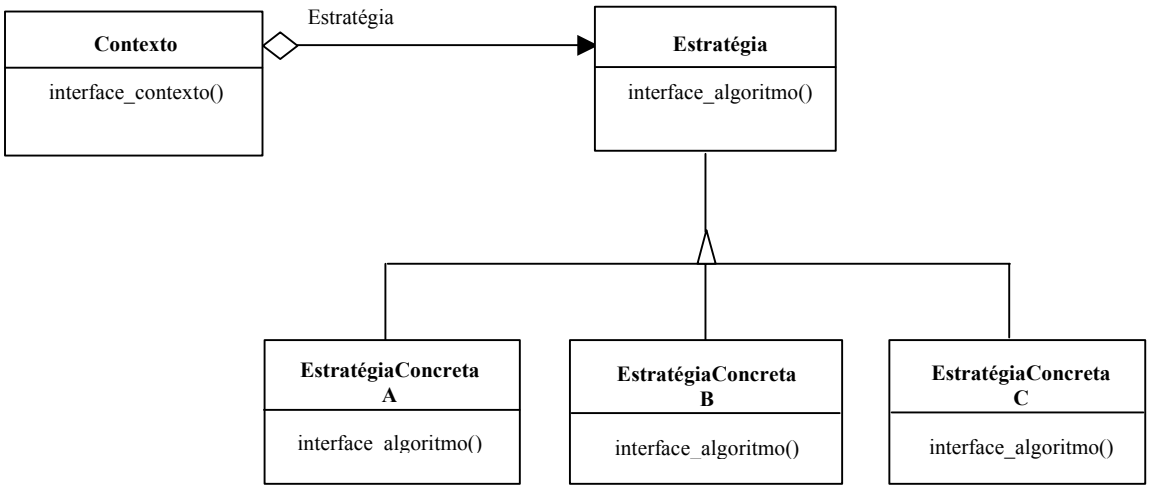

FIGURA 3.5 - Padrão *Strategy*.

FONTE: adaptada de Chung e Cooper(2002, p.257).

Na Figura 3.5 o objeto "Estratégia" declara uma interface comum a todos os algoritmos suportados. O objeto "Contexto" usa essa interface para chamar o algoritmo definido pelo objeto "EstratégiaConcreta". O objeto "EstratégiaConcreta", por sua vez, implementa o algoritmo usando a interface fornecida pelo objeto "Estratégia". O objeto "Contexto" é configurado com um objeto "EstratégiaConcreta" e mantém uma referência para ele.

O objeto "Estratégia" e o objeto "Contexto" interagem para implementar o algoritmo escolhido. O contexto repassa as requisições dos seus clientes para a estratégia. Normalmente há uma família de classes de "EstratégiaConcreta" para que cada cliente possa escolher dentre uma delas (Gamma et al., 1995).

Se regras de negócio mais poderosas são necessárias, as estratégias podem evoluir para se tornar mais complexas. Essas regras podem ser regras primitivas ou uma combinação de regras de negócio através da aplicação do padrão *Composite* (Item 3.1.5). Assim sendo, o padrão *Composite* permite que regras possam ser compostas de outras regras.

Por exemplo, regras que representam predicados são compostas de conjunções e disjunções, regras que representam valores numéricos são compostas de regras de adição e subtração, regras que representam conjuntos são compostas pelas regras união e intersecção. Essas estratégias mais complexas são chamadas de *RuleObjects* (Item 3.1.6).

3.1.5 Padrão *Composite*

O padrão *Composite* compõe objetos em estruturas de árvore para representar hierarquias todo-parte. Ele faz com que clientes tratem objetos individuais e composições de objetos uniformemente, descrevendo como se usar a composição recursivamente de forma que os clientes não necessitem fazer essa distinção (Gamma et al., 1995). A estrutura do padrão *Composite* é mostrada na Figura 3.6.

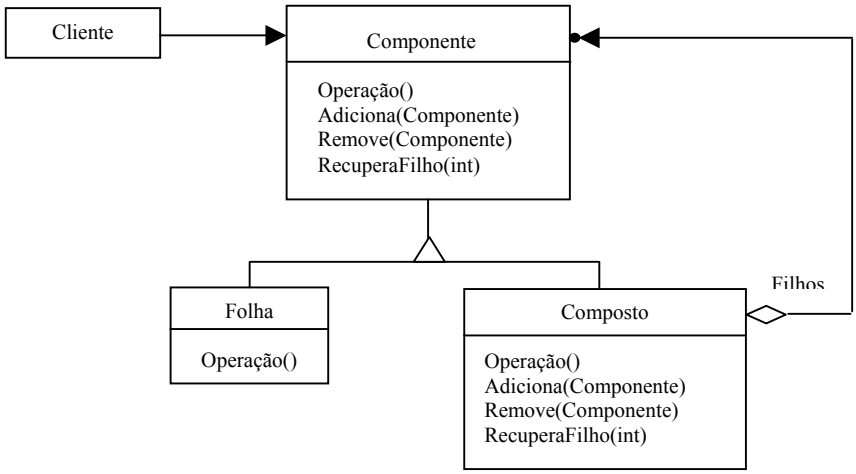

FIGURA 3.6 – Padrão *Composite*.
FONTE: adaptada de Gamma et al.(1995, p.164).

Dessa forma, o objeto "Componente": **(1)** declara uma interface para os objetos na composição, **(2)** implementa o comportamento padrão para a interface comum a todas as classes, **(3)** declara uma interface para o acesso e gerenciamento dos componentes de

seus filhos e **(4)** define uma interface para acessar o pai de um componente na estrutura recursiva e a implementa.

O objeto "Folha" representa um objeto folha na composição, o que significa que ele não tem filhos. Ele define o comportamento para objetos primitivos na composição.

O objeto "Composto" define o comportamento para objetos que têm filhos. Ele guarda os componentes dos filhos e implementa operações relacionadas aos filhos na interface do objeto "Componente".

O objeto "Cliente" manipula objetos na composição através da interface provida por "Componente".

Assim, os clientes usam a interface da classe "Componente" para interagir com objetos na estrutura de composição. Se o receptor é uma "Folha", então a requisição é tratada diretamente. Se o receptor é um "Composto", então ele usualmente repassa as requisições para os componentes de seus filhos, possivelmente realizando operações adicionais antes e depois do repasse.

Dessa forma, o padrão *Composite* define hierarquias de classes que consistem de objetos primitivos e objetos compostos. Objetos primitivos podem ser compostos em objetos mais complexos, que por sua vez, podem também ser compostos, e assim por diante, recursivamente (Gamma et al.,1995).

3.1.6 Padrão *RuleObject*

O padrão *RuleObject* faz com que o projeto e a implementação de processos de negócio computadorizados sejam extensíveis e adaptáveis, sem causar uma cadeia de mudanças, pois as regras que governam os processos de negócio se tornam "plugáveis", ou facilmente intercambiáveis (Arsanjani,2000).

Na verdade, com o padrão *RuleObject* pode-se ter o conceito de estratégias configuráveis, já que novas estratégias podem ser criadas a partir de regras já existentes. A Figura 3.7 mostra a estrutura do padrão *RuleObject.*

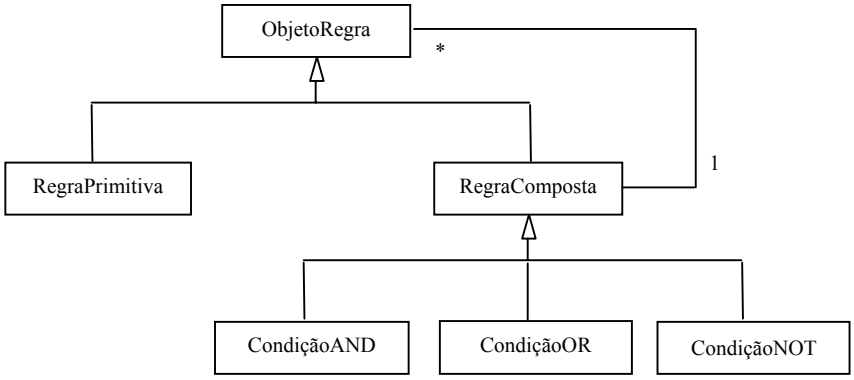

FIGURA 3.7 – Padrão *RuleObject*.

FONTE: adaptada de Yoder e Johnson(2003, p. 17).

Para se construir as regras de negócio em tempo de execução pode-se usar diversas abordagens, desde o uso de *table driven systems*, o uso de uma abordagem orientada à gramática *(Grammar-oriented object design)*, ou até o uso de uma abordagem orientada por *workflow.*

Se as regras de negócio descrevem um *workflow*, a arquitetura *Micro-Workflow* descrita em Manolescu(2000) pode ser usada. Ela descreve classes que representam uma estrutura *workflow* como uma combinação de regras como, por exemplo, repetição, condição, seqüência, bifurcação e regras primitivas. Essas regras podem ser construídas em tempo de execução para representar um processo *workflow* particular.

As regras também podem ser construídas a partir de *Table Driven Systems*, como descrito em Perkins(2000). Nessa abordagem as diferenças nas regras de negócio podem ser parametrizadas e armazenadas em um banco de dados. O sistema sendo executado pode interpretar essas mudanças da tabela do banco de dados ou uma função apropriada

pode ser chamada com os valores modificados no banco de dados. Isso pode ser feito através de *triggers* ou de *stored procedures.*

Pode-se usar também uma abordagem orientada à gramática, que é conhecida como *Grammar-oriented Object Design (GOOD).* Essa abordagem aplica uma combinação de princípios de linguagens específicas de domínio para a modelagem do negócio e os mapeia para arquiteturas de software baseadas em componentes. Mais detalhes sobre essa abordagem para a construção de regras podem ser encontrados em Arsanjani(2001).

Prover essa construção de regras mais complexas é, geralmente, a parte mais difícil do projeto de modelos de objetos adaptáveis (Yoder, 2001).

3.2 Arquitetura Comum do Modelo de Objetos Adaptáveis

Os modelos de objetos adaptáveis são usualmente construídos através da aplicação de um ou mais padrões citados anteriormente. A Figura 3.8 apresenta a arquitetura obtida pela combinação dos padrões *Type-Square*, *Accountability* e *RuleObjects*.

Como ilustrado na figura, a arquitetura de modelos de objetos adaptáveis pode ser dividida em dois níveis: **(1)** o nível operacional, onde estão as classes que guardam informações relacionadas exclusivamente ao domínio do problema e **(2)** o nível de conhecimento, onde estão as classes que guardam informações a respeito das classes do nível operacional.

As classes envolvidas pela elipse onde se lê “Classes com atributos e relacionamentos”, representam as classes responsáveis por guardar os tipos de entidades, tipos de relacionamentos e tipos de atributos existentes no domínio.

As classes envolvidas pela elipse onde se lê “Comportamento” representam as classes que especificam o comportamento do sistema através de regras.

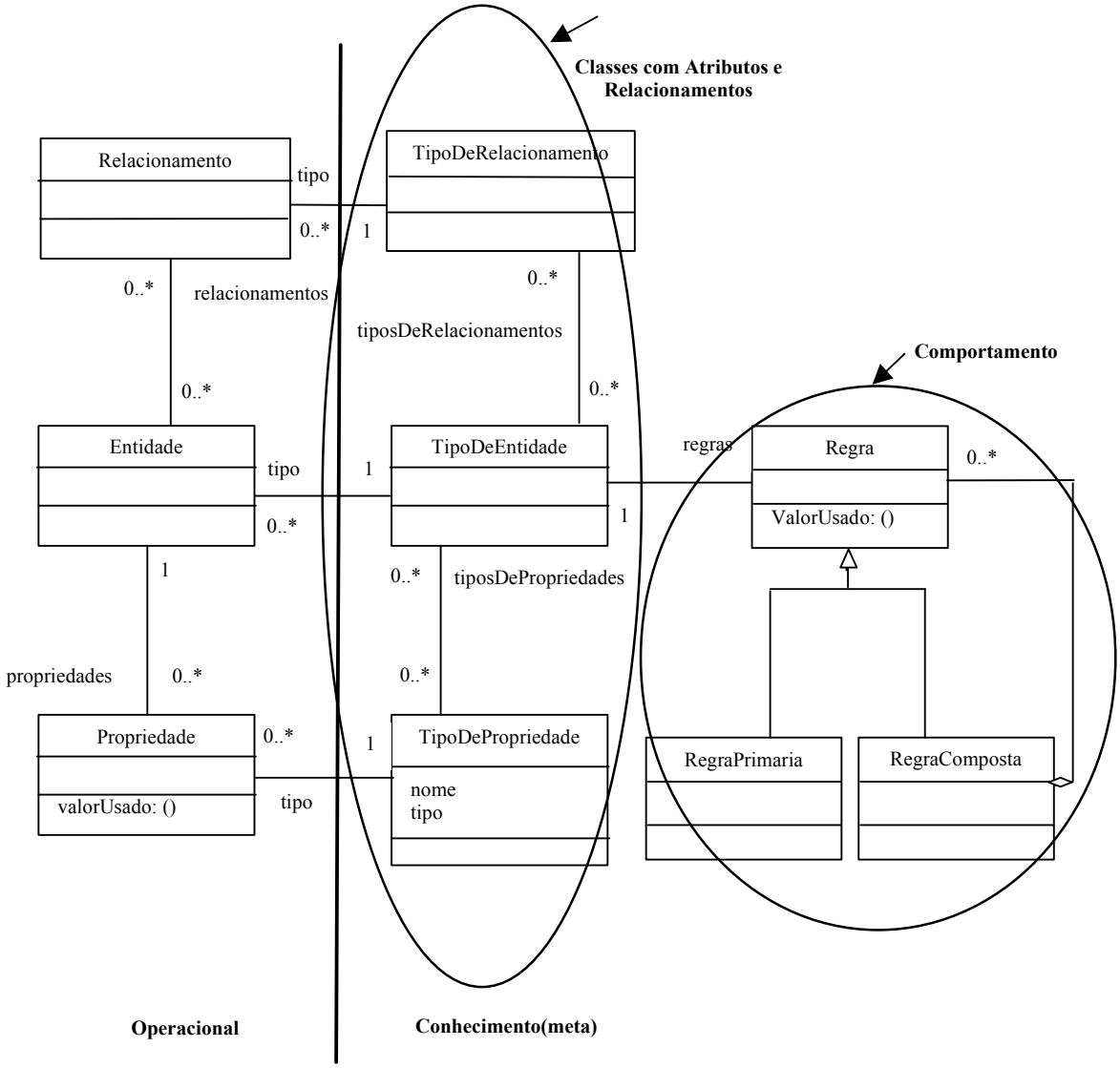

FIGURA 3.8 - Arquitetura comum dos modelos de objetos adaptáveis.
FONTE: adaptada de Yoder e Johnson(2003, p. 17).

Deve-se notar que, apesar da arquitetura DOM poder ser obtida através da aplicação dos padrões, isto não significa que ela constitua necessariamente um *framework* para a construção de modelos de objetos adaptáveis.

Na realidade, na literatura pesquisada cada modelo de objetos adaptável é um *framework* de algum tipo de domínio. Ou seja, não existe ainda um *framework* genérico para construí-los. Um *framework* genérico, para a construção de *TypeObjects*, *Properties* e seus relacionamentos, pode provavelmente ser construído, porém, esses

elementos não são fáceis de serem definidos e a dificuldade está geralmente associada às regras descritas pela linguagem do domínio. Isso é algo que normalmente é bem específico do domínio e varia de acordo com cada um deles.

3.3 Metadados

Dois fatores contribuem para a alta flexibilidade dos sistemas DOM. Primeiro, a parte genérica, ou seja, o núcleo comum a uma ampla faixa de problemas, é definida no próprio código. Segundo, a parte variável é definida como metadados de forma que os usuários possam modificá-los para adaptar o sistema para atender suas necessidades.

Metadados podem ser descritos dizendo que se algo vai variar de uma maneira previsível, é melhor armazenar a descrição da variação em um banco de dados para facilitar futuras mudanças. Em outras palavras, se algo tem tendência de mudar, uma boa solução é projetar um sistema de forma a incorporar mais facilmente esta mudança. O problema é que pode ser difícil descobrir o que muda. E mesmo quando se sabe pode ser difícil descrever esta mudança na base de dados. O código é complexo,e pode ser muito difícil substituí-lo por dados sem fazer estes dados tão complexos quanto ele. Mas quando se consegue criá-los da maneira correta, os metadados podem, a longo prazo, compensar os custos iniciais e diminuir bastante a necessidade de manutenção (Johnson e Wolf, 1998).

Desta forma, a abstração da parte genérica e a identificação das partes mais sujeitas a modificações exigem projetistas com grande experiência. Normalmente isto é alcançado somente após um certo número de iterações. Esta é a razão que faz os sistemas DOM´s tão difíceis de serem construídos.

Os objetos dos padrões TypeObject, Property e Strategy, que normalmente são os blocos de construção da parte genérica da arquitetura DOM, são montados a partir de dados de configuração, ou metadados. Portanto, modificações nos metadados alteram o modelo de objetos e conseqüentemente o comportamento do sistema.

Em geral, os metadados para a descrição das regras de negócio e do modelo de objetos são interpretados em dois momentos: **(1)** quando os objetos são construídos, isto é, quando o modelo de objetos é instanciado e **(2)** durante a interpretação das regras de negócio em tempo de execução (Johnson e Yoder, 2002).

Se um banco de dados orientado a objetos está sendo usado para armazenar os metadados, os tipos dos objetos e relacionamentos podem ser construídos pela simples instanciação das classes genéricas (Entidade, TipoDeEntidade, etc.) com os metadados armazenados no banco.

Caso contrário, os metadados deverão ser lidos do banco de dados para que se possa construir esses objetos. Nesta construção podem ser usados os padrões *Interpreter* e *Builder* (Manolescu e Johnson, 1999 ; Johnson e Yoder, 2002). Os padrões *Builder* (Item 3.3.1) e *Interpreter* (Item 3.3.2) são comumente usados para, respectivamente, a construção das estruturas do meta-modelo e interpretação de suas regras.

Um outro momento onde os metadados têm que ser interpretados é quando acontecem mudanças no domínio, por exemplo, novos tipos de objetos são criados com seus respectivos atributos e operações. Quando isso acontece, essas mudanças têm que ser refletidas na aplicação sendo executada, e para isso o repositório de metadados deve ser atualizado. Devido aos vários relacionamentos existentes entre os elementos dos metadados, o padrão *Observer* (Item 3.3.3) pode ser utilizado para indicar quais elementos devem ser atualizados, como conseqüência de mudanças em outros elementos, de modo a manter a integridade do meta-modelo.

Quando as novas operações criadas são simples estratégias, os metadados descrevem o método que precisa ser invocado. Essas estratégias podem ser conectadas ao objeto apropriado durante a instanciação de tipos.

Contudo, se forem necessárias regras mais dinâmicas, uma linguagem específica de domínio deve ser projetada através de *RuleObjects*. Por exemplo, regras primitivas podem ser definidas e compostas em objetos que formem uma estrutura de árvore que é interpretada em tempo de execução.

Às vezes, as necessidades do sistema se tornam tão complexas que a única solução é criar uma linguagem de regras usando gramática, árvores de sintaxe abstrata, linguagens de restrições, e interpretadores complexos. Deve-se lembrar que a linguagem deve ser evoluída se houver realmente necessidade, pois muitas vezes a nova linguagem na realidade acaba tornando a manutenção e a evolução da aplicação mais difíceis do que se essas regras fossem simplesmente modeladas através de uma linguagem de programação normal (Yoder, Balaguer e Johnson, 2001).

Regras e gramática requerem habilidade para escrever e manter. Desta forma é fundamental que esta linguagem seja colocada de uma forma que seja o mais natural possível para os usuários finais. Além disso, ferramentas especiais e linguagens visuais podem ser criadas para auxiliar os usuários finais nesta tarefa. (Manolescu e Johnson, 1999) (Johnson e Yoder, 2002)

3.3.1 Construção do Modelo Utilizando Metadados

A ativação de um sistema baseado na arquitetura do modelo DOM requer a instanciação de seus objetos com base nos metadados que representam os objetos do mundo real. Esta construção deve levar em conta os relacionamentos entre os objetos criados, fazendo as devidas conexões conforme descrições contidas nos metadados, e a possibilidade de mudança dos metadados em tempo de execução.

O padrão *Builder* é utilizado para separar a construção de um objeto de sua representação, permitindo que a sua representação interna possa ser modificada facilmente, sem que essas alterações afetem o sistema globalmente, além de isolar seu código de construção e representação, aumentando a modularidade. Ele é utilizado quando a forma de se compor um objeto deve ser independente de suas partes e de como essas se conectam, ou quando o objeto pode ter diferentes configurações, que só serão informadas em tempo de execução. A Figura 3.9 ilustra a aplicação do padrão *Builder*.

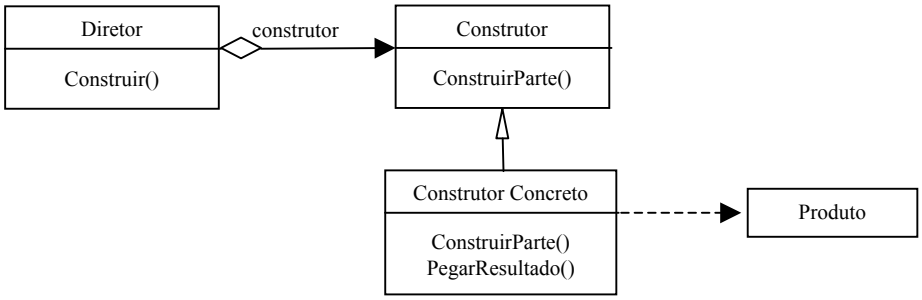

FIGURA 3.9 – Padrão *Builder*.
FONTE: adaptada de Gamma et al.(1995, p. 98).

O “Construtor” especifica uma interface abstrata para a criação das partes de um objeto “Produto”.

O “ConstrutorConcreto” constrói e monta as partes de um produto implementando a interface “Construtor”. Ele define e guarda a representação que criou, provendo uma interface para a recuperação do produto.

O “Diretor” constrói um objeto usando a interface “Construtor”. O “Produto” representa o objeto complexo em construção.

3.3.2 Interpretação das Regras

No modelo DOM, as regras assim como os próprios objetos são descritos através de metadados. Após a construção do modelo a partir dos metadados uma regra se transforma em uma hierarquia de objetos que devem ser interpretados quando aquela regra for ativada. A seqüência de ativação dos objetos da hierarquia depende do contexto corrente dos objetos do modelo.

O padrão *Interpreter* descreve como representar sentenças em uma linguagem e interpretar essas sentenças. Um dos benefícios do padrão *Interpreter* é que ele permite que se adicione novas formas de se interpretar expressões (Nakamura e Johnson, 1998).

Assim sendo, ele define a representação da gramática de uma linguagem, juntamente com seu interpretador. A Figura 3.10 mostra a estrutura do padrão *Interpreter*.

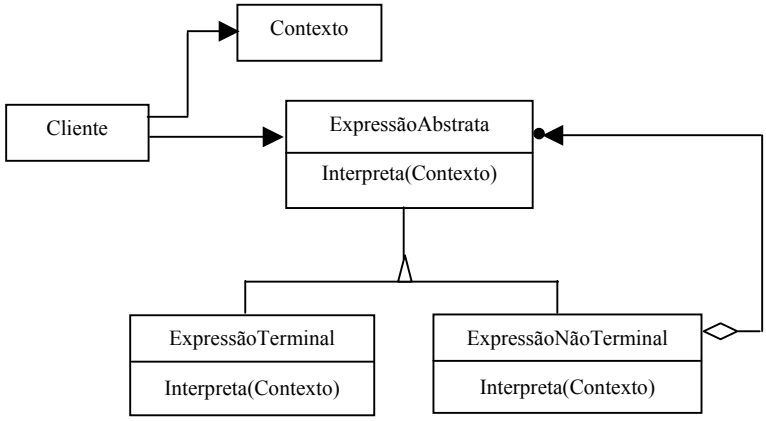

FIGURA 3.10 – O padrão *Interpreter*.
FONTE: adaptada de Gamma et al.(1995, p. 245).

Um objeto "ExpressãoAbstrata" declara uma operação "Interpreta" que é comum a todos os nós na árvore de sintaxe abstrata.

Uma "ExpressãoTerminal" implementa uma operação "Interpreta" associada a símbolos terminais na gramática. É requerida uma instância desse objeto para cada símbolo terminal em uma seqüência.

Uma "ExpressãoNãoTerminal" representa, por exemplo, uma expressão de repetição, seqüência ou condição em uma gramática. Uma classe desse tipo é requerida para cada regra $R ::= R_1R_2...R_n$ na gramática, e são necessárias variáveis de instância do tipo "Expressão Abstrata" para cada símbolo desde R_1 até R_n para implementar uma operação "Interpreta" para símbolos não terminais na gramática. Essa operação tipicamente chama a si mesmo recursivamente nas variáveis representando R_1 até R_n.

O "Contexto" contém informações globais ao interpretador. O "Cliente" constrói uma árvore de sintaxe abstrata representando uma sentença particular na linguagem que a gramática define. A árvore de sintaxe abstrata é montada das instâncias das classes "ExpressãoTerminal" e "ExpressãoNãoTerminal".

Assim sendo, o “Cliente” constrói a sentença como uma árvore abstrata de instâncias de “ExpressãoTerminal” e “ExpressãoNãoTerminal”, e inicializa o “Contexto” invocando a operação “Interpreta”. Cada nó “ExpressãoNãoTerminal” define “Interpreta” em termos de “Interpreta” em cada sub-expressão. A operação “Interpreta” para cada nó usa o “Contexto” para armazenar e acessar o estado do interpretador (Gamma et al., 1995).

3.3.3 Mudança dos Metadados

Os objetos do mundo real são representados em um sistema DOM através de um conjunto de objetos que se colaboram de modo a representar o estado, os relacionamentos e o comportamento dos objetos reais. Como os objetos do modelo DOM são construídos a partir dos metadados, qualquer mudança dos metadados acarreta a necessidade de mudança dos objetos diretamente relacionados àqueles metadados e também de todos os outros objetos que lhe são de alguma forma dependentes.

Um efeito colateral do particionamento de um sistema em uma coleção de classes cooperantes é a necessidade de manter consistência entre os objetos relacionados. Para se garantir a consistência sem reduzir a reusabilidade deve-se evitar que as classes cooperantes tenham um alto grau de acoplamento. Uma solução para este tipo de problema é o padrão *Observer* (Gamma et al., 1995) o qual descreve como estabelecer estes relacionamentos. Ele é utilizado em uma dependência 1-N entre objetos de modo que quando um objeto muda seu estado, todos os objetos dependentes sejam notificados. Os objetos chaves são os *subject* e *observer*. Um *subject* pode ter qualquer número de *observers* dependentes dele. Todos os *observers* são notificados sempre que o *subject* sofre uma mudança de estado. A Figura 3.11 ilustra a estrutura do padrão *Observer*.

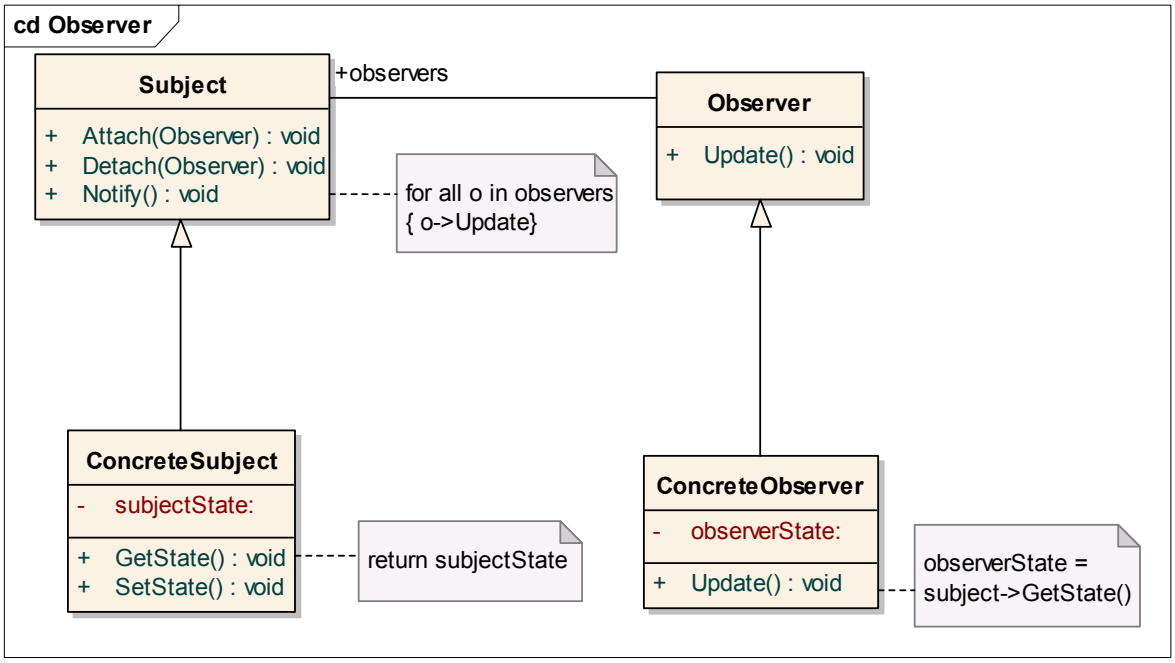

FIGURA 3.11 – O padrão *Observer*.

FONTE: adaptada de Gamma et al.(1995, p. 294).

Um objeto *Subject* fornece uma interface para que objetos *Observers* possam se registrar (*Attach*) para receber notificações de suas mudanças ou cancelar este registro (*Detach*). Para se registrar os objetos *Observers* precisam ter uma interface padrão para receber as notificações. Esta interface é definida através da classe *Observer*. Um objeto *subject* concreto, representado pela classe *ConcreteSubject,* armazena o estado de interesse para os *observers* e envia uma notificação sempre que este estado mudar. Por outro lado os objetos *observers*, representados através da classe *ConcreteObserver*, mantêm uma referencia para o objeto *ConcreteSubject*, armazenam o estado que deveria permanecer consistente com o do *subject* e implementa a interface de atualização da classe *Observer* de modo manter a consistência.

3.4 Técnicas de Reflexão

O modelo DOM propõe que um sistema possa ser adaptado utilizando-se de uma combinação adequada de objetos primitivos de forma a compor ou modificar objetos

mais complexos. Como explicado no Item 3.3.2, a interpretação de uma regra gera um processo recursivo que é finalizado sempre que se atinge uma expressão terminal. No nosso caso isto equivale a executar uma regra de um objeto primitivo. Ativar as regras de objetos primitivos com base nas descrições dos metadados requer a utilização de técnicas reflexivas.

Reflexão computacional é a atividade executada por um sistema computacional quando realizando computação sobre sua própria computação e possivelmente afetando-[a] Neste sentido, um sistema reflexivo incorpora estruturas representando aspectos de si próprio e pode até mesmo modificar estas estruturas e o seu próprio comportamento (Maes, 1987).

A programação reflexiva nos leva a um programa que usa reflexão em tempo de execução para alterar valores de campos na classe alvo (Fowler, 2002). Essas técnicas estão relacionadas com os modelos de objetos adaptáveis no sentido de que estas também definem um nível de abstração mais alta onde os metadados são mantidos. Porém, as técnicas de reflexão não estabelecem que o metadado deve ser armazenado em um banco de dados. Desta forma, pode-se dizer que os modelos de objetos adaptáveis usam técnicas reflexivas.

O núcleo de reflexão da linguagem *Java* fornece uma pequena e segura API que suporta introspecção sobre as classes e objetos na corrente máquina virtual *Java*. Se permitido pelas políticas de segurança, a API pode ser usada para: construir novas instâncias de classes, acessar e modificar campos de objetos e classes, e invocar métodos sobre objetos e classes.

Esta API é utilizada por duas categorias de aplicações. A primeira categoria é composta de aplicações que precisam descobrir e usar todos os membros públicos de um objeto alvo em tempo de execução. Estas aplicações requerem acesso em tempo de execução a todos os campos públicos, métodos e construtores de um objeto. Exemplos nesta categoria são serviços tais como *Java Beans*, e ferramentas como inspetores de objetos. A segunda categoria consiste de aplicações sofisticadas que precisam descobrir e usar os membros declarados por uma classe. Estas aplicações precisam ter acesso em tempo de

execução à implementação de uma classe. Exemplos desta categoria são ferramentas de desenvolvimentos, tais como interpretadores, inspetores e serviços em tempo de execução.

Instâncias das classes *Field*, *Method* and *Constructor* implementam a interface *Member* e podem ser criadas somente pela máquina virtual do *Java*, a qual as utiliza para manipular os demais objetos.

Um objeto *Field* representa um campo refletido que pode ser uma variável da classe (campo estático) ou uma variável da instância. Métodos da classe *Field* são usados para obter o tipo de um campo e também para obter ou definir seu valor.

Um objeto *Method* representa um método refletido que pode ser abstrato, de uma instância ou de uma classe (estático). Métodos da classe *Method* são usados para obter os tipos dos parâmetros, o tipo do retorno e os tipos de exceções de um método. Em adição, o método "*invoke*" é usado para invocar o método em objeto alvo.

Um objeto *Constructor* representa um construtor refletido. Métodos da classe *Constructor* são usados para obter os tipos dos parâmetros formais e tipos das exceções de um construtor. Em adição o método *newInstance* da classe *Constructor* é usado para criar e inicializar uma nova instância de uma classe.

O gerenciamento de segurança do *Java* controla o acesso da API de reflexão aos objetos. Código privilegiado pode ter acesso completo, passando por cima das regras de controle de acesso padrão da linguagem, usando o método *setAccessible*. Este método esta disponível nas classes *Field*, *Method* e *Constructor*.

CAPÍTULO 4

SOLUÇÃO PARA UTILIZAÇÃO DO MODELO ADAPTATIVO NOS SISTEMAS DE CONTROLE DE SATÉLITES

Como visto no Capítulo 2, o grupo de desenvolvimento de sistemas de controle para os satélites do INPE, através do sistema de controle para os micros satélites científicos, conseguiu construir um importante *framework* que minimiza e facilita mudanças para atender requisitos específicos de futuras missões espaciais. No desenvolvimento deste *framework*, os pontos mais sujeitos a mudanças foram identificados e levados em conta no projeto de modo a facilitar a criação de classes para atender necessidades específicas. Porém, por menor que seja a mudança ela exige conhecimento do *framework* e codificação das novas classes.

Este trabalho esta propondo que para a realização destas mudanças o *framework* aceite a conexão com um sistema de objetos adaptáveis. Deste modo a responsabilidade por estas mudanças pode ser transferida para a equipe de controle dos satélites, desde que as condições abaixo sejam satisfeitas:

- Os usuários responsáveis pelas mudanças devem ter um grande conhecimento no controle de satélites como, por exemplo, é o caso dos engenheiros responsáveis pela operação;

- Deve ser fornecida uma ferramenta de edição, para auxiliar estes usuários, que seja o mais amigável possível e que minimize a inserção de erros;

- Para realizar as mudanças os usuários terão acesso às regras de negócio oferecidas por objetos básicos do domínio do problema de controle de satélites, ou seja, deve ser oferecida uma linguagem de alto nível;

- A linguagem e/ou o conjunto de objetos oferecidos devem ser ricos o suficiente para maximizar a possibilidade de que as mudanças sejam realizadas pelo próprio usuário.

Além disso, como no futuro pode ocorrer a evolução do *framework* de controle de satélites e a identificação de novos pontos de mudança, o modelo de objetos adaptáveis a ser criado não deve estar restrito aos tipos de objetos associados aos pontos de mudança conhecidos atualmente. Isto é, dentro do possível, ele também deve ser um *framework.*

A arquitetura da solução proposta está ilustrada na Figura 4.1 e é composta dos metadados armazenados em um banco de dados; um *framework* DOM construído com base na arquitetura comum descrita no Item 3.2; uma ferramenta de edição dos metadados; e uma interface externa para permitir que o sistema de controle possa acessar os objetos do *framework* DOM.

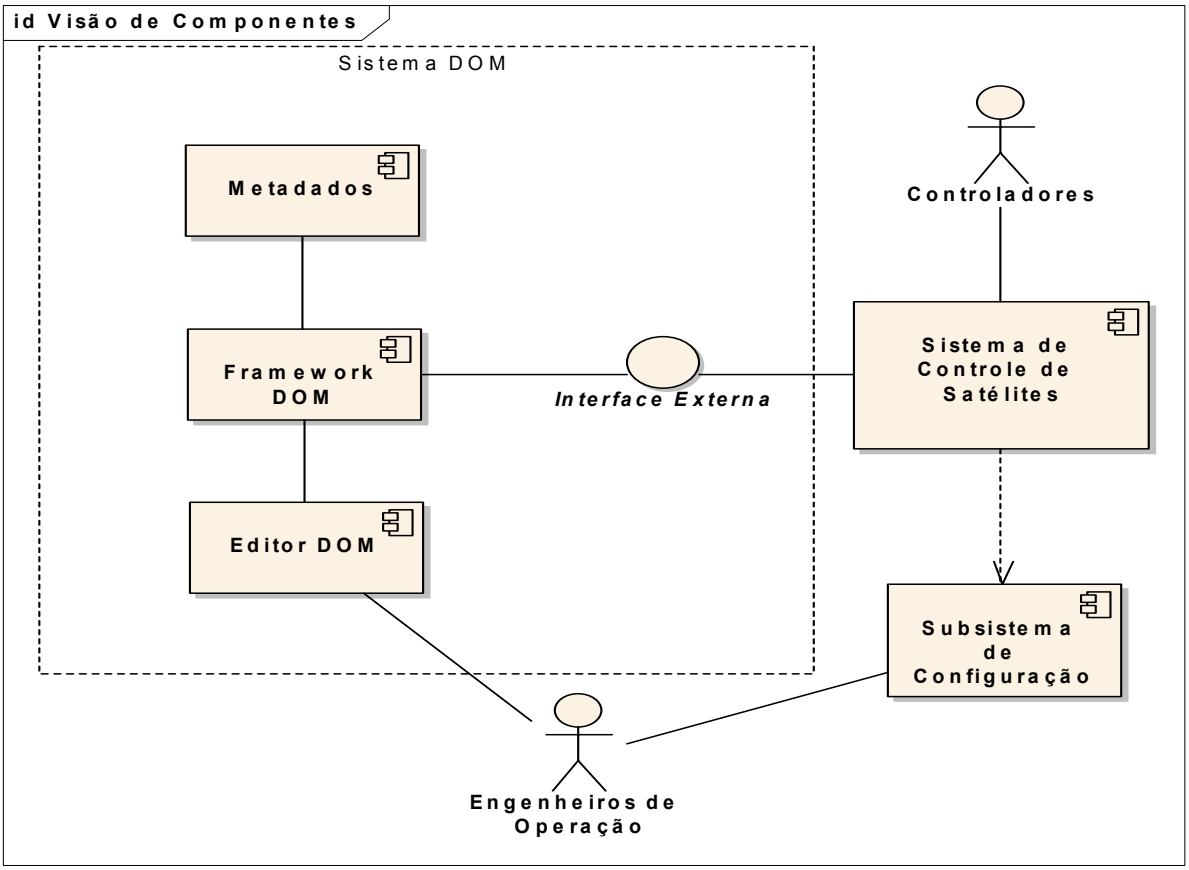

FIGURA 4.1 – Arquitetura da solução proposta.

Para atender novos requisitos de operação os engenheiros devem configurar tanto os metadados, que descrevem os objetos adaptativos, quanto o *software* do Sistema de Controle de Satélites. Para facilitar a configuração dos metadados eles terão à

disposição uma ferramenta de edição que faz uso do *framework* DOM para atualizar os metadados e através da qual será possível criar novos objetos e/ou tipos de objetos. Por outro lado eles também deverão configurar o software de controle para que o mesmo tenha acesso aos objetos adaptativos via a interface externa disponibilizada pelo *framework* DOM.

Uma vez realizadas estas duas configurações os controladores dos satélites terão à sua disposição uma nova versão do sistema de controle que contempla as mudanças feitas no modelo de objetos adaptativos para atender os novos requisitos de operação. Para acessar os objetos definidos no sistema adaptativo o sistema de controle usa a interface existente entre os dois sistemas.

O maior desafio encontrado no estudo da arquitetura proposta foi a definição de um modelo para representar as regras de negócio. A solução encontrada propõe uma nova abordagem para implementação do modelo DOM, segundo a qual o padrão *Property* é utilizado também para modelar as regras dos objetos criados pelo usuário. Ao final se chegou a um modelo no qual Objetos, Tipos de Objetos e Regras são representados por uma árvore de entidades e suas propriedades. Esta concepção resultou em uma menor complexidade na modelagem dos metadados e do *framework* DOM.

No restante deste Capítulo, serão apresentados com mais detalhes: a abordagem para a implementação do modelo DOM (Item 4.1); a especificação de casos de uso para definição de um sistema segundo a arquitetura proposta (Item 4.2); as classes que compõem o *framework* gerado a partir destes casos de uso e da abordagem de implementação do modelo DOM (Item 4.3); os metadados que foram definidos (Item 4.4), a interface externa com o sistema de controle de satélites (Item 4.5) e uma visão da dinâmica do funcionamento das classes do *framework* (Item 4.6). Em seguida, no Capítulo 5, será mostrado um protótipo da arquitetura proposta e um exemplo da utilização deste protótipo pelo sistema de controle de satélites, apresentado no Capítulo 2.

4.1 Abordagem para Implementação do Modelo DOM

Durante a primeira parte do estudo do modelo de objetos adaptativos, finalizada com a apresentação da proposta de dissertação, foi feita a implementação de um protótipo de estudo do modelo DOM e notou-se que a maior dificuldade de sua implementação estava na definição de regras de negócio configuráveis através da base de dados. Esta definição podia se tornar tão mais complexa quanto maior for a flexibilidade a ser oferecida aos usuários finais.

Além disso, na literatura pesquisada se encontrou muitas referências sobre a implementação dos padrões *Type-Object* and *Properties,* mas muito pouco sobre o padrão *Strategy*. O princípio para a configuração das regras encontrado na literatura é que os usuários devem poder combinar regras básicas pré-definidas (funções primitivas) para construir uma regra de maior complexidade. Mas, para oferecer maior flexibilidade, pelo menos dois aspectos importantes devem ser também considerados na configuração das regras: a) acesso ao conjunto de entidades que compõem o modelo (contexto); b) tratamento de condições.

Como conseqüência destas constatações o foco desta pesquisa foi dirigido em primeiro lugar para alcançar um meta-modelo para as regras.

Fazendo-se uma analogia com as linguagens de programação existentes, uma regra pode, por exemplo, ter um tipo de retorno e usar parâmetros para que um dado contexto de Entidades possa ser utilizado e/ou modificado pela combinação de funções primitivas que compõe o procedimento a ser executado. Em relação às condições, este procedimento deve comportar pelo menos, a execução condicional e iterativa de comandos. Um comando, por sua vez, pode ser uma função primitiva ou um dos tipos de seqüência.

A regra em si é então uma entidade complexa. Ela pode ser vista como uma entidade que tem como propriedades: a conexão com o contexto das entidades externas, que pode ser necessário para a execução da mesma, e o procedimento a ser executado. Cada uma

destas propriedades também é bastante complexa e pode ser vista como composta de uma ou mais entidades.

Desta forma, uma regra pode ser modelada como uma entidade que tem propriedades, tais como tipo de retorno, parâmetros e procedimento. Um procedimento pode ser uma entidade que tem uma seqüência de propriedades do tipo comando. Uma propriedade do tipo comando pode estar associada a um dos seguintes tipos de propriedades: regra de um objeto primitivo, regra de um outro tipo de entidade definido pelo usuário, e seqüências condicional e iterativa de comandos.

Através da generalização desta idéia o modelo de uma regra foi então concebido como sendo uma árvore de entidades e propriedades. Qualquer regra pode ser instanciada com base neste modelo. A estrutura desta árvore está ilustrada na Figura 4.2.

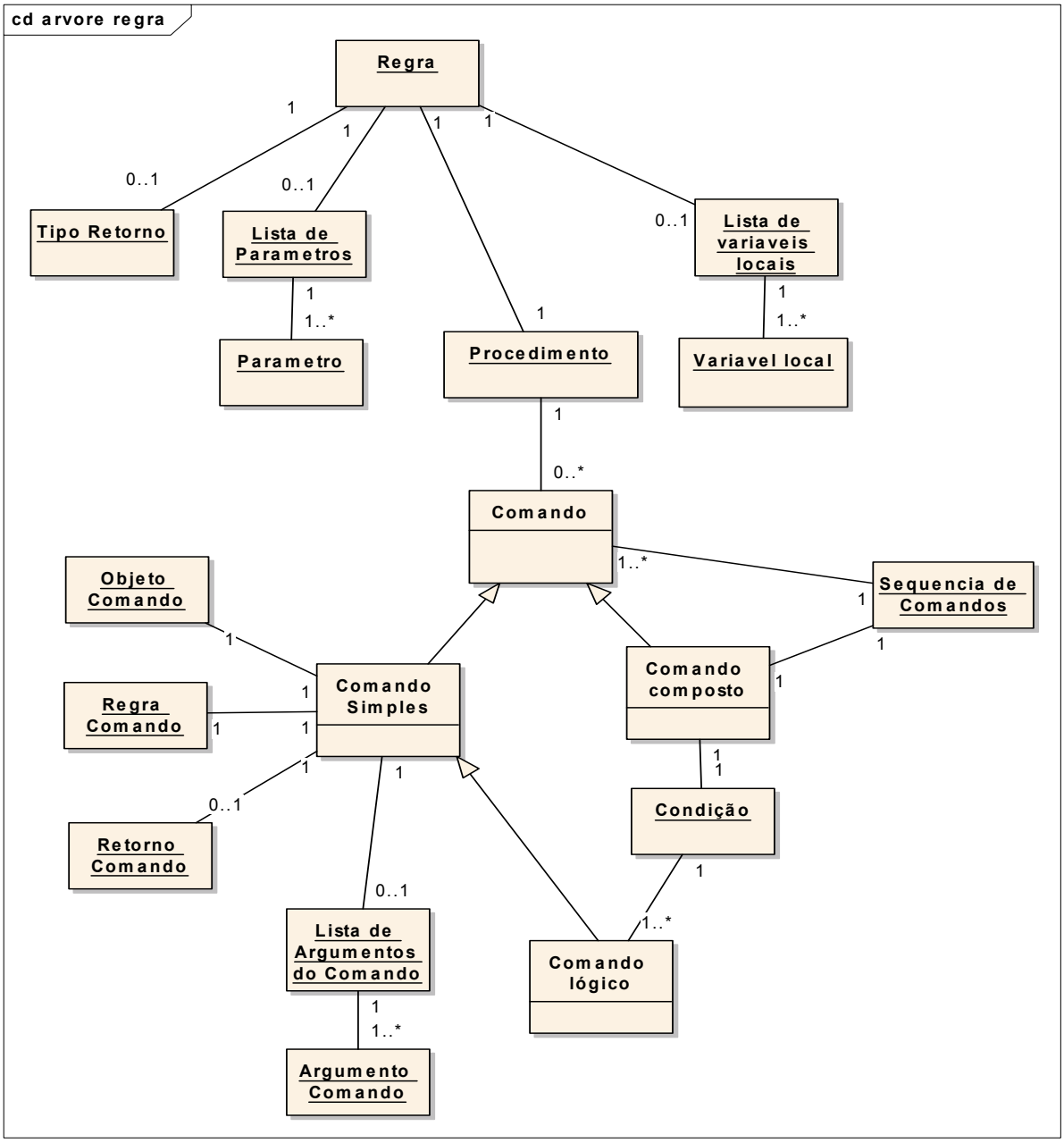

FIGURA 4.2 – Modelo dos tipos de entidade de uma regra.

Assim, uma entidade do tipo “Regra” é composta dos seguintes tipos de propriedades:

- Tipo Retorno – é parte do contexto e representa o tipo de objeto do valor que será retornado como resultado da execução da regra;

- Lista de Parâmetros – representa o conjunto de possíveis parâmetros que sejam necessários para passar um dado contexto de objetos para ser acessado / modificado pela regra;

- Lista de Variáveis Locais – representa o conjunto de possíveis variáveis locais necessárias para execução da regra;

- Procedimento – representa a seqüência de comandos que devem ser executados pela regra.

Cada parâmetro contém como atributos a sua identificação e o tipo de objeto representado por ele. De modo semelhante cada variável local contem os atributos de identificação e o tipo de objeto representado.

A entidade comando foi concebida com base no padrão *RuleObject,* onde cada comando pode ser classificado como sendo de um dos seguintes tipos:

- Comando simples – regra de um objeto primitivo ou regra de um outro tipo de objeto definido pelo usuário ou;

- Comando composto – seqüência iterativa ou condicional de comandos.

Os comandos simples são compostos das seguintes propriedades:

- Objeto Comando – representa o objeto que recebe a mensagem (comando);

- Regra Comando– identifica qual das regras de Objeto Comando será ativada;

- Retorno Comando – representa o objeto que será atualizado com o valor retornado como resultado da execução do comando;

- Lista de Argumentos do Comando – representa o conjunto de argumentos (contexto de objetos externos) que devem ser utilizados durante a execução do comando.

Cada Argumento Comando contém os atributos:

- identificação do argumento
- valor do argumento ou referencia para um objeto representado pelo argumento.

Os comandos compostos possuem as seguintes propriedades:

- Condição – que representa a seqüência de comandos lógicos que compõem a condição
- Seqüência de Comandos – que representa a seqüência de comandos a serem executados se (seqüência condicional) ou enquanto (seqüência iterativa) a condição for válida

O padrão *Type-Square* é utilizado na relação da especificação do contexto de tipos de objetos aceitos por uma regra (Tipo Retorno e Parâmetros) com:

- Os objetos reais que são utilizados pela regra durante sua execução;
- especificação do contexto de um comando simples.

Em seguida foi observado que, dada sua generalidade, a concepção de uma árvore de entidades e propriedades poderia ser uma boa solução não apenas para a representação de uma regra mas para a implementação de toda a arquitetura do Modelo de Objetos Adaptáveis descrita no Capítulo 3. A Figura 4.3 exibe a estrutura concebida para o modelo completo.

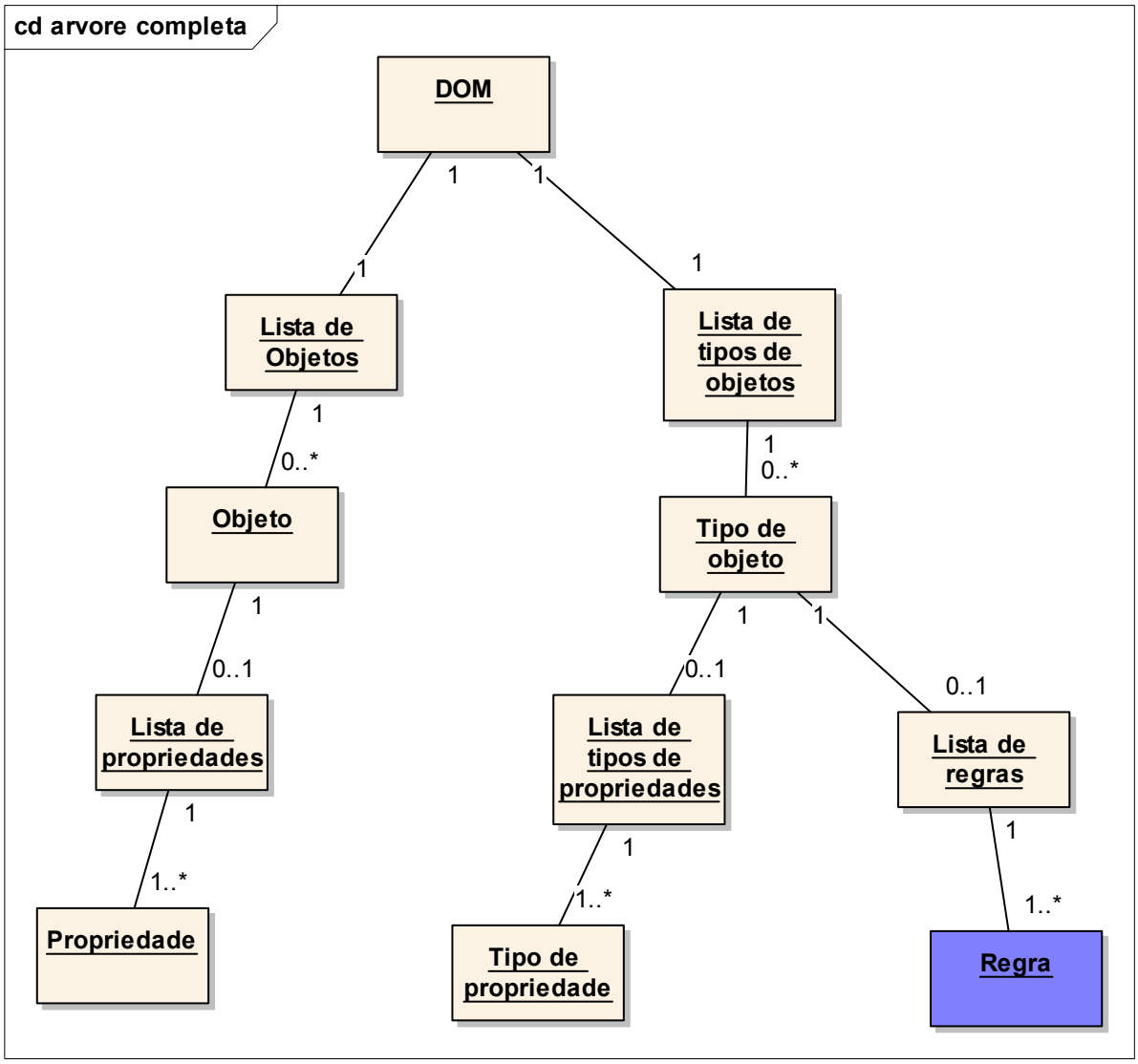

FIGURA 4.3 – Estrutura de representação do modelo DOM.

A raiz da árvore representada pelo modelo se divide em duas listas: Tipos de Objetos e Objetos.

Cada Tipo de Objeto contém duas sub-listas: Lista de Tipos de Propriedades (atributos) e Lista de Regras (serviços). Cada Tipo de Propriedade contem a identificação da Propriedade e uma referência para o tipo de objeto por ela representada. A representação das regras segue a estrutura ilustrada na Figura 4.2.

De acordo com o padrão *Type-Square* cada Objeto está associado a um dos Tipos de Objetos previamente definidos e contem uma lista de Propriedades definidas pelo Tipo

de Objeto ao qual está associado. De modo semelhante, cada Propriedade contem uma referência para o Tipo de Propriedade por ela representada e o seu valor.

Baseado neste modelo o usuário final pode: alterar as regras do negócio instanciando novos tipos ou modificando os tipos existentes; ou executar as regras existentes através da instanciação de objetos dos tipos existentes.

4.2 Casos de Uso

A Figura 4.4 apresenta o diagrama de casos de usos definidos para a especificação de um protótipo que foi construído para verificar a viabilidade da arquitetura proposta.

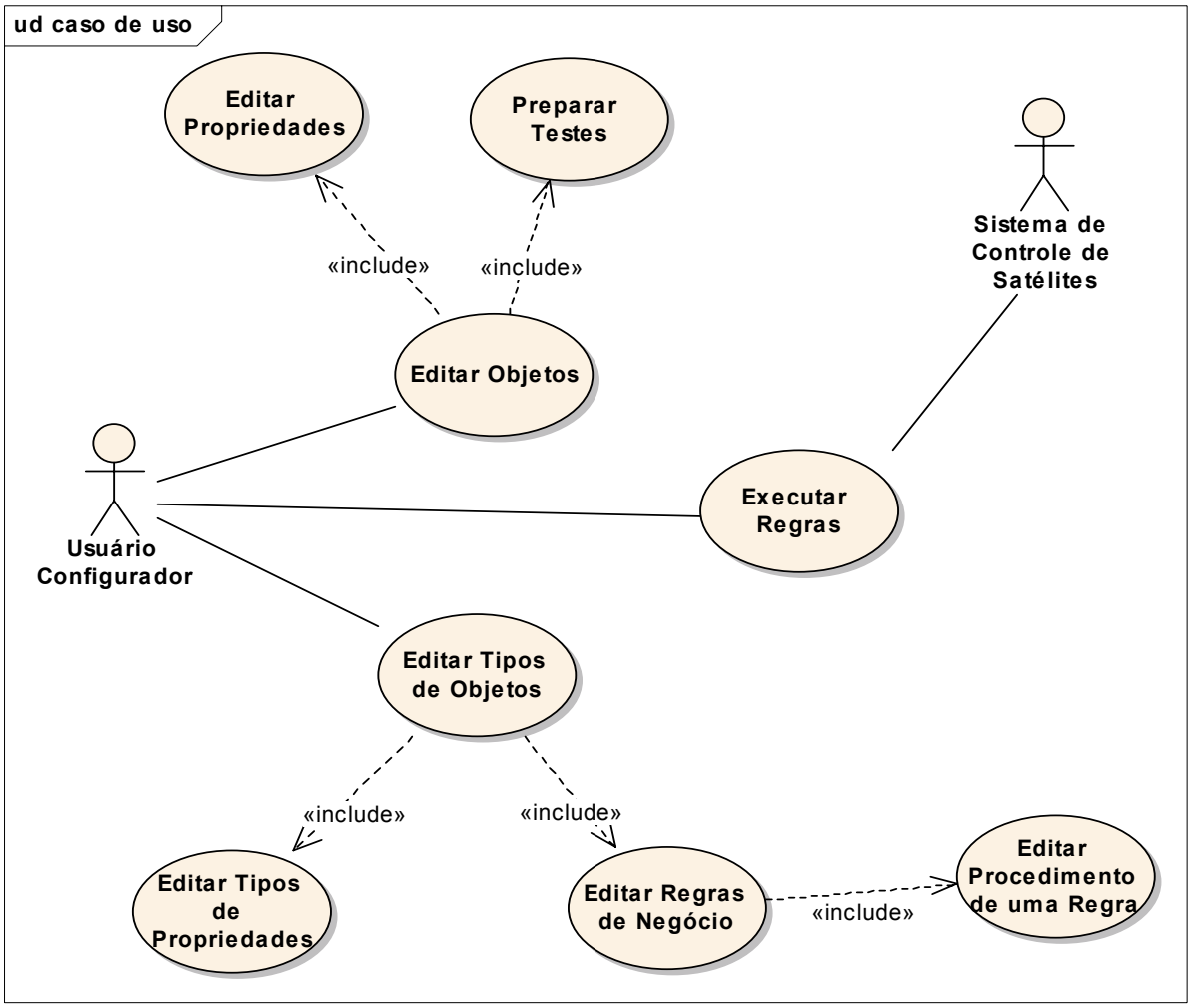

FIGURA 4.4 – Diagrama de casos de uso.

Foram definidos três casos de uso principais: Editar Objetos, Editar Tipos de Objetos e Executar Regras. Os dois primeiros se referem à ferramenta de edição e o último à interface para que o modelo de objetos criados pela ferramenta possa ser acessado pelo sistema de controle de satélites. O terceiro caso de uso também representa uma interface disponível na ferramenta de edição para permitir que os tipos de objetos e objetos criados possam ser testados antes de serem utilizados pelo sistema de controle de satélites.

A edição dos Tipos de Objetos inclui dois casos de uso: Editar Tipos de Propriedades e a Editar Regras de Negócio. Na definição de um Tipo de Propriedade o usuário deve definir o nome da Propriedade e definir o seu tipo através da seleção de um dos Tipos de Objetos existentes. Para adicionar uma nova Regra a um Tipo de Objeto o usuário deve especificar o nome da Regra, um possível conjunto de Tipos de Objetos utilizados para definir um contexto de entidades a ser utilizado pela regra (tipo de retorno e parâmetros), possíveis variáveis locais que sejam necessárias e o Procedimento a ser executado.

Na edição de um Procedimento, o usuário pode incluir / eliminar comados dos seguintes tipos: funções de objetos primitivos, funções de outros Tipos de Entidades definidos através da ferramenta, seqüência condicional de comandos e seqüência iterativa de comandos.

A edição de um Objeto inclui a Edição de suas Propriedades e a preparação de um ambiente para testes das Regras criadas ou modificadas. Ao incluir um novo Objeto o usuário deve definir o nome do Objeto e selecionar o seu tipo do conjunto de Tipos de Objetos pré-definidos pela ferramenta. Ao se criar um novo Objeto a ferramenta cria automaticamente as Propriedades associadas ao seu tipo. Posteriormente o usuário deve definir o valor de cada propriedade, atribuindo-lhe um valor dos tipos primitivos básicos (Inteiro, Texto, Real ou Boleano) ou criar uma referência para um outro Objeto (Relacionamento de Associação). Para facilitar a verificação do modelo de objetos criados ou modificados, a ferramenta adicionalmente cria um ambiente para que o usuário possa testar a execução de novas regras ou testar a execução de regras após

modificações ocorridas em algum elemento do modelo associado a elas. Na definição do ambiente necessário para testar uma regra o usuário deve definir o contexto que será utilizado na execução da regra: Objetos associados ao valor de retorno da regra e aos seus parâmetros. O contexto pode também ser definido através de valores no caso de objetos dos tipos primitivos básicos.

A especificação do caso de uso Executar Regras estabelece que a ferramenta de edição e o sistema de controle de satélites devem poder ter acesso aos Objetos e Tipos de Objetos existentes e executar qualquer uma de suas regras.

4.3 *Framework* DOM

Neste Item serão apresentadas as classes que foram criadas para representar a árvore descrita no Item 4.1. A tabela 4.1 apresenta o mapeamento entre os nós das árvores, ilustradas nas Figuras 4.2 e 4.3, as classes que foram criadas para representá-los, e os casos de uso, ilustrados na Figura 4.4, nos quais estas classes colaboram.

TABELA 4.1 – Relacionamento entre casos de uso, classes e nós da árvore

Nó	**Classe**	**Caso de Uso**
Lista de tipos de objetos	CdomListaTipos	Editar Tipos de Objetos
Tipo de objeto	CtipoEntidade	
Lista de tipos de propriedades	ClistaAtributos	
Lista de regras	ClistaServicos	
Lista de Objetos	CDomListaEntidades	Editar Objetos
Objeto	CEntidade	
Lista de propriedades	ClistaPropriedades	
[1]	CListaFuncoes	
Propriedade	CPropriedade	CPropriedade
Tipo de propriedade	CAtributo	Editar Tipos de Propriedades

(continua)

TABELA 4.1 (continuação)

Regra	CServico	Editar Regras de Negócio
Tipo Retorno	CTipoRetorno	
Lista de Parâmetros	CListaParametros	
Parâmetro	CParametro	
Lista de variáveis locais	CListaVarLocais	
Variável local	CVarLocal	
Procedimento	CSequencia	
Comando	CComando	Editar Procedimento de uma Regra
Objeto Comando	CObjetoComando	
Regra Comando	CMetodoComando	
Retorno Comando	CRetornoComando	
Lista de argumentos do Comando	CListaArgumentos	
Argumento Comando	CArgumento	
Comando lógico	CComandoLogico	
Comando composto	CSeqIterativa	
Comando composto	CSeqCondicionada	
Condição	CCondicao	

(continua)

TABELA 4.1 (conclusão)

[1]	CFuncao	Preparar Testes
[1]	CTipoRetorno	
[1]	CListaArgumentos	
[1]	CArgumento	
Lista de Objetos	CDomListaEntidades	Executar Regras
Lista de tipos de objetos	CDomListaTipos	

[1] Estes nós foram acrescentados adicionalmente ao modelo de árvore proposto com a finalidade de permitir os testes dos Objetos criados através do modelo DOM

Os principais relacionamentos estáticos entre as classes da Tabela 4.1 serão apresentados a seguir através de alguns diagramas de classes.

Em todos os diagramas de classe o estereótipo “Nó” tem o propósito de identificar que instância(s) de uma classe são nós de instância de outra classe. Por exemplo, na Figura 4.5 a raiz do modelo se ramifica em uma instância da classe CDomListaEntidades e em uma instância da classe CDomListaTipos. Outra informação importante a respeito do modelo é que todos os nós da árvore do modelo são instâncias de classes derivadas da classe base CDomNode.

Para evitar confusões na utilização das palavras “objeto”, “propriedade” e “regra”, na continuação deste texto os objetos dinâmicos, as propriedades dos objetos dinâmicos, os tipos de objetos dinâmicos e as regras dos objetos dinâmicos serão chamados de Objetos, Propriedades, Tipos de Objetos e Regras, respectivamente.

Na Figura 4.5 está ilustrada a raiz da hierarquia de entidades e propriedades cuja composição contêm todos os Objetos, representados pela classe CEntidade, e todos os Tipos de Objetos, representados pela classe CTipoEntidade, criados pelo modelo.

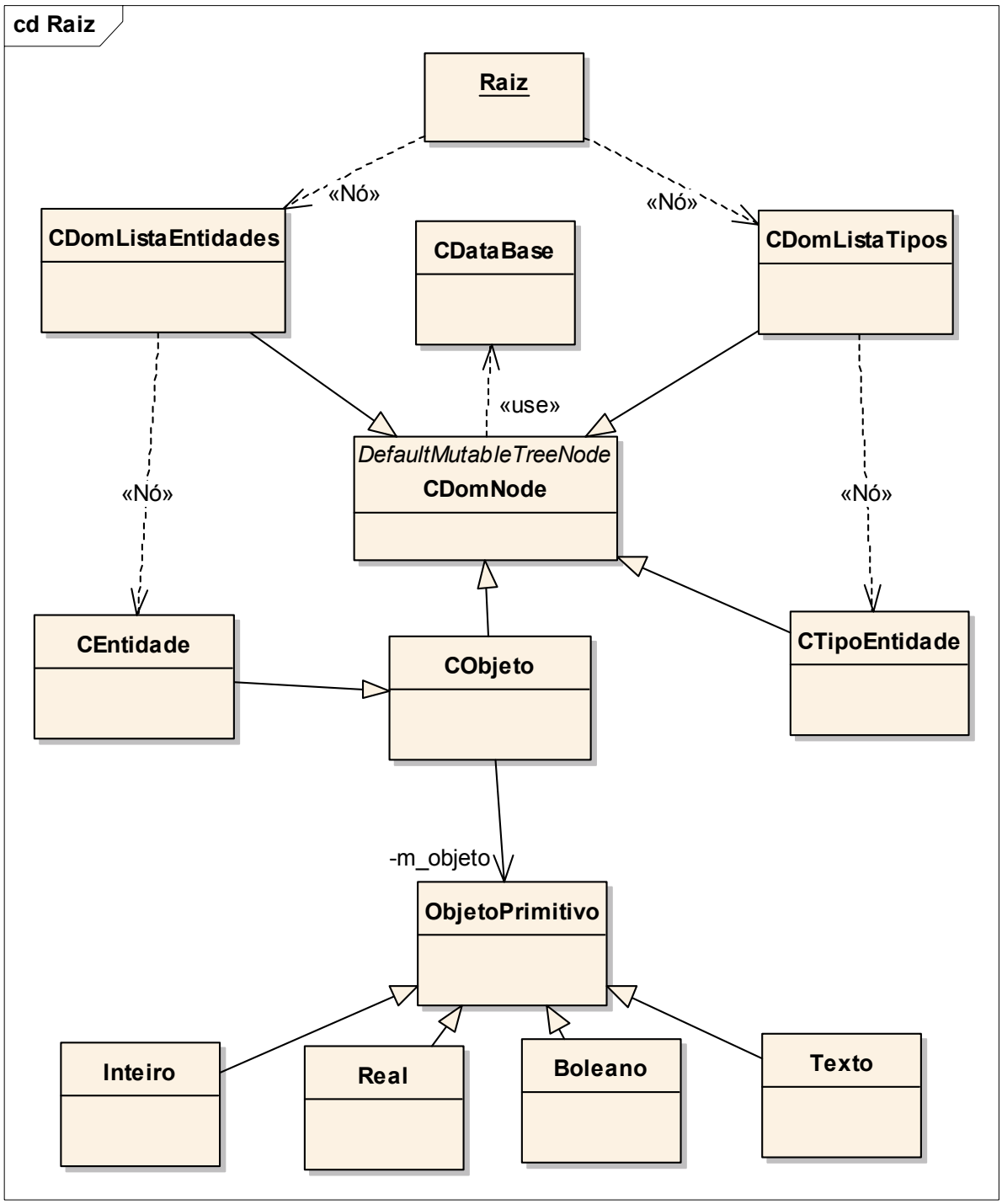

FIGURA 4.5 – Diagrama de classes “Raiz”.

A persistência de todas as informações que compõem o modelo DOM é garantida através do relacionamento entre a classe base CDomNode e a classe CDataBase. A classe CDataBase é a única interface com o banco de dados provendo todos os serviços necessários para garantir o armazenamento, recuperação, atualização e eliminação dos dados do modelo durante o processo de edição do modelo, ou atualização dos valores dos Objetos durante a execução de suas Regras.

A Figura 4.5 ilustra também o relacionamento com os objetos primitivos. A classe CObjeto é uma interface comum aos vários papéis representados pelos Objetos em diferentes contextos: Objetos, Propriedades, Argumentos e Valor de Retorno. Cada instância de CObjeto, ou suas derivadas, contém uma instância da classe CObjetoPrimitivo. A classe CObjetoPrimitivo define a interface comum a todos objetos primitivos de qualquer domínio de problema que podem ser agregados ao modelo. As classes Inteiro, Real, Boleano e Texto representam objetos primitivos básicos que são utilizados por Objetos primitivos mais complexos.

Para inserir um objeto primitivo no sistema DOM proposto é suficiente: (1) criar uma classe derivada de CObjetoPrimitivo na linguagem Java e gerar o correspondente arquivo .class e; (2) criar os metadados correspondentes ao novo Tipo de Objeto, juntamente com a especificação de seus Tipos de Propriedades e Regras. Estas especificações são realizadas com base nos Tipos de Objetos primitivos anteriormente cadastrados.

O próximo diagrama de classes (Figura 4.6) ilustra a aplicação do padrão *Type-Square* na definição do relacionamento entre os Objetos, suas Propriedades, seus Tipos e os Tipos de suas Propriedades. Cada Objeto, representando pela classe CEntidade, se associa com uma instância da classe que define o seu Tipo, representada pela classe CTipoEntidade. As Propriedades de um Objeto, representadas pela classe CPropriedade, são definidas com base na especificação dos Tipos de Propriedade, representados pela classe CAtributo, associados à classe CTipoEntidade. Um Objeto pode executar qualquer Regra, representada pela classe CServiço, associada ao seu Tipo. As classes

CListaPropriedades, CListaAtributos e CListaServiços representam, respectivamente, as coleções de Propriedades, Tipos de Propriedades e Regras.

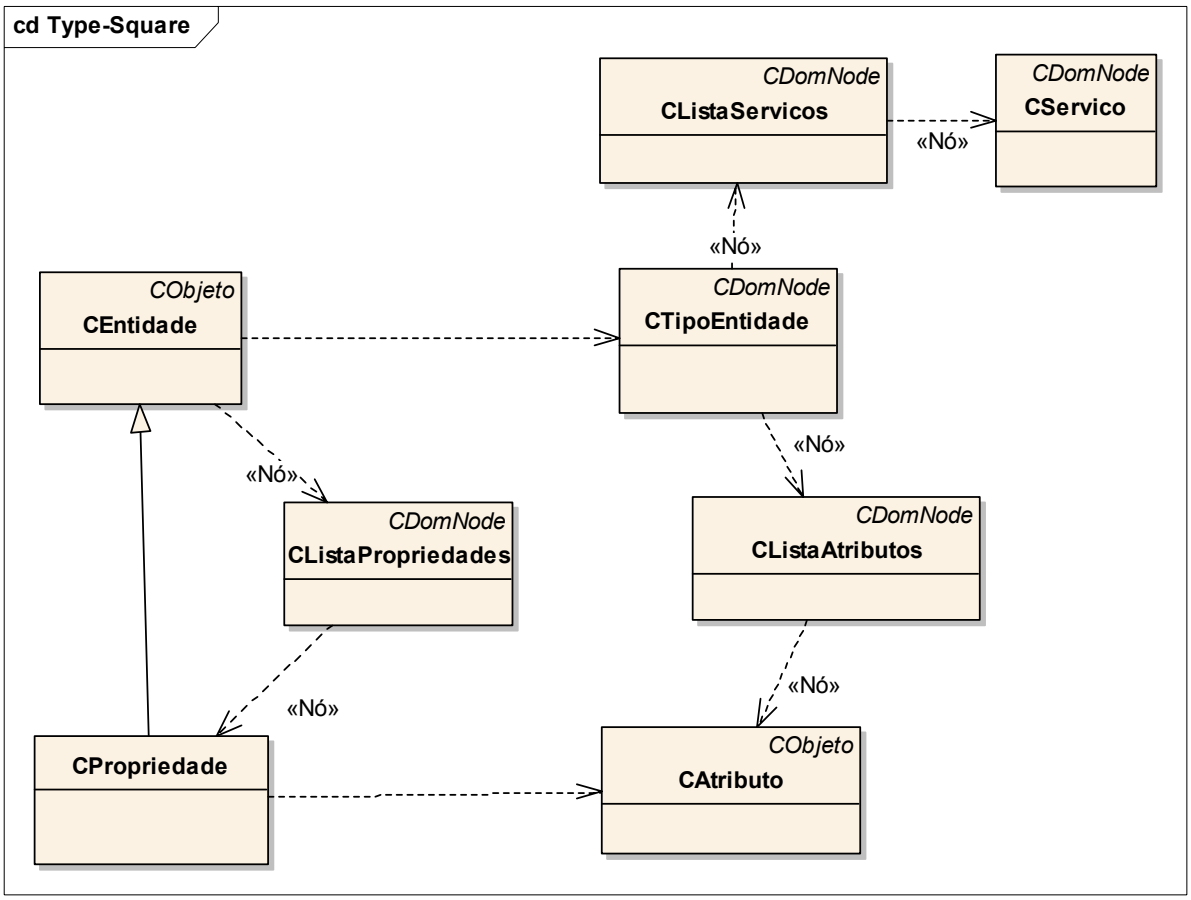

FIGURA 4.6 – Diagrama de classes *Type-Square.*

Uma vez que um Tipo de Objeto esteja totalmente definido, através dos seus Tipos de Propriedades e suas Regras, é possível criar Objetos a partir dele. Juntamente com cada instância da classe CEntidade são criadas instâncias das classes CListaPropriedades, CListaFuncoes e CPropriedade. Cada uma das instâncias da classe CPropriedade criada corresponde a uma instância da classe CAtributo associada ao Tipo de Propriedade em questão. Na definição de uma Propriedade é possível:

- atribuir valores a Propriedades correspondentes aos Tipos primitivos básicos;
- criar referências destas Propriedades para outros Objetos do mesmo tipo que existam de forma a representar relacionamentos de associação entre Objetos;

- definir as Propriedades de uma Propriedade levando em conta que a classe CPropriedade é derivada de CEntidade.

A descrição dos componentes de uma Regra está ilustrada na Figura 4.7. A classe CSequencia representa a seqüência de comandos que devem ser executados pela Regra de negócio em questão. A figura apresenta também os componentes de um comando e suas diferentes especializações para oferecer uma flexível capacidade de edição de uma Regra. Cada comando da seqüência é uma instância da classe CComando. A classe CSequencia tem uma referência para o primeiro comando da seqüência e cada instância da classe CComando possui uma referência para o próximo comando da seqüência.

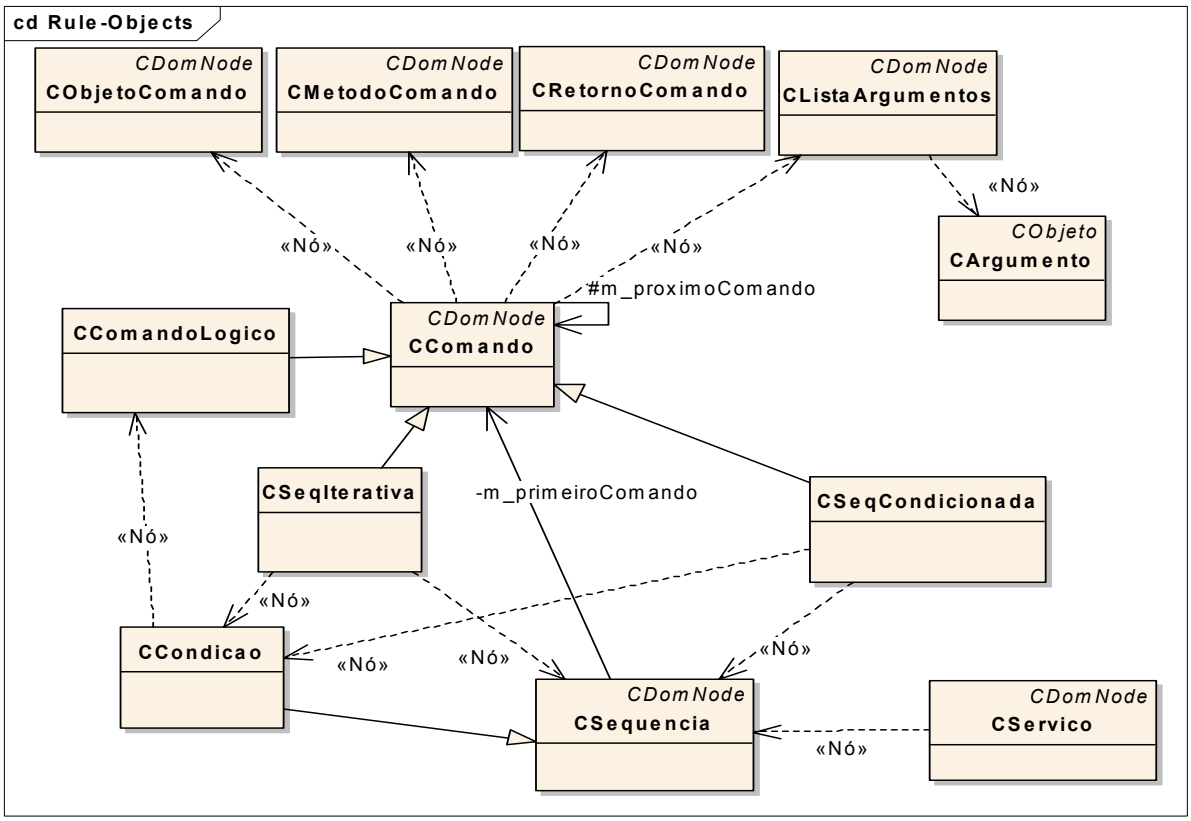

FIGURA 4.7 – Diagrama de classes dos componentes de uma regra.

Um comando de uma Regra A pode corresponder à execução de uma outra Regra do mesmo tipo de objeto da Regra A ou uma Regra de qualquer outro objeto que faça parte do contexto de Objetos visível pela Regra A. Nestes dois casos o comando, representado pela classe CComando, possui como componentes instâncias das classes

CObjetoComando (Objeto que recebe o comando) , CMetodoComando (Regra associada ao comando), CRetornoComando (Objeto que recebe o valor retornado pelo comando) e CListaArgumentos (Lista de Argumentos utilizados pelo comando).

Por outro lado um comando pode representar um bloco de comandos condicionais ou iterativos. Nestes casos a classe CComando é especializada nas classes CSeqIterativa e CSeqCondicionada. Estas duas classes possuem dois componentes: uma seqüência de comandos e a condição que deve ser satisfeita para a execução da seqüência. A seqüência de comandos destes blocos é representada pela mesma classe CSequencia utilizada para descrever uma regra e a condição é representada pela classe CCondicao que é uma especialização da classe CSequencia. Uma condição é composta por uma série de comandos lógicos que são representados pela classe CComandoLogico. Este conjunto de classes foi modelado tendo como referência o padrão *Rule-Objects*.

Para facilitar os testes dos Objetos foi incluída no modelo uma nova ramificação de propriedades, que está ilustrada na Figura 4.8, de modo a ser possível definir o contexto e ativar as Regras dos Objetos através da ferramenta de edição. Nesta nova ramificação a classe CFuncao representa um teste aplicado a uma Regra. Para cada teste são associados os Objetos correspondentes aos argumentos da regra e o Objeto que irá receber o valor retornado pela Regra, após sua execução. Estes Objetos estão representados pelas classes CArgumento e CTipoRetorno, respectivamente. Na definição dos argumentos é possível atribuir o valor de um tipo primitivo básico, ou fazer referência ao conjunto de Objetos existentes e também ao conjunto de Propriedades do Objeto em questão.

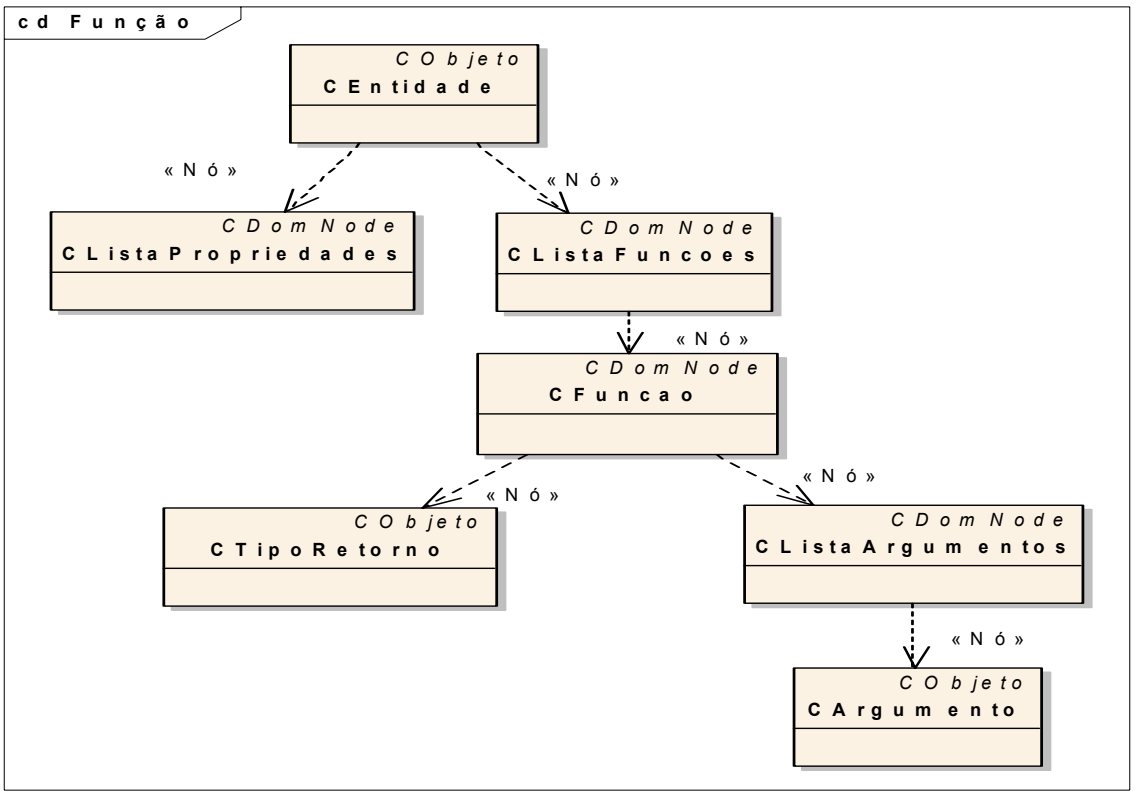

FIGURA 4.8 – Diagrama de classes “Função de Teste”.

Como pode ser visto, os componentes de uma função de teste são representados pelas mesmas classes utilizadas anteriormente para definir os comandos de uma regra.

4.4 Metadados

Todas as operações de edição do modelo e execução de regras, descritas nos itens anteriores, resultam na atualização dos elementos que compõem a árvore de entidades e propriedades em uma Base de Dados de modo a garantir a persistência dos dados. A Figura 4.9 ilustra o modelo da base de dados que foi idealizada. Este modelo é composto de três relações:

- Relação tblEntidade, na qual são armazenadas as descrições de todas as entidades descritas no Item 4.1 (Objetos, Tipos de Objetos, Propriedades, Tipos de Propriedades, Regras, etc).

- Relação tblPropriedade, na qual são armazenados os relacionamentos entre uma entidade e suas propriedades

- Relação, na qual são armazenados os valores dos tipos primitivos básicos que são entrada ou saída do processo de execução de uma Regra

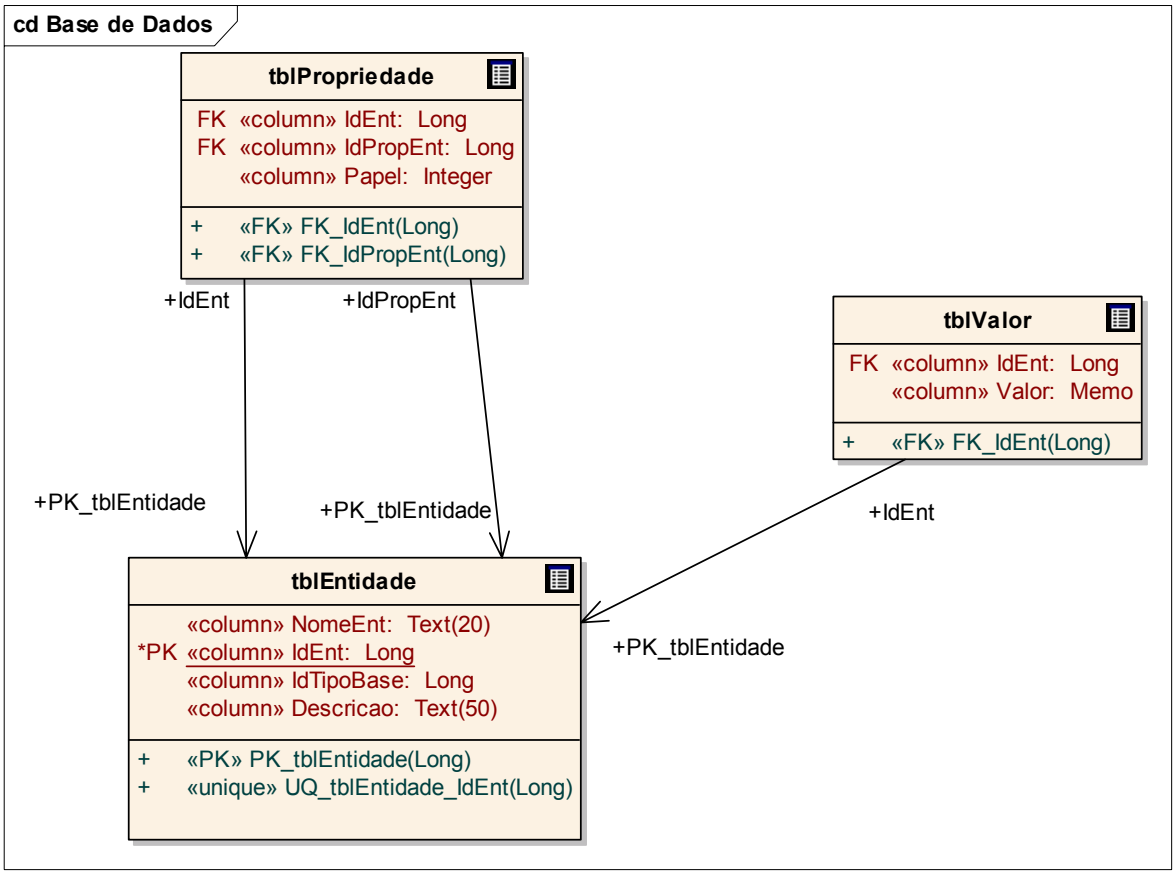

FIGURA 4.9 – Diagrama do modelo da base de dados.

A tabela tblEntidade é composta dos seguintes campos:

- IdEnt: identificação numérica e única de cada entidade

- NomeEnt: nome atribuído à entidade

- IdTipoBase: identificação do tipo de nó que a entidade representa na árvore descrita no Item 4.1

- Descrição: texto descritivo da Entidade

A tabela tblPropriedade é composta dos seguintes campos:

- IdEnt: identificação da entidade
- IdPropEnt: identificação da propriedade
- Papel: tipo de relacionamento entre uma Entidade e sua propriedade (Tipo de Entidade, Referência, Sequenciamento). O papel "Tipo de Entidade" é utilizado na associação de um Objeto com outra entidade que define o seu Tipo de Objeto. O papel "Referência" é utilizado na definição dos argumentos de uma Regra ou para definir o relacionamento de associação entre Objetos. E o papel "Sequenciamento" é utilizado para garantir a ordem dos comandos que compõem uma Regra.

A tabela tblValor é composta dos seguintes campos:

- IdEnt: identificação de uma entidade que representa um Objeto
- Valor: valor de um tipo primitivo básico (Inteiro, Real, Boleano ou Texto)

4.5 Interface Externa

Após a criação e validação, via a ferramenta de edição, os Objetos armazenados no repositório podem ser utilizados pelo sistema de controle de satélites. Para que o acesso a estes Objetos seja possível é necessário definir uma interface externa para ativação dos mesmos.

Para que possam executar uma de suas Regras de negócio definidas no repositório, o sistema de controle precisa definir as seguintes informações:

- Nome do Objeto ou do Tipo de Objeto, contido no repositório, associado à Regra de negócio que se deseja executar;

- Nome da Regra de negócio contida no repositório;

- Nome do Objeto, contido no repositório, que deverá receber o valor retornado pela Regra de negócio ou referência para um objeto interno ao sistema de controle de satélites;

- Nomes dos objetos, contidos no repositório, correspondentes aos argumentos utilizados pela regra de negócio ou referencias para objetos internos ao sistema de controle de satélites;

Quando se desejar executar uma Regra do repositório os sistemas de controle deverão executar um serviço da interface de acesso ao repositório passando estas informações. Os objetos internos do sistema de controle de satélites, que fizerem parte destas informações, devem ser instancias de classes derivadas da classe ObjetoPrimitivo definida no Item 4.3, ou seja devem ser também objetos do domínio de conhecimento das classes que controlam o acesso ao repositório de objetos adaptáveis.

O serviço da interface de acesso ao repositório, por sua vez, será responsável pela ativação da Regra e atualização dos valores dos Objetos associados ao valor de retorno e argumentos da Regra.

Além do serviço de ativação das Regras de negócio, a interface de acesso ao repositório deve também prover serviços para que módulos de configuração dos sistemas externos possam selecionar os nomes dos Objetos, Tipos de Objetos e das Regras de negócio armazenados no repositório.

A definição desta interface de acesso permite que futuros requisitos possam ser implementados por um usuário devidamente capacitado do sistema de controle de satélites, de duas maneiras:

- mudança do comportamento interno de uma Regra do repositório que o sistema de controle de satélites já conhece, através da utilização da ferramenta de edição do repositório;

- criação de novas Regras no repositório e configuração do sistema de controle de satélites para acessá-las.

4.6 Construção e Interpretação dos Objetos Dinâmicos

A seguir, através de diagramas de seqüência que representam as principais funções da arquitetura proposta, serão apresentados mais detalhes sobre a troca de mensagens entre os objetos das classes utilizadas para representar os Objetos Dinâmicos. Além disso, nos casos em que são utilizadas funções reflexivas será também mostrada a implementação em Java. Desta forma, serão detalhadas a inicialização do sistema DOM, as funções de edição (inserção, remoção e atualização) dos Objetos Dinâmicos através da ferramenta de edição e a execução de Regras dos Objetos Dinâmicos.

A Figura 4.10 ilustra a inicialização do sistema DOM, quando todos os objetos dinâmicos são instanciados através da sua recuperação da base de dados. Inicialmente o serviço CriarPropriedade instancia o nó raiz. Em seguida é chamado o serviço Expandir para instanciar os nós contidos no nó raiz. O serviço Expandir da classe base CDomNode é um serviço recursivo que extrai cada nó da base de dados, o instancia utilizando o serviço CriarPropriedade e em seguida chama o serviço Expandir para o novo nó criado. O serviço Expandir pode ser redefinido nas classes que representem algum tipo de nó especial. Por exemplo, no caso de nós que representam comandos este serviço é redefinido de modo a apresentar na tela a ordem correta dos comandos.

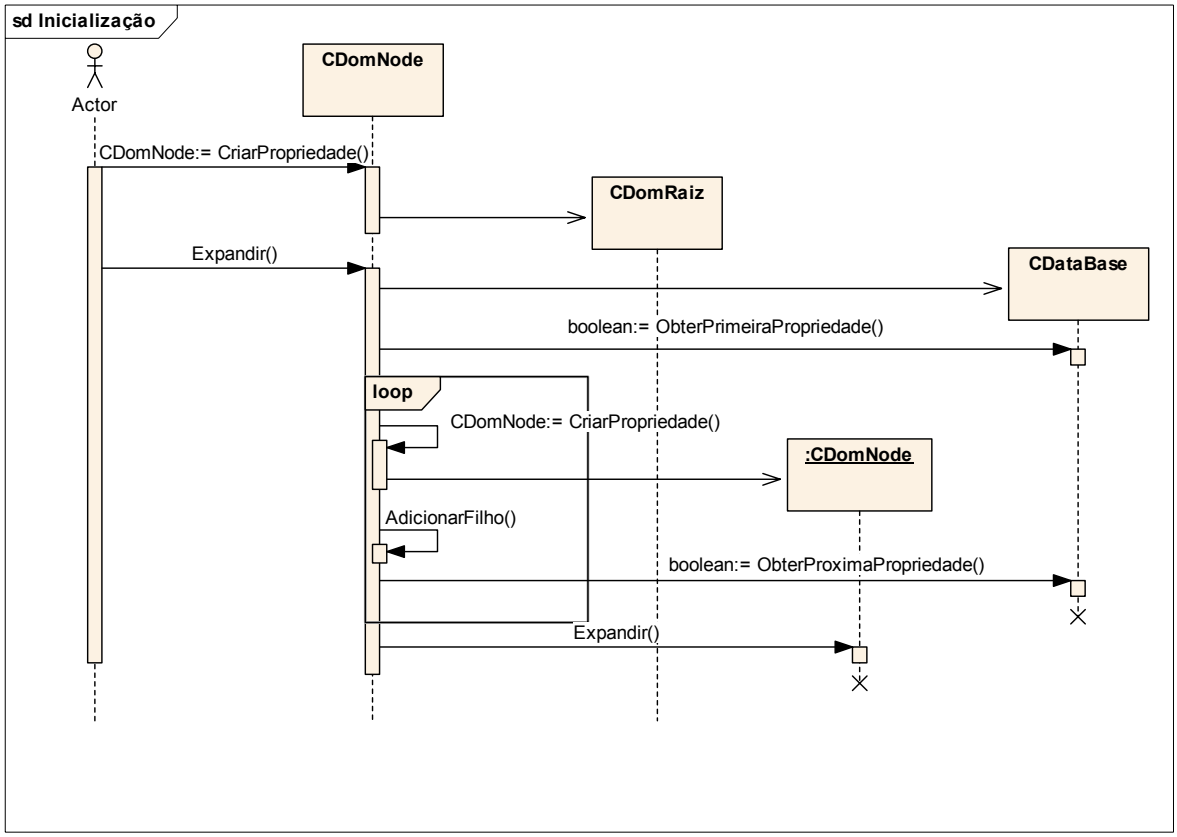

FIGURA 4.10 – Diagrama de seqüência de inicialização.

Uma vez que todos os nós estejam instanciados o usuário pode navegar pela árvore gerada expandindo os nós desejados. Selecionando um nó com o botão direito do "*mouse*", o usuário tem acesso a um menu "*popup*" com as opções de inserção, atualização e remoção de nós. No caso de um nó que corresponda a uma função de teste é possível também selecionar a execução daquela função.

A Figura 4.11 ilustra um caso de inserção de nó: inserção de um novo Tipo de Objeto. Ao selecionar o nó correspondente à lista de tipos de objetos e a opção de inserção, um diálogo é exibido para que o usuário possa definir o nome e a descrição de um novo Tipo de Objeto. Se um usuário confirmar a operação o novo Tipo de Objeto é adicionado à base de dados e um novo nó é acrescentado à árvore. Juntamente com a inserção do novo Tipo de Objeto são inseridas as suas correspondentes listas de atributos (Tipos de Propriedades) e serviços (Regras). O usuário pode, posteriormente,

selecionar estas listas para adicionar os Tipos de Propriedades e as Regras associadas àquele Tipo de Objeto.

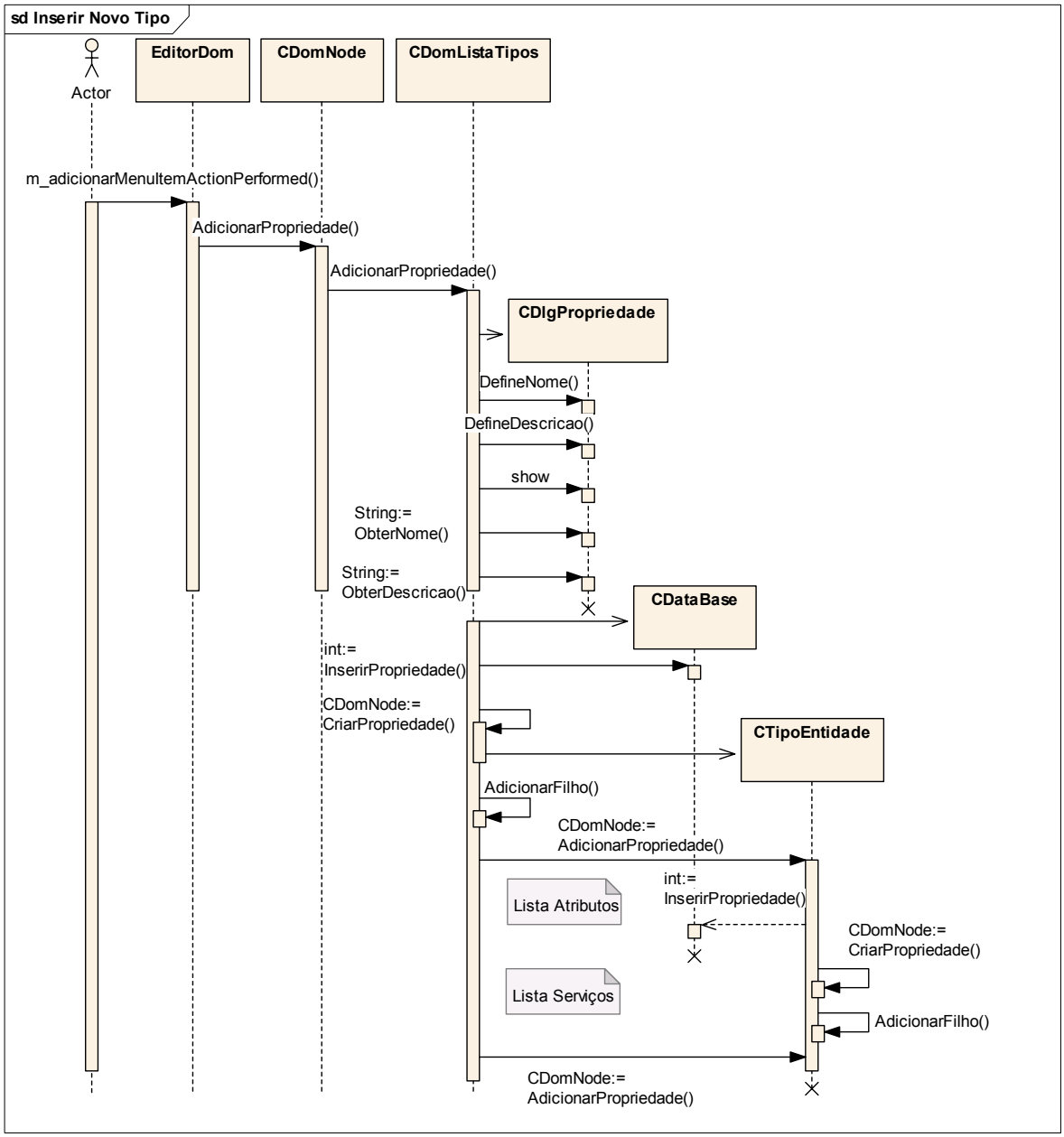

FIGURA 4.11 – Diagrama de seqüência para inserir novo tipo de objeto.

As Figuras 4.12 e 4.13 a seguir exemplificam o processo de atualização de um nó da árvore. Estas figuras também ilustram o processo de definição de valores dos tipos primitivos usando as funções reflexivas da linguagem Java.

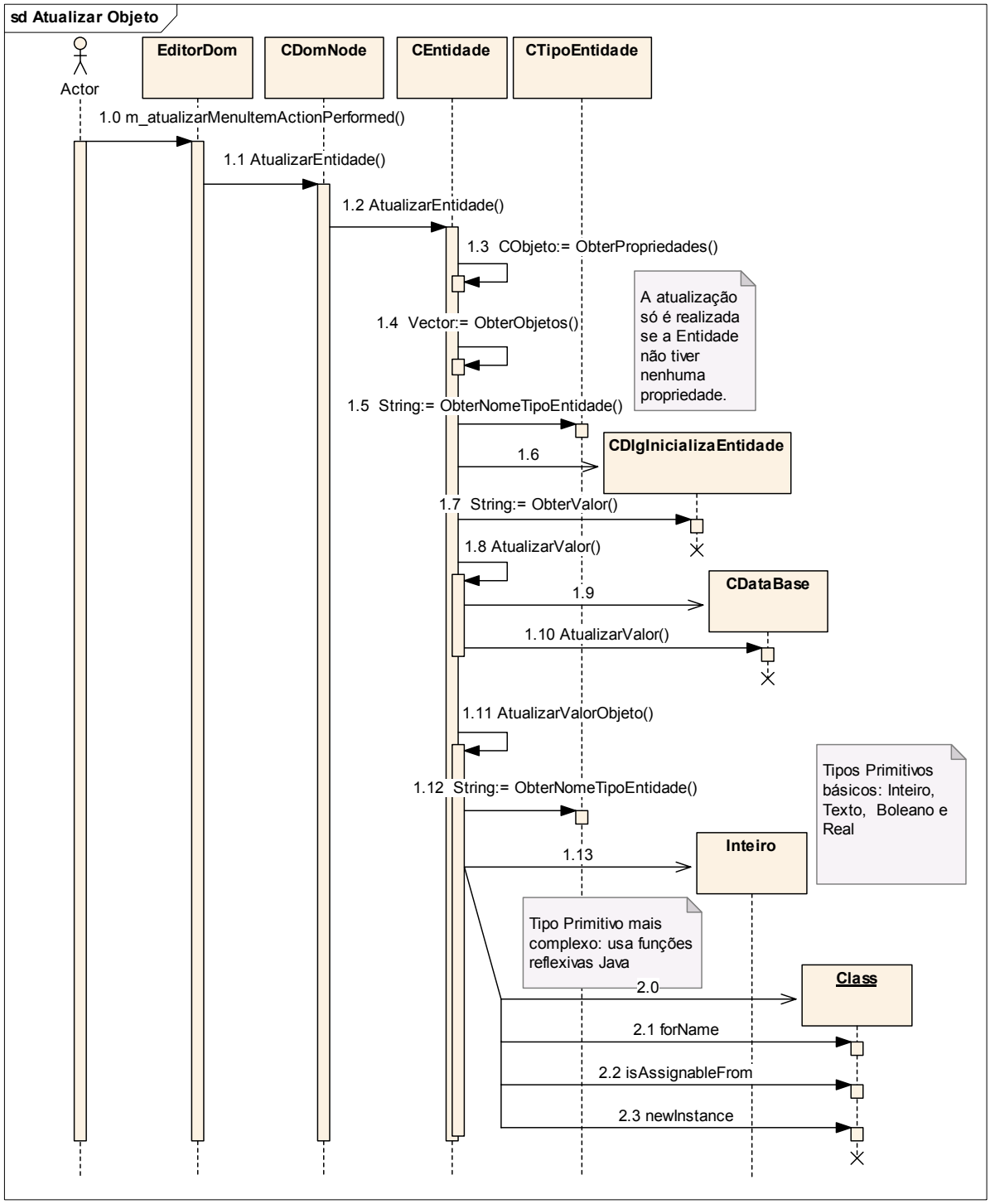

FIGURA 4.12 – Diagrama de seqüência para atualizar valor de um objeto.

Na Figura 4.12 é exibido o processo de atualização do valor de um Objeto. Ao selecionar um nó correspondente a um Objeto que não tenha nenhuma Propriedade é exibido um diálogo para que o usuário possa definir uma referência deste Objeto para

um outro Objeto, ou definir o seu valor caso o Objeto correspondente ao nó selecionado for de um tipo primitivo. Levando em conta que a classe CPropriedade é uma especialização da classe CEntidade a referência para um outro Objeto é principalmente utilizada no caso de uma Propriedade que representa uma associação com outro Objeto. Em seguida se o Tipo do Objeto for um dos tipos primitivos o serviço AtualizarValorObjeto é ativado para instanciar um objeto do tipo primitivo correspondente e associá-lo com o Objeto. Se o tipo primitivo for um dos tipos básicos o objeto é instanciado de modo a conter o valor digitado pelo usuário. Por outro lado, se o objeto for um tipo primitivo mais complexo é criada uma instância utilizando as funções reflexivas do Java definidas na classe "Class" e o nome do Tipo de Objeto associado. Por último, o valor obtido ao final da interação do usuário com o diálogo é armazenado na base de dados.

No trecho de código Java apresentado a seguir "tipo" contém o nome do tipo do Objeto cujo valor está sendo atualizado e "ObjetoPrimitivo" é o nome da classe base de todos os objetos primitivos. A linha de código com o comando "isAssignableFrom" é utilizada para verificar se a classe associada com o tipo do Objeto é derivada da classe ObjetoPrimitivo. Caso seja derivada, o comando "newInstance" é utilizado para criar uma instancia da mesma.

```
try
{
  Class classe = Class.forName(tipo);
  Class classePai = Class.forName("ObjetoPrimitivo"); // linha 2.1 no diagrama
  if(classePai.isAssignableFrom(classe))  // linha 2.2
        m_objeto = (ObjetoPrimitivo)classe.newInstance();  // linha 2.3
}
```

```
catch (Exception exception)
{
    ....
}
```

A Figura 4.13 exibe o processo de atualização do valor de uma Propriedade. O serviço AtualizarEntidade é especializado na classe CPropriedade. Em primeiro lugar este serviço chama o serviço de mesmo nome da classe base CEntidade. Em seguida, no caso do Objeto que possui esta Propriedade ser de um dos tipos de objetos primitivos o seu correspondente atributo (m_objeto da classe CObjeto) é atualizado de modo a refletir a mudança do valor da Propriedade.

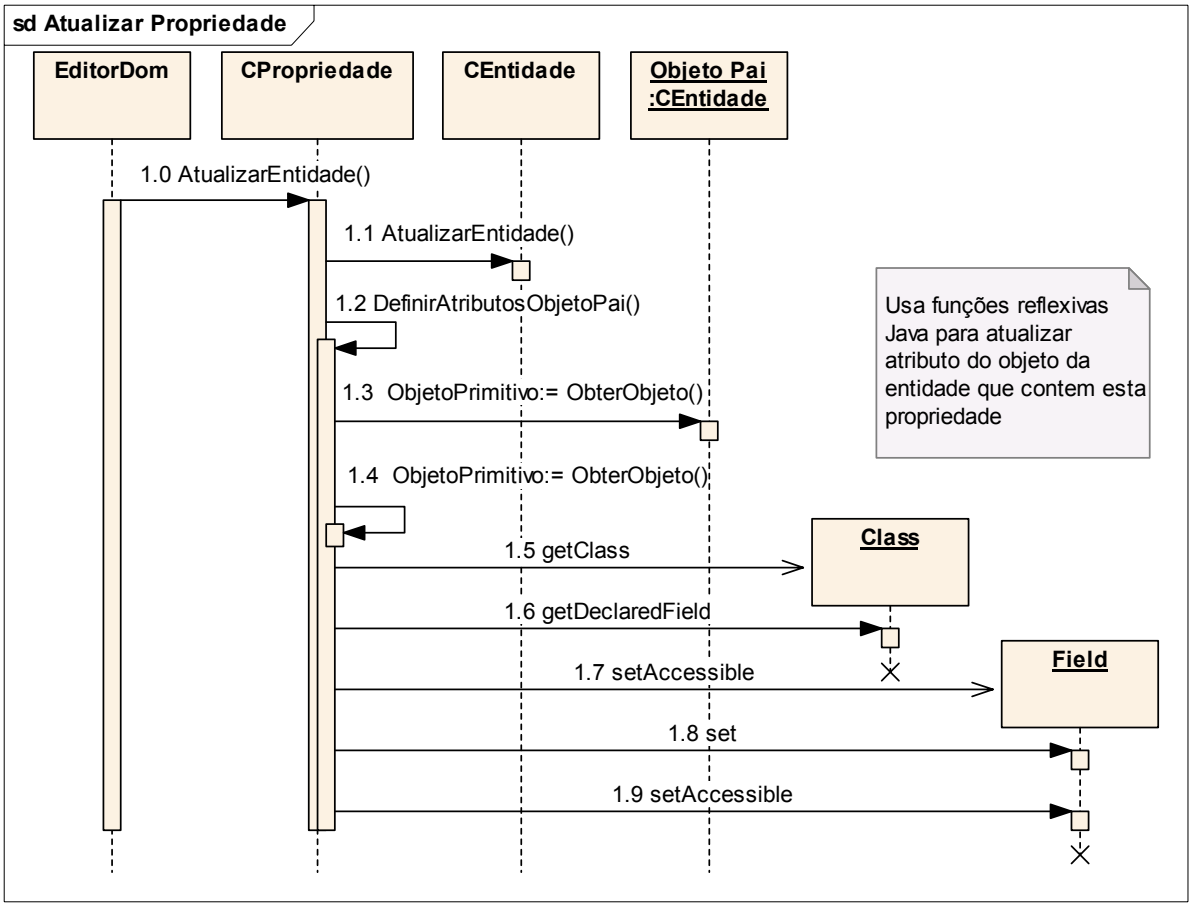

FIGURA 4.13 – Diagrama de seqüência para atualizar valor de uma propriedade.

No trecho de código abaixo objPai é o objeto primitivo associado com o Objeto que possui a Propriedade sendo modificada, objProp é o objeto primitivo associado com a Propriedade, classePai é a classe da qual objPai é uma instância e campoProp é o atributo de classePai que tem o mesmo nome da Propriedade (m_nome). A linha com o comando set é utilizada para atualizar o atributo de classePai com o valor de objProp. O comando SetAccessible tem a finalidade de permitir que atributos que não sejam públicos possam ser acessados.

```
ObjetoPrimitivo objPai = entPai.ObterObjeto();  // linha 1.3 do diagrama
if (objPai == null) return;
ObjetoPrimitivo objProp = ObterObjeto(); // linha 1.4
if (objProp == null) return;


try
{
   Class classePai = objPai.getClass();  // linha 1.5
   Field campoProp = classePai.getDeclaredField(m_nome);  // linha 1.6
   campoProp.setAccessible(true);  // linha 1.7
   campoProp.set(objPai, objProp);  // linha 1.8
   campoProp.setAccessible(false);  // linha 1.9
}
catch (Exception exception)
{
```

....

}

Além das funções para adicionar e atualizar nós da árvore a ferramenta de edição também inclui a função de eliminação de nós. A Figura 4.14 exibe a seqüência para eliminar um Objeto com o objetivo de ilustrar o processo de eliminação de um nó da árvore e a correspondente atualização da base de dados. No protótipo a eliminação de um nó foi restrita a aqueles nós que não tivessem nenhum nó filho. Deste modo para eliminar um nó com nós filhos é necessário primeiro eliminar estes.

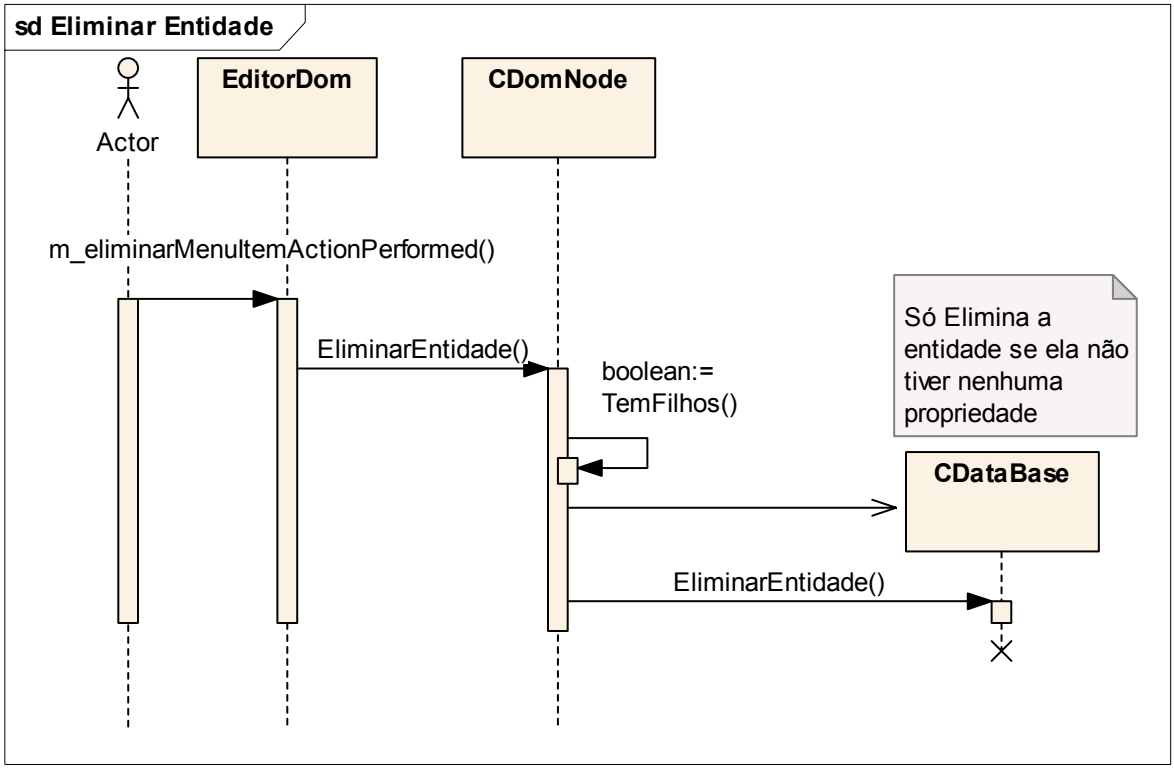

FIGURA 4.14 – Diagrama de seqüência para eliminar um nó da árvore.

Uma vez completado o processo de edição dos Objetos é necessário oferecer um meio de executar as Regras associadas com os Tipos de Objetos correspondentes. As Figuras 4.15, 4.16 e 4.17 ilustram como os elementos do modelo se relacionam para atingir este objetivo.

Como comentado anteriormente o protótipo da ferramenta de edição incluiu uma funcionalidade de adicionar funções de testes para permitir a verificação dos Objetos e Tipos de Objetos criados. A Figura 4.15 ilustra a seqüência de troca de mensagens que ocorre quando o usuário seleciona uma destas funções e a opção de executar a regra associada à mesma.

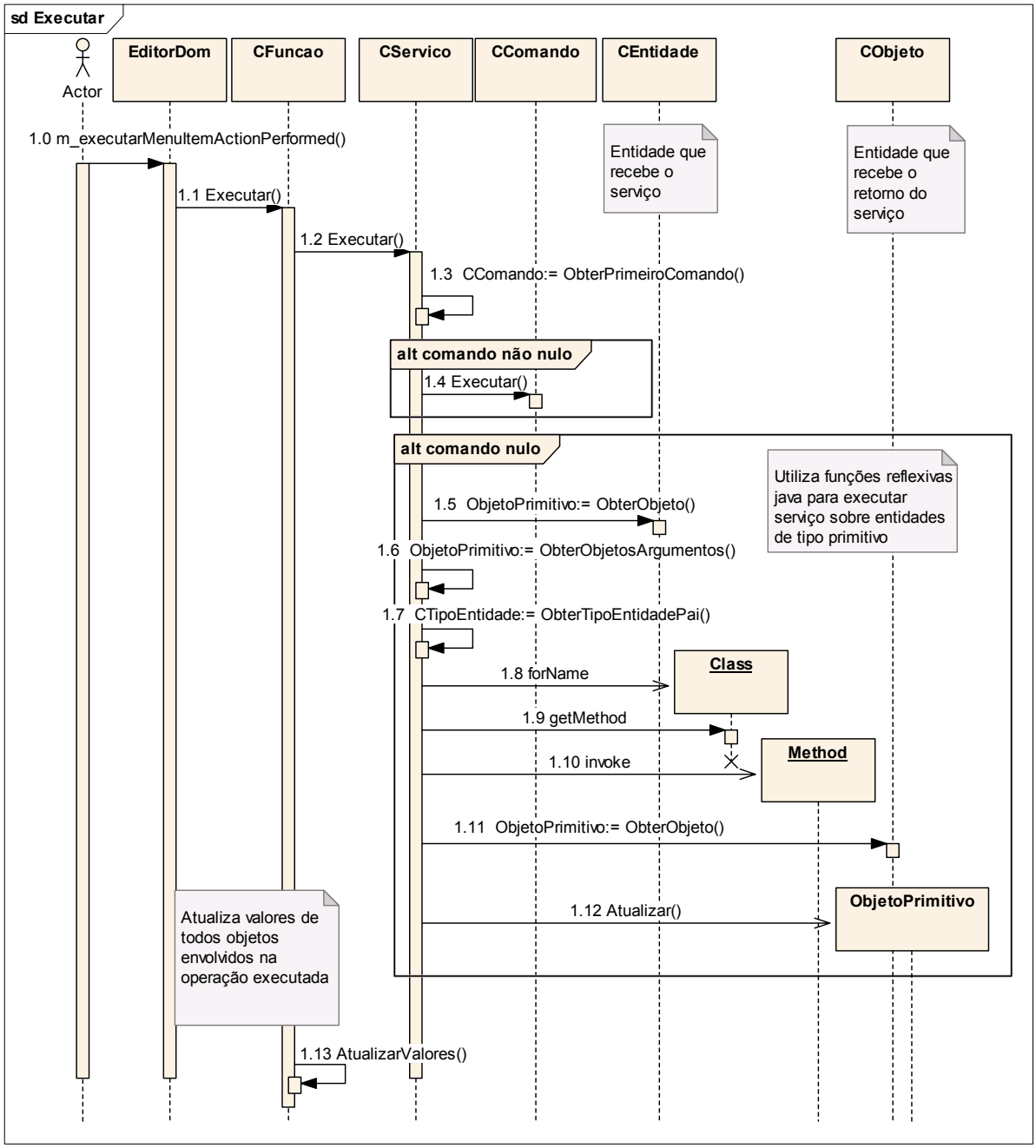

FIGURA 4.15 – Diagrama de seqüência para executar uma regra.

Uma instância da classe CFuncao recebe a mensagem Executar e ativa o serviço de mesmo nome para Executar a Regra associada. Na figura esta Regra é representada pela instância da classe CServico. Caso a Regra não possua nenhum comando ela é encarada como uma Regra de um tipo primitivo. Neste caso são utilizadas as funções reflexivas do Java para invocar o correspondente método da classe primitiva. O trecho de código a seguir detalha como ocorre a interação entre o modelo de objetos criado e as funções reflexivas do Java. O objeto "nomeTipo" define o nome do Tipo de Objeto do qual a regra faz parte, "nomeRegra" é o nome da regra, "nomeTiposParametros" é um vetor com os nomes dos Tipos de Objetos associados com os parâmetros utilizados pela regra. Desta forma é obtido o objeto "metodo" que representa o método da classe primitiva, associada com o Tipo de Objeto, que tenha a assinatura definida pelo nome da Regra e pelos nomes dos tipos dos seus parâmetros. Em seguida é ativado o comando "invoke" para executar o método em questão. Neste comando, "retornoExec" representa o objeto que recebe o valor retornado pelo método, "objeto" é o objeto que recebe a mensagem "método" e "argObjetos" é um vetor com os objetos correspondentes aos argumentos utilizados pelo método durante sua execução. Em seguida, caso o método tenha algum retorno (tipo diferente de "void") o objeto primitivo correspondente a este retorno é atualizado.

```
try
{
    Class classe = Class.forName(nomeTipo); // linha 1.8 do diagrama de sequência
    Method metodo = classe.getMethod(nomeRegra,
                                     nomeTiposParametros); // linha 1.9
    Object retornoExec = metodo.invoke(objeto, argObjetos); // linha 1.10
    if (a_retorno != null)
```

```
        {
            Object retorno = a_retorno.ObterObjeto(); // linha 1.11
            ((ObjetoPrimitivo)retorno).Atualizar(
                                (ObjetoPrimitivo)retornoExec);  // linha 1.12
        }
    }
    catch (Exception exception)
    {
      ....
    }
```

Ao final da execução da função selecionada o serviço AtualizarValores é ativado para atualizar os valores dos Objetos modificados durante a execução, incluindo a atualização dos valores destes Objetos na base de dados. Os valores são modificados somente ao final da execução para melhorar o desempenho do processo de execução. Se a cada modificação de um Objeto o valor da mesma fosse atualizado na base de dados, este desempenho ficaria claramente bastante comprometido.

Se a Regra possuir pelo menos um comando a instância da classe CServico ativa então o primeiro dos comandos que a compõem. A execução de um comando é ilustrada de forma mais detalhada na Figura 4.16. Ao receber da Regra ao qual pertence (regra A) uma mensagem de execução o comando, representado na figura por uma instância da classe CComando (comando Ai), obtém a Regra associada a ele (Regra B) e prepara o contexto exigido pela mesma. O contexto é composto dos seguintes objetos:

- Objeto que está recebendo o comando;

- Objeto que receberá o valor de retorno,
- Objetos que devem ser passados como argumentos.

Em seguida é enviada a mensagem Executar para a Regra B, que está representada no diagrama por uma instância da classe CServiço. Após a execução da Regra B o comando verifica se existe um próximo comando a ser executado (comando Ai+1). Caso exista, este próximo comando recebe uma mensagem para ser executado.

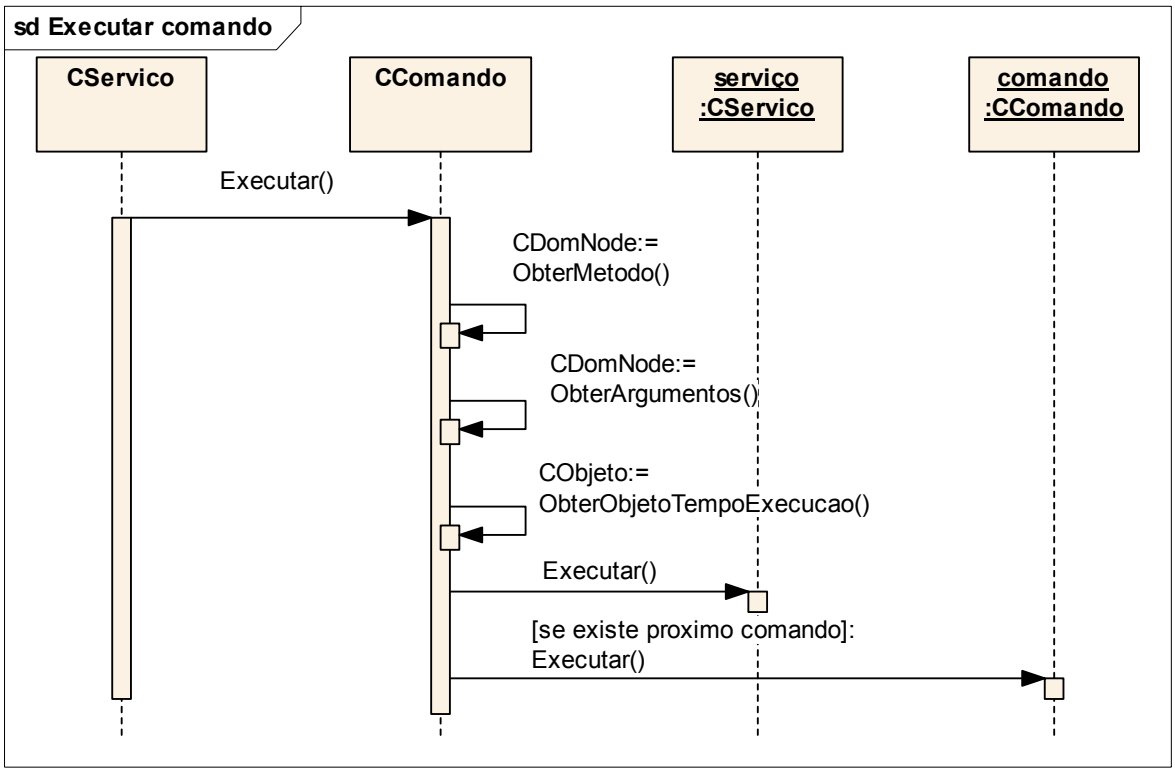

FIGURA 4.16 – Diagrama de seqüência para executar um comando.

Como comentando anteriormente uma Regra pode, além de comandos simples, conter tipos de comandos especiais representados por seqüências condicionais ou iterativas de comandos. O processo de execução destes tipos de comandos está ilustrado na Figura 4.17, onde está representada a troca de mensagens que ocorre na execução de uma seqüência condicionada de comandos. Ao receber a mensagem de execução uma Seqüência Condicionada, representada pela classe CSeqCondicionada, obtém a condição associada a ela e verifica se a condição está satisfeita. Para fazer esta

verificação o objeto correspondente à condição executa os comandos que fazem parte da mesma. Os comandos de uma condição são representados na figura por uma instância da classe CComandoLogico. Como esta classe é uma especialização da classe CComando o processo de execução dos comandos lógicos é o mesmo descrito anteriormente para a execução de comandos simples contidos em uma Regra. Se a condição for satisfeita a Seqüência Condicionada obtém a seqüência de comandos que devem ser executados. Uma vez que a seqüência de comandos obtida é representada pela classe CSequencia o processo de execução de seus comandos é o mesmo descrito anteriormente para a execução dos comandos de uma regra.

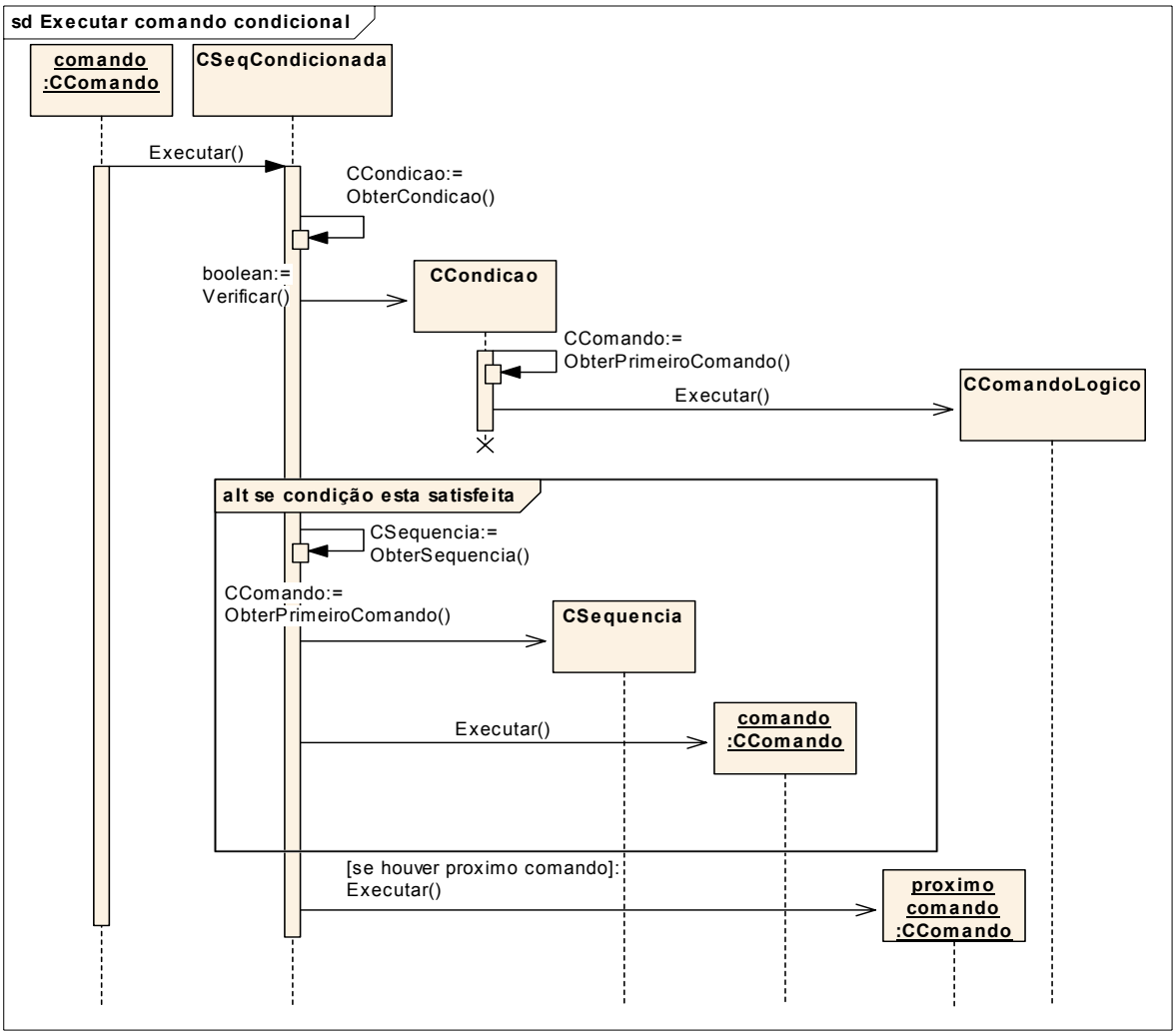

FIGURA 4.17 – Diagrama de seqüência para executar um comando condicional.

O procedimento para a execução de uma seqüência iterativa é semelhante ao descrito com a diferença que a cada iteração a condição é verificada e a execução prossegue enquanto a condição permanecer válida.

CAPÍTULO 5

IMPLEMENTAÇÃO DE UM PROTÓTIPO DA ARQUITETURA

Neste Capítulo será apresentada uma visão mais prática da arquitetura proposta no Capítulo 4, através da descrição da interface com o usuário da ferramenta que foi construída para facilitar a edição dos objetos dinâmicos (Item 5.1) e de um exemplo de utilização do modelo para introduzir uma nova regra de monitoração no subsistema de telemetria (Item 5.2), que é parte integrante do *framework* do sistema de controle de satélites citado no Capítulo 2.

Na implementação do protótipo da arquitetura proposta foram utilizados:

- linguagem Java para codificação das classes
- gerenciador de base de dados Microsoft Access para armazenar os metadados
- ferramenta Sun ONE Studio para a construção da interface gráfica.

A linguagem Java foi selecionada principalmente devido às facilidades de reflexão oferecidas. O Microsoft Access foi selecionado simplesmente por ser um gerenciador de banco de dados disponível. A ferramenta Sun ONE Studio foi selecionada por ser de acesso livre, pela integração com a linguagem Java, e pela capacidade e facilidade de desenvolvimento de interfaces gráficas com os usuários.

5.1 Ferramenta de Edição

Oferecer ao usuário uma ferramenta que torne a configuração das regras o mais natural possível é com certeza o maior obstáculo na construção de um sistema DOM. Para vencer este desafio a ferramenta de edição apresenta uma interface gráfica, como ilustrado na Figura 5.1, que modela todos os elementos do modelo DOM como uma árvore cuja raiz se ramifica em lista de tipos e lista de objetos. O usuário pode selecionar um elemento da árvore para adicionar novas propriedades ao elemento selecionado, ou eliminar / atualizar o elemento selecionado. Para facilitar os testes dos

objetos criados a ferramenta também proporciona uma facilidade de executar as suas regras.

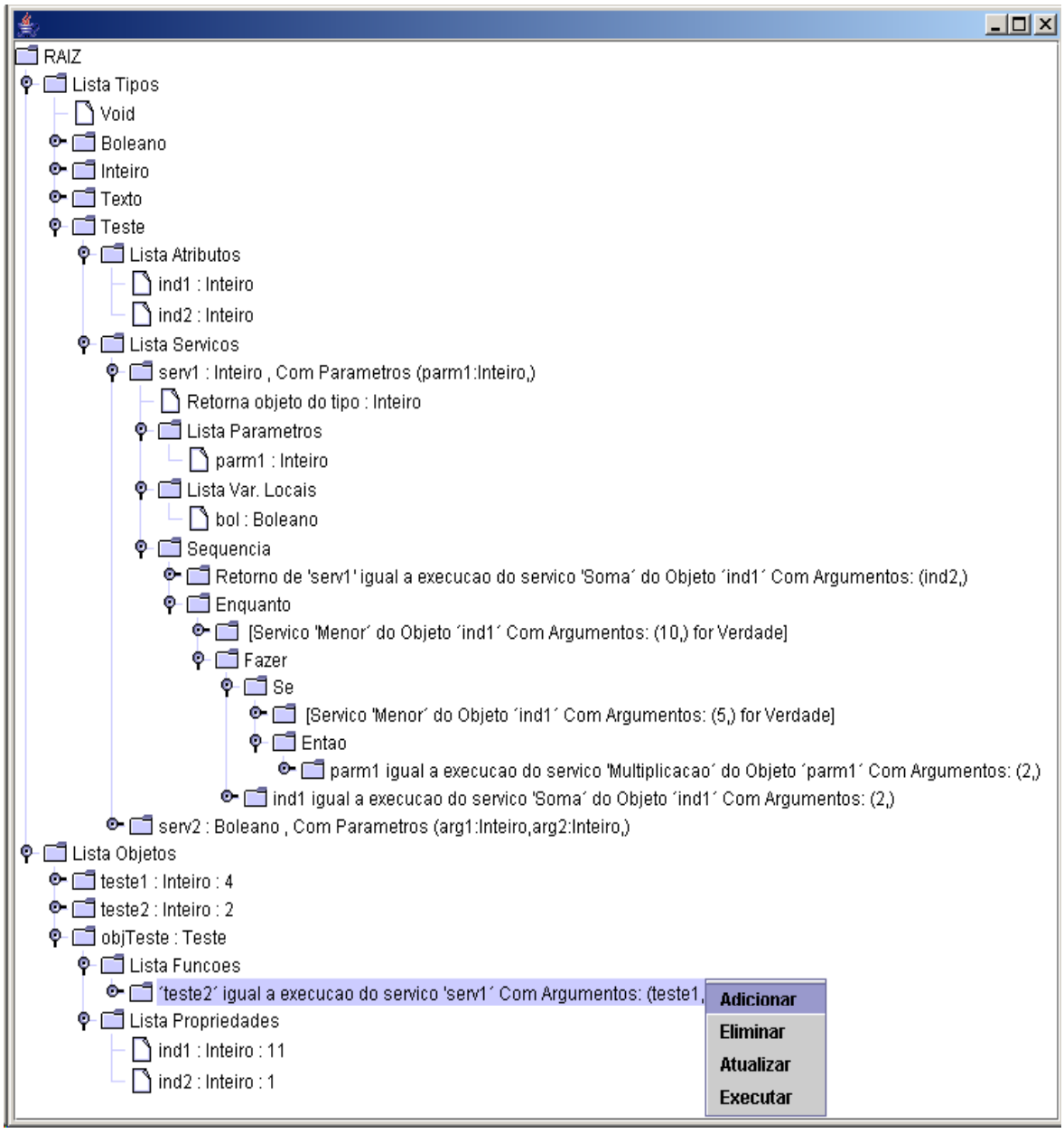

FIGURA 5.1 – Tela do protótipo da ferramenta de edição.

Ao selecionar um item da árvore com o botão direito do *mouse* um menu *popup* é exibido, permitindo que o usuário possa inserir um novo item ou modificar / eliminar o item selecionado e no caso de uma função de teste, executá-la.

Utilizando o ramo “Lista Tipos” o usuário pode adicionar novos tipos de objetos, seus atributos, serviços e procedimentos a serem executados pelos serviços. É também possível modificar ou eliminar um tipo ou alguma de suas propriedades. O ramo “Lista Objetos” permite que o usuário possa adicionar novos objetos dos tipos definidos, definindo o valor de suas propriedades. De modo semelhante ao caso dos tipos é também possível eliminar um objeto ou modificar suas propriedades. Todas estas operações instanciam, eliminam ou modificam objetos na memória e simultaneamente atualizam a base de dados.

Na construção da interface com o usuário, buscou-se atingir o objetivo de proporcionar ao usuário uma ferramenta que minimize a necessidade de conhecimento de programação. O usuário não precisa se preocupar com a sintaxe de uma linguagem de programação porque a ferramenta permite que ele, na maior parte do processo de edição, apenas possa selecionar elementos que estão restritos ao contexto que ele está configurando. Por exemplo, para definir o tipo de um atributo a ferramenta permite que o usuário possa apenas selecionar um dos tipos anteriormente definidos.

A Figura 5.2 é apresentada com o objetivo de exemplificar as facilidades de edição que são oferecidas ao usuário no caso da adição de um novo comando a uma regra. O diálogo representado pela figura é exibido ao usuário após ele ter selecionado um dos comandos da regra em questão e a opção “Adicionar” do menu “*popup*”.

Utilizando o diálogo, caso o usuário selecione um comando simples são ativados os elementos “Objetos Disponiveis”, que é uma “*listbox*” contendo somente os objetos que fazem parte do contexto de entidades visíveis pela regra, e o botão “Sel Objeto” para selecionar um dos objetos desta “*listbox*”. Após a seleção, o campo abaixo deste botão é preenchido com o nome do objeto selecionado (parm1), e a “*listbox*” “Métodos Disponíveis” e o botão “Sel Método” são ativados. A “*listbox*” ativada contém as regras do tipo de objeto associado ao objeto “parm1”. Após a seleção de uma das regras usando o botão “Sel Método”, o campo abaixo deste botão é preenchido com o nome da regra (Divisão), e os demais elementos do diálogo que permitem a definição do retorno e argumentos da regra são ativados. Na definição do retorno o usuário só pode

selecionar um dos objetos que são oferecidos via uma "*listbox*". Os possíveis objetos de retorno oferecidos são apenas aqueles que fazem parte do contexto da regra que está sendo configurada e, ao mesmo tempo, estão associados ao tipo de objeto retornado pela regra "Divisão". No caso da definição dos argumentos ocorre um processo semelhante com a diferença que para cada argumento é apresentada uma "*combobox*" de modo a permitir que o usuário possa selecionar um dos objetos do contexto ou definir o valor caso o parâmetro em questão seja associado a um dos tipos primitivos básicos.

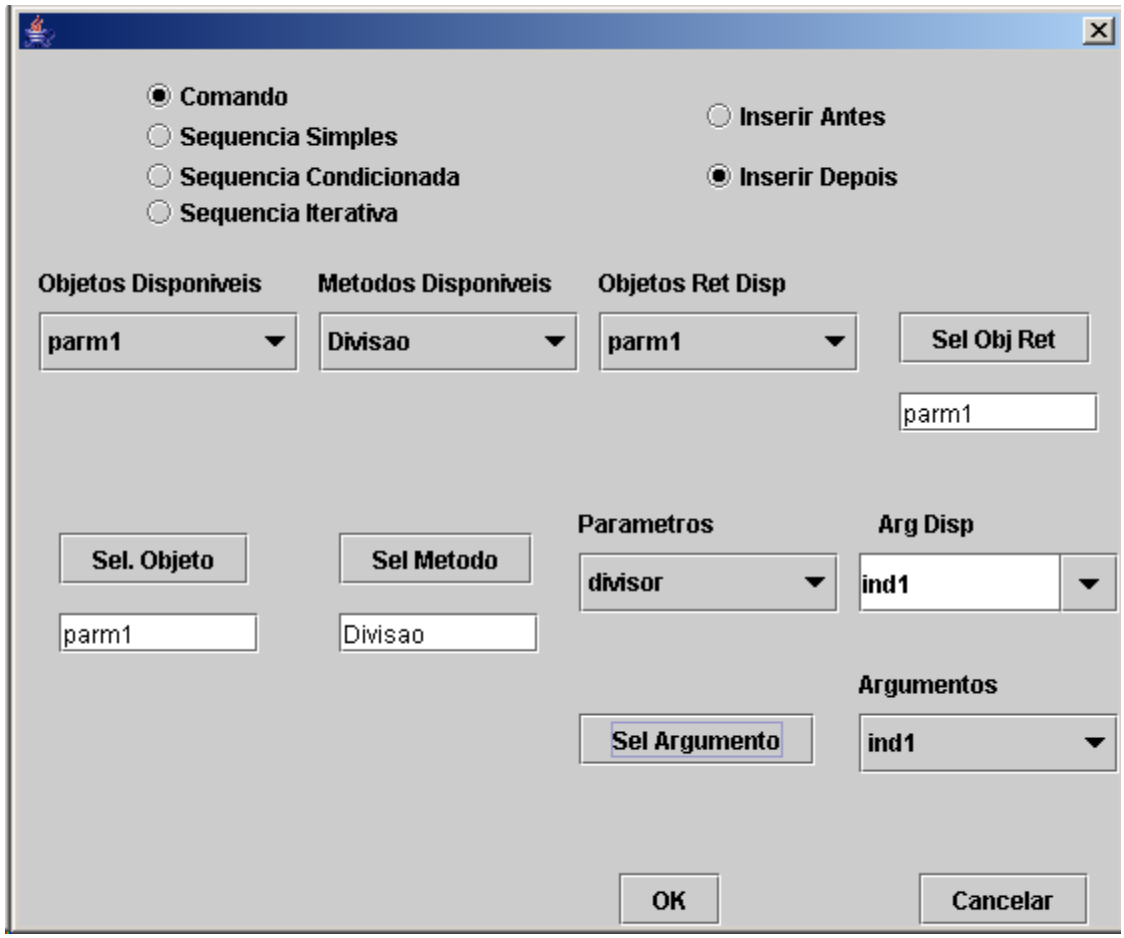

FIGURA 5.2 – Diálogo para adição de um comando.

Não importa que possam ser considerados programação de alto nível, os objetos e as regras inseridas ou modificadas pelo usuário devem ser verificados para comprovar que estão refletindo o comportamento esperado. A ferramenta de edição provê uma facilidade para que os objetos e regras sejam verificados localmente antes de serem

utilizados pelo sistema de controle de satélites. Mas esta facilidade não dispensa os testes de cadeia, ou seja, testes que integram o sistema de controle de satélites e o sistema de objetos dinâmicos, para ter certeza que o conjunto formado por estes dois sistemas esteja funcionando corretamente.

5.2 Exemplo do Uso do Sistema DOM pelo Subsistema de Telemetria

Vamos supor que uma nova missão espacial apresente a necessidade de criar um tipo de monitoração não previsto pelo sistema de controle de satélites. O novo requisito estabelece que é necessário alertar os controladores do satélite sempre que um subsistema consumir mais potência do que a porcentagem máxima prevista em relação à potência disponível no subsistema de fornecimento de energia.

Para utilizar a arquitetura DOM proposta será assumido que:

- O sistema de controle sabe como extrair, dos pacotes de telemetrias recebidos do satélite, os valores dos parâmetros de telemetria correspondentes à potência consumida pelo subsistema em questão e à potência total fornecida pelo subsistema de fornecimento de energia;

- Existe uma classe em Java (ParamTelemetria), que representa um parâmetro de telemetria, é derivada da classe CObjetoPrimitivo, definida no Item 4.3, e é compartilhada pelo sistema de controle e pelo sistema DOM. A classe ParamTelemetria tem um método (ObterValor) que obtém o valor corrente do parâmetro de telemetria. Este método retorna um valor do tipo “Real”.

- Existem duas classes, Boleano e Real, que são também derivadas da classe CObjetoPrimitivo e compartilhadas por ambos sistemas. A classe Boleano representa um valor lógico (*true* ou *false*) e a classe Real representa um número do conjunto dos números reais. A classe Real tem dois métodos: Multiplicação e Maior. Ativando o método Multiplicação de um objeto do tipo Real, é realizada a multiplicação do valor representando por este objeto com outro objeto do tipo Real, que é passado como argumento, e o resultado da operação é retornado

como um objeto do tipo Real. Ativando o método Maior em um objeto do tipo Real é realizada a comparação do valor representado por este objeto com outro objeto do tipo Real, que é passado como argumento, e o resultado da operação é retornado como um objeto do tipo Boleano (*true* se o argumento for menor ou igual ao objeto que recebe a mensagem Maior, ou *false*, caso contrário).

- O sistema de controle conhece a interface com o sistema DOM e prevê a ativação de Objetos Dinâmicos para a inserção de novas regras de monitoração. Esta interface é definida pela seguinte linha de comando Java: "void Executar(String idObjDom, String regraDom, CObjetoPrimitivo valorRetorno, CObjetoPrimitivo[] listaObjetosPrimitivos)". Este comando é responsável pela ativação da regra, identificada por "regraDom", de um objeto dinâmico, identificado por "idObjDom", existente no sistema DOM. Para a execução da regra podem, opcionalmente, ser passados como argumentos um conjunto de objetos primitivos, representados pelo parâmetro "listaObjetosPrimitivos". Se a regra ativada tiver um valor de retorno, ele será recebido pelo sistema de controle através do parâmetro "valorRetorno". No caso de objetos de monitoração o valor de retorno deverá ser um objeto do tipo "Boleano", que retorna o valor *true* se o valor da potência consumida superar o esperado ou *false*, caso contrário.

Tendo em vista estas suposições, para que o sistema de controle possa ativar uma nova regra de monitoração, criada no sistema DOM, é necessário: a) configurar o sistema DOM para introduzir a regra (Item 5.2.1); b) configurar o sistema de controle para acessar a regra no sistema DOM e executá-la (Item 5.2.2).

5.2.1 Configurando o Sistema DOM

Para incluir a nova regra de monitoração no sistema DOM, utilizando a ferramenta de edição, serão executados os seguintes passos:

- A partir da raiz da árvore mostrada pela ferramenta, selecionar o nó "Lista Tipos" e adicionar um novo tipo de objeto utilizando a opção "Adicionar", exibida após pressionar o botão direito do *Mouse*. Como resultado um diálogo será exibido para definir o nome e a descrição do novo tipo (Figura 5.3).

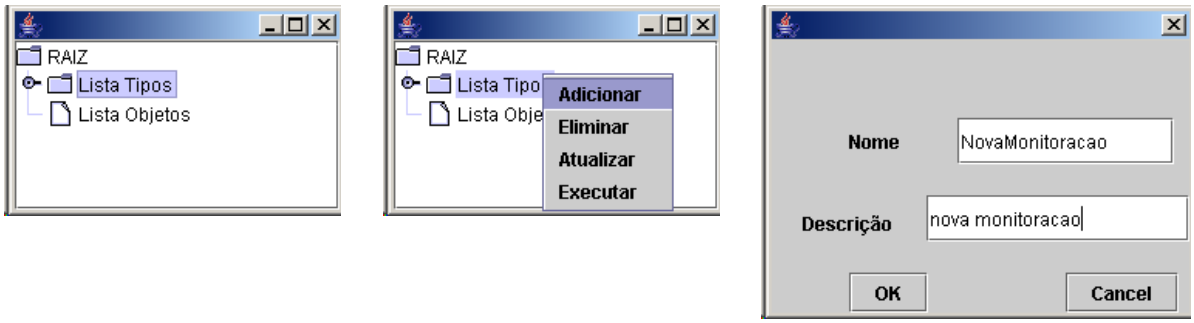

FIGURA 5.3 – Adicionando o novo tipo de monitoração.

- No diálogo exibido como resultado da ação anterior, digitar o nome do novo Tipo (NovaMonitoracao) e sua descrição. Como resultado um novo nó, com o nome "NovaMonitoracao", será incluído na árvore como filho do nó "Lista Tipos". O novo nó será criado com dois nós filhos: "Lista Atributos" e "Lista Serviços" (Figura 5.4).

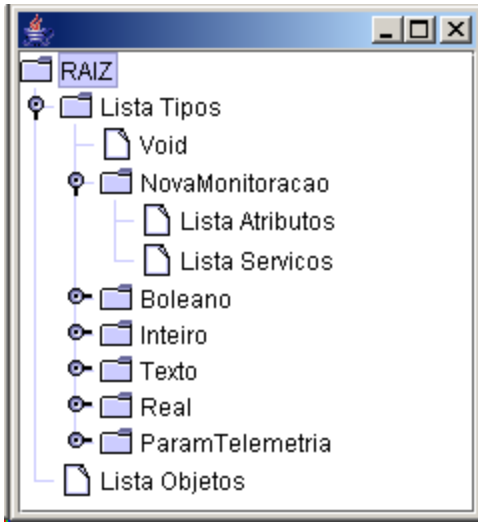

FIGURA 5.4 – Árvore com o novo tipo de monitoração.

- Com a finalidade de definir o valor da porcentagem máxima prevista, em relação à potência disponível no subsistema de fornecimento de energia, será adicionado o atributo "Porcentagem" ao tipo "NovaMonitoracao". Para fazer esta configuração selecionar o nó "Lista Atributos" e, em seguida, utilizando o botão direito, a opção "Adicionar". Como resultado será exibido um diálogo para definir o nome do novo atributo (lado esquerdo da Figura 5.5). No diálogo exibido, digitar o nome do atributo (Porcentagem), selecionar o seu tipo ("Real"), utilizando a lista de tipos disponíveis, e digitar sua descrição. Como resultado um novo nó, com o nome "Porcentagem", será incluído na árvore como filho do nó "Lista Atributos". Na representação do novo nó o tipo do atributo será exibido juntamente com o nome do atributo: "Porcentagem : Real" (lado direito da Figura 5.5).

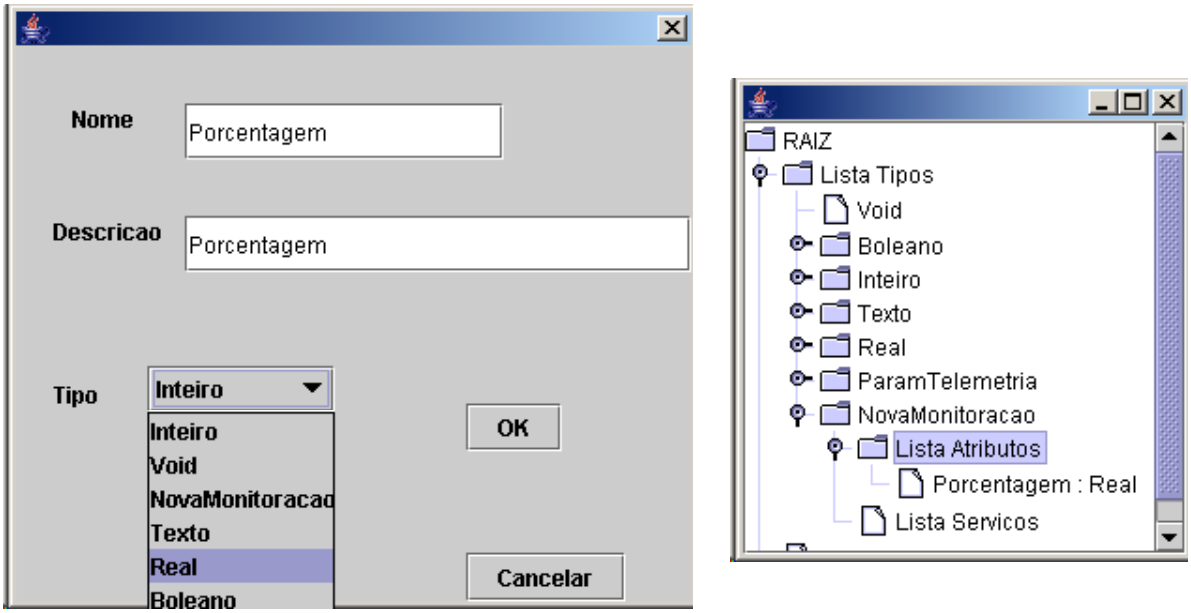

FIGURA 5.5 – Adicionando um atributo ao novo tipo.

- Em seguida a nova regra de monitoração deve ser adicionada ao tipo "NovaMonitoracao". Para fazer esta configuração devem ser selecionados o nó "Lista Servicos" e, em seguida, utilizando o botão direito do *Mouse,* a opção "Adicionar". Como resultado será exibido um diálogo para definir o nome de um novo serviço (lado esquerdo da Figura 5.6). Utilizando o diálogo exibido, digitar o nome do serviço (Monitorar) e sua descrição. Como resultado um novo

nó, com o nome “Monitorar : void”, será incluído na árvore como filho do nó “Lista Servicos”. O novo nó será criado com os seguintes nós filhos: “Retorna objeto do tipo: void”, “ListaParametros”, “Lista Var. Locais” e “Seqüência”, que são utilizados para definir, respectivamente, o tipo do valor retornado pelo serviço, a lista de possíveis parâmetros do serviço, a lista de possíveis variáveis locais e o procedimento a ser executado pelo serviço (lado direito da Figura 5.6).

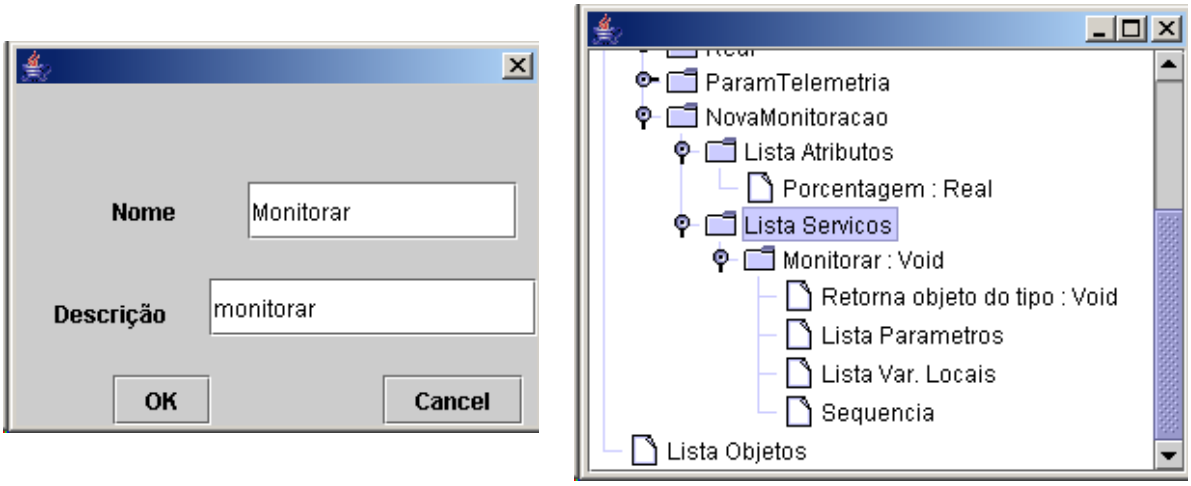

FIGURA 5.6 – Adicionando a regra de monitoração ao novo tipo.

- A regra “Monitorar” deve retornar um valor lógico: “*true*” no caso da potência de um subsistema exceder o máximo previsto ou “*false*”, caso contrário. Desta forma devemos alterar o tipo do retorno de “void” para “Boleano. Para realizar esta mudança é necessário selecionar o nó “Retorna objeto do tipo: void” e, em seguida, utilizando o botão direito do *Mouse,* ativar a função “Atualizar”. Como resultado é exibido um diálogo para redefinir o tipo do valor de retorno (lado esquerdo da Figura 5.7). No diálogo exibido é necessário selecionar o tipo “Boleano”, que é exibido como parte da lista de tipos disponíveis. Como conseqüência o nó que define o tipo de retorno muda sua aparência para “Retorna objeto do tipo: Boleano” e o nó corresponde ao serviço muda sua aparência para “Monitorar : Boleano” (lado direito da Figura 5.7).

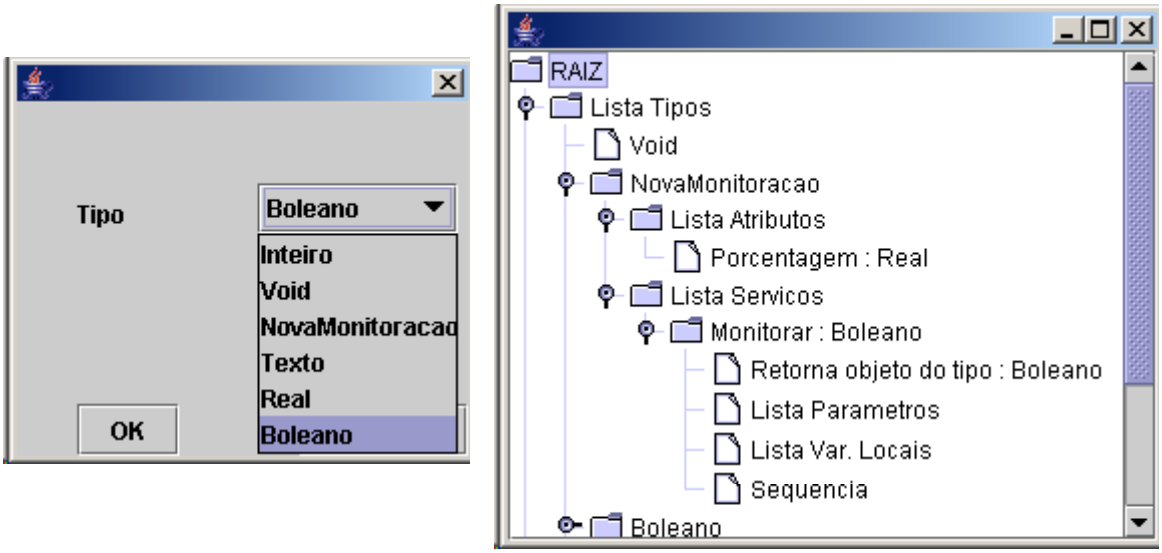

FIGURA 5.7 – Alterando o tipo de retorno da regra de monitoração.

- Em seguida serão definidos dois parâmetros para o serviço "Monitorar": a Potência de um Subsistema do Satélite que será monitorada e a Potência Total disponível no Subsistema de Energia. Estes parâmetros receberão os nomes de "PotenciaSub" e "PotenciaTotal". Para configurar cada parâmetro, deve ser selecionado o nó "Lista Parametros" e, em seguida, a opção "Adicionar", exibida após pressionar o botão direito do *Mouse*. Como resultado será exibido um diálogo para definir o nome do novo parâmetro, seu tipo e sua descrição (parte superior da Figura 5.8). Utilizando o diálogo, digitar o nome do parâmetro (PotenciaSub ou PotenciaTotal), selecionar "ParamTelemetria" para definir seu tipo e digitar sua descrição. Como resultado, novos nós serão incluídos na árvore como filhos do nó "Lista Parametros": "PotenciaSub : ParamTelemetria" e "PotenciaTotal : "ParamTelemetria". Além da inclusão nos novos nós, a aparência do nó correspondente ao serviço será alterada para "Monitorar : Boleano, Com Parâmetros (PotenciaSub:ParamTelemetria, PotenciaTotal:ParamTelemetria,)" (parte inferior da Figura 5.8).

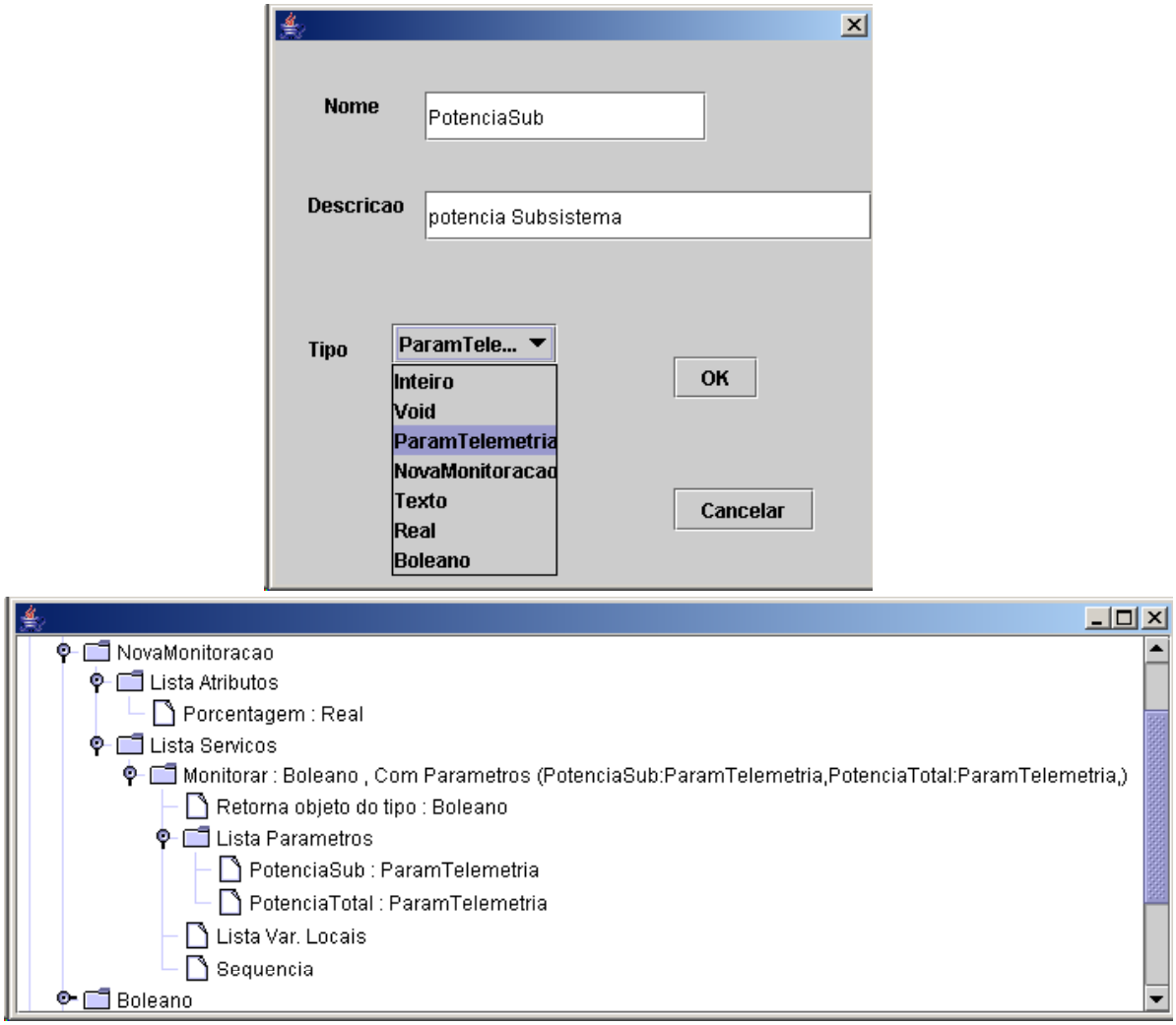

FIGURA 5.8 – Incluindo os parâmetros na regra de monitoração.

- Em seguida serão definidas duas variáveis locais para armazenar temporariamente os valores dos dois parâmetros de telemetria. Estas variáveis deverão receber os nomes “ValPotSubsistema” e “ValPotTotal” e serão definidas como sendo do tipo “Real”. Elas deverão ser criadas como nós filhos do nó “Lista Var. Locais”, segundo um processo semelhante ao realizado para definir os parâmetros do serviço. Como resultado serão criados dois novos nós com as seguintes aparências “ValPotSubsistema : Real” e “ValPotTotal : Real” (Figura 5.9).

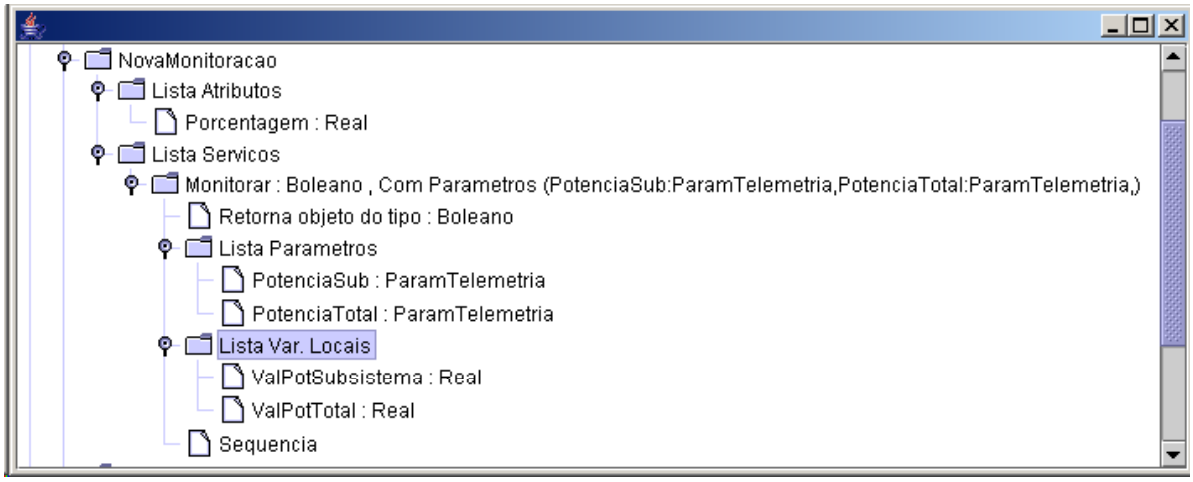

FIGURA 5.9 – Incluindo variáveis locais para a regra de monitoração.

- O próximo passo é adicionar comandos para definir o procedimento do serviço "Monitorar". Isto será realizado através da seleção do nó "Seqüência", filho do nó "Monitorar", e ativação da função "adicionar", que é exibida após pressionar o botão direito do Mouse. Como resultado será exibido um diálogo para a construção do primeiro comando do procedimento (Figura 5.10).

FIGURA 5.10 – Definindo um comando.

- Na *listbox* “Objetos Disponiveis”, do diálogo exibido, estarão incluídos todos os objetos do contexto visível pelo método “Monitorar”: o próprio objeto de monitoração (Este Objeto), atributo (Porcentagem), parâmetros (PotenciaSub e PotenciaTotal), valor de retorno e variáveis locais (ValPotSubsistema e ValPotTotal). Dentre estes objetos será selecionado o objeto correspondente ao parâmetro “PotenciaSub” e em seguida o botão “Sel. Objeto”. Como conseqüência a *listbox* “Metodos Disponíveis” será atualizada com os métodos definidos para objetos do tipo “ParamTelemetria”.

- Selecionar o método “ObterValor” na *listbox* “Métodos Disponíveis” e em seguida o botão “Sel. Método”. Como o método “ObterValor” foi definido com um tipo de retorno igual a “Real” a *listbox* “Objetos Ret Disp” será atualizada com o nome dos objetos do tipo “Real”: “Porcentagem”, ValPotSubsistema” e “ValPotTotal”.

- Na *listbox* “Objetos Ret Disp” selecionar o objeto “ValPotSubsistema”, para receber o valor retornado pelo método “ObterValor”, e em seguida selecionar o botão Sel Obj Ret”. Como conseqüência o campo de edição abaixo deste botão será atualizado com o nome do objeto selecionado. Como o método “ObterValor” não tem nenhum parâmetro a *listbox* “Parâmetros”, que é utilizada na definição de argumentos, não terá nenhum objeto.

- Após a seleção do botão “OK” um nó é criado como filho do nó “Seqüência”. Este novo nó representa o comando preparado nos passos anteriores e tem a seguinte aparência: “ValPotSubsistema igual a execução do serviço ‘ObterValor’ do Objeto ‘PotenciaSub’” (Figura 5.11).

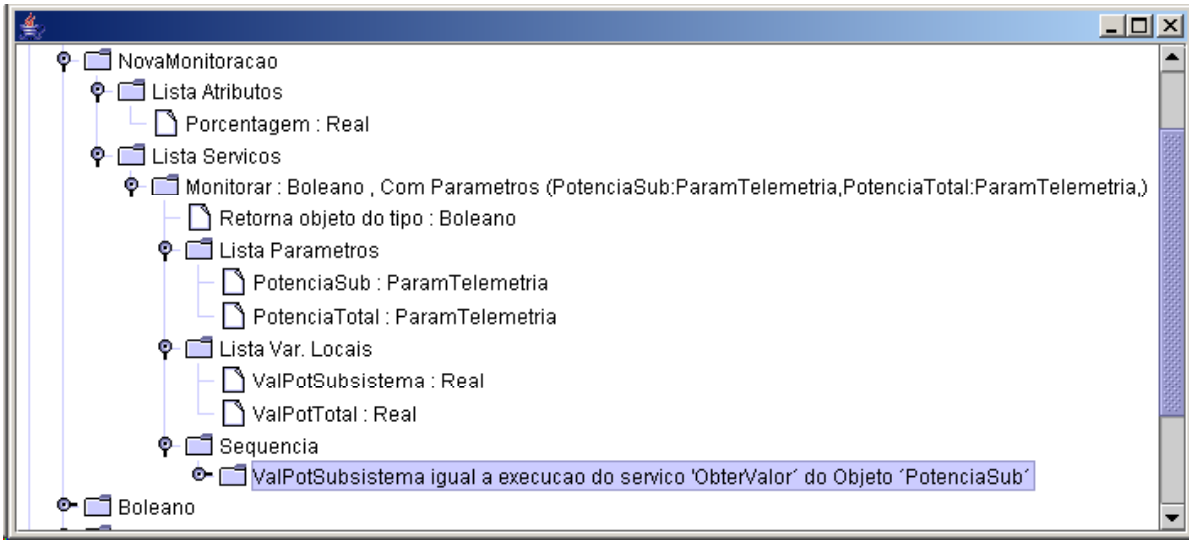

FIGURA 5.11 –Primeiro comando da regra.

- Para obter o valor da potência total deverá ser incluído um novo comando. O processo de criação do novo comando é semelhante ao descrito para obter o valor da potência do subsistema, com as seguintes diferenças: o objeto "PotenciaSub" é substituído por "PotenciaTotal" e o objeto "ValPotSubsistema" é substituído por "ValPotTotal". Como resultado um novo comando é adicionado como filho do nó "Seqüência". O novo nó é adicionado após o nó correspondente ao comando anterior e tem a seguinte aparência "ValPotTotal igual a execução do serviço 'ObterValor' do Objeto 'PotenciaTotal'".

- Em seguida, um terceiro comando deve ser adicionado para obter a porcentagem da potência total que será utilizada na comparação. De modo semelhante ao processo descrito para adicionar o primeiro comando, deverão ser selecionados: o objeto "ValPotTotal", o método "Multiplicacao" do objeto "ValPotTotal" e objeto "ValPotTotal" como valor de retorno.

- Levando em consideração que o método "Multiplicacao" tem como parâmetro o segundo termo da operação de multiplicação, a *listbox* "Parâmetros" exibe a identificação deste parâmetro ("Valor"). Como "Valor" é um objeto do tipo "Real", após sua seleção a *combobox* "ArgDisp" é atualizada com os objetos do tipo "Real": "Porcentagem", "ValPotSubsistema" e "ValPotTotal".

- Selecionado o objeto “Porcentagem” na *combobox* “Arg Disp” e o botão “Sel Argumento” resulta em que a *listbox* “Argumentos” é atualizada com o objeto “Porcentagem”. Após a confirmação da inclusão, através do botão “OK”, o novo comando é adicionado ao nó “Seqüência”. Este terceiro nó tem a seguinte aparência: “ValPotTotal igual a execução do serviço ‘Multiplicacao’ do Objeto ‘ValPotTotal’ com Argumentos: (Porcentagem,)”.

- Para finalizar o procedimento da nova regra de monitoração, um último comando deve ser adicionado para comparar os dois valores obtidos nos comandos anteriores. Utilizando o diálogo de definição de comandos deverão ser selecionados: o objeto “ValPotSubsistema”, o método “Maior” do objeto “ValPotSubsistema” e o valor de retorno correspondente ao retorno da regra “Monitorar” (objeto “Retorno”).

- Levando em consideração que o método “Maior” do tipo “Real” tem como parâmetro o valor de comparação, a *listbox* “Parâmetros” exibe a identificação deste parâmetro (“Valor”). Como “Valor” é um objeto do tipo “Real”, após a sua seleção a *combobox* “ArgDisp” é atualizada com os objetos do tipo “Real”: “Potencia”, “ValPotSubsistema” e “ValPotTotal”.

- Selecionando o objeto “ValPotTotal” na *combobox* “Arg Disp” e o botão “Sel Argumento” resulta em que a *listbox* “Argumentos” é atualizada com o objeto “ValPotTotal”. Após a confirmação da inclusão, através do botão “OK”, o novo comando é adicionado, encerrando a definição do procedimento da nova regra de monitoração. Este último nó tem a seguinte aparência: “Retorno igual a execução do serviço ‘Maior’ do Objeto ‘ValPotSubsistema’ com Argumentos: (ValPotTotal,)” (Figura 5.12).

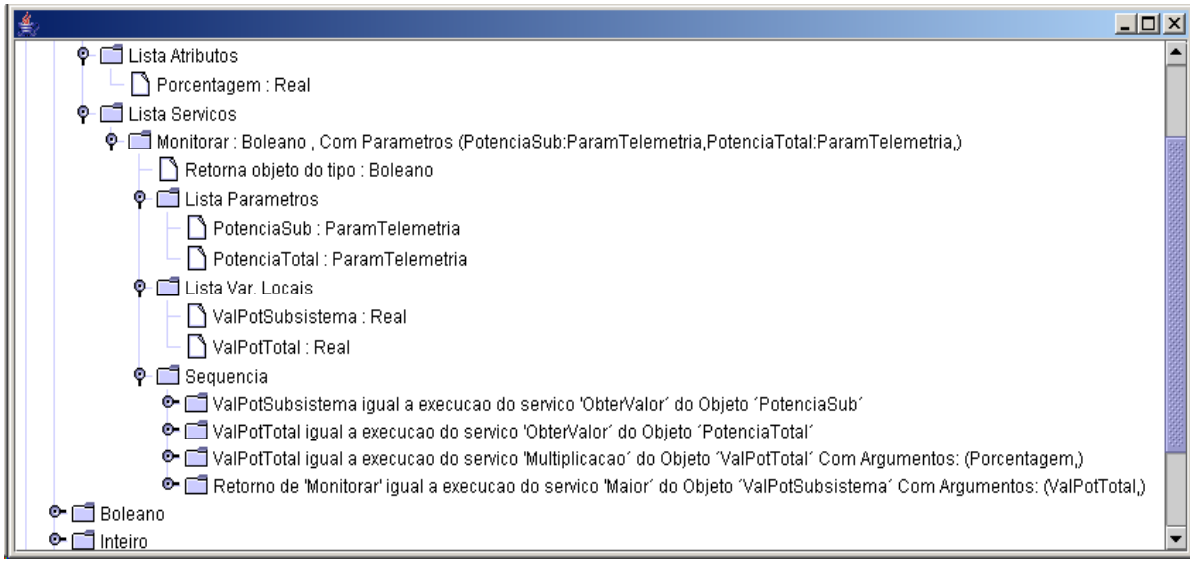

FIGURA 5.12 – Definição do procedimento completo da regra.

- Após a finalização do processo de inclusão de um novo tipo de objeto, para representar a nova regra de monitoração, é possível criar o objeto que será acessado pelo sistema de controle. Para criar este objeto é necessário selecionar o nó "Lista Objetos" e, em seguida, a função "Adicionar", exibida após pressionar o botão direito do *Mouse*. Como resultado será exibido um diálogo para que o usuário possa definir o nome do objeto, qual o seu tipo e uma descrição do objeto.

- No diálogo exibido será definido o nome do objeto, como sendo "ObjNovaMonitoracao", e será selecionado o tipo do objeto, como sendo "NovaMonitoração". Como resultado um novo nó será criado como filho do nó "Lista Objetos". Este novo nó terá a seguinte aparência: "ObjNovaMonitoracao : NovaMonitoracao". Junto com este nó serão criados dois ramos: "Lista Funções e ListaPropriedades". O ramo "Lista Funções" não terá nenhum nó filho" e o ramo "Lista Propriedades" terá um nó filho, correspondente ao tipo de atributo definido para o tipo "Nova Monitoração". O nó correspondente ao atributo terá a seguinte aparência: "Porcentagem : Real : Indefinido", onde "Indefinido" representa que o valor do atributo ainda não foi definido (Figura 5.13).

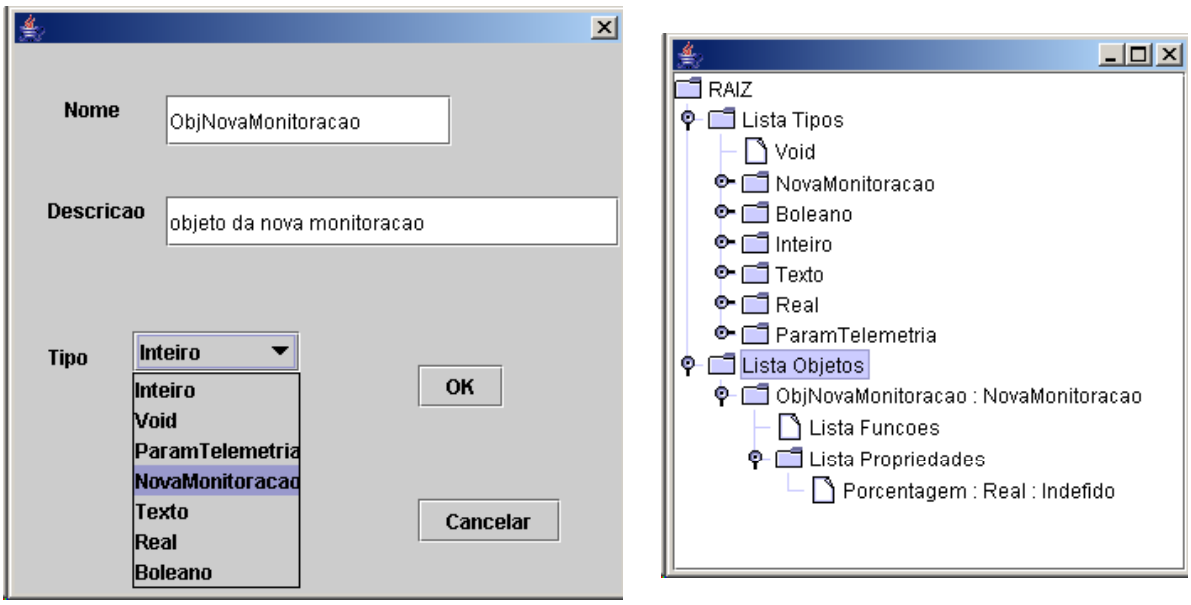

FIGURA 5.13 – Criando o objeto dinâmico de monitoração.

- Para definir o valor do atributo "Porcentagem" é necessário selecioná-lo e, em seguida, ativar a função "Atualizar", após pressionar o botão direito do *Mouse*. Como resultado será exibido um diálogo com o valor corrente do atributo. Modificando o valor do atributo para 0.4 resulta na alteração da aparência do seu nó para: "Porcentagem : Real : 0.4". O resultado final da edição da nova regra de monitoração e criação de um objeto associado a este tipo está ilustrado na Figura 5.14.

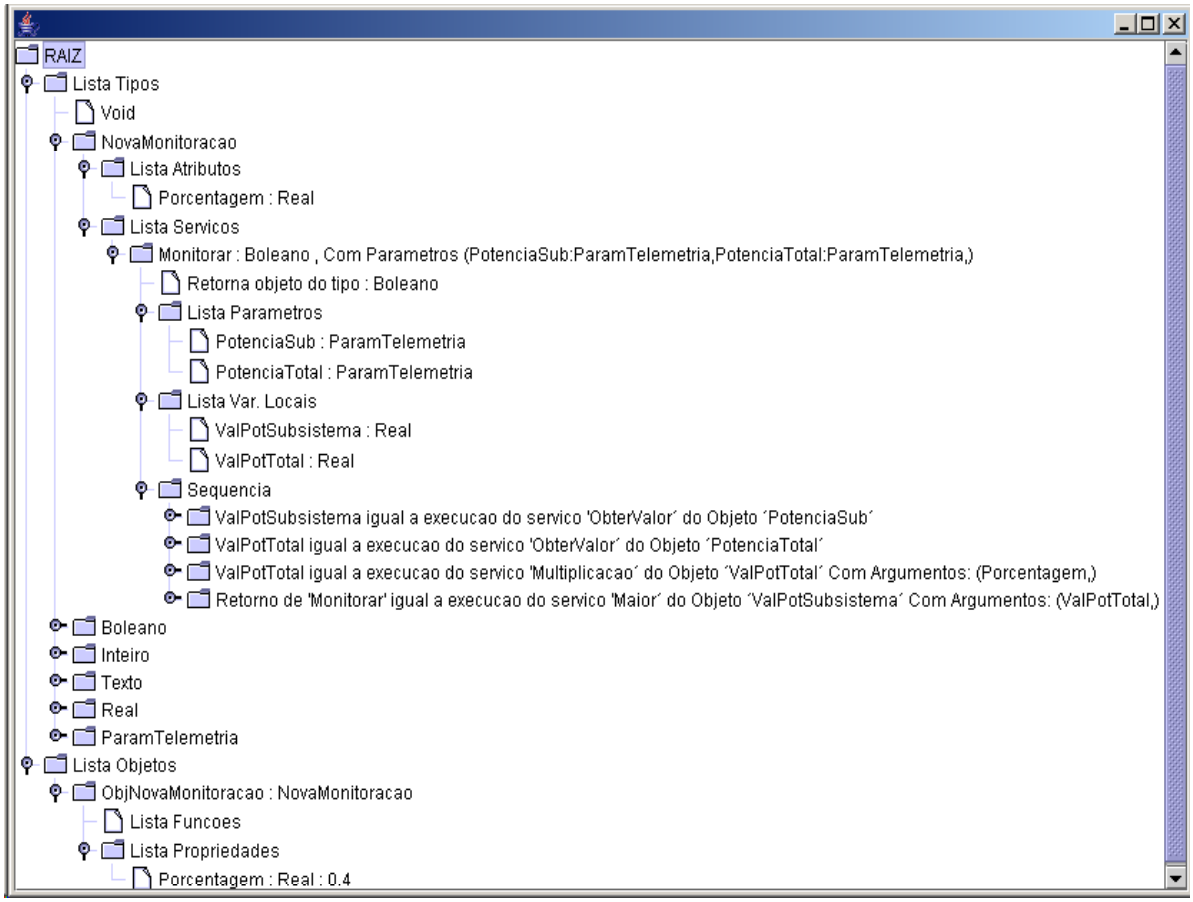

FIGURA 5.14 – Árvore do modelo DOM ao final da edição.

5.2.2 Configurando o Sistema de Controle e Executando a Regra de Monitoração

Para que o sistema de controle tenha acesso aos novos objetos e regras, definidos pela ferramenta de edição do sistema DOM, é necessário configurá-lo. O *framework* do sistema de controle deve permitir que novas monitorações sejam agregadas através de seu módulo de configuração sem necessidade de alterações no código.

A interface externa, definida no sistema DOM, permite que as listas das identificações de todos objetos dinâmicos e suas regras sejam acessadas. O módulo de configuração do sistema de controle deverá usar esta interface para permitir que o usuário configurador possa selecionar as identificações do objeto (“ObjNovaMonitoracao”) e da regra que verifica o novo tipo de monitoração de telemetria (“Monitorar”), que foram definidos através da ferramenta de edição do sistema DOM. Através desta mesma interface

podem ser obtidos os tipos dos parâmetros e do valor de retorno da regra. Este processo corresponde ao trecho de código Java a seguir:

String[] listaIdObjetos = objInterfaceDom.ObterIdentificacoesObjetos(); //obter objetos

// Usuário seleciona identificação do objeto (“ObjNovaMonitoracao”)

String objSelec = listaIdObjetos[i];

String[] listaRegras = objInterfaceDom.ObterRegrasObjeto(objSelec); //obter regras do objeto

//Usuário seleciona identificação da regra (“Monitorar”)

String regraSelec = listaRegras[j];

String idObjRetorno = objInterfaceDom.ObterTipoRetorno(regraSelec); //obter tipo de retorno da regra

//Verifica se tipo é Boleano (tipo definido no *framework* de controle de satélites para o resultado de uma monitoração)

if (idObjRetorno != “Boleano”) //tratamento de exceção

String[] listaParametros = objInterfaceDom.ObterTipoParametros(regraSelec); // obter tipo dos parâmetros da regra

//Verifica se todos os parâmetros são do tipo “ParamTelemetria” (tipo definido no *framework* de controle de satélites para os argumentos de uma monitoração)

for (int i=0;i<listaParametros.length;o++) if (listaParametros[i] != “ParamTelemetria”) // tratamento de exceção

Além disso, o módulo de configuração do sistema de controle deve também permitir que o usuário possa selecionar o contexto que deverá ser visível pelo sistema DOM, ou seja, definir os parâmetros de telemetria que devem ser passados como argumentos para

a execução da nova regra de monitoração pelo sistema DOM. Este processo corresponde ao trecho de código Java a seguir:

```
//usuário seleciona os dois parâmetros de telemetria que serão utilizados na monitoração

ParamTelemetria[] listaArgumentos = new ParamTelemetria[listaParametros.length];

listaArgumentos[0] = listaParamTelemetria.ObterParametroTelemetria(idParam);
//idParam = "PotenciaSubsistema"

listaArgumentos[1] = listaParamTelemetria.ObterParametroTelemetria(idParam);
//idParam = "PotenciaTotal"
```

Realizadas estas configurações, a nova monitoração será agregada ao sistema de controle e conseqüentemente executada durante os futuros processamentos de pacotes de telemetria recebidos do satélite. Este processo de execução corresponde ao trecho de código Java a seguir:

```
Boleano resultadoMonitoracao;

objInterfaceDom.Executar(objSelec, regraSelec, resultadoMonitoracao,
listaArgumentos);

if (resultadoMonitoracao.toString.equals("true")) // gera alarme para o controlador do
satélite
```

CAPÍTULO 6

CONCLUSÃO

Este trabalho buscou pesquisar e implementar uma solução de configuração de regras de negócio pelos próprios usuários, com base no modelo de objetos adaptáveis considerando-se o escopo e as restrições inicialmente estabelecidas:

- Redução do custo e tempo de desenvolvimento dos sistemas de controle para atender requisitos específicos de cada missão espacial;

- Existência de uma equipe de desenvolvimento com longa experiência acumulada no desenvolvimento dos sistemas de controle de satélites;

- Existência de *framework* de controle de satélites razoavelmente maduro composto de um núcleo de processamento bem estabelecido e de pontos de maior probabilidade de mudança, para atender necessidades especificas de cada satélite, bem identificados (Item 2.4);

- Transferência da responsabilidade de pequenas mudanças no sistema de controle para os engenheiros de operação da equipe de controle.

A transferência de mudanças no comportamento dos sistemas de controle para os usuários, ainda que mínimas, é um grande desafio técnico. Este trabalho propôs que esta transferência se reduza a apenas pequenas alterações em pontos já identificados do *framework* de controle de satélites existente. Além disso, para realizar estas mudanças os usuários terão a seu dispor uma ferramenta de edição com uma interface bastante amigável e uma linguagem o mais natural e próxima possível do seu domínio de conhecimento. Embora este trabalho tenha gerado uma ferramenta que facilitou a criação dos Objetos e seus Tipos, é importante esclarecer que a definição da interface com o usuário e da linguagem não foram o objetivo principal.

6.1 Resultados Obtidos

Através do desenvolvimento deste trabalho, e especialmente após a construção do protótipo, pôde-se constatar que o estabelecimento de um modelo de objetos adaptáveis para ser utilizado pelo *framework* do Sistema de Controle de Satélites mostrou-se viável e perfeitamente adequado aos propósitos estabelecidos inicialmente.

Ao longo do desenvolvimento deste trabalho, contudo, pôde-se identificar alguns pontos favoráveis e alguns pontos desfavoráveis da arquitetura proposta, que são detalhados a seguir:

6.1.1 Pontos Desfavoráveis:

- A geração e manutenção do metamodelo são tarefas complexas que exigem um alto grau de abstração, pois se trabalha não com objetos do mundo real, mas sim com uma coleção de objetos genéricos que são usados para compor os primeiros.

- Desenvolvimento da interface gráfica da ferramenta de edição, que deve ser o mais simples possível para que os próprios usuários possam fazer as configurações necessárias, porém não tão simples a ponto de limitar a possibilidade das mudanças serem realizadas por estes usuários.

- As configurações do *framework* do controle de satélites via este modelo se limitam aos pontos mais suscetíveis a mudanças previamente identificados. Caso surjam requisitos de operação que exijam mudanças externas a estes pontos haverá de necessidade de modificar o *framework* de controle de satélites.

- Apesar dos limites definidos pela proposta (utilização do modelo apenas para pequenas modificações em pontos bem identificados do *framework* de controle de satélites; utilização de uma ferramenta de edição com interface gráfica amigável; e linguagem com base em objetos do domínio de conhecimento dos usuários) a configuração exige que estes usuários tenham um conhecimento, ainda que mínimo, de técnicas de programação.

- Embora não tenha sido analisado durante o trabalho, um ponto desfavorável diz respeito ao desempenho do sistema, uma vez que o modelo interpreta as regras armazenadas em um banco de dados. Para minimizar esta dificuldade, no projeto do protótipo adotou-se as seguintes estratégias: a) instanciar todos os objetos durante a ativação do sistema; b) os valores dos objetos somente são atualizados na base de dados ao final da execução completa do procedimento de uma regra ativada pelo usuário.

6.1.2 Pontos Favoráveis:

- O modelo permite que os requisitos específicos de uma nova missão possam ser acomodados através da configuração dos metadados, sem a necessidade de se criar um sistema específico para o satélite a ser lançado, e transferindo a responsabilidade destas mudanças dos especialistas de *software* para os especialistas de operação dos satélites.

- Facilidade de inserção de novos objetos primitivos sem necessidade de alteração da ferramenta, contribuindo para que a evolução do *framework* de controle de satélites possa ser facilmente acomodada pelo modelo. Esta facilidade também permite que o modelo e ferramenta de edição possam ser utilizados por outros domínios de problema pela simples inserção no repositório da definição dos objetos primitivos adequados e disponibilização dos respectivos arquivos *.class* (gerados pelo compilador Java*)*.

- O modelo apresentado é uma contribuição importante para a área de modelos de objetos adaptáveis, uma vez que define, dentro do escopo e restrições estabelecidas, um modelo para representação da regras, uma ferramenta de edição e uma interface de acesso por sistemas externos que constituem um *framework* que pode ser facilmente utilizado por outros domínios de problema, além do domínio de controle de satélites.

- Outra característica importante do modelo apresentado é a possibilidade do usuário, através da ferramenta, criar objetos com base nos objetos primitivos que podem ser utilizadas pelo próprio usuário para construir objetos mais complexos. Isto representa um avanço em relação aos trabalhos pesquisados, que propõem que o usuário possa criar objetos apenas tendo como base objetos primitivos.

6.2 Trabalhos Futuros

A conclusão deste trabalho não encerra, de forma alguma, as considerações a serem feitas em relação ao desenvolvimento de um sistema de objetos configuráveis pelo usuário que possa ser utilizado no Sistema de Controle de Satélites. Na realidade, este trabalho abre uma série de questões que ainda podem ser exploradas, como por exemplo:

- Fornecer condições para garantir a integridade dos objetos do repositório é ainda mais importante quando se considera que os usuários responsáveis pelas mudanças não são necessariamente especialistas em software. É também importante auxiliar este usuário a avaliar as conseqüências de uma mudança.Uma possível solução para prover ao usuário tal auxílio é a utilização do padrão *Observer* nas implementações da ferramenta de edição e na interface com os sistemas externos. Sempre que for criado o relacionamento entre uma entidade do repositório ou um sistema externo com uma segunda entidade do repositório, esta segunda entidade deverá armazenar a identificação da primeira entidade ou do sistema externo. Quando esta segunda sofrer qualquer tipo de mudança as demais entidades ou sistemas externos que lhe são dependentes serão notificados. De maneira semelhante, este mecanismo pode ajudar o usuário a prever o impacto de uma mudança informando ao mesmo quais as entidades que serão afetadas pela mudança antes que a mesma seja efetuada.

- Estudo do problema de desempenho do modelo levando em conta: desempenho de diferentes tecnologias de gerenciadores de banco de dados (relacionais e

orientados a objetos); desempenho dos mecanismos de interpretação de uma regra; desempenho do mapeamento dos objetos do contexto externo para o contexto local de uma regra; estratégia para armazenamento na base de dados dos valores dos objetos à medida que forem sendo modificados durante a execução de uma regra.

- Em casos em que o desempenho seja muito critico pode-se pensar em acoplar à ferramenta de edição uma outra função, que a um simples toque do usuário leia o repositório, gere código em uma linguagem de programação compilável e sem seguida realize sua compilação.

- Criação de um mecanismo para o controle das modificações nos objetos que mantenha no repositório um histórico das mudanças realizadas e que tenha a capacidade de recuperar qualquer versão existente. Este mecanismo é importante para o acompanhamento das modificações realizadas e para que, em caso de problemas em uma versão, seja fácil recuperar uma versão anterior.

- Apesar de não ser uma característica importante no caso dos sistemas de controle de satélites, para outros domínios de problema talvez seja necessário realizar estudos para avaliar a mudança dos objetos em tempo de execução.

- A evolução da interface da gráfica da ferramenta de edição, talvez agregada com ferramentas de projeto que utilizam UML, pode possibilitar ou a transferência da responsabilidade de configuração de regras mais complexas para os usuários ou a disponibilização aos desenvolvedores de *software* de uma ferramenta de projeto que gere código executável a partir dos diagramas UML.

6.3 Considerações Finais

Apesar de muitos aspectos apontarem para o fato de que, pelos menos inicialmente, as arquiteturas baseadas em modelos de objetos adaptáveis exigirem um esforço muito grande para serem construídas, acredita-se com este trabalho tenha sido dado um passo significativo em direção ao aumento da reusabilidade, pelo menos do ponto de vista dos

desenvolvedores de *software* que sejam responsáveis por sistema baseados em *frameworks* maduros. Foi afirmado "Do ponto de vista dos desenvolvedores de software" porque, em caso de necessidade, as pequenas adaptações, já previstas em um *framework,* não são eliminadas mas a responsabilidade por sua execução é transferida para os especialistas do domínio de problema. Além disso, a ferramenta de edição e a interface de acesso ao repositório de objetos constituem um *framework* que pode ser adaptado para diferentes domínios de problemas pela simples adição dos objetos primitivos adequados àqueles domínios.

È importante enfatizar as seguintes contribuições inovadoras para o modelo de objetos adaptáveis alcançadas através deste trabalho: ênfase na utilização do padrão *Property*, gerando uma visão de que "tudo pode ser visto como uma coleção de propriedades"; a abordagem de considerar os objetos adaptáveis como uma árvore acarreta simplificações na implementação dos metadados e na estrutura de classes utilizadas para representar o nível de abstração criado pelo modelo DOM; capacidade do usuário poder criar novos tipos de objetos tendo como base os tipos de objetos primitivos e outros tipos criados pelo próprio usuário. De acordo com o estado da arte da literatura de objetos adaptáveis, que foi pesquisada, o usuário somente pode criar um novo tipo usando os tipos primitivos.

A proposta de utilização de um sistema de objetos adaptáveis para ser utilizado nos sistemas de controle dos satélites, que está sendo defendida através deste trabalho, coincide com a orientação que está sendo seguida, no desenvolvimento da plataforma multi-missão MMP, pelo grupo de tecnologia espacial. Tanto lá como aqui, chegou-se à conclusão que o desenvolvimento de arquiteturas mais flexíveis e configuráveis, apesar de sua complexidade, possibilitam que futuras missões espaciais possam ser realizadas a um custo e prazos menores, compensando o esforço inicial.

As idéias deste trabalho já foram publicadas, até o momento, em Cardoso, Cardoso e Ferreira(2004) e Cardoso e Ferreira(2004b).

Espera-se, através do desenvolvimento deste trabalho, colaborar, mesmo que de forma modesta, para o avanço da pesquisa no Brasil, e para o sucesso da missão espacial

brasileira, oferecendo uma possível solução para redução dos custos e prazos de desenvolvimento dos sistemas de controle de futuros satélites, e, adicionalmente, abrindo novos campos de estudo em direção aos sistemas adaptáveis.

REFERÊNCIAS BIBLIOGRÁFICAS

Arsanjani, A. Rule Object: A Pattern language for adaptable and scalable business rule construction. In: Conference on Patterns Languages of Programs (PloP'2000), 2000, Monticello, Illinois, USA. **Proceedings…** St Louis: Washington University Department of Computer Science & Engineering, 2000, Technical Report number: wucs-00-29.

Arsanjani, A.; Alpigini, J. Using grammar-oriented object design to seamlessly map business models to component-based software architectures. In: International Symposium of Modelling and Simulation, 2001, Pittsburgh, PA, USA. **Proceedings…** Calgary, Canada: International Association of Science and Technology for Development, 2001, p. 186-191.

Cardoso , P. E.; Gonçalves, L. S. C.; Ferreira, M. G. V.; Ambrosio, A. M. Brazilian experiences in Upgrading a satellite control system. In: IV International Symposium on Space Mission Operations and Ground Data Systems. 1996, Munich, German. **Proceedings…** Darmstadt, Germany: ESA/ESOC, 1996. Disponível na biblioteca digital URLib: <http://www.esoc.esa.de/external/mso/SpaceOps/2_31/2_31.htm>. Acesso em 15 mar. 2005.

Cardoso , P. E.; Gonçalves, L. S. C.; Ambrosio, A. M. Lessons learned in adopting PCs at the Brazilian satellite control center. In: V International Symposium on Space Mission Operations and Ground Data Systems, SPACEOPS98. 1998, Tokyo, Japan. **Proceedings…** Tokyo, Japan, 1998. Disponível na biblioteca digital URLib: <http://track.sfo.jaxa.jp/spaceops98/paper98/track5/5c007.pdf>. Acesso em 15 mar. 2005.

Cardoso, P. E.; Cardoso, L. S. ; Ferreira, M. G. V. Dynamic object architecture applied to satellite control systems. In: Eighth International Conference on Space Operations – SpaceOps 2004. 2004, Montreal, Canadá. **Proceedings…** Montreal, Canadá: American Institute of Aeronautics and Astronautics, 2004.

Cardoso, P. E.; Ferreira, M. G. V. Estrutura de dados e regras de negócio configuráveis pelo usuário final. In: IV Workshop dos Cursos de Computação Aplicada do INPE. 2004, São José dos Campos. **Anais…** São José dos Campos: INPE, 2004.

Chung, L.; Cooper, K.; Yi, A. Developing adaptable software architectures using design patterns: an NFR approach. **Computer Standards & Interfaces.** v.25, n.5, p. 253-260. 2003.

Cunha, J. B. S. **Uma abordagem de qualidade e produtividade para desenvolvimento de sistemas de software complexos utilizando a arquitetura de placa de software – SOFTBOARD.** Tese (Doutorado em Computação Aplicada) – INPE, São José dos Campos. 1997.

Ferreira, M. G. V. **Uma arquitetura flexível e dinâmica para objetos distribuídos aplicada ao software de controle de satélites.** 2001. 244p. (INPE-8602-TDI/787). Tese (Doutorado em Computação Aplicada) – INPE, São José dos Campos. 2001.

Fowler, M. **Analysis patterns:** reusable object models. Reading, Massachusetts: Addison-Wesley, 1997. 357p.

Fowler, M. Using metadata. **IEEE Software.** v.19, n.6**,** p. 13-17, 2002.

France, R.; Ghosh, S.; Song, E.; Kim, D. A metamodeling approach to pattern-based model refactoring. **IEEE Software.** v.20, n.5**,** p. 52-58, 2003.

Gamma, E.; Helm, R. ; Johnson, R. ; Vlissides, J. **Design Patterns: elements of reusable object-oriented software**. Reading, Massachusetts: Addison-Wesley. 1995. 395p.

Gonçalves, L. S. C.; Cardoso, P. E.; Ambrosio, A. M. Satellite Control Center: solution for an Adaptable System. In: IV International Symposium of Small Satellites Systems and Services – SSSS, 1998, Antibes, France. **Proceedings...** Toulouse, France: Centre National d'Etudes Spatiales, 1998. Book - Cote : TL795.4.S27.

Instituto Nacional de Pesquisas Espaciais (INPE). **Multi-mission platform specification.** São José dos Campos: INPE. 2001. **(**Number A822000-PRR-01/03).

Johnson, R.; Wolf, B. Type object. In: Martin, R. C.; Riehle, D.; Buschmann, F. (ed.) **Pattern languages of program design 3.** Reading, Massachusetts: Addison-Wesley, 1998. cap. 4, p. 47-66.

Johnson, R.; Yoder, J. W. The adaptive object-model architectural style. In: IEEE/IFIP Conference on Software Architecture 2002 (WICSA3´02)**,** 2002**.** Canada. **Proceedings…**Trier, Germany: University of Trier, Science Computer Department, 2002. p. 3-27. Disponível na biblioteca digital URLib: < http://dblp.uni-trier.de>. Acesso em 15 mar. 2005.

Killijian, M.; Fabre, J. C. Implementing a reflective fault-tolerant CORBA system. In: 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), 2000, Nürnberg, Germany. **Proceedings…**Washington: IEEE Computer Society, 2000. p. 154-163.

Ledeczi, A.; Bapty T. ; Karsai G. Synthesis of self-adaptive software. In: IEEE Aerospace Conference, 2000**,** Big Sky, MT. **Proceedings…**Piscataway, NJ: IEEE Inc.**,** 2000. v. 4, p. 501-507.

Lee, E. A. Computing for embedded systems. In: IEEE Instrumentation and Measurement Technology Conference, 18. 2001, Budapest. **Proceedings…** Budapest: IEEE, v. 3, p. 1830-1837, 2001.

Maes, P. Concepts and experiments in computational reflection. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), 1987, Orlando, Florida. **Proceedings…** Orlando: ACM SIGPLAN Notices, 1987. v. 22, n.12, p. 147-155.

Manolescu, D. A.; Johnson, R. E. Dynamic object model and adaptive workflow. In: Metadata and Active Object-Model Pattern Mining Workshop (OOPSLA'99), 1999, Denver, Colorado. Disponível em: <http://www.uiuc.edu/ph/www/manolesc/Workflow>. Acesso em: 28 março 2005.

Manolescu, D. "**Micro-Workflow": a workflow architecture supporting compositional object-oriented software development.** PhD Thesis - University of Illinois at Urbana, Champaign. 2000.

Nakamura, H.; Johnson, R. Adaptive framework for the REA accounting model. In: Workshop on Business Object Design and Implementation, 4., (OOPSLA´98), 1998, Vancouver, Canadá. **Proceedings…** Orlando: ACM SIGPLAN Notices, 1998. Disponível em <http://jeffsutherland.org/oopsla98/>. Acesso em 15 mar. 2005.

Nguyen-Tuong, A.; Grimshaw, A. S. Using reflection for incorporating fault-tolerance techniques into distributed applications. **Parallel Processing Letters,** v. 9, n. 2, p. 291-301, 1999.

Perkins, A. Business rules=meta-data. In: International Conference on Technology of Object-Oriented Languages and Systems, 34., 2000, Santa Barbara, California. **Proceedings….**Washington: IEEE Computer Society, 2000. p. 285-294.

Roberts, D.; Johnson, R. Patterns for Evolving Frameworks. In: Martin, R. C.; Riehle, D.; Buschmann, F. (ed.) **Pattern languages of program design 3.** Reading, Massachusetts: Addison-Wesley, 1998. cap. 29, p. 471-486.

Stehling, R. O. **Projeto e implementação de uma arquitetura de software reflexiva para linguagem Xchart.** Dissertação (Mestrado - Instituto de Computação da Universidade Estadual de Campinas) – UNICAMP, Campinas. 1999.

Thomé, A. C. **SICSDA - Uma arquitetura de software distribuída configurável e adaptável aplicada às várias missões de controle de satélites.** Dissertação (Doutorado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2004.

Yamaguti, W.; Vieira, A. E. C.; Oliveira, J. L.; Cardoso, P. E., Costa, P. O. Satellite Control System Nucleus for the Brazilian Complete Space Mission. In: International Symposium on Ground Data Systems for Spacecraft Control, 1990, Darmstadt,

Germany. **Proceedings…** Noordwijk, The Netherlands: ESA Publications Division, 1990.

Yoder, J. W.; Balaguer, F. ; Johnson, R. Architecture and design of adaptive object-models. In: OCM Conference On Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001). 2001, Tampa Bay, Florida. **Proceedings…** Orlando: ACM Sigplan Notices, 2001 v. 36, n. 12, p. 50-60.

Yoder, J.; Johnson, R. **Architecture and Design of Adaptive Object-Models**. São Paulo: IME/USP, 2003. Cursos e Palestras sobre Desenvolvimento de Software Orientado a Objetos realizados no Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP) em 23 e 24 ago. 2003.

GLOSSÁRIO

Comando – linha de comando que faz parte da seqüência de ações de uma regra de negócio (*statement*)

Comando simples – comando que faz uso de um objeto primitivo ou regra de um outro tipo de objeto dinâmico definido pelo usuário.

Comando composto - seqüência iterativa ou condicional de comandos.

Entidade – algo que existe como uma particular e discreta unidade.

Objeto Comando – representa o objeto dinâmico que recebe uma mensagem (comando).

Objeto Dinâmico – nível operacional de uma entidade do mundo real que foi modelada de acordo com a tecnologia de objetos adaptativos.

Propriedades – características que definem o estado de uma entidade.

Regras – características que definem o comportamento de uma entidade.

Regra Comando – identifica qual das regras de um Objeto Comando será ativada.

Retorno Comando – representa o objeto que será atualizado com o valor retornado como resultado da execução de um comando.

Regra de Negócio – uma característica que define ou restringe a estrutura e/ou o comportamento de um negócio.

Tipo de Objeto Dinâmico – nível de conhecimento de uma entidade do mundo real que é modelada de acordo com a tecnologia de objetos adaptativos.