UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

LUCIANA NUNES LEAL

NATURAL MDA: UMA LINGUAGEM DE ESPECIFICAÇÃO DE AÇÕES BASEADA EM LINGUAGEM NATURAL CONTROLADA

Rio de Janeiro

2006

Luciana Nunes Leal

NATURAL MDA: UMA LINGUAGEM DE ESPECIFICAÇÃO DE AÇÕES BASEADA EM LINGUAGEM NATURAL CONTROLADA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática - Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Orientador: Maria Luiza Machado Campos
Co-orientador: Paulo de Figueiredo Pires

Rio de Janeiro

2006

Luciana Nunes Leal

NATURAL MDA: UMA LINGUAGEM DE ESPECIFICAÇÃO DE AÇÕES BASEADA EM LINGUAGEM NATURAL CONTROLADA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática - Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Aprovada em

_____
Maria Luiza Machado Campos, Ph.D., UFRJ

_____
Paulo de Figueiredo Pires, D.Sc., UFRJ

_____
Cláudia Maria Lima Werner, D.Sc., UFRJ

_____
Marcos Roberto da Silva Borges, Ph.D., UFRJ

RESUMO

LEAL, Luciana Nunes. **Natural MDA: Uma linguagem de especificação de ações baseada em linguagem natural controlada.** Rio de Janeiro, 2006. Dissertação (Mestrado em Informática) – Instituto de Matemática - Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2006.

Atualmente, empresas de desenvolvimento de sistemas buscam alcançar maior facilidade de adaptação frente às mudanças tecnológicas, maior produtividade no processo de desenvolvimento e maior qualidade do produto final. As abordagens de desenvolvimento orientadas a modelos são as mais promissoras no sentido de alcançar estes objetivos. No entanto, estas abordagens ainda não possuem mecanismos suficientes para derivar completamente a implementação a partir de modelos em alto nível de abstração. Este trabalho descreve a linguagem Natural MDA, uma linguagem de especificação de ações baseada em linguagem natural controlada, a qual visa complementar o desenvolvimento orientado a modelos, de forma a obter uma especificação precisa e ao mesmo tempo em alto nível de abstração, capaz de ser computacionalmente processável.

ABSTRACT

LEAL, Luciana Nunes. **Natural MDA: An action specification language based on controlled natural language.** Rio de Janeiro, 2006. Dissertação (Mestrado em Informática) – Instituto de Matemática - Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2006.

Currently, software development companies search for greater capability to adapt to technological changes, process productivity and quality of the final product. Model driven development approaches are the most promising initiatives to reach these goals. However, these approaches still lack from adequate mechanisms to fully derive implementation from models described at a high level of abstraction. This work describes Natural MDA language, an action specification language based on controlled natural language, that focuses on complementing model driven development in order to reach precise specifications, besides being at a high level of abstraction and processable by computers.

SUMÁRIO

ÍNDICE DE FIGURAS

1 INTRODUÇÃO

Na evolução da Engenharia de Software, foi sempre um desafio a busca do aumento de produtividade. Acreditando que o aumento do nível de abstração das linguagens de programação influencia no alcance desse objetivo, ao longo dos anos migramos desde as linguagens de máquina até as linguagens de quarta geração. Nessa evolução das linguagens, um dos fatores de sucesso mais relevantes é a geração automática de linguagens de baixo nível a partir das linguagens de mais alto nível.

Desenvolvimento de sistemas orientado a modelos (*Model Driven Development*) (MDD) (KLEPPE; WARMER, 2002) é uma abordagem de desenvolvimento de sistemas onde, ao longo do ciclo de vida do projeto, os modelos são refinados progressivamente até conterem informações suficientes para geração de código. O principal objetivo dessa abordagem é tornar a lógica de aplicação independente da tecnologia da plataforma utilizada.

O sucesso da abordagem MDD depende da capacidade de agregar valor às empresas envolvidas com o desenvolvimento de software. Em geral, essas organizações objetivam o aumento de produtividade e a redução de esforços no desenvolvimento de sistemas. A adoção de padrões de projeto e a independência de tecnologias são fatores que influenciam o alcance desses objetivos.

A adoção da Arquitetura Orientada a Modelos (*Model Driven Architecture*) (MDA) (KLEPPE; WARMER, 2002) constitui uma solução de desenvolvimento de sistemas independente de tecnologia. Essa abordagem é baseada em padrões definidos pelo *Object Management Group* (OMG) (OMG, 2005) e a linguagem de modelagem de sistemas adotada é a *Unified Modeling Language* (UML) (PENDER, 2004).

A noção central do MDA é criar modelos em diferentes níveis de abstração estabelecendo uma ligação entre eles até chegar à implementação do sistema. Alguns destes modelos são independentes de plataforma, enquanto outros são voltados a uma plataforma

específica. Uma plataforma é a especificação de um ambiente de execução para um conjunto de modelos. Exemplos de plataforma incluem: Java (JAVA, 2006), CORBA (CORBA, 2006) e .NET (NET, 2006).

MDA permite desenvolvimento incremental e iterativo, já que os mapeamentos entre modelos podem ser aplicados mais de uma vez. Isto permite expressar cada aspecto do sistema no nível de abstração apropriado mantendo os vários modelos em sincronia. Um mapeamento entre modelos assume um ou mais modelos como entrada e produz um modelo como saída. As regras definidas no mapeamento determinam as transformações entre os modelos.

A partir de modelos em UML padrão, é possível gerar uma parte da implementação, obtendo apenas os esqueletos de código. A capacidade de geração de código pode ser aumentada através da utilização dos mecanismos de extensibilidade da UML (estereótipos e valores etiquetados) (PENDER, 2004). No entanto, esses mecanismos ainda não são suficientes para descrever completamente o comportamento de um sistema. Consideramos comportamento como sendo um conjunto de ações a serem realizadas para a concretização de casos de uso, ou seja, o refinamento e complementação dos casos de uso. Devido a essa limitação, o OMG incorporou a teoria de semântica de ações (MOSSES, 1996) à UML, não se restringindo à nenhuma linguagem de especificação de ações. Com essa incorporação, o OMG visa tornar a especificação de sistemas completamente independente de plataforma.

Como o OMG não recomendou nenhuma linguagem em particular, algumas linguagens de especificação de ações foram propostas, tais como: *Kabira Action Semantics* (KABIRA, 2005), *Object Action Language* (OAL) (BRIDGEPOINT, 2005) e *Action Specification Language* (ASL) (WILKIE *et al*., 2001). Utilizando essas linguagens, consegue-se obter completude na geração de código, já que toda a implementação pode ser gerada automaticamente. Apesar da existência de tais linguagens, elas ainda se apresentam muito

próximas de linguagens de programação, pois necessitam de mecanismos explícitos para declaração de variáveis, atribuição, instanciação, destruição e associação de objetos. Por serem diretamente relacionados a linguagens de programação, esses mecanismos não podem ser considerados de alto nível de abstração e, portanto, não são adequados como linguagem de especificação voltada a projetistas de sistemas e analistas de negócio. Por essa razão, julgamos necessária uma solução em que a especificação do comportamento do sistema fique mais próxima da definição do problema e, ao mesmo tempo, forneça a precisão necessária para ser processada e transformada em código executável.

O objetivo deste trabalho é apresentar a linguagem Natural MDA, que é uma linguagem de especificação de ações, de alto nível de abstração, alinhada aos objetivos da MDA. Desenvolvemos duas ferramentas associadas à linguagem: uma para processamento e integração com as demais ferramentas MDA e outra para auxíliar o uso da linguagem. A contribuição esperada é fornecer um mecanismo que permita aumentar o nível de abstração na especificação de sistemas, em complemento à UML, e, como consequência, reduzir a distância entre o domínio do problema e a linguagem de implementação da solução.

Uma linguagem que se propõe a ser complementar à UML e, ao mesmo tempo, ser mapeada em linguagens de programação, impõe requisitos de alto nível de abstração e processabilidade computacional. Com base nesses requisitos, consideramos a hipótese de que linguagens naturais controladas (ALLEN, 1995) podem ser utilizadas na especificação de sistemas e, posteriormente, transformadas em códigos executáveis. Apesar de haver diversas linguagens naturais controladas (BRYANT; LEE, 2002) (SILVA; PINHEIRO, 2004a), nenhuma delas é alinhada aos objetivos da MDA. Estas linguagens naturais controladas especificam o sistema de forma completa, sem a necessidade de serem acompanhadas por modelos.

Este trabalho encontra-se organizado em cinco capítulos. No capítulo 2, apresentamos as principais características de especificação de sistemas, a abordagem de desenvolvimento baseada em modelos, os recursos da UML já existentes e outras abordagens alternativas para especificação de sistemas. No capítulo 3, descrevemos a linguagem Natural MDA, o impacto da utilização dessa linguagem no processo de desenvolvimento de sistemas e as ferramentas de apoio desenvolvidas para o processamento da linguagem. No capítulo 4, apresentamos o exemplo de aplicação que direcionou o projeto da linguagem e os testes das ferramentas desenvolvidas e, um estudo de observação que visa experimentar a linguagem proposta. Finalmente, no capítulo 5, descrevemos as conclusões e trabalhos futuros. Nos apêndices apresentamos: a modelagem e implementação da extensão de uma ferramenta MDA; os questionários de avaliação da linguagem; e o detalhamento da gramática proposta.

2 DESENVOLVIMENTO ORIENTADO A MODELOS

Sistemas corporativos devem ser modelados para elaborar a "planta" da solução antes de construí-la, além de facilitar a documentação e entendimento por parte dos desenvolvedores e clientes. Isto também se aplica a aplicações menores, mas os benefícios da modelagem são mais evidentes em aplicações maiores devido à complexidade inerente destas.

A modelagem é uma parte essencial dos projetos de software pois permite estruturar a aplicação em módulos, identificando suas responsabilidades, contratos e as dependências entre si. Uma vantagem disso é permitir ciclos paralelos de desenvolvimento e testes para cada módulo.

Neste capítulo, apresentaremos a MDA (uma abordagem de desenvolvimento de sistemas baseada em modelos), os recursos da UML para modelagem e outras abordagens para especificação de sistemas alternativas a UML.

2.1 MODEL DRIVEN ARCHITECTURE

As abordagens mais recentes para o desenvolvimento de sistemas visam alcançar um maior nível de abstração na especificação de sistemas de software. MDA é uma iniciativa do OMG que tem como objetivo prover uma maior separação entre a especificação e a implementação de sistemas (KLEPPE; WARMER, 2002). Na abordagem MDA, os modelos de sistemas são classificados em:

- Modelo Independente de Computação (*Computation Independent Model*) (CIM): é a representação do sistema do ponto de vista independente de computação. Esse modelo foca nos requisitos do sistema, onde os detalhes de estrutura e processamento estão ocultos ou ainda indeterminados. Esse modelo é também conhecido como modelo de domínio.

- Modelo Independente de Plataforma (*Platform Independent Model*) (PIM): é a representação das funcionalidades de um sistema, com foco nas regras de negócio, utilizando-se de uma especificação que não dependa de plataforma.
- Modelo Específico de Plataforma (*Platform Specific Model*) (PSM): é a complementação do Modelo Independente de Plataforma com detalhes tecnológicos específicos da plataforma de implementação do sistema.

A abordagem MDA adotou a UML como linguagem de modelagem de sistemas. A UML é uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de sistemas orientados a objetos (PENDER, 2004). Ela é basicamente composta de elementos e diagramas. Os elementos compõem o modelo do sistema e os diagramas são uma forma de visualizar apenas um subconjunto destes elementos. Através dos diagramas, podemos ter diferentes visões de um mesmo modelo (RATIONAL, 2003).

Os principais diagramas da UML são classificados em dinâmicos e estáticos. Os diagramas estáticos definem os elementos do modelo da aplicação. Os diagramas dinâmicos definem a interação entre esses elementos de modo a obter o comportamento desejado pelo cliente. Atualmente, a UML é a notação mais utilizada na modelagem de sistemas orientados a objetos.

A adoção da UML pela MDA permite que a especificação se mantenha independente de plataforma e possa ser transformada em um modelo específico. Modelos UML podem ser classificados em seis níveis de maturidade (WARMER; KLEPPE, 2002), de acordo com a precisão da especificação:

- Nível 0: Não há documentos de especificação. O sucesso do projeto depende dos desenvolvedores.

- Nível 1: A especificação está em documentos escritos em linguagem natural. Como a linguagem natural é ambígüa, os desenvolvedores tomam suas decisões de negócio baseadas na interpretação pessoal do texto.
- Nível 2: A especificação está em documentos escritos em linguagem natural acompanhada de alguns diagramas para explicar a estrutura.
- Nível 3: A especificação é composta por modelos, tanto diagramas como textos, com sentido bem definido e específico. Os textos em linguagem natural apenas explicam a motivação e o contexto dos modelos e complementam alguns detalhes. Os modelos são a parte mais importante da análise e do projeto. Nesse nível, os desenvolvedores ainda tomam suas próprias decisões, mas com menos influência sobre a arquitetura.
- Nível 4: A especificação é composta de modelos acompanhados de textos em linguagem natural. Os modelos são precisos o suficiente para ter uma associação direta com o código atual, apesar de estarem em níveis de abstração diferentes. Esse nível de modelagem é o objetivo da abordagem MDA atual. Nesse nível, os desenvolvedores não podem tomar suas próprias decisões de projeto. Desta forma, os modelos e a implementação se mantêm sincronizados. O desenvolvimento iterativo e incremental é facilitado pela transformação direta de modelo para código.
- Nível 5: A especificação é baseada somente em modelos. Os modelos são completos, precisos e detalhados. Portanto, são suficientes para geração de código, e nenhuma interferência é necessária para esta transformação. As linguagens em que esses modelos são escritos podem ser vistas como a próxima geração das linguagens de programação.

No seu estágio atual de desenvolvimento, a MDA não explicita como a geração da implementação de um sistema pode ser derivada de modelos UML. Desta forma, foi feita uma pesquisa para levantar abordagens que visam suprir essa lacuna, tendo sido encontrado o

método Shlaer-Mellor (WIERINGA; SAAKE, 1997). Na seção seguinte, detalharemos esse método e veremos como ele se relaciona com a MDA.

2.2 MÉTODO SHLAER-MELLOR

O método Shlaer-Mellor (WIERINGA; SAAKE, 1997) tem como principal objetivo especificar sistemas de forma que as especificações possam ser executadas, validadas e depuradas. Neste método, o desenvolvimento do software começa com a divisão de um projeto de software em domínios, de forma que cada um seja reconhecido como uma unidade simples. Se um domínio é muito grande, o mesmo é quebrado em subsistemas independentes. Em seguida, os domínios são analisados, começando pelo domínio da aplicação, que é responsável por descrever o objetivo principal do sistema. O resultado da análise de um domínio é a criação de um modelo de informações que descreve os elementos do sistema e os relacionamentos entre eles. Modelos de estado são construídos para descrever o comportamento de cada um dos elementos do sistema identificados no modelo de informações. Então, cria-se um diagrama de fluxo de ações, que descreve o processamento necessário em cada um dos modelos de estado. Em nenhum momento a análise especifica como o sistema realiza o processamento. A análise apenas detalha os dados, os passos e a ordem de aplicação dos passos que a aplicação precisa.

A análise do domínio da aplicação resulta em um modelo que descreve por completo o comportamento da aplicação. É possível simular a aplicação usando este modelo e validar a corretude da análise. Este processo de simulação e validação por resultados esperados permite verificar se a aplicação funcionará corretamente.

Como resultado da análise do domínio da aplicação, requisitos e restrições são aplicados no domínio seguinte, o domínio de serviços. O domínio de serviços descreve todas as funções ou serviços que o software deve prover para executar os modelos. A análise deste

domínio se dá da mesma forma que o domínio de aplicação, produzindo um conjunto de requisitos no domínio de serviços.

Uma vez finalizada a análise da aplicação e dos serviços, já é possível começar a examinar a arquitetura do software. A análise do domínio de serviço produz informações sobre os tipos e quantidades de dados que o sistema deve tratar, bem como os eventos e as suas respectivas ordens que afetam a manipulação dos dados. Em função destes requisitos, a análise de arquitetura trata de detalhes como, por exemplo, linguagens de programação, plataformas, sistemas operacionais, organização de dados, controle de fluxo e sincronização.

O passo final no método é traduzir os modelos de processo em código, seguindo as regras criadas pelas estruturas da arquitetura. Em vários casos, ferramentas podem automatizar este passo.

Podemos fazer uma analogia entre o modelo independente de plataforma, definido pela MDA, e o domínio de aplicação, definido pelo método Shlaer-Mellor, pois a idéia de ambos é descrever o objetivo principal do sistema, sem especificar detalhes de processamento. Uma outra analogia é entre o modelo específico de plataforma, definido pela MDA, e o domínio de serviços, definido pelo método Shlaer-Mellor, pois a idéia de ambos é descrever detalhes de implementação.

A partir dos conceitos do método Shlaer-Mellor e como conseqüência da idéia de especificar sistemas de forma completa a partir de modelos, foi criado o conceito de UML Executável (xUML) (MELLOR; BALCER, 2002), que veremos na seção seguinte.

2.3 UML EXECUTÁVEL

UML Executável (xUML) é uma derivação do método Shlaer-Mellor, que foca na execução e especificação de uma solução abstrata. xUML combina a notação da UML com os conceitos de execução do método Shlaer-Mellor. As características de modelo executável e

transformável podem ser vistas como a maior contribuição do método. De fato, a evolução do desenvolvimento de software é uma busca permanente por aumentar o nível de abstração e a xUML surgiu neste contexto.

xUML reúne as idéias de modelo de programação abstrato complementado por um compilador de modelo específico de plataforma que gera código executável. Atualmente, as principais ferramentas que atendem a estes conceitos da xUML são: BridgePoint (BRIDGEPOINT, 2005) e iUML (WILKIE *et al*., 2001). Essas ferramentas foram investigadas e ambas geram apenas código C e C++. Cada uma delas trabalha com a sua própria linguagem de especificação de ações, *Object Action Language* (OAL) e *Action Specification Language* (ASL), respectivamente. Somente a ferramenta iUML oferece uma versão não comercial, a iUML Lite. Essas linguagens de ações possuem declarações como: if-then-else, switch, criação e destruição de objetos, criação de objetos com atribuição, escrita e leitura de atributos, seleção de objetos, criação e destruição de associações entre objetos, navegação entre objetos e invocação de operações.

Mellor et al. (MELLOR *et al*., 1998) explora a idéia de complementar a UML com mecanismos para especificar precisamente semânticas de ações independentes de plataforma, destacando que as vantagens dessa abordagem são possibilitar a verificação da especificação mais cedo e o reuso no nível de domínio. Simulações de sistemas podem ser realizadas a partir de especificações precisas e, após verificar a corretude da especificação, pode-se mapear para diferentes tecnologias de implementação. A seguir, veremos alguns recursos da UML que viabilizam a idéia de Mellor para descrever ações.

2.4 RECURSOS DA UML PARA DESCRIÇÃO DE AÇÕES

A UML oferece recursos para a representação de comportamento, mas ainda não existe um consenso sobre quais são os mais adequados para o contexto MDA. Em (MCNEILE;

SIMONS, 2004), encontramos uma classificação destas técnicas e uma discussão das vantagens e desvantagens de cada uma.

A UML divide o universo de artefatos de modelagem em modelos estruturais e modelos comportamentais. Uma parte considerável do código de infra-estrutura de uma aplicação pode ser gerada a partir dos modelos estruturais. Existem fornecedores que oferecem ferramentas que criam esqueletos de código, mas a lógica de negócio deve ser feita pelos programadores. Em casos de operações relativamente simples, é possível gerar parte da implementação das operações a partir de pré e pós-condições OCL (KLEPPE; WARMER; BAST, 2002). Na próxima seção, veremos como a OCL contribui na especificação de ações e suas limitações.

2.4.1 Object Constraint Language

O primeiro passo da UML no sentido de aumentar a expressividade foi incorporar a *Object Constraint Language* (OCL) (KLEPPE; WARMER, 2000), desde o início, como uma maneira de definir restrições de negócio de forma clara e não-ambígua. Com a OCL, temos a especificação precisa de invariantes, pré e pós-condições e condições de guarda em diagramas de estados.

Restrições em OCL são condições semânticas em um ou mais valores de um modelo orientado a objetos, expressas por sentenças em linguagem textual. As restrições expressam situações, detalhes ou regras que não podem ser representados com as construções visuais da UML. Invariantes são um tipo de restrição que estabelece uma condição que deve ser sempre atendida por todas as instâncias do modelo. Assim, temos modelos mais completos e robustos. A seguir, veremos alguns exemplos de invariantes.

A Figura 1 ilustra um exemplo de diagrama UML que representa a modelagem simplificada de um sistema de controle de vôo de uma compania aérea. Apenas com esta modelagem, não é possível representar as seguintes restrições:

- A quantidade de assentos disponíveis é igual à quantidade de assentos do avião menos a quantidade atual de passageiros do vôo;
- O número de passageiros de um vôo deve ser menor ou igual à quantidade de assentos do avião correspondente.

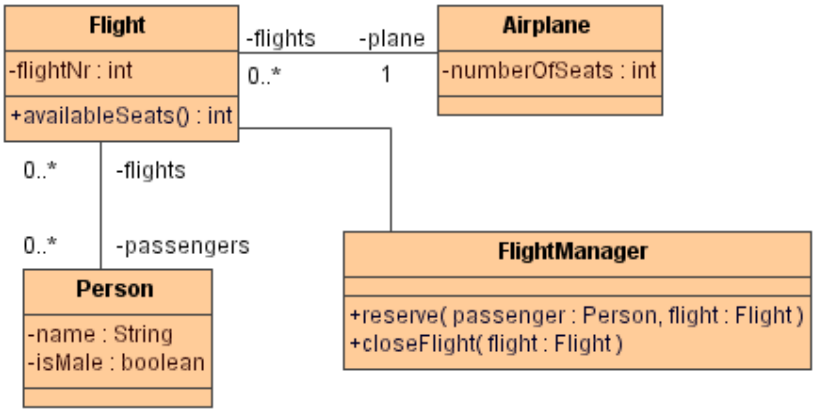


Figura 1. Modelagem de um sistema de controle de vôo

A Tabela 1 mostra como as restrições listadas anteriormente são descritas usando a OCL.

Tabela 1. Restrições OCL do sistema de controle de vôo

```
Context Flight
Inv: passengers->size() <= plane.numberOfSeats

Context Flight::availableSeats() : Integer
Body: plane.numberOfSeats – passengers.size()
```

Entretanto, não conseguimos representar em OCL a especificação das ações do método “reserve” que pode ser descrito como: “Verifique se há assento disponível no vôo em questão. Se houver, verifique a forma de pagamento. Se a forma de pagamento for autorizada, efetue a reserva.”.

A OCL não é completa na definição precisa de ações, pois ela expressa apenas aspectos estáticos do sistema. A OCL não possui mecanismos para expressar a ocorrência ou disparo de eventos e ações do tipo estímulo-resposta. Nos processos de modelagem

tradicionais (RATIONAL, 2003), a utilização de descrições textuais é a única forma de capturar essas ações e eventos, mas isto acarreta em problemas de ambigüidade (KLEPPE; WARMER, 2000). Devido a essa limitação, o OMG incorporou a teoria de semântica de ações na UML – denominada *Precise Action Semantics for UML* (MELLOR *et al*., 1998), descrita na próxima seção.

2.4.2 Linguagens de especificação de ações

A proposta da semântica de ações é permitir a especificação de sistemas em detalhe suficiente para que esta seja transformada em uma implementação de uma plataforma específica e ser executada (PENDER, 2004). A semântica de ações define construções simples e cada linguagem de ações, em particular, pode implementar cada uma dessas construções ou fazê-lo em um nível mais alto; ou seja, compondo construções simples. A abordagem de definir primeiro a semântica e posteriormente a sintaxe de ações tem a vantagem de focar em questões críticas de semântica. Uma vez que haja acordo na semântica, as notações e sintaxes para ações podem ser definidas.

O principal objetivo das linguagens de ações é complementar as especificações das operações com detalhe suficiente para permitir verificação de modelos e traduzi-los em código (OMG, 1999). O OMG determinou que as linguagens de ações devem seguir um padrão da indústria, permitir uma especificação completa e processável, permitir verificação através de simulação e gerar código completo a partir de modelos UML.

Com o objetivo de estabelecer o alinhamento à MDA, o OMG também impôs que as linguagens de ações devem ser compatíveis com a UML, executáveis, completas, independentes de implementação e manter o nível de abstração acima da implementação.

Apesar do OMG ter incorporado semântica de ações na UML, ele não especificou e nem recomendou alguma linguagem para especificação de semântica de ações. Visando preencher essa lacuna, alguns autores anunciaram suas próprias linguagens de ações, como a

Kabira Action Semantics (KABIRA, 2005), a *Object Action Language* (BRIDGEPOINT, 2005), a *Action Specification Language* (ASL) (WILKIE *et al*., 2001) e a *Java like Action Language* (JAL) (DINH-TRONG, 2005). Essas linguagens atendem aos três primeiros requisitos, mas não estão em um nível de abstração acima da implementação. Essas linguagens têm construtores especialmente relacionados à programação tais como: leitura e modificação de atributos, declaração de variáveis e manipulação de objetos. A Tabela 2 e a Tabela 3 mostram exemplos de uma especificação usando ASL onde esses contrutores estão presentes. As linhas que iniciam com "#" contém apenas comentários.

Tabela 2. Exemplo de manipulação de objetos em ASL

```
#create an instance of Person
person = create Person with name = "Luciana"
#set an attribute of person
person.isMale = false
#delete the person
delete Person where name = "Luciana"
```

Tabela 3. Exemplo de seqüência lógica em ASL

```
for <local variable> in {<set of instances>} do
   if <condition 1> then
      <statements>
      loop
         <statements>
         breakif <condition 2>
      endloop
   else
      statements
   endif
Endfor
```

Com o objetivo de também oferecer um mecanismo de especificação de ações, foi proposta uma extensão da OCL para contemplar ações, dado que a OCL não é capaz de atender a todas as características de semânticas de ações (principalmente, especificação de eventos e chamadas de operações) (KLEPPE; WARMER, 2000). Essa extensão da OCL preocupou-se em manter a compatibilidade com a semântica da UML e evitar sobreposição de recursos. Desta forma, os projetistas podem especificar requisitos dinâmicos de forma declarativa. Essa extensão atende às regras de negócio do tipo estímulo-resposta e consiste

basicamente da inclusão da cláusula *action*. A cláusula *action* é composta de três partes (chamada de sintaxe abstrata):

- *TargetSet*: um conjunto de instâncias-alvo do evento
- *EventSet*: um conjunto de eventos a serem enviados às instâncias-alvo
- *Condition*: uma condição para que os eventos ocorram

Como sintaxe concreta, os autores propuseram a seguinte:

```
action: if <condition> to <targetSet> send <eventSet>
```

Usando como base o modelo UML da Figura 1, a Tabela 4 mostra um exemplo desta extensão da OCL.

Tabela 4. Exemplo de OCL incluindo ações

```
Context FlightManager
Action: if flight.availableSeats()==0 to self send closeFlight(flight)
```

Como podemos observar na Tabela 4, essa linguagem de ações ainda não está em um nível de abstração desejado pelo OMG; ou seja, ainda está muito próxima das linguagens de programação. As especificações de ações não devem se restringir a detalhes específicos de linguagens de programação se a intenção for alcançar uma maior separação entre a especificação e a implementação de um sistema. Outro ponto a ser considerado é a fraca aceitação da OCL na especificação de sistemas. A OCL tem sido mais utilizada na especificação de metamodelos (CORREA; WERNER, 2004).

Com vistas nesta separação entre a especificação e a implementação de sistemas, a precisão do modelo não deve depender do nível de detalhe de implementação, ou seja, o modelo deve ser preciso ainda no nível da especificação. As regras de negócio devem ser especificadas sem que o especialista de domínio tome decisões relativas à implementação do software. Mellor *et al*. (1998) defendem que linguagens de especificação de ações devem

mapear diretamente os conceitos da UML que são apropriados no nível necessário para a especificação, devendo ser um subconjunto das linguagens de programação existentes.

Consideramos que as transformações entre modelos não apresentam muitas vantagens quando eles estão no mesmo nível de abstração do código. Por exemplo, as linguagens de especificação de ações citadas neste trabalho utilizam os mesmos conceitos que encontramos no código: parâmetros, classes, retornos, atribuições, entre outros. Este fato não ocorre com a UML pois, apesar dela ser especialmente voltada a elementos de modelagem orientada a objetos (classes, atributos e operações), ela é baseada em uma representação gráfica de mais alto nível que esconde detalhes relacionados à sintaxe de programação.

Apesar de existirem iniciativas no sentido de implementar semântica de ações, que já possibilitam especificações mais precisas, o nível de abstração dessas especificações está especialmente voltado às linguagens de programação. Como a especificação das funcionalidades dos sistemas é papel do projetista, que não é obrigatoriamente um especialista em programação, torna-se necessária a definição de uma linguagem mais adequada à fase de projeto de sistemas. Nas seções seguintes, veremos algumas abordagens alternativas que possibilitam especificar sistemas em um nível de abstração acima da implementação.

2.5 ALTERNATIVAS PARA DESCRIÇÃO DE AÇÕES

Nesta seção, apresentaremos as principais características, vantagens e desvantagens de algumas alternativas para especificação de ações: Linguagens Específicas de Domínio, Programação Orientada a Domínio, Programação por Usuários Finais e Linguagens Naturais Controladas.

A afirmativa seguinte nos leva a acreditar que linguagens específicas de domínio são uma boa alternativa à UML: “In the longer term, defining domain-specific languages (DSLs)

using the MDA framework is likely to be an important alternative to the UML." (MELLOR *et al*., 2004)

2.5.1 Linguagens específicas de domínio

Uma linguagem específica de domínio é uma linguagem processável por máquina cujos termos são derivados de um domínio particular (área de negócio), que é usada para definição de componentes daquele domínio.

Linguagens específicas de domínio auxiliam os desenvolvedores na representação dos sistemas utilizando conceitos familiares aos analistas de negócio. Essas linguagens emergiram como uma forma de reduzir a complexidade de projetos de desenvolvimento de software através de um vocabulário natural para os conceitos do domínio do problema. Essas linguagens, ao invés de focarem em problemas tecnológicos, como a segurança, o armazenamento de dados e as linguagens de programação tradicionais, focam no problema a ser resolvido.

As principais vantagens das linguagens específicas de domínio são:

a) permitir a conservação e reutilização do conhecimento do domínio;
b) possibilitar a expressão de soluções no idioma e no nível de abstração do domínio do problema;
c) possibilitar a validação dos programas no nível do domínio.

Existem algumas ferramentas para definição de linguagens de modelagem específicas de domínio. A ferramenta MetaEdit+ (TOLVANEN; ROSSI, 2003), um dos módulos do ambiente MetaCase (METACASE, 2005), permite construir tais linguagens e geradores de código específicos da aplicação. Nessa ferramenta, um especialista define a linguagem específica de domínio como um metamodelo, contendo os conceitos, propriedades, relacionamentos, símbolos e regras do domínio e especifica o mapeamento, em linguagem de transformação proprietária, desses elementos para código. Com a linguagem criada, a equipe

pode iniciar a modelagem da aplicação e gerar código. As linguagens criadas nessa ferramenta podem ser consideradas como uma UML específica para um determinado domínio. No entanto, esta ferramenta é adequada para modelar domínios e não comportamentos - ela não possui recursos para especificar ações.

A Microsoft também desenvolveu uma ferramenta visual para facilitar a criação de linguagens específicas de domínio, chamada *Visual Studio 2005 DSL Tools* (DSLTOOLS, 2006). Nessa ferramenta, o desenvolvedor define os conceitos, a notação da linguagem específica de domínio e as transformações do modelo criado nessa linguagem em artefatos. O produto final da *DSL Tools* é um plugin para o *Visual Studio 2005*, onde os desenvolvedores modelam suas aplicações respeitando a notação da linguagem e têm a opção de disparar a transformação dos artefatos.

Apesar da modelagem específica de domínio alcançar maior abstração, maior produtividade, independência de plataforma e redução da distância entre os termos de negócio e a solução do problema, o projeto de uma linguagem implica em aumento de custos porque requer:

- Definição do domínio do problema: Nesta fase é feita uma análise abrangente do domínio para oferecer uma definição completa e precisa de todos os conceitos do domínio do problema.
- Definição do escopo do problema: Nesta fase é feita uma delimitação do escopo do domínio do problema. Essa delimitação pode levar ao encontro de conceitos irrelevantes, redundantes, variáveis, entre outros.
- Formulação da linguagem: Nesta fase é feita uma avaliação formal da sintaxe e semântica da linguagem a ser definida.

A técnica de programação orientada a domínio utiliza linguagens específicas, que incorporam diretamente abstrações do domínio, para permitir que usuários finais expressem

suas necessidades de forma computacional (THOMAS; BARRY, 2003). O objetivo é aproveitar o conhecimento dos desenvolvedores de aplicação, que não necessariamente têm formação em Ciência da Computação.

Esta iniciativa também visa aproximar os termos da linguagem dos conceitos do domínio, assim como nas linguagens específicas de domínio. Pode-se considerar que linguagens de programação orientadas a domínio são utilizadas como base para implementar linguagens específicas de domínio (THOMAS; BARRY, 2003). Esse autor defende a idéia de que quando um domínio é mapeado para a UML (que não tem domínio de aplicação específico) e, posteriormente, para código, há uma perda de conhecimento na tradução. Ele sugere que o domínio seja mapeado para uma linguagem de domínio e então para código.

A técnica de programação orientada a domínio está diretamente relacionada à utilização de linguagens específicas de domínio. Logo, a mesma desvantagem dessas linguagens (custos relacionados à formulação da linguagem) vale para essa técnica de programação.

2.5.2 Linguagem natural controlada

Uma linguagem natural controlada é um subconjunto de linguagens naturais que pode ser processado por computador e ainda é expressivo o suficiente para permitir o uso intuitivo por não especialistas. Uma linguagem natural controlada visa reduzir ou eliminar a complexidade e a ambigüidade existentes na linguagem natural (ALTWARG, 2000). Apesar do termo "linguagem natural controlada" ter uma certa contradição porque o termo "natural" implica em não ser "controlada" e vice-versa, optamos por utilizá-lo pois ele é amplamente encontrado na literatura.

Neste trabalho, estamos interessados em melhorar a legibilidade por humanos e possibilitar o processamento computacional. Os elementos de uma linguagem controlada são os mesmos de uma linguagem natural: palavras, regras e pontuação. A linguagem controlada

prescreve estes elementos de maneira limitada e formal através de uma gramática mais restrita do que a gramática das linguagens naturais.

Alguns trabalhos propõem o uso de uma linguagem natural controlada como uma linguagem de especificação declarativa específica de aplicação com o objetivo de reduzir as diferenças conceituais entre o domínio da aplicação e o desenvolvimento de software. Assim, essas especificações podem ser automaticamente transformadas em alguma linguagem de programação, tornando-se portanto, executáveis.

Alguns autores defendem a idéia de que linguagens de especificação formais podem eliminar os problemas associados à linguagem natural, tais como ambigüidade e imprecisão. No entanto, devido à necessidade de comunicação e compreensão entre os membros de uma equipe de desenvolvimento, nem sempre podemos substituir documentos escritos em linguagem natural por especificações formais. Como forma de melhorar a qualidade das especificações sem perder a legibilidade, alguns trabalhos propõem linguagens naturais controladas com sintaxe bem definida (SILVA; PINHEIRO, 2004a) (LINHALIS; MOREIRA, 2005) (BRYANT; LEE, 2002) (SCHWITTER; FUCHS, 1996) (HARS; MARCHEWKA, 1996) (LIU; LIEBERMAN, 2005).

A utilização de linguagem natural controlada é fundamental na área de programação por usuários finais (também conhecida como EUP – *End User Programming*), cujo objetivo é permitir que os usuários finais modifiquem suas aplicações. Nessa abordagem, não é necessário que o usuário tenha a priori qualquer conhecimento de programação ou do funcionamento interno de um sistema computacional.

Em (SILVA; PINHEIRO, 2004a), o autor buscou a obtenção de uma linguagem-tipo na forma de uma linguagem natural controlada para programação por usuários finais. Um ponto importante nesse trabalho é a identificação de dois tipos de conhecimento: o procedimental (quando especificam o processo a ser executado sobre os recursos) e o

declarativo (quando declaram os recursos que podem ser utilizados no processo). O procedimental corresponde aos comandos das linguagens de programação e o declarativo corresponde à definição de variáveis de um programa ou da ontologia de um domínio. Apesar da análise desse trabalho referir-se à lingua inglesa, os comentários e conclusões são válidos para outras línguas, já que foram analisados mecanismos lingüísticos universais. O autor tratou as expressões lógicas mas não as expressões aritméticas, pois acredita que estas variam de acordo com o domínio.

Existem alguns trabalhos na literatura que exploram mecanismos de representação e comunicação de conhecimento. Esses trabalhos são geralmente baseados no uso de figuras de linguagem e estruturas lingüísticas comumente usadas pelas pessoas (SILVA; PINHEIRO, 2004b). Entre esses trabalhos, podemos citar a proposta de um *framework* baseado no uso de uma linguagem natural controlada como uma linguagem-tipo para a instanciação de linguagens específicas de domínio (SILVA; PINHEIRO, 2004a). O objetivo desse *framework* é facilitar a aprendizagem e o uso de linguagens de programação. No *framework*, os mecanismos linguísticos são limitados para torná-lo eficiente no âmbito computacional. Esse *framework* oferece um parser e um processador para uma linguagem-tipo (com sintaxe fixa) de controle e referenciação a objetos (SILVA; PINHEIRO, 2004b). Os projetistas de linguagens podem adaptar esta sintaxe instanciando um analisador léxico específico de domínio através da definição de uma ontologia.

> Visando obter um sistema modular, decidimos empregar uma sintaxe de controle, de referenciação a objetos e de metalinguagem fixa, como ocorre nas linguagens naturais, separando os elementos léxicos que são específicos do domínio. Esta decisão torna possível desenvolvermos um ambiente de programação composto de um parser e um processador para esta sintaxe antes mesmo dela ser instanciada pelo projetista da linguagem. (SILVA; PINHEIRO, 2004b)

A Figura 2 mostra o processo do *framework* proposto por (SILVA; PINHEIRO, 2004b). O processo se inicia quando um engenheiro de linguagem define a ontologia do domínio. Através da combinação da linguagem-tipo, previamente definida, com a ontologia

de domínio, é instanciada a linguagem de domínio específico. Neste momento, o usuário pode utilizar a linguagem instanciada para descrever as ações. O parser e o processador da linguagem-tipo utilizam a ontologia de domínio na resolução de referências a objetos.

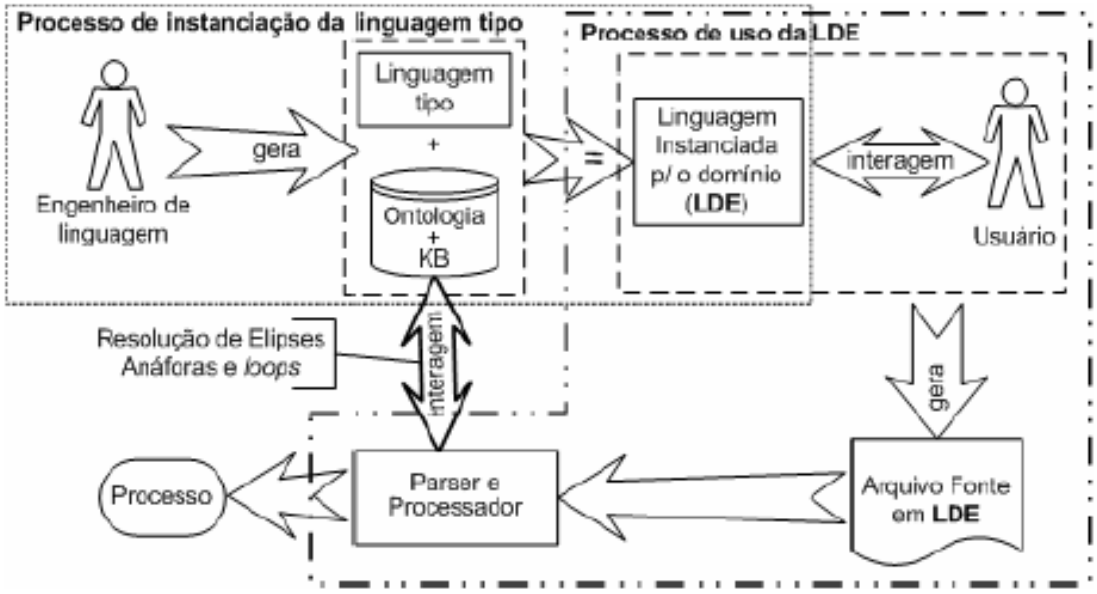


Figura 2. Processo de instanciação e uso da linguagem-tipo (Fonte: SILVA; PINHEIRO, 2004b)

Os autores do *framework* visam reduzir o ciclo de desenvolvimento e a curva de aprendizado de linguagens específicas de domínio. Utilizando este *framework*, o ciclo de desenvolvimento se reduz à definição da ontologia do domínio. A curva de aprendizado é atenuada devido à utilização de uma linguagem natural controlada - um subconjunto da linguagem natural com o léxico, a sintaxe e a semântica restritos ao domínio específico. Apesar do autor ter definido uma gramática para linguagem natural controlada, o *framework* proposto não é compatível com a abordagem MDA pois ele não considera modelos UML.

A idéia de utilizar linguagem natural controlada como base para linguagens específicas de domínio está baseada no fato de que as linguagens naturais controladas são corretas na linguagem natural. Porém, em função das restrições de linguagens controladas, nem todas as sentenças em linguagem natural são corretas na linguagem controlada. Dessa

forma, os usuários da linguagem controlada terão que aprender somente os tipos de sentenças que ele pode utilizar. Esse processo é mais simples do que aprender uma nova linguagem.

Em (LINHALIS; MOREIRA, 2005) (LINHALIS; MOREIRA, 2004) é apresentada uma solução de tratamento de requisições escritas em linguagem natural controlada para invocação de componentes de software. Esta solução está baseada no fato de que os componentes estão descritos segundo uma ontologia. Como pode ser visto na Figura 4, essa ontologia descreve as funcionalidades dos componentes, as entradas e saídas de cada funcionalidade e os tipos dessas entradas e saídas.

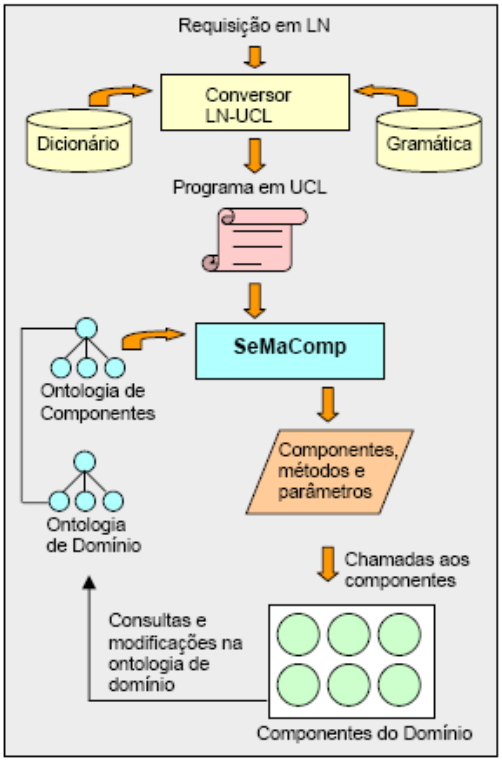


Figura 3. Requisições em linguagem natural (Fonte: LINHALIS; MOREIRA, 2004)

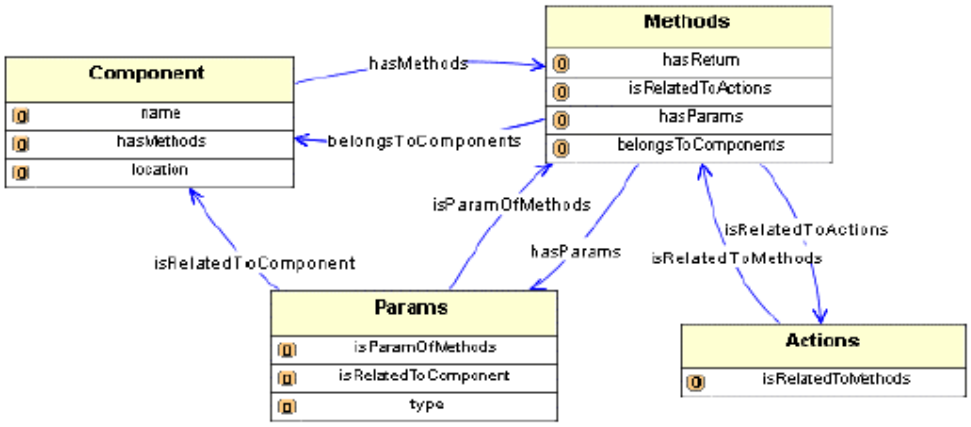


Figura 4. Ontologia de componentes (Fonte: LINHALIS; MOREIRA, 2004)

A Figura 3 ilustra o funcionamento da tradução da requisição, que é passada por um conversor que transforma a entrada em uma linguagem intermediária (aceita pelo módulo de descoberta de componentes - SeMaComp). Para descobrir o componente adequado, o módulo SeMaComp (*Semantic Mapping between Components*) faz uma relação entre as ontologias disponíveis e a requisição na linguagem intermediária. Esse trabalho foca no reuso de componentes de software, mas não faz uso de modelos e portanto não está alinhado aos objetivos da MDA.

Considerando necessária uma linguagem de especificação de requisitos conveniente para os especialistas de domínio, mas também capaz de ser mapeada para uma implementação, em (BRYANT; LEE, 2002) foi definida a *Two-Level Grammar* (TLG). TLG é uma linguagem de especificação de requisitos orientada a objetos baseada em linguagem natural, mas com formalismo suficiente para derivar a implementação correspondente. A técnica utilizada na transformação é identificar objetos no domínio do problema baseando-se em substantivos e relações através dos verbos. Essa gramática foi inicialmente criada como uma linguagem de especificação de linguagens de programação. Com o surgimento de modelos executáveis, ela passou a ser utilizada como uma linguagem de especificação executável, possibilitando a transformação de requisitos expressos em linguagem natural para uma especificação formal. Na Tabela 5, mostramos exemplos de requisitos em linguagem

natural controlada que são aceitos pela TLG. Na Tabela 6, temos a representação correspondente em uma transformação intermediária.

Tabela 5. Exemplo de requisitos em TLG

```
       The bank keeps the list of accounts.
       Each account has three integer data fields; ID, PIN, and balance.
The ATM machine has 3 service types; withdraw, deposit, and balance
check. For each service first it verifies ID and PIN from the bank.
       Withdraw service withdraws an amount from the account of ID with
PIN in the bank in the following sequence: First it gets the balance of
the account of ID from the bank, if the amount is less than or equal to
the balance then it decreases the balance by Amount, updates the balance
of the account of ID in the bank, and then outputs the new balance.
       Deposit service deposits an amount to the account of ID with PIN
in the bank in the following sequence: First it gets the balance of the
account of ID from the bank, it increases the balance by Amount, updates
the balance of the account of ID in the bank, and then outputs the new
balance.
       Balance check service checks the balance of the account of ID with
PIN in Bank in the following order: It gets the balance of the account of
ID from the bank, and then outputs the balance.
Transfer service withdraws an amount from the account of ID with PIN in
the bank and deposits the amount to the account of ID2.
```

Tabela 6. Exemplo de requisitos na linguagem intermediária da TLG

```
       class Account.
            Id, Pin, Balance, Amount :: Integer;

            withdraw Amount giving Balance1 :
                  Amount <= Balance,
                  Balance1 := Balance - Amount,
                  set balance to Balance1.

            deposit Amount giving Balance1 :
                  Balance1 := Balance + Amount,
                  set balance to Balance1.
       end class.

       class Bank.
            Accounts :: AccountList.
            Id, Pin :: Integer.

            get account using Id giving Account :
                  select Account from Accounts
                  with id of Account = Id.

            get account using Id and Pin giving Account :
                  select Account from Accounts with
                  id of Account = Id and pin of Account = Pin.
       end class.

       class ATM.
            Id, Pin, Balance, Amount :: Integer.

            withdraw Amount from account of Id with Pin in Bank
                  giving Balance :
```

```
                get account from Bank using Id and Pin
                giving Account,
                withdraw Amount from Account giving Balance.

        deposit Amount account of Id with Pin in Bank
                giving Balance :
                get account of Bank using Id and Pin
                giving Account,
                deposit Amount to Account giving Balance.

        check balance of Id with Pin in Bank giving Balance :
                get account of Bank using Id and Pin giving Account,
                get balance of Account giving Balance.

        transfer Amount from account of Idi with Pini to
                account of Id2 in Bank :
                withdraw Amount from account of Idi
                with Pini in Bank giving Balancei,
                get account of Bank using Id2 giving Account2,
                deposit Amount to Account2 giving Balance.
end class.
```

Apesar da TLG permitir especificações independentes de plataforma e uma maior proximidade dos termos do domínio, essas especificações são apenas textuais - escritas em linguagem natural. Isto torna a especificação muito extensa e pouco visual, o que prejudica o entendimento.

Outra proposta de linguagem natural controlada é a *Attempto Controlled English* (ACE) que é utilizada pelo sistema Attempto (SCHWITTER; FUCHS, 1996) na especificação de requisitos. A razão da utilização de linguagem natural controlada é a derivação de especificações formais em Prolog (PROLOG, 2006). Desta forma, a base de conhecimento resultante pode ser simulada e validada. Esta linguagem natural controlada e o sistema Attempto foram definidos antes da definição da MDA e, portanto, não são aderentes aos objetivos das abordagens baseadas em modelos.

Na literatura, encontramos algumas ferramentas que tratam o processamento de linguagens naturais para geração de aplicações. Uma delas visa mapear regras de negócio em sistemas de informação, identificando sentenças condicionais, da forma if-then-else, e baseando-se em um vocabulário de vinte e três mil palavras (HARS; MARCHEWKA, 1996). Apesar desta ferramenta já possuir um alto nível de abstração nas especificações, ela não é

compatível com a abordagem MDA pois não é baseada em modelos. Uma desvantagem dessas soluções que não são baseadas em modelos é o fato de elas misturarem a definição dos conceitos do domínio com a descrição das ações na gramática.

Segundo Liu e Lieberman (2005), programar é a arte de construir uma história sobre objetos em um programa e como eles se comportam em determinadas situações. Histórias são contadas em uma lingugem natural controlada onde, basicamente, os substantivos são traduzidos para objetos, os verbos para funções e os adjetivos para propriedades. A ferramenta desenvolvida, Metafor, dinamicamente converte as histórias do usuário em programas Python (PYTHON, 2006).

Segundo Liu e Lieberman (2004), as descrições de procedimentos geralmente contêm semântica programática, que pode ser facilmente mapeada em construções de linguagens de programação. Esse trabalho baseou-se em algumas características da linguagem natural tais como: tipagem sintática, herança, referência, condições, iterações e composição funcional. Liu e Lieberman (2005) fizeram um estudo de viabilidade de programação em linguagem natural. O resultado desse trabalho foi uma abordagem baseada nos avanços de processamento de linguagem natural e programação por exemplo. Para avaliar a abordagem, foi feito um experimento onde usuários não desenvolvedores descreviam alguns requisitos. O que pôde ser observado é que os requisitos mais importantes puderem ser tratados pela abordagem proposta. O principal objetivo da ferramenta utilizada é a elicitação de conhecimento e, portanto, ela não gera programas completos. A saída da ferramenta é uma estrutura sequencial de ações, com suas respectivas entradas.

Um outro trabalho que também envolve processamento de linguagens naturais é a proposta de uma ferramenta para elicitar requisitos com os especialistas de domínio com o objetivo de guiar o projeto de interface com o usuário (TAM; MAULSBY; PUERTA, 1998). Esta ferramenta é chamada de *User-Task Elicitation Tool* (U-TEL).

Apesar de existirem várias iniciativas e ferramentas que utilizam linguagem natural controlada para geração de código, não encontramos nenhuma que faça uso de modelos UML, não sendo, portanto, compatíveis com a MDA. Algumas dessas linguagens são completamente textuais e as demais utilizam outras formas de representação, tal como ontologias. Uma notação gráfica, como a UML, facilita a visualização e pode ser complementada por uma gramática em linguagem natural, ao invés de substituí-la completamente.

3 NATURAL MDA: A LINGUAGEM DE ESPECIFICAÇÃO DE AÇÕES

No capítulo 2, vimos os principais objetivos da abordagem MDA e as principais vantagens e limitações das linguagens de especificação de ações e das linguagens naturais controladas. Percebemos que, absorvendo as vantagens das linguagens naturais controladas (nível de abstração acima da implementação) e das linguagens de ações (compatível com a UML, executável, completa e independente de plataforma), podemos alcançar o principal objetivo da abordagem MDA que é a modelagem completa de sistemas, de forma precisa e independente de detalhes tecnológicos.

Neste capítulo, apresentamos a linguagem Natural MDA, uma linguagem de ações baseada em linguagem natural controlada. Iniciamos apresentando o projeto da linguagem e, em seguida, o impacto da utilização dessa linguagem no processo de desenvolvimento de sistemas e um conjunto de ferramentas construído para apoiar a utilização da linguagem.

3.1 PROJETO DA LINGUAGEM

A Natural MDA é uma linguagem de especificação de ações, complementar a UML, baseada em linguagem natural controlada. O projeto desta linguagem foi baseado nos seguintes requisitos: fornecer um nível de abstração adequado à fase de projeto, ser processável por máquina e complementar a especificação corrente da UML (UML, 2004) (aderente a MDA). A utilização de linguagens naturais controladas está vinculada ao alto nível de abstração (melhorar a legibilidade por humanos) e à possibilidade de processamento computacional.

Nosso objetivo é permitir que os especialistas especifiquem o comportamento do negócio utilizando os seus próprios conceitos; ou seja, uma abordagem compatível com o método de modelagem específica de domínio. A concepção e projeto desta linguagem foram inicialmente apresentados em (LEAL; PIRES; CAMPOS, 2006).

A gramática da Natural MDA está baseada no trabalho de (SILVA; PINHEIRO, 2004b), onde linguagens naturais controladas podem ser utilizadas como base para definição de linguagens específicas de domínio. No primeiro trabalho, a gramática referencia elementos de ontologias. Neste trabalho, a gramática referencia elementos de modelos UML. A idéia principal é que a gramática da linguagem defina apenas a regra de formação das sentenças válidas. Os termos que não são palavras reservadas da linguagem devem ser representados como referência aos elementos do modelo UML (classes, atributos e operações) através de suas descrições. Utilizamos o mecanismo de extensão da UML conhecido como valores etiquetados (*tagged values*) para associar descrições às operações (assim como para os parâmetros de entrada e valor de retorno), classes, atributos e associações. Essas descrições dos elementos do modelo devem ser utilizadas para descrever as ações das demais operações.

A proposta de combinar alto nível de abstração, geração de código e o uso de conceitos de um domínio específico na especificação de sistemas oferece as seguintes vantagens:

- A especificação é mais legível e, por conseqüência, facilitando a manutenção do sistema pois a solução é descrita em alto nível de abstração, utilizando somente conceitos do domínio de negócio que está sendo modelado;
- Melhoria da qualidade de código, reduzindo erros sintáticos e lógicos devido às transformações serem automatizadas, praticamente sem interferência humana;
- Concentração de esforços na modelagem do domínio, resolvendo o problema apenas uma vez, em alto nível de abstração, e gerando código automaticamente.

Assim como os compiladores transformam código-fonte em linguagens de programação de mais baixo nível (código de máquina), visamos oferecer uma linguagem que permita especificar o comportamento de sistemas em um nível de abstração acima da implementação. Desta forma, reduzimos a diferença conceitual entre o domínio do problema e

o código pois são dois mundos significantemente diferentes, com suas próprias linguagens, especialistas e maneiras de pensar.

Seguindo a classificação dos níveis de maturidade das linguagens de modelagem apresentados no capítulo 2, modelos UML complementados pela linguagem Natural MDA são um passo na direção de uma linguagem de modelagem de nível 5; ou seja, modelos completos e precisos.

Nas seções seguintes, detalhamos a gramática da linguagem-tipo que serviu como base para a definição desta linguagem, apresentamos a linguagem Natural MDA com seus elementos base e funções, e identificamos algumas limitações.

3.1.1 Gramática de uma linguagem-tipo

Esta gramática faz parte do *framework* para criação de linguagens de domínio específico apresentada no Capítulo 2. Apresentaremos, em seguida, a sublinguagem de referência a objetos, a sublinguagem de controle e as expressões lógicas.

A sublinguagem de referencia a objetos, mostrada na **Error! Not a valid bookmark self-reference.**, permite o uso de determinantes para os objetos e um conjunto restrito de figuras de linguagens, como elipses[1] e anáforas[2]. Os determinantes são classificados em quantificadores, qualificadores e seletores.

A sublinguagem de controle, mostrada na Tabela 8, permite o uso de aglutinação, condicionamento e iterações de ações. As expressões lógicas, mostradas na Tabela 9, também estão embutidas na sublinguagem de controle.

[1] Elipse é uma figura de estilo e, mais concretamente, uma figura de sintaxe que consiste na omissão de uma ou mais expressões numa frase que podem, contudo, ser facilmente subentendidas. Geralmente, o contexto em que se insere a frase dá ao receptor da mensagem todas as informações necessárias para a compreensão do enunciado (WIKIPEDIA, 2005).

[2] Anáfora é uma figura de estilo que consiste na repetição do início em frases, versos ou orações sucessivas. Pode aparecer associada à enumeração e à gradação ("tão simples, tão franco, tão honesto..." - onde se repete o elemento "tão") (WIKIPEDIA, 2005).

Tabela 7. Sublinguagem de referencia a objetos (Fonte: SILVA; PINHEIRO, 2004b)

Tipo da referência	Exemplo
"noun" (um nome próprio)	"personal"
[a/an] noun	[a] message
Noun(s)	Messages
Cardinal noun(s)	7 messages
The/this noun	The message
A [set/list/sequence] of noun(s)	A set of messages
The nounip of [the] noun	The sender of the message
Pronouns (it/them)	It belongs to …/… and copy them
The ordinal noun (next/last/first/previous)	The last message
All [the] noun(s)	All the messages
Each / every noun	Each message

Tabela 8. Sublinguagem de controle (Fonte: SILVA; PINHEIRO, 2004b)

Sintaxe da sentença na linguagem-tipo	Exemplo
actions → action (',' | 'and') actions '.'	reply to the message and mark it as read.
action → verb [preposition] object_ref [preposition object_ref [prepositon object_ref]]] '.'	send the message to all addresses from the receiver of the message.
action → ('if' | 'when') object_ref logic_expr ',' actions '.' [('if not' | 'otherwise') ',' actions '.']	if the group belongs to the address-book, send the… . if not, show the message… .
action → 'repeat' actions ('until' | 'up to' | 'while') logic_expr '.'	Repeat if the color of folder is blue, go to its places and pick it up to the last…
action → 'for' object_ref ',' actions '.'	for all important appointments of the agenda, set its day's color to red.
action → 'from' object_ref ('at' | 'on' | 'to' | 'in') object_ref ',' actions '.'	from the first mail to the last, change their status to 'read'.

Tabela 9. Expressões lógicas (Fonte: SILVA; PINHEIRO, 2004b)

logic_expr → simple_expr [rel_operator simple_expr]
simple_expr → [unary_operator] term {add_operator term}
term → factor {multi_operator factor}
factor → literal | object_ref
rel_operator → '***belongs to***' | '***is***' | '***are***' | '***is accepted***' | '***is bigger than***' | '***is smaller than***' | '***is bigger or equal than***' | '***is smaller or equal than***'
unary_operator → '***there is***' | '***there are***' | '***not***'
Add_operator → '***and***'
multi_operator → '***or***'

3.1.2 Gramática da linguagem Natural MDA

Para definirmos a gramática da linguagem natural controlada, identificamos os elementos base de linguagens de programação. A partir desses elementos, identificamos abstrações para reduzir a complexidade para os analistas de negócio. Essas abstrações são representadas no que chamamos de funções auxiliares e são apresentadas na próxima seção.

Na Tabela 10, apresentamos a gramática da Natural MDA, que foi baseada na linguagem apresentada em (SILVA; PINHEIRO, 2004b) e segue os princípios de programação por usuários finais. A principal diferença está na substituição da ontologia de domínio por um modelo UML; ou seja, as referências a objetos citadas naquele trabalho correspondem às referências às classes e operações existentes no modelo.

Tabela 10. Gramática da linguagem natural controlada

```
1  …
2  rule : ((operations DOT) | (sentence) | (after_all) )+;
3  operations: (operation ((COMMA | AND) operation)*) |
4              (show_error_warning) | (return_statement);
5  show_error_warning : SHOW (ERROR | WARNING) ASPAS message ASPAS;
6  return_statement: RETURN alpha_numeric;
7  sentence: (conditional (OTHERWISE COMMA operations DOT)?) |
8            (loop) | (exceptional);
9  conditional : IF conditions COMMA
10               ((operations DOT) | loop);
11 loop: FOR EACH item OF THE collection COMMA
12       ((operations DOT) | (conditional ));
13 exceptional : WHEN event COMMA operations DOT;
14 after_all : AFTER_ALL alpha_numeric COMMA
15             ((operations DOT) | (sentence));
16 conditions: condition ((AND | OR ) condition)*;
17 condition : operand verb ( (equal | not_equal |
18             greater_or_equal_to | greater_than |
19             lower_or_equal_to | lower_than) ) operand;
20 equal : EQUAL TO;
21 not_equal : NOT EQUAL TO;
22 greater_or_equal_to : GREATER OR EQUAL TO;
23 greater_than : GREATER THAN;
```

```
24 lower_or_equal_to : LOWER OR EQUAL TO;
25 lower_than : LOWER THAN;
26 verb : IS | ARE;
27 alpha_numeric : (CHARACTER | DIGIT)(DIGIT | CHARACTER)*;
28 collection: alpha_numeric;
29 event: alpha_numeric;
30 operation: alpha_numeric;
31 message: alpha_numeric;
32 operand: alpha_numeric;
33 …
```

O detalhamento da gramática da linguagem encontra-se no Apêndice 6. Com base na linguagem apresentada, veremos como os elementos (classes, atributos e métodos) do modelo UML são referenciados na linguagem. Para aproximar os conceitos do negócio com os conceitos do modelo, utilizamos o mecanismo de extensão da UML conhecido como valores etiquetados. Assim, em cada elemento do modelo, o projetista deve atribuir um valor a esse valor etiquetado. Esse valor etiquetado serve para ligar a linguagem aos elementos do modelo. Esse valor provê um nome mais amigável e natural aos elementos do modelo, funcionando como um mapeamento entre eles e os termos de negócio. Os valores etiquetados criados foram:

- *element.description* → é utilizado para atribuir descrição aos elementos do modelo;
- *return.description* → é utilizado para atribuir descrição ao valor de retorno das operações;
- *operation.behaviour* → é utilizado para atribuir a especificação de ações das operações.

Voltando à definição da gramática da linguagem para explicitar sua ligação com os elementos do modelo, ressaltamos que apesar do termo “operation” da gramática ser definido como um conteúdo alfa numérico, ele deve necessariamente possuir um conteúdo equivalente

ao valor etiquetado *element.description* de alguma operação do modelo. Da mesma forma, o termo "collection" da gramática está associado ao valor etiquetado *element.description* de algum parâmetro de entrada de alguma operação, ou ao valor etiquetado *return.description* de alguma operação. O termo "event" da gramática também deve ter conteúdo relacionado ao valor etiquetado *element.description* de alguma exceção declarada no modelo. Esses valores etiquetados de descrição de elementos são utilizados para referenciar os elementos do modelo quando o projetista especifica uma lógica de negócio.

3.1.3 Funções auxiliares

Em geral, as linguagens de programação oferecem uma série de funções em suas bibliotecas. Essas funções são, por exemplo: obtenção de data e hora atuais, manipulação de conjuntos (união, interseção, diferença, tamanho, etc), manipulação de cadeias de caracteres (concatenação, substituição, etc) e operações aritméticas.

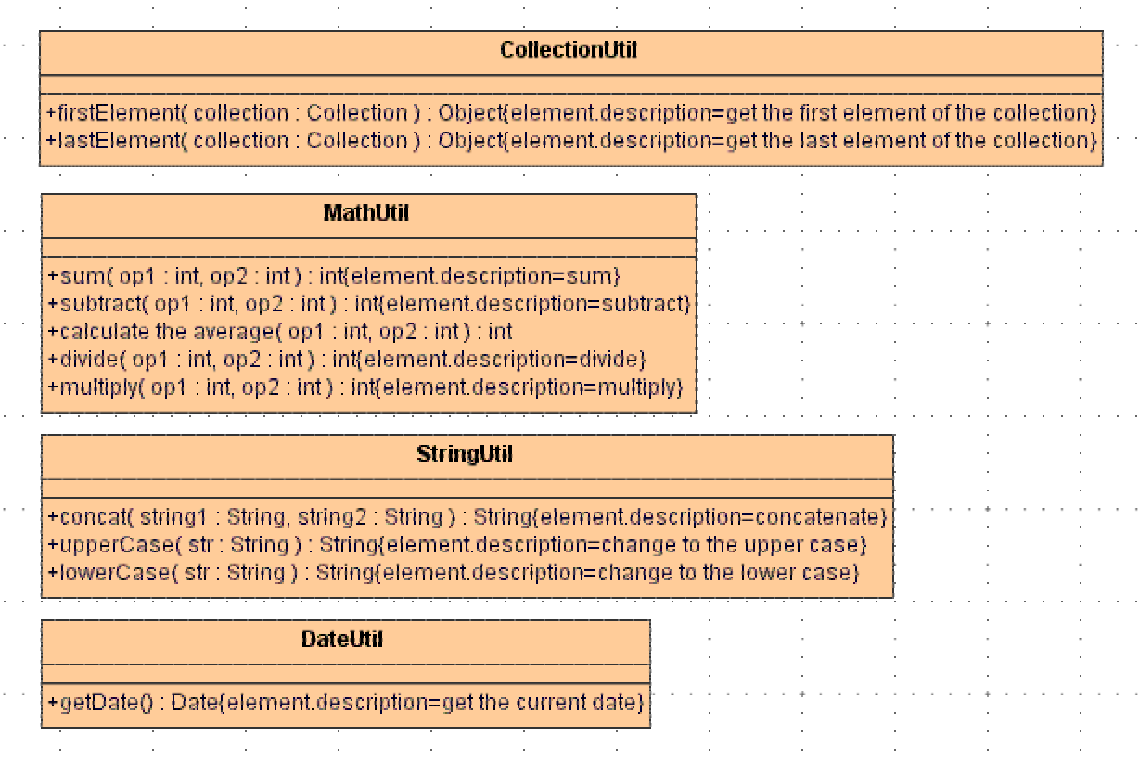


Figura 5. *Profile* das funções auxiliares

Para permitir que o projetista utilize funções auxiliares, optamos por criar algumas classes empacotadoras para encapsular a utilização da biblioteca. Para tal, criamos um *profile* contendo essas classes e os valores etiquetados necessários para a utilização da linguagem Natural MDA. A Figura 5 mostra esse *profile*, que pode ser importado em outros modelos.

Todas as operações dessas classes têm um valor etiquetado de descrição associado. É essa descrição que deve ser utilizada para invocar as funções. Na Tabela 11 vemos um exemplo de especificação de ações que utiliza a função auxiliar de obtenção de data. Desta forma, havendo uma implementação dessa função auxiliar, basta importar a biblioteca correspondente no novo projeto para torná-lo completo e executável.

Tabela 11. Exemplo de uso de funções auxiliares na especificação

```
Find the book, get the current date and reserve the book.
```

A Tabela 12 mostra como a utilização de funções auxiliares no nível de especificação é mapeada para a implementação após as transformações de modelos.

Tabela 12. Exemplo de uso de funções auxiliares na implementação

```
...
java.util.Date var1 = DateUtil.getDate();
...
```

3.1.4 Linguagens visuais versus textuais

Na definição da Natural MDA foram pesquisadas e analisadas as características de linguagens visuais em comparação com linguagens textuais e optamos pela criação de uma linguagem textual. Nossa decisão se baseia nos aspectos descritos a seguir.

Acreditamos que quando a complexidade dos sistemas aumenta, a representação diagramática não é a mais adequada. A própria UML, que é uma linguagem de modelagem visual, adotou linguagens textuais, como a OCL e linguagens de especificações de ações, para descrever restrições e ações. Neste sentido, uma outra atitude do OMG que demonstra certa preocupação com a notação gráfica foi a adoção de um padrão de notação textual voltada à

humanos, a *Human-Usable Textual Notation* (HUTN) (HUTN, 2004). HUTN é uma especificação que provê uma solução genérica para geração de linguagens textuais a partir de modelos *Meta Object Facility* (MOF) (MOF, 2006). A principal meta de projeto do HUTN é a usabilidade por humanos. O critério de usabilidade está voltado a uma abordagem de projeto centrada no usuário, considerando as necessidades do usuário antes dos detalhes tecnológicos inerentes ao desenvolvimento de sistemas.

Um outro exemplo de avaliação de linguagens textuais x visuais é a linguagem de programação visual chamada Visula (VISULA, 2004). Apesar dessa linguagem ser visual, o próprio autor reconhece algumas vantagens das linguagens textuais: i) não requerem uma notação especial; ii) não requerem editores especiais; iii) são mais fáceis para discussão em fóruns.

3.1.5 Aplicabilidade e limitações da Natural MDA

Apesar da Natural MDA ter se mostrado adequada para a especificação de sistemas de informação tradicionais, consideramos que alguns sistemas, devido aos seus requisitos específicos, podem não ser adequados a aplicação da Natural MDA. Um exemplo são os sistemas especialistas, onde a especificação é fortemente baseada no uso de cálculos e fórmulas matemáticas. Nessa categoria de sistemas, a linguagem Natural MDA não é adequada porque esse tipo de comportamento pode ser melhor representado através de notações formais mais apropriadas do que a linguagem natural. Um outro exemplo são os sistemas complexos, onde nem sempre é possível reduzir a complexidade através de fatorações ou outras alternativas de especificação.

Durante a implementação da Natural MDA foram detectadas pequenas limitações que são descritas a seguir:

- As descrições das operações do modelo não podem conter vírgula (“,”) e nem ponto final (“.”). O mesmo vale para a descrição das classes e atributos. Para

resolver essa limitação, poderíamos utilizar delimitadores para identificar o início e fim de uma descrição. Por exemplo, a descrição “a, b, c and d.” poderia ser utilizada na especificação de ações na forma “<a, b, c and d>.”.

- Na especificação das condições de uma expressão condicional, não deve existir vírgula (“,”), mesmo que as condições sejam compostas de mais de duas condições. Por exemplo, se a condição for “a is equal to b, b is equal to c and c is equal to d”, então ela deve ser escrita como “a is equal to b and b is equal to c and c is equal to d”. Essa limitação pode ser resolvida trocando o termo que indica o final da condição. Atualmente, utilizamos “,” para isso, mas poderíamos utilizar o termo “then”. Dessa forma, é possível identificar cada parcela que compõe a condição completa e o final da condição.
- A descrição da mensagem de erro ou alerta não pode conter aspas. Por exemplo, a mensagem “O campo “nome” não foi informado” deve ser escrita como “O campo ‘nome’ não foi informado”. Essa limitação não foi considerada significante.
- A linguagem não possui operações aritméticas, de datas e de manipulação de *strings* e conjuntos. Como trabalho futuro, pode ser feita uma análise para verificar se é adequado incluir estas operações na gramática. Sendo adequado, é necessário alterar a gramática e as ferramentas de apoio da linguagem. Ao analisar a possibilidade de incluir operações aritméticas na gramática, deve-se levar em consideração o fato de manter o nível de abstração acima da implementação e a legibilidade por pessoas não especialistas em Computação.

3.2 UTILIZAÇÃO DA NATURAL MDA NO PROCESSO DE DESENVOLVIMENTO DE SISTEMAS

Na seção anterior, vimos o projeto da linguagem Natural MDA. Nesta seção, veremos o impacto da utilização da linguagem no processo de desenvolvimento de sistemas.

Construir soluções utilizando a abordagem MDA requer mudanças no processo de desenvolvimento (BROWN; CONALLEN, 2005). A utilização da linguagem Natural MDA afeta o processo de desenvolvimento de sistemas pois requer a inclusão de atividades e elimina a necessidade de outras, como discutida anteriormente em (LEAL *et al.*, 2006). Para ilustrar como a linguagem Natural MDA influencia no processo de desenvolvimento de sistemas, utilizaremos o *framework Rational Unified Process* (RUP) (KRUCHTEN, 2000) como exemplo, ressaltando as mudanças necessárias nesse processo para tal adequação. A escolha do RUP se deve a sua compatibilidade com MDA (pois ambos são baseados em modelos), à completude e à capacidade de customização do *framework*.

As principais características do RUP são:

- Orientado a casos de uso → O processo impõe que os casos de uso orientem o desenvolvimento desde a concepção até a entrega do sistema.
- Centrado na arquitetura → O processo busca entender os aspectos mais significantes em termos da arquitetura, que é derivada através das necessidades dos usuários e é capturada nos casos de uso.
- Iterativo e incremental → O processo considera que é prático dividir projetos grandes em projetos menores ou sub-projetos. Cada sub-projeto equivale a uma iteração, que resulta em um incremento.

A Figura 6 mostra as relações entre os conceitos do *framework* RUP. O processo é organizado em disciplinas e é expresso em *workflows*. Cada papel executa suas atividades e é

responsável por seus artefatos. Cada atividade tem seus artefatos de entrada e de saída. Para a criação de cada artefato, existem *checkpoints*, *templates* e relatórios.

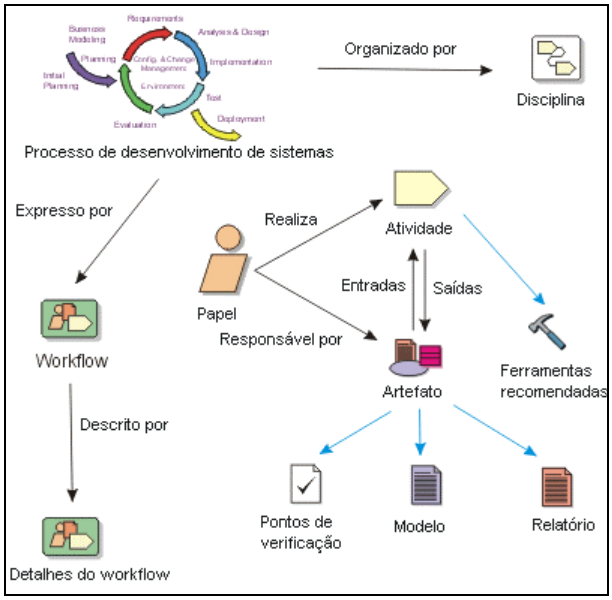


Figura 6. Elementos do RUP (Fonte: baseado em RATIONAL, 2006)

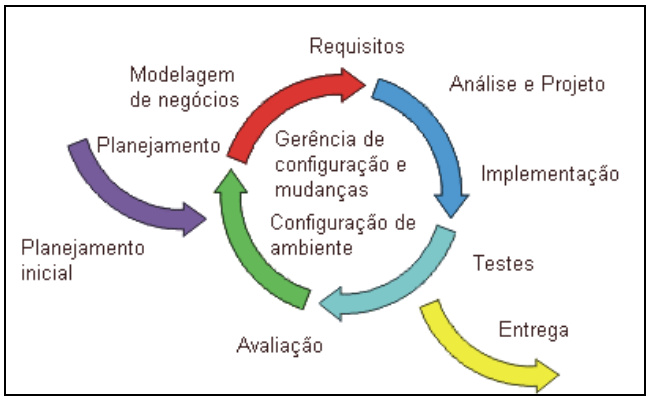


Figura 7. Ciclo de vida do RUP (Fonte: baseado em RATIONAL, 2006)

O ciclo de vida do RUP é mostrado na Figura 7. O processo é iniciado com a modelagem do negócio e um planejamento inicial. Então, inicia-se o ciclo de levantamento de requisitos, análise, projeto, implementação, teste e avaliação até que se obtenha o aceite para a instalação do produto. A configuração de ambiente e a gerência de configuração e mudanças acontecem durante todo o ciclo de vida.

O RUP consiste de fases e disciplinas. As fases representam os aspectos do ciclo de vida do processo. As disciplinas representam um agrupamento lógico de atividades. A Figura 8 mostra a arquitetura do *framework* segundo essas duas dimensões (RATIONAL, 2006). Nessa figura, podemos observar que existem disciplinas cuja ênfase é praticamente constante ao longo das fases, enquanto outras variam a ênfase conforme a fase. Por exemplo, a disciplina "Gerência de Projetos" e "Configuração de Ambiente" requerem a mesma ênfase em todas as fases. Já a disciplina "Implementação" tem nitidamente maior ênfase na fase de "Construção".

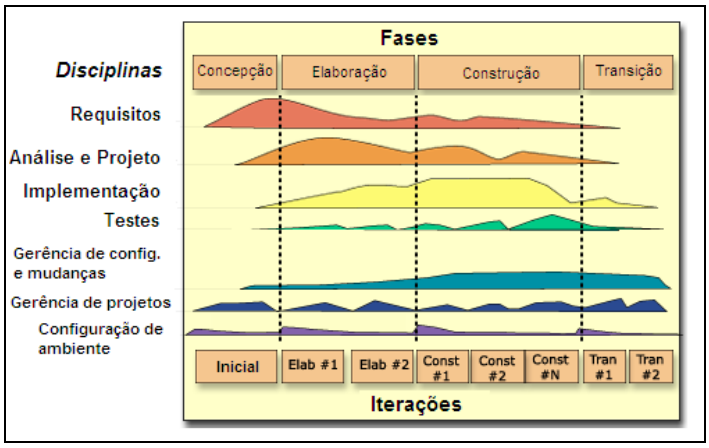


Figura 8. Arquitetura do *framework* RUP (Fonte: baseado em RATIONAL, 2006)

As fases do *framework* RUP são (MELOCHE, 2002):

- Concepção → O objetivo desta fase é definir a viabilidade e escopo do projeto e ter um acordo de entendimento comum entre os analistas e os clientes.
- Elaboração → O objetivo desta fase é detalhar os casos de uso, definir a arquitetura, identificar os riscos significantes e preparar o cronograma.
- Construção → O objetivo desta fase é implementar os casos de uso.
- Transição → O objetivo desta fase é certificar de que os requisitos identificados e desenvolvidos satisfizeram as necessidades dos clientes.

As disciplinas do *framework* RUP são: Requisitos, Análise e Projeto, Implementação, Teste, Gerência de Mudanças e Configuração, Gerência de Projetos e

Configuração de Ambiente. A seguir, mostraremos o *workflow* de cada uma dessas disciplinas e o impacto devido à utilização da linguagem Natural MDA.

Figura 9. Workflow de análise e projeto (Fonte: baseado em RATIONAL, 2006)

O *workflow* de requisitos não é afetado pela utilização da linguagem Natural MDA pois ele é voltado para a captura dos requisitos da aplicação. A Figura 9 apresenta o *workflow* de análise e projeto. Nesse *workflow*, identificamos que a utilização de desenvolvimento orientado a modelos e da linguagem Natural MDA apresenta algumas vantagens. Esse *workflow* tem como objetivo implementar a prova de conceito arquitetural, que inclui as decisões mais críticas do sistema. Como é possível gerar o código a partir de modelos complementados pela linguagem Natural MDA, a prova de conceito arquitetural pode ser rapidamente construída, permitindo assim testar diferentes transformações de modelos a fim de escolher as mais apropriadas de acordo com os requisitos do sistema. Dessa forma, a aplicação das transformações nos demais casos de uso será facilitada, visto que a prova de conceito engloba os casos de uso mais complexos e com maior risco até o momento. Consideramos necessária a inclusão de uma atividade cujo objetivo é definir as transformações apropriadas às necessidades do projeto e dentro dos padrões da organização, se for o caso.

Outra observação é o fato de que o projetista, ao criar as classes, deve associar descrições (através do valor etiquetado correspondente) aos atributos e operações que serão utilizados na especificação das ações das operações, assim como os estereótipos ou marcações necessárias. No projeto dos casos de uso, o projetista pode descrever o comportamento das operações utilizando a Natural MDA (através do valor etiquetado correspondente) ou utilizar os próprios diagramas de interação da UML. Utilizando MDD, a atividade de projeto do banco de dados é simplificada, consistindo no refinamento do esquema gerado a partir do modelo.

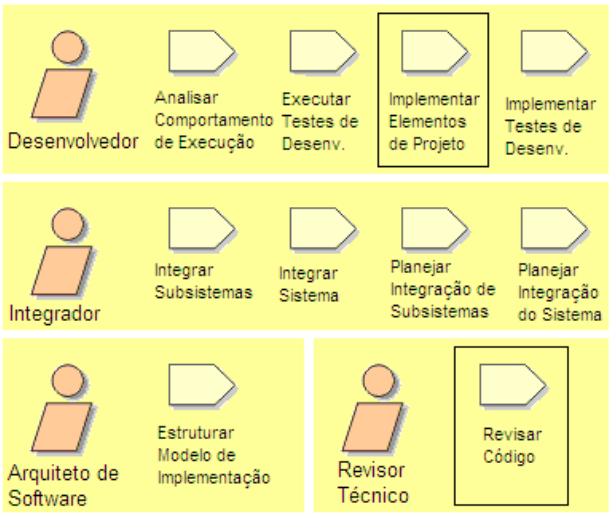


Figura 10. Workflow de implementação (Fonte: baseado em RATIONAL, 2006)

A Figura 10 apresenta o *workflow* de implementação. Nesse *workflow*, a atividade "Implementar elementos de projeto" é substancialmente afetada pois grande parte da implementação é gerada a partir das especificações de ações contidas no modelo. Observamos também que a atividade de revisão de código também é impactada, pois com a utilização da abordagem MDA o mais adequado é revisar a especificação que originou o código. E ainda, grande parte da implementação não precisa ser revisada já que a transformação foi validada inicialmente durante a elaboração da prova de conceito.

O *workflow* de testes não é afetado pela utilização da linguagem Natural MDA, porque a versão atual das ferramentas de apoio não contempla a geração de testes

automatizados. O *workflow* de gerência de configuração e mudanças também não é afetado. No *workflow* de gerência de projeto, a única consideração com a adoção da linguagem é em relação à atividade de alocação de equipe, em que o gerente de projeto deve ter a preocupação de alocar uma pessoa com habilidade para definir as transformações entre os modelos.

Como o *workflow* de configuração de ambiente visa ajustar o processo às necessidades do projeto e preparar o ambiente de desenvolvimento com *hardware* e *software* adequados, a utilização da linguagem Natural MDA também não implica em alterações nas atividades previstas para tal.

Apesar da estimativa de tempo e esforço variar entre projetos, o RUP sugere uma estimativa inicial típica para projetos de médio porte, como pode ser vista na Tabela 13. Consideramos que as abordagens de desenvolvimento orientadas a modelo tendem a reduzir o tempo e o esforço necessários na fase de Construção, visto que o principal objetivo dessas abordagens é concentrar esforços na especificação da solução e gerar código a partir da especificação. Por outro lado, consideramos que a fase de Elaboração exigirá maior tempo e esforço dado que a especificação final precisa ser mais completa e uma nova atividade será necessária para a definição das transformações de modelos.

Tabela 13. Estimativa de tempo e esforço proposto pelo RUP (Fonte: RATIONAL, 2006)

	Concepção	Elaboração	Construção	Transição
Tempo	10%	30%	50%	10%
Esforço	5%	20%	65%	10%

Neste trabalho, nos concentramos em identificar o impacto da utilização da Natural MDA em um processo de desenvolvimento típico, o RUP. Na literatura, existem outros trabalhos que apresentam em detalhes o impacto da utilização de MDA nos processos de desenvolvimento (RODRIGUES, 2004) (LARRUCEA; DIEZ; MANSELL, 2004).

Rodrigues (2004) propõe uma adequação do RUP ao desenvolvimento de sistemas baseado em MDA. Nesse trabalho, a autora identificou detalhadamente: (i) as atividades

desnecessárias e a necessidade de novas atividades; (ii) a geração de novos artefatos e os artefatos desnecessários de cada *workflow*; (iii) a geração automática total ou parcial dos artefatos.

Um outro trabalho que avalia a adequação do RUP à MDA é a definição de um processo de desenvolvimento orientado a modelos baseado no RUP e que é composto pelas seguintes fases: (i) captura de requisitos dos usuários; (ii) definição do contexto do PIM; (iii) especificação dos requisitos do PIM; (iv) análise do PIM; (v) projeto; (vi) codificação e integração; (vii) teste e (viii) instalação. Além da definição das fases, os autores identificaram as atividades com suas respectivas entradas e saídas e atribuiram papéis para executá-las (LARRUCEA; DIEZ; MANSELL, 2004).

3.3 FERRAMENTAS DE APOIO

Com base na linguagem Natural MDA, desenvolvemos duas ferramentas para apoiar sua utilização: o processador da linguagem (analisador sintático e semântico) e um editor para auxiliar o uso da linguagem na especificação de ações. Como o objetivo deste trabalho é gerar a implementação das operações, para obter o código executável, tivemos que definir uma forma de integração com as ferramentas MDA de forma a complementar o esqueleto de código gerado por elas. De modo geral, essas ferramentas têm mecanismos que permitem estender suas funcionalidades.

A Figura 11 representa o funcionamento de uma ferramenta MDA típica. Os modelos UML são feitos em ferramentas de modelagem, que os exportam no formato XMI. A ferramenta então, com base no XMI e nos templates de transformação, inicia a geração dos artefatos.

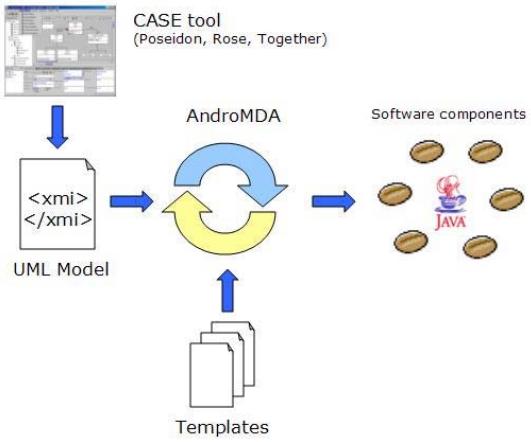


Figura 11. Funcionamento do AndroMDA (Fonte: ANDROMDA, 2006)

A Figura 12a mostra o processo convencional das ferramentas MDA onde o projetista: (1) especifica o sistema em uma ferramenta de modelagem, (2) exporta o modelo UML no formato XMI, (3) importa o modelo na ferramenta MDA, (4) aciona a ferramenta e (5) obtém o esqueleto de código.

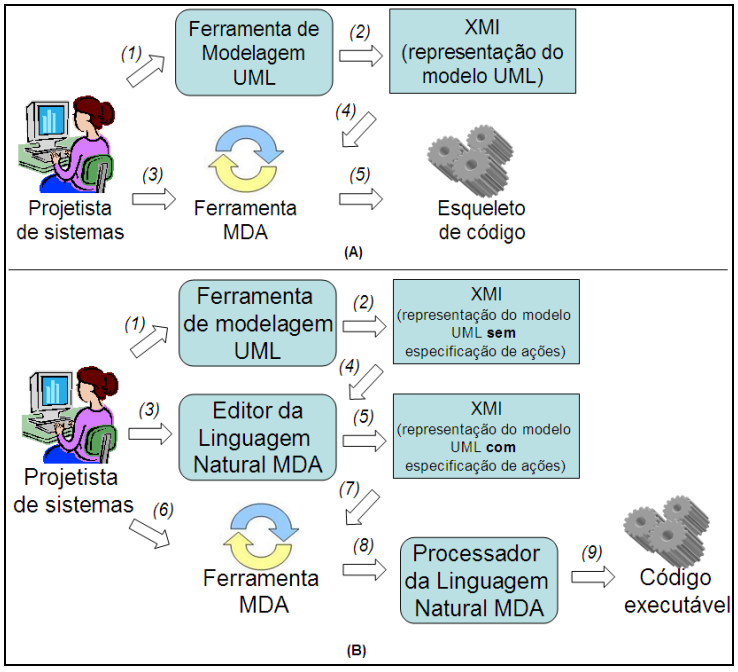


Figura 12. Arquitetura das ferramentas de apoio

A Figura 12b mostra como essas ferramentas interagem com as ferramentas MDA. Assim como no processo convencional, o projetista especifica o sistema em uma ferramenta de modelagem e exporta o modelo UML correspondente (passos 1 e 2). Em seguida, o

projetista deve importar esse modelo no editor da linguagem Natural MDA para especificar as ações das operações do modelo e salvá-las (passos 3, 4 e 5). Esse novo modelo, complementado pela especificação de ações, pode ser importado novamente na feramenta de modelagem. Finalmente, o projetista importa o modelo do sistema na ferramenta MDA e a aciona (passos 6 e 7). Isso faz com que o processador da linguagem Natural MDA seja acionado e obtenha-se o código completo (passos 8 e 9). O editor da linguagem não interage com a ferramenta MDA; ele apenas lê a representação XMI (para obter as descrições dos elementos do modelo) e salva a especificação de ações no XMI. O processador da linguagem interage diretamente com a ferramenta MDA. A fim de gerar o código da aplicação, o processador da linguagem combina o esqueleto de código, gerado pela ferramenta MDA, com a implementação de cada operação previamente especificada pelo projetista no editor.

3.3.1 Processador da linguagem

Um compilador é um programa que transforma um código escrito em uma linguagem em um programa equivalente em outra linguagem (AHO; SETHI; ULLMAN, 1986). Em nosso estudo, transformamos a linguagem natural controlada em código Java. O processo de compilação é composto de análise e síntese. A análise tem como objetivo entender o código fonte e representá-lo em uma estrutura intermediária. A síntese constrói o código objeto a partir dessa representação intermediária. Nesta implementação, a análise é realizada em três etapas:

- Análise léxica → é responsável por processar o texto contendo a linguagem natural controlada e transformá-lo em uma seqüência de tokens que pode ser facilmente tratada pelo analisador sintático. Este passo é responsável também por validar se a especificação do comportamento do método é válida segundo a gramática.
- Análise sintática → é responsável por construir a árvore sintática, representando de forma estruturada a especificação do comportamento do método.

- Análise semântica → é responsável por verificar erros semânticos na especificação do comportamento e preparar as informações necessárias para a transformação em código Java. Um exemplo de erro semântico seria a invocação de um método não existente no modelo.

Na fase de análise léxica, utilizamos a ferramenta de código aberto chamada *Another Tool for Language Recognition* (ANTLR) (ANTLR, 2005), que a partir de uma gramática, gera um programa capaz de reconhecer sentenças nessa gramática.

Na fase de síntese, optamos por utilizar o *String Template* (STRINGTEMPLATE, 2005), que é uma ferramenta de código aberto para geração de textos a partir de *templates*. A vantagem de utilizarmos uma ferramenta como essa é que podemos alterar a linguagem de saída simplesmente alterando os *templates*, sem alterar a lógica da transformação. No momento, geramos código Java, mas se for necessário gerar código em outra linguagem, basta alterar os *templates*.

Apesar de já existirem essas duas ferramentas, ainda não existe uma integração entre elas capaz de aplicar uma transformação (definida nos *templates*) em uma árvore sintática construída pelo ANTLR. Por esse motivo, implementamos os analisadores sintático e semântico. Um outro motivo para termos implementado o analisador semântico é que precisávamos verificar se as descrições dos métodos utilizadas na especificação das ações estavam especificadas no modelo UML.

Nas seções seguintes, detalhamos a implementação dos analisadores sintático e semântico, a síntese e a integração com uma ferramenta MDA.

3.3.1.1 Analisador sintático

O papel de um analisador sintático é ler tokens e construir a árvore de expressões. Nesta seção, apresentamos a implementação do analisador sintático para a linguagem de ações. Como podemos ver na Figura 13, temos expressões simples e compostas, representadas pelas

classes *SingleExpression* e *BlockExpression*, respectivamente. Chamamos de expressões compostas aquelas que englobam outras expressões. Classificamos as expressões compostas em condicionais (*ConditionalExpression*), iterativas (*LoopExpression*) e de tratamento de eventos (*TryExpression* e *EventExpression*).

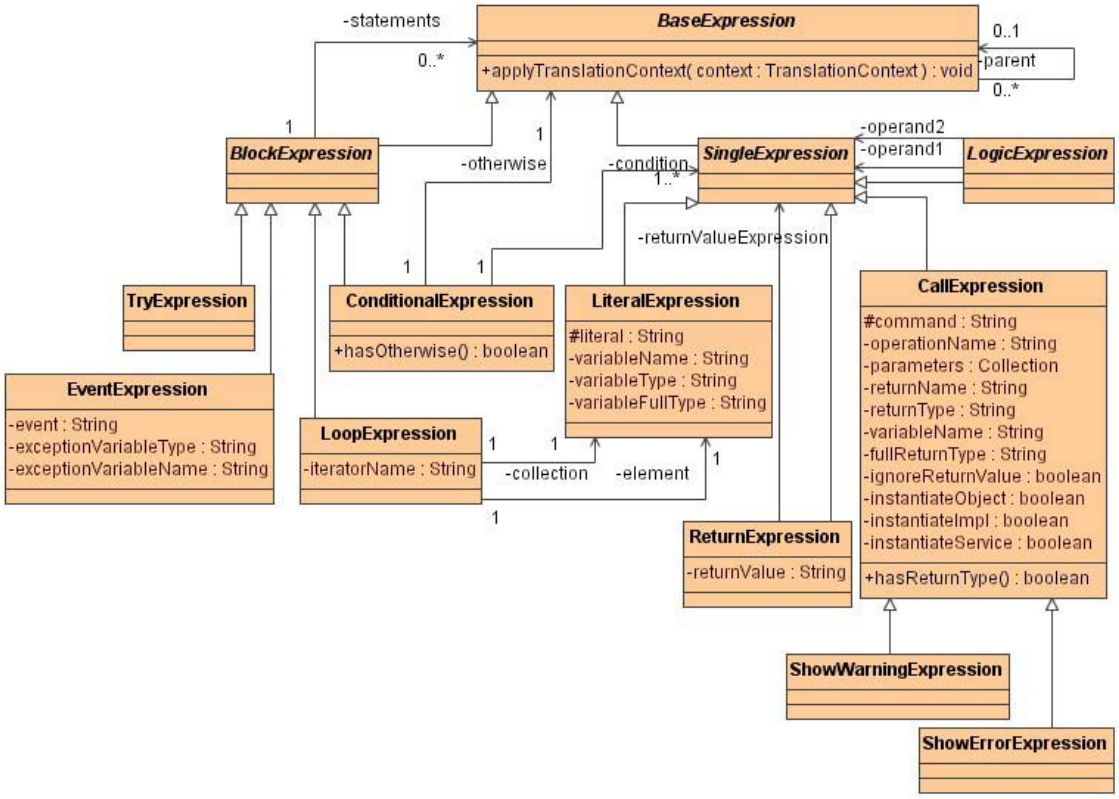


Figura 13. Diagrama de classes das expressões

Consideramos como expressões simples as expressões de invocação de métodos, as de retorno, as lógicas e as literais. As expressões de invocação de métodos são representadas pela classe *CallExpression* e as suas subclasses *ShowWarningExpression* e *ShowErrorExpression*. As expressões lógicas são representadas pelas subclasses de *LogicExpression*, que estão ilustradas na Figura 14. Uma expressão lógica possui dois operandos. Consideramos estes operandos como sendo *SingleExpression* porque eles podem ser uma outra expressão lógica ou até mesmo uma invocação de método. As expressões literais estão representadas pela classe *LiteralExpression*.

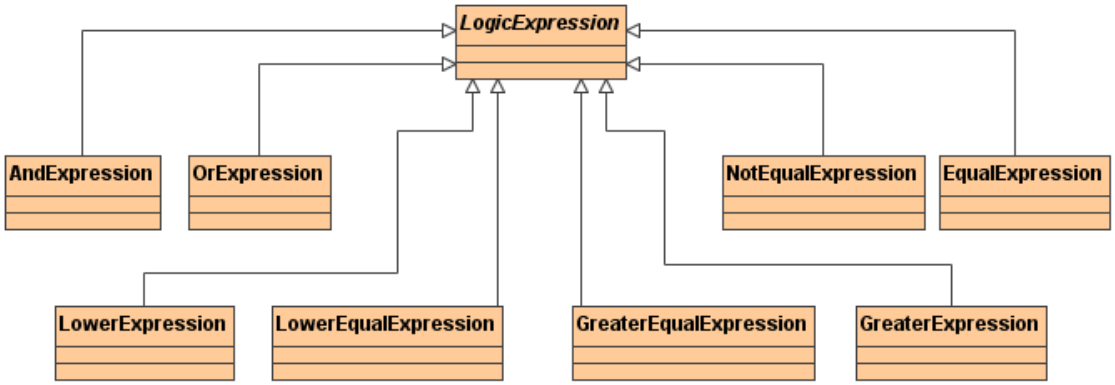


Figura 14. Diagrama de classes das expressões lógicas

3.3.1.2 Analisador semântico e síntese

Implementamos um parser que lê a especificação de ações e constrói a árvore de expressões. Depois, implementamos um transformador que percorre a árvore de expressões e a traduz em código. No transformador, criamos um gerenciador de contexto para controlar a criação das variáveis e o escopo durante a tradução. O gerenciador de contexto armazena as variáveis do escopo corrente. O contexto de tradução é sempre iniciado com as variáveis que representam os parâmetros da operação. O contexto então é aplicado ao nó raíz da árvore de expressões, que por conseqüência dispara a aplicação do contexto para os seus filhos e assim sucessivamente, de modo que eles possam descobrir, em função do contexto corrente, quais variáveis devem ser envolvidas na tradução da respectiva expressão. Dessa forma, novas variáveis podem ser criadas e adicionadas ao contexto corrente. A Figura 15 ilustra a modelagem do gerenciador de contexto.

Quando uma expressão complexa recebe um contexto para ser aplicado, ela cria um novo contexto com base no recebido, e propaga este novo contexto para os seus filhos. Após a tradução desta expressão complexa, este contexto que foi criado é descartado. Este comportamento simula o conceito de que uma variável declarada dentro do escopo de uma iteração, por exemplo, não pode ser acessada fora dele.

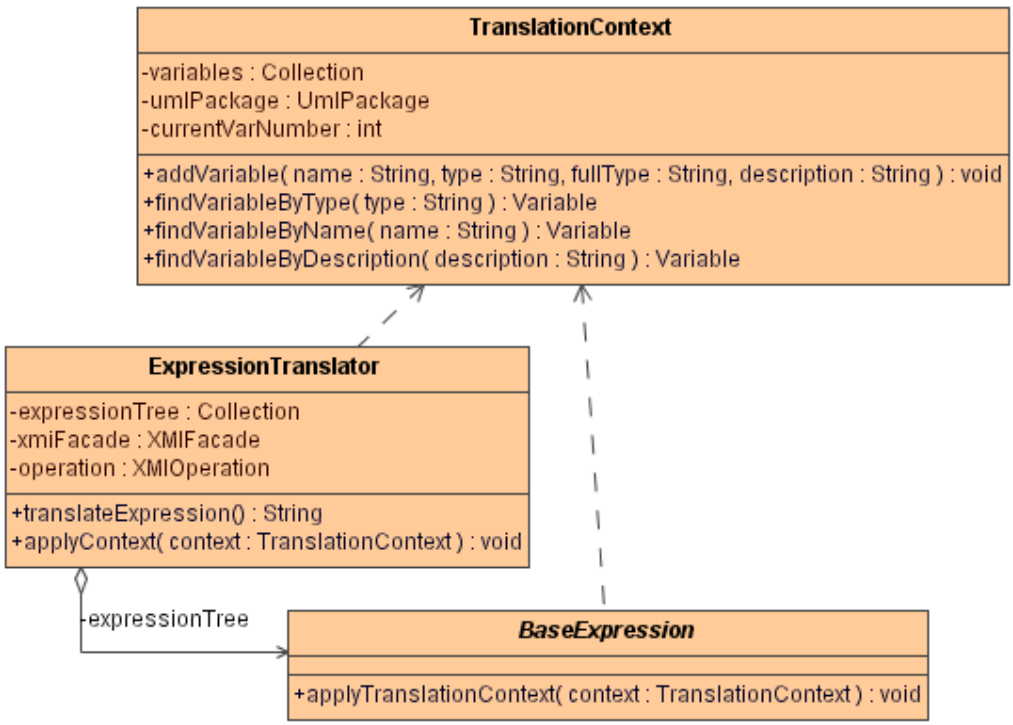


Figura 15. Diagrama de classes de tradução

A partir da especificação de ações, o parser gera a árvore de expressões contendo a seqüência lógica das expressões de acordo com a gramática. Essas expressões consistem de invocações de operações do modelo e estruturas de controle (condicionais, iterativas e eventuais). Depois que a árvore de expressões é criada, iniciamos a busca em profundidade solicitando a tradução de cada expressão de acordo com o seu tipo.

Quando a expressão é uma invocação de método, é buscado no modelo o método que tem a descrição correspondente. Se o método não for encontrado, é mostrado um erro. Quando o método é encontrado, o contexto de tradução é analisado para descobrir se já há uma instância da classe que contém o método desejado. Se houver, essa instância é selecionada. Caso contrário, é verificado se o método é estático ou não. Se for estático, a classe não é instanciada para fazer a chamada ao método. Se não for estático, é verificado se a classe tem estereótipo *Entity* ou *Service*. Se for *Service*, a interface de negócio correspondente é obtida e se for *Entity*, a classe *DAOImpl* associada à entidade é instanciada. Depois, é

verificado se o método tem parâmetros e retorno. Se tiver parâmetros, é feita uma busca no contexto de tradução por variáveis que tenham a mesma descrição do parâmetro (a busca privilegia as variáveis mais recentemente utilizadas). Se nenhuma variável for encontrada, é verificado se as variáveis do contexto contêm atributos ou associações com a descrição procurada (as associações são navegadas recursivamente e existem condições de parada para evitar navegação infinita em associações circulares). Para finalizar, se a variável não for encontrada, uma busca por tipo é feita. A busca por tipo é análoga à busca por descrição. Ainda assim, se a variável não for encontrada, um erro é mostrado. Quando o método tem retorno, o retorno é adicionado como uma variável no contexto, que poderá ser utilizada na tradução das expressões seguintes.

Quando a expressão se refere ao tratamento de eventos, é feita uma busca no modelo pela exceção que tem a descrição correspondente. Se a exceção for encontrada, ela é utilizada para tratar o evento. Caso contrário, um erro é mostrado.

Quando a expressão é condicional, é verificado se existe mais de uma condição. Cada condição pode ser uma invocação de método ou uma expressão lógica. Se a condição for uma invocação de método, é verificado se o método existe no modelo. Se o método existe, ele é processado. Caso contrário, um erro é mostrado. Se a condição for uma expressão lógica, é verificado se os operandos já existem no contexto de tradução. Se os operandos não forem encontrados, um erro é mostrado.

Quando a expressão é uma iteração, é verificado se a coleção a ser iterada já existe no contexto de tradução. Se existir, o tipo de cada elemento da coleção é inferido através da busca por relacionamentos de dependência dessa operação com as demais classes do modelo. Se a coleção ou a dependência (que determina o tipo dos elementos) não for encontrada, um erro é mostrado.

O processo de tradução da especificação de ações em código é apresentado na Figura 16.

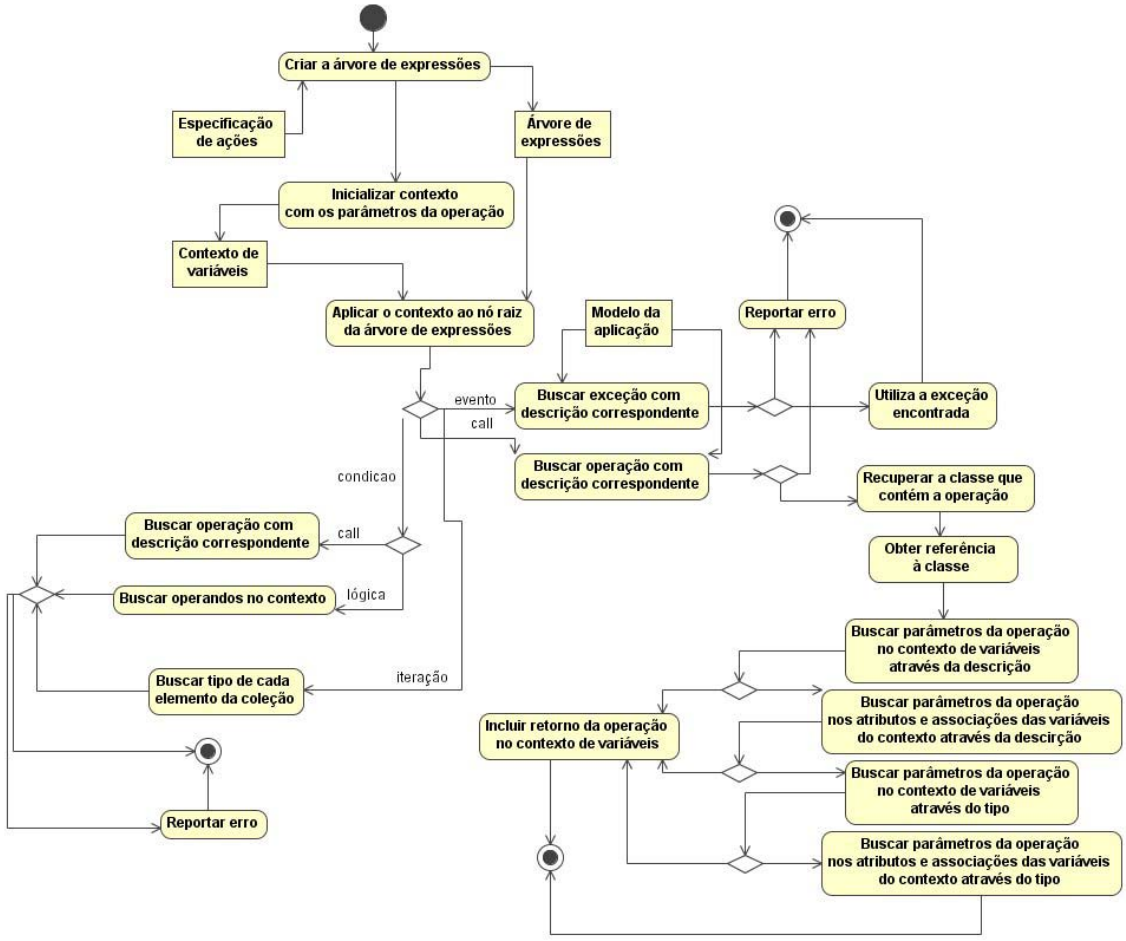


Figura 16. Processo de tradução da especificação de ações

3.3.1.3 Integração com uma ferramenta MDA

Na concepção da solução de integração com uma ferramenta MDA, consideramos como principal requisito a necessidade de manter a independência da ferramenta. Para isso, definimos um conjunto de interfaces para desacoplar as responsabilidades, como mostrado na Figura 17. Desta forma, quando for necessário integrar com alguma outra ferramenta MDA basta implementar estas interfaces de integração.

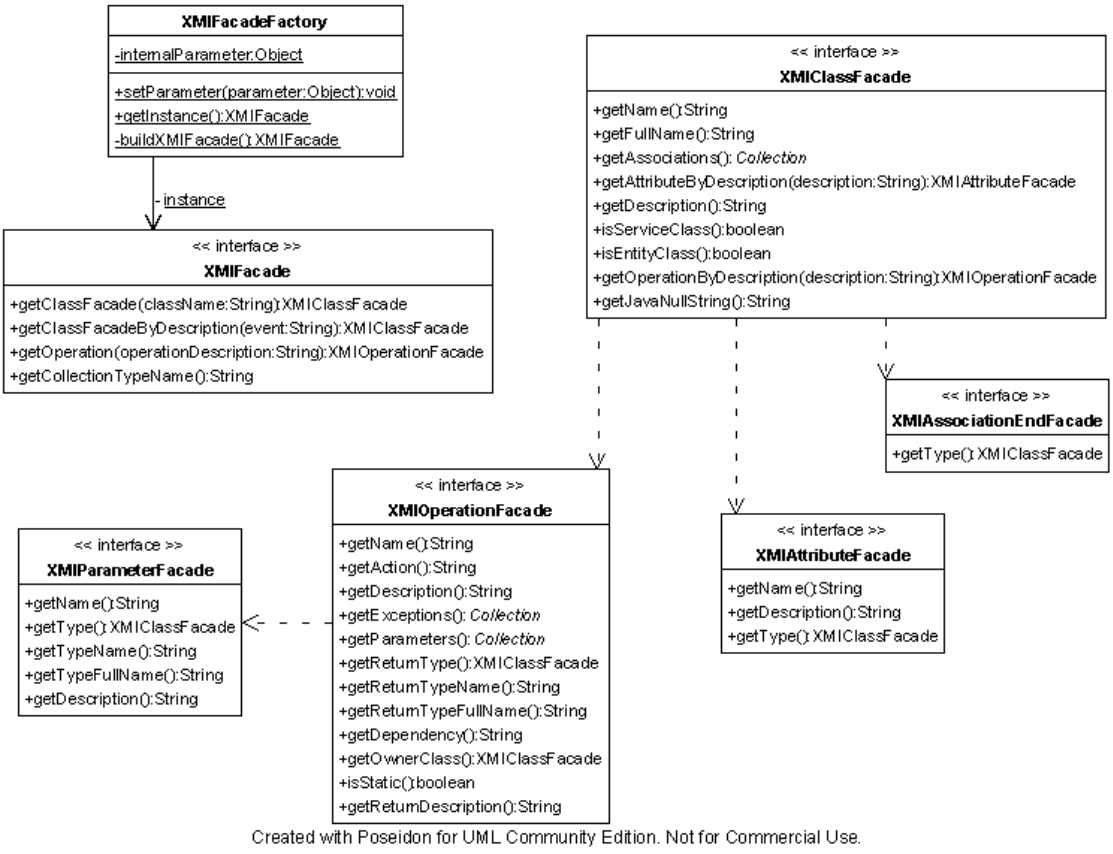


Figura 17. Modelagem das interfaces de integração com uma ferramenta MDA

As principais interfaces são:

- *XMIFacade* → Representa a interface de uma classe utilitária, que consiste de operações para obter uma determinada classe a partir do nome ou da descrição e obter uma determinada operação a partir da sua descrição.
- *XMIClassFacade* → Representa a interface de uma fachada para a classe correspondente às classes do modelo. Esta interface consiste basicamente das operações para obter: o nome, o nome completo, as associações, o atributo e a operação a partir de uma descrição e a identificação da classe como um componente de serviço ou entidade.

- *XMIAssociationFacade* → Representa a interface de uma fachada para a classe correspondente às associações entre as classes do modelo. Esta interface consiste de uma operação para obter o tipo da classe da associação.
- *XMIAttributeFacade* → Representa a interface de uma fachada para a classe correspondente aos atributos do modelo. Esta interface consiste de operações para obter: o nome, o tipo e a descrição do atributo.
- *XMIOperationFacade* → Representa a interface de uma fachada para a classe correspondente às operações do modelo. Consiste de operações para obter: o nome, a ação, a descrição, os parâmetros, as exceções, o tipo de retorno e as dependências de uma operação.
- *XMIParameterFacade* → Representa a interface de uma fachada para a classe correspondente aos parâmetros das operações do modelo. Consiste de operações para obter: o nome, o tipo e a descrição do parâmetro.

A classe *XMIFacadeFactory* é responsável pela criação da classe XMIFacade. Escolhemos a ferramenta AndroMDA (ANDROMDA, 2005), uma ferramenta de código aberto que é extensível, bem documentada e amplamente utilizada, para desenvolver nosso estudo. Uma breve comparação entre as ferramentas MDA existentes pode ser encontrada em (CUNHA, 2005). Para integrar com o AndroMDA, criamos um conjunto de classes que implementam essas interfaces (como mostra a Figura 18). Essas classes são como adaptadores. Se quisermos integrar nossa ferramenta com outra ferramenta MDA, é necessário apenas implementar essas interfaces de adaptação.

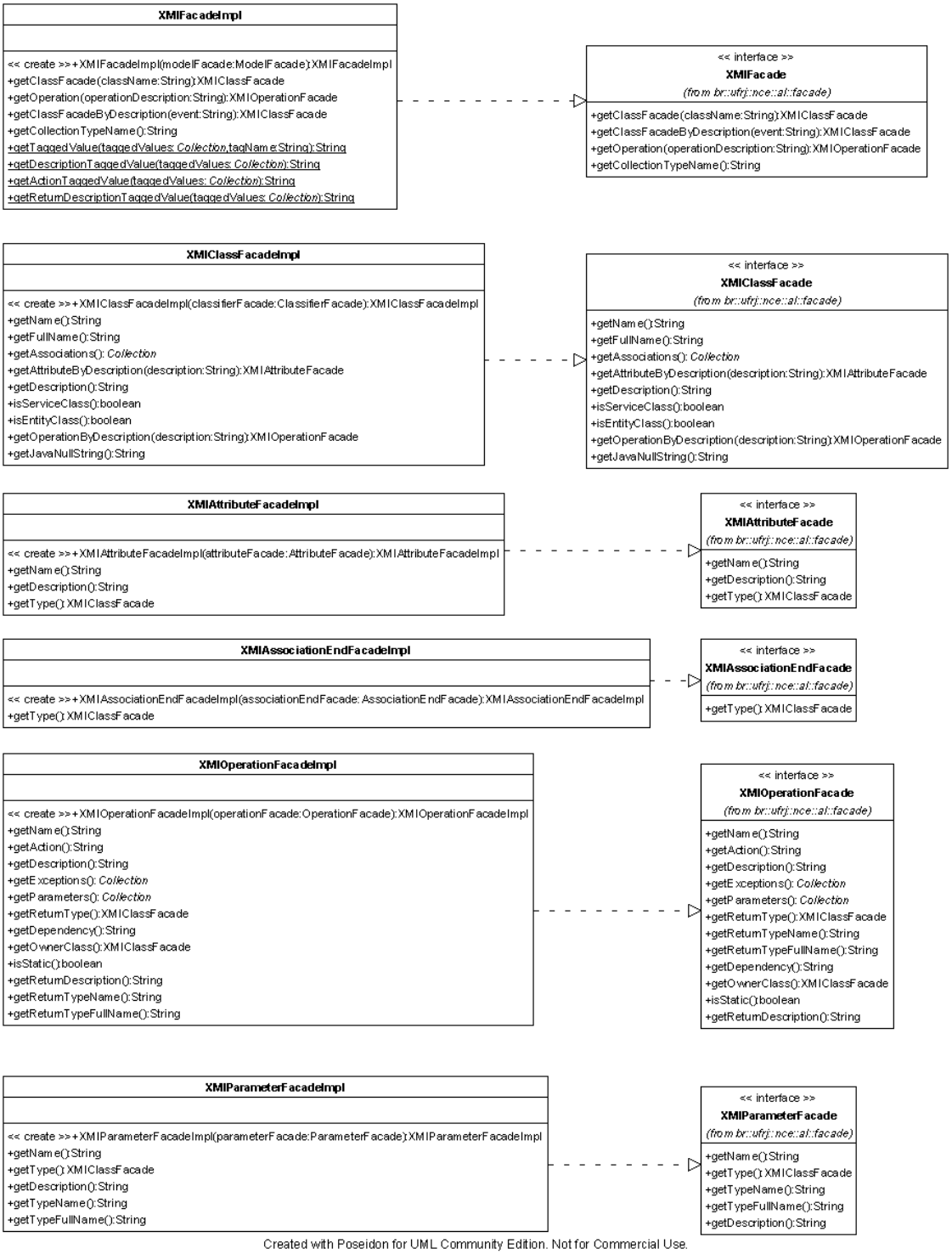


Figura 18. Modelagem das classes de integração

As transformações do AndroMDA são baseadas em cartuchos. Um cartucho é um conjunto de *templates*, escritos em Velocity (VELOCITY, 2005), para uma plataforma

específica. Hibernate (HIBERNATE, 2006) (um *framework* de persistência objeto-relacional) e Enterprise Java Beans (EJB) (EJB, 2003) (uma arquitetura de componentes que simplifica o processo de construção de aplicações corporativas em Java) são alguns exemplos de cartuchos disponíveis no AndroMDA. O AndroMDA sabe qual cartucho deve ser aplicado a partir dos estereótipos e da configuração de cada projeto. Alguns estereótipos pré-definidos são Entity e Service. Classes estereotipadas como Entity podem ser transformadas em classes e descritores Hibernate, por exemplo, caso essa seja a plataforma escolhida. Nesse caso, o AndroMDA gera todos os artefatos necessários para aplicações de cadastro (classes e arquivos de mapeamento). Classes estereotipadas como Service podem ser transformadas em Session Beans (que é um tipo de componente EJB), por exemplo, caso essa seja a plataforma escolhida. Nesse caso, o AndroMDA gera apenas o esqueleto de código e os descritores de instalação. É este o ponto que mereceu a integração com a nossa ferramenta. A ferramenta AndroMDA tem os seguintes mecanismos de extensão: cartuchos, *metafacades* e *translation libraries*. Para implementar essa integração, utilizamos *metafacades*. Desiderati (2006) descreve detalhadamente esses mecanismos de extensão do AndroMDA. O Anexo 1 mostra a modelagem da extensão da *metafacade* do cartucho EJB. Como pode-se obervar, criamos a classe *ActionOperationFacade*, que estende a metafacade *EJBOperationFacade*. Esta classe consiste da inclusão do método *translateAction*, que é responsável pela tradução das ações da operação. O Anexo 2 mostra o *template* que utiliza essa extensão que implementamos. O Anexo 3 mostra a implementação da extensão da *metafacade*.

Uma customização do AndroMDA que implementamos foi a geração de esqueleto de código para os métodos definidos em classes com estereótipos Entity. Isto foi necessário para que no nível do modelo, seja possível invocar operações da camada de domínio. Quando ocorre uma invocação de operação de domínio, instanciamos a classe DAO correspondente para então invocar a operação.

3.3.2 Editor da linguagem

Reconhecemos que não é prático para o projetista ter que conhecer ou buscar as descrições de todos os elementos a fim de utilizá-las na especificação de ações. Para facilitar a utilização da linguagem, foi desenvolvida uma ferramenta em nosso grupo de pesquisa, como projeto final de graduação (CIANNI; CABEÇO, 2006). A principal característica desse editor é considerar os princípios básicos de projetos de interface: consistência, uso de atalhos, simplicidade, antecipação e familiaridade. Essa ferramenta guia o usuário na especificação de ações e foi desenvolvida utilizando a biblioteca *Standard Widget Toolkit* (SWT) (SWT, 2006).

Para especificar ações no editor da linguagem, o usuário seleciona um modelo, representado no padrão XML Metadata Interchange (XMI) (XMI, 2005), para especificar as ações de cada operação. Em seguida, a ferramenta mostra todas as classes agrupadas por pacote, na mesma forma em que são apresentadas nas ferramentas de modelagem. Quando uma classe é expandida, a ferramenta mostra suas operações, e quando uma operação é expandida, a ferramenta mostra seus parâmetros de entrada (seta apontando para a direita) e valor de retorno (seta apontando para a esquerda). Com um clique duplo em uma operação, a ferramenta abre a especificação de ações correspondente no lado direito e na parte superior. O editor oferece algumas facilidades de *auto-complete* para editar as ações. As outras funcionalidades do editor são a validação e tradução da especificação, e a geração de fluxogramas.

Para validar a especificação de ações de uma determinada operação, o editor dispõe de uma função de validação. Nesse caso, o editor aciona as ferramentas de análise sintática e semântica para validar as ações de acordo com a gramática e o modelo UML, e mostra as mensagens de erro ou sucesso na parte inferior da tela. O editor também dispõe de uma função de tradução que exibe o código-fonte correspondente às ações da operação selecionada.

Ainda, é possível visualizar o fluxograma que representa graficamente as ações da operação corrente. Na ferramenta Visual Rules (VISUALRULES, 2006), os usuários podem visualizar a lógica de negócio, onde as regras podem ser modeladas graficamente e facilmente integradas nas aplicações. Seguindo a mesma idéia, quando os usuários descrevem as ações de determinada operação, eles podem visualizar as ações em um fluxograma. Apesar da linguagem ser textual, ela pode ser apresentada graficamente.

Os usuários também têm a opção de salvar as ações que foram especificadas. Quando esta opção é selecionada, as ações são salvas no modelo no valor etiquetado "operation.behaviour" da operação correspondente. A Seção 4.1 apresenta em detalhes a utilização do editor.

4 AVALIAÇÃO DA LINGUAGEM E DAS FERRAMENTAS

Experimentação é um modo sistemático e controlado para avaliação. Novos métodos, técnicas, linguagens e ferramentas devem ser experimentados para obter uma comparação com os já existentes (TRAVASSOS; GUROV; AMARAL, 2002).

A linguagem Natural MDA, proposta neste trabalho, tem como principal objetivo complementar os modelos UML fornecendo mecanismos para a especificação do comportamento de sistemas de informação de forma a manter um equilíbrio entre os requisitos de precisão, completude e alto nível de abstração nas especificações. Esses requisitos juntos possibilitam o mapeamento automático para código e o uso e compreensão por não especialistas em linguagens de programação.

Neste capítulo, apresentamos um exemplo de aplicação construído durante o desenvolvimento das ferramentas de apoio à linguagem e um estudo de observação realizado ao final do desenvolvimento. Esses estudos permitiram avaliar parcialmente a precisão e correção da linguagem Natural MDA com relação à geração de código. A característica relacionada ao nível de abstração, fornecido pela linguagem, é uma medida difícil de ser avaliada dado que depende de questões subjetivas. Desta forma, foi necessário realizar uma avaliação qualitativa, a qual foi concretizada pelo estudo de observação.

4.1 EXEMPLO DE APLICAÇÃO

O sistema desenvolvido neste exemplo de aplicação constitui uma parte do Sistema Acadêmico da Universidade Federal do Rio de Janeiro. Esse sistema é responsável por controlar as atividades administrativas e acadêmicas da Universidade, como por exemplo, previsão de turmas, controle de histório escolar, emissão de pautas e inscrição em disciplinas. Nesse exemplo, veremos apenas os casos de uso de inscrição em disciplinas, cadastro de turmas e cadastro de notas. Nas seções seguintes, veremos a modelagem e a especificação de

ações desse sistema. A especificação foi descrita em inglês devido ao fato da gramática da linguagem Natural MDA estar em inglês. A gramática está em inglês porque foi uma adaptação da linguagem proposta por Silva e Pinheiro (SILVA; PINHEIRO, 2004).

4.1.1 Modelagem da aplicação

A Figura 19 mostra os casos de uso do Sistema Acadêmico que foram considerados neste exemplo.

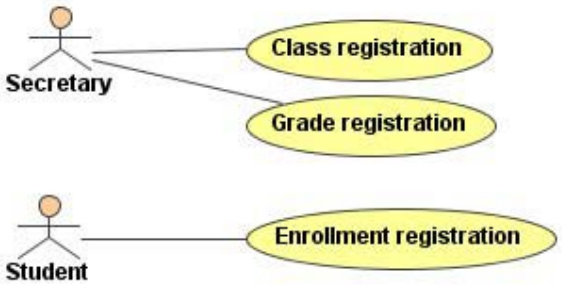


Figura 19. Casos de uso do Sistema Acadêmico

Como o exemplo de aplicação visa avaliar a utilização da linguagem, não detalhamos as atividades relacionadas a levantamento de requisitos, elaboração de cronograma e planejamento do projeto. Na fase de Concepção, obtivemos os requisitos já definidos com a equipe que desenvolveu o projeto real. Em seguida, entramos na fase de Elaboração para definir a arquitetura candidata e implementar uma prova de conceito. A Figura 20 ilustra a interação entre as camadas da aplicação. Seguindo o modelo arquitetural em camadas, a camada de interface aciona os componentes da camada de serviço, que por sua vez acionam os componentes de domínio.

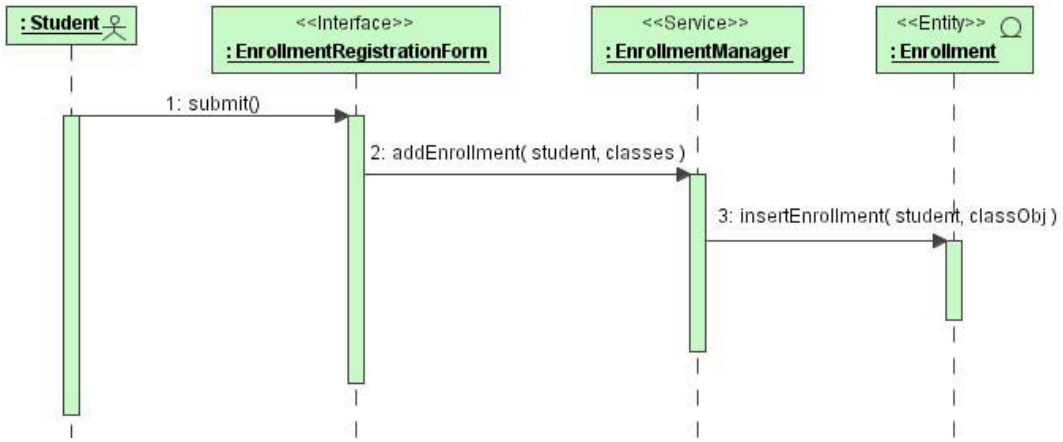


Figura 20. Arquitetura das camadas da aplicação

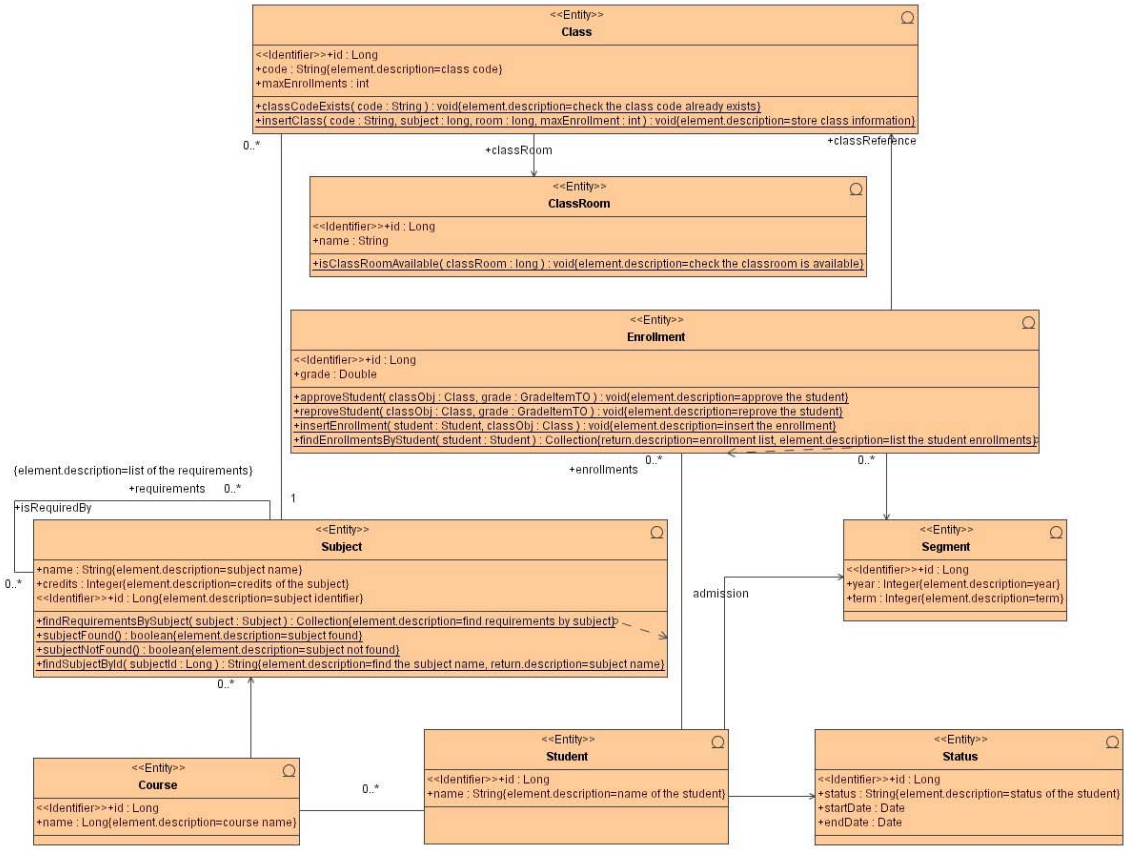


Figura 21. Diagrama de classes de domínio do sistema acadêmico

No domínio de aplicação do Sistema Acadêmico, identificamos que um curso (Course) tem disciplinas (Subject). Uma disciplina pode ter outras disciplinas como pré-requisito e são oferecidas várias turmas (Class) para uma mesma disciplina. Uma turma está

associada a uma sala de aula (ClassRoom). A cada período (Segment), os alunos (Student) se inscrevem em disciplinas.

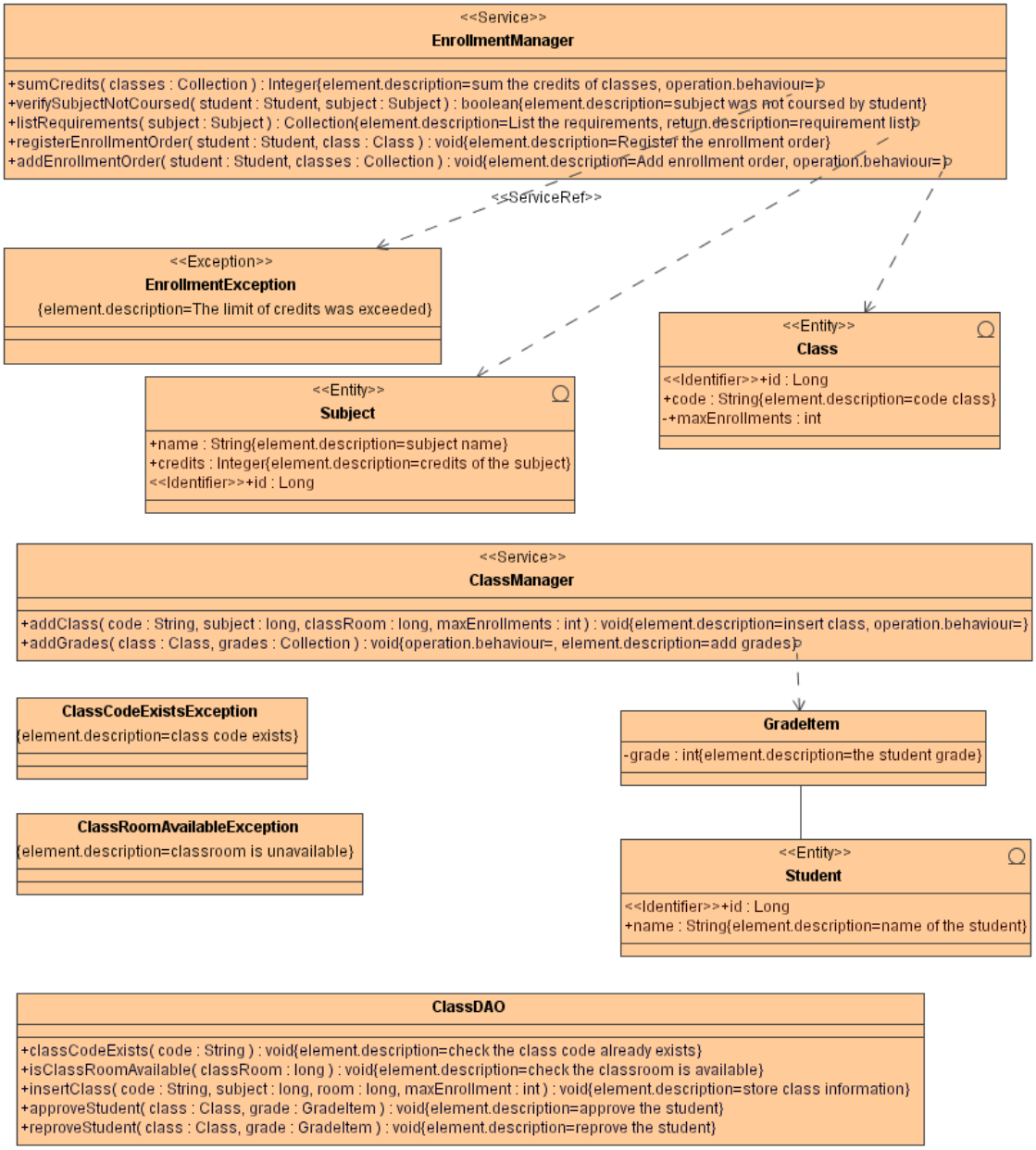


Figura 22. Diagrama de classes de serviço do sistema acadêmico

Na Figura 21, apresentamos a modelagem das classes do domínio da aplicação, com seus respectivos atributos e relacionamentos.

O passo seguinte foi o detalhamento dos casos de uso da Figura 19. A Figura 22 mostra a modelagem inicial dos componentes de serviço resultantes da análise dos casos de

uso. O método “addEnrollmentOrder” do componente EnrollmentManager realiza o caso de uso de inscrição de aluno em turmas. Os métodos “addClass” e “addGrade” do componente ClassManager realizam os casos de uso de cadastro de turmas e de cadastro de notas, respectivamente. Para especificar as ações destes métodos, definimos outros métodos – apenas para fatorar as responsabilidades envolvidas. Repare que todos os métodos estão associados à tag “element.description”. O valor atribuído nesta tag foi utilizado nas especificações de ações dos métodos.

Em seguida, iniciamos o refinamento dos casos de uso utilizando a linguagem Natural MDA e o editor da linguagem apresentado anteriormente. Importamos o modelo no editor, e a Figura 23 e a Figura 24 mostram como a modelagem das camadas de domínio e de serviço é apresentada no editor da linguagem, respectivamente.

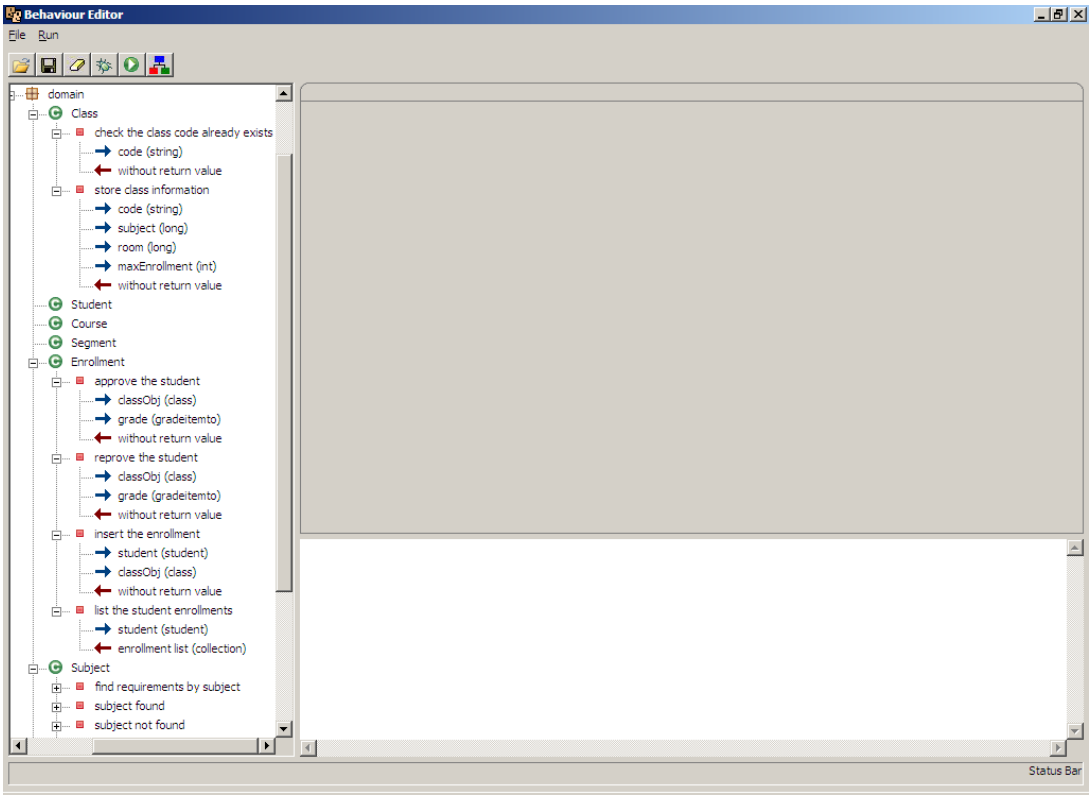


Figura 23. Elementos da camada de domínio apresentada no editor

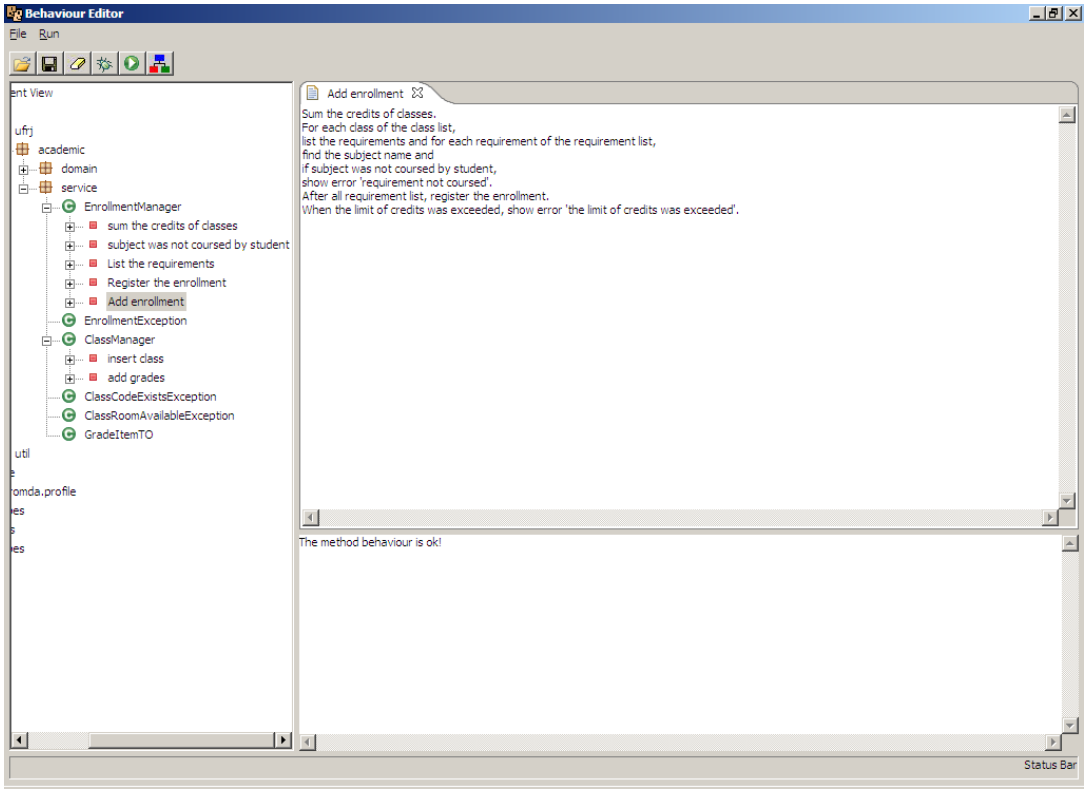


Figura 24. Elementos da camada de serviço apresentada no editor

No lado esquerdo da Figura 23 e da Figura 24, temos exemplos de descrições dos elementos do modelo importado. Para cada operação, temos também as descrições de cada parâmetro de entrada e valor de retorno. Assim, para a classe “Enrollment”, temos: (i) a operação “approveStudent” com descrição “approve student” e parâmetros de entrada “classObj” e “grade”; (ii) a operação “reproveStudent” com descrição “reprove the student” e parâmetros de entrada “classObj” e “grade”; (iii) a operação “insertEnrollment” com descrição “insert the enrollment” e parâmetros de entrada “student” e “classObj”; (iv) a operação “findEnrollmentsByStudent” com descrição “list the student enrollments”, parâmetro de entrada “student” e parâmetro de saída “enrollment list”.

4.1.2 Especificação de ações

Após a modelagem do sistema, incluindo as devidas marcações (estereótipos e valores etiquetados) no modelo, iniciamos a especificação de ações de cada operação. Com base nos

elementos do modelo, mostrados na Figura 23 e na Figura 24, e na gramática da linguagem Natural MDA, a Tabela 14 mostra a especificação das ações da operação "addEnrollmentOrder". Para realçar os elementos da gramática e as referências aos elementos do modelo UML, utilizamos a seguinte notação: palavras sublinhadas se referem aos elementos do modelo; palavras em itálico se referem aos elementos da gramática.

Tabela 14. Inscrição de Aluno em Turma

1	Sum the credits of classes.
2	*For each* class *of the* class list, list the requirements *and for each*
3	requirement *of the* requirement list, *if* subject was not coursed by student,
4	*show error* 'Requirement not coursed'.
5	*After all* requirement list, register the enrollment order.
6	*When* the limit of credits was exceeded, *show error* 'The limit of credits was
7	exceeded'.

Na Tabela 14, descrevemos as ações da operação de inscrição de aluno em turma. Na linha 1 da Tabela 14, fazemos a invocação de um método para verificar se o limite de créditos permitido por período foi excedido. Em seguida, verificamos se o aluno cursou os pré-requisitos das disciplinas selecionadas e se a disciplina já foi cursada anteriormente. Em caso de sucesso, o pedido de inscrição é registrado. Quando o limite de créditos é excedido, o tratamento descrito nas linhas 6 e 7 é aplicado.

A Tabela 15 mostra a implementação gerada pelo transformador a partir da especificação das ações da Tabela 14. Os passos realizados pelo transformador durante a tradução são:

- Inicialização do contexto de variáveis: os parâmetros de entrada do método a ser traduzido são incluídos no contexto. Nesse caso, foram incluídas duas variáveis: a variável "student" do tipo "Student" e a variável "classes" do tipo "Collection" e com descrição "class list".

- Mapeamento das ações com as operações do modelo: para cada ação utilizada na especificação de ações buscamos uma operação no modelo que tenha a mesma descrição. Se a operação tem um valor de retorno, este é incluído como uma variável no contexto. Como exemplos, podemos ver: (i) a ação “sum the credits of classes” que teve correspondência com a operação “sumCredits” devido ao conteúdo do valor etiquetado nessa operação; (ii) a ação “list the requirements” que também teve correspondência com a operação “listRequirements” devido ao conteúdo do valor etiquetado nessa operação.
- Manipulação das variáveis do contexto com as entradas e saídas de cada operação: quando precisamos invocar uma operação que possui parâmetros de entrada, buscamos no contexto variáveis que atendam a essa necessidade através do tipo da variável e/ou descrição. Como exemplo, veja a operação “sum the credits” que espera receber uma coleção com descrição “class list”. Como há uma variável no contexto com essa descrição (foi adicionada no primeiro passo), esta foi utilizada como entrada para a operação em questão.
- Tratamento de iterações: localizamos no contexto a variável com descrição da coleção a ser iterada e incluimos uma nova variável que corresponde a cada elemento da coleção. Nesse caso, podemos ver a iteração sobre “class list” em “for each class of class list”, que corresponde à variável já citada anteriormente. Em relação à inclusão de uma variável que corresponde a cada item da coleção, incluimos a variável do tipo “Class” devido à dependência declarada no modelo dessa coleção com a classe “Class”.
- Tratamento de eventos: localizamos no modelo a exceção com a descrição equivalente ao evento citado na descrição de ações e a utilizamos na tradução.

Nesse caso, podemos ver o mapeamento entre o evento “the limit of credits was exceeded” e a exceção “EnrollmentException”.

Tabela 15. Código gerado para Inscrição de Aluno em Turma

```
public void addEnrollment(br.ufrj.academic.domain.Student student,
        java.util.Collection classes) {
    try {
        java.lang.Integer var1 = this.sumCredits(classes);
        for (java.util.Iterator var2 = classes.iterator(); var2.hasNext();) {
            br.ufrj.academic.domain.Class var3 = (br.ufrj.academic.domain.Class)
var2.next();
            java.util.Collection var4 = this.listRequirements(var3.getSubject());
            for (java.util.Iterator var5 = var4.iterator(); var5.hasNext();) {
                br.ufrj.academic.domain.Subject var6 =
(br.ufrj.academic.domain.Subject) var5.next();
                java.lang.String var7 =
br.ufrj.academic.domain.Subject.findSubjectById(var6.getId());
                if (this.verifySubjectNotCoursed(student, var7)) {
                    logger.error("requirement not coursed");
                }
            }
            this.registerEnrollment(student, var3);
        }
    } catch (br.ufrj.academic.service.EnrollmentException var2) {
        logger.error("the limit of credits was exceeded");
    }
}
```

Para facilitar a visualização da especificação de ações, o usuário pode solicitar ao editor a geração do fluxograma correspondente. O fluxograma gerado mostra as iterações, condições e invocações de operações na ordem em que elas ocorrem. A Figura 25 mostra o fluxograma correspondente à especificação de ações apresentada na Tabela 14.

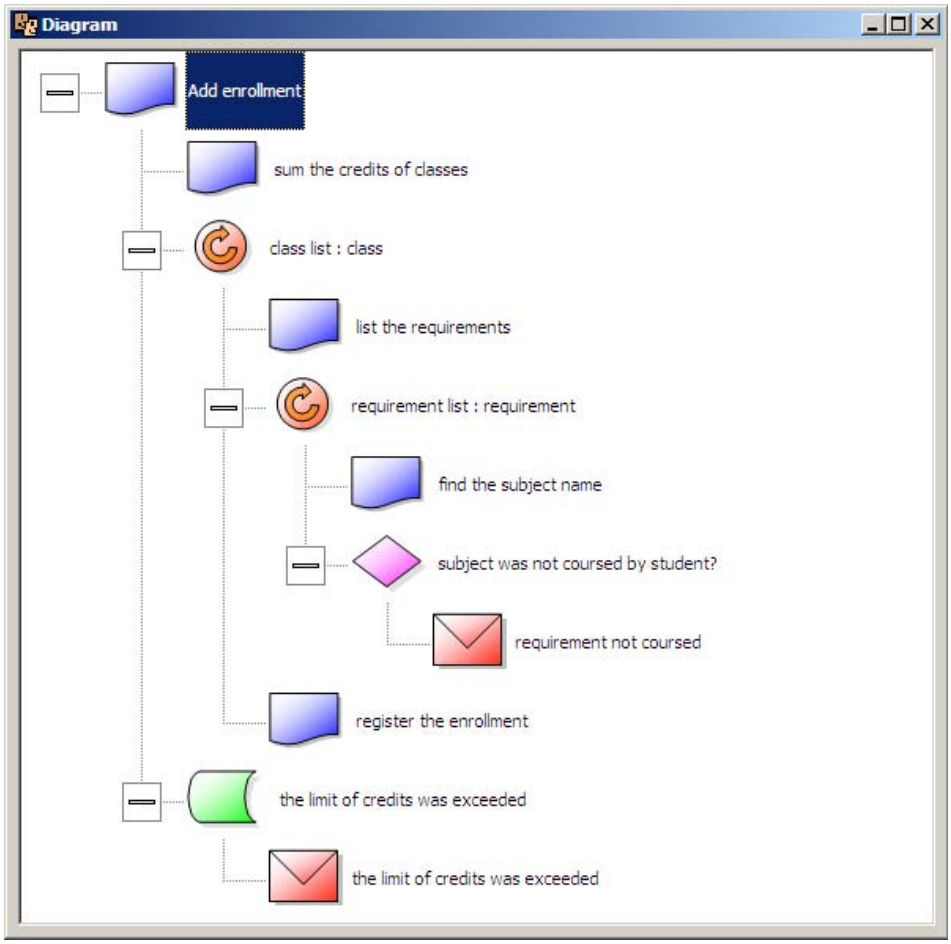


Figura 25. Fluxograma das ações de inscrição de aluno em turma

A Tabela 16 mostra o significado de cada símbolo do fluxograma.

Tabela 16. Símbolos do fluxograma

	Representa uma chamada de método (ao lado do símbolo aparece a descrição do método)
	Representa uma iteração (ao lado do símbolo aparece o nome da coleção e a referência a cada item da coleção)
	Representa uma decisão (ao lado do símbolo aparece a condição)
(em vermelho)	Representam as mensagens de erro
(em amarelo)	Representam as mensagens de alerta
	Representa um tratamento de evento (ao lado do símbolo aparece a descrição do evento)

Na Tabela 17, descrevemos as ações da operação de cadastro de turma. Na linha 1 da Tabela 17, verificamos se já existe uma turma com o código fornecido. Em seguida, verificamos se a sala a ser alocada para a turma já está ocupada e cadastramos a turma.

Quando a sala não estiver disponível ou o código da turma já existir, temos o tratamento como descrito nas linhas 4 e 5, respectivamente.

Tabela 17. Cadastro de turma

1	Check the class code already exists.
2	Check the classroom is available.
3	Store class information.
4	*When* classroom is unavailable, *show warning* 'Classroom is in use'.
5	*When* class code exists, *show warning* 'Class code exists'.

A Tabela 18 mostra a implementação gerada a partir da especificação das ações da Tabela 17. Os mesmos passos executados no processo de transformação explicado no primeiro caso de uso são válidos neste.

Tabela 18. Código gerado para cadastro de turma

```
public void addClass(java.lang.String code, long subject, long classRoom,
        int maxEnrollments) {
    try {
        br.ufrj.academic.domain.Class.classCodeExists(code);
        br.ufrj.academic.domain.ClassRoom.isClassRoomAvailable(classRoom);
        br.ufrj.academic.domain.Class.insertClass(code, subject, classRoom,
                maxEnrollments);
    } catch (br.ufrj.academic.service.ClassRoomAvailableException var1) {
        logger.warn("classroom is in use");
    } catch (br.ufrj.academic.service.ClassCodeExistsException var1) {
        logger.warn("classcode exists");
    }
}
```

A Figura 26 mostra o fluxograma, gerado pelo editor da linguagem, correspondente à especificação de ações apresentada na Tabela 17.

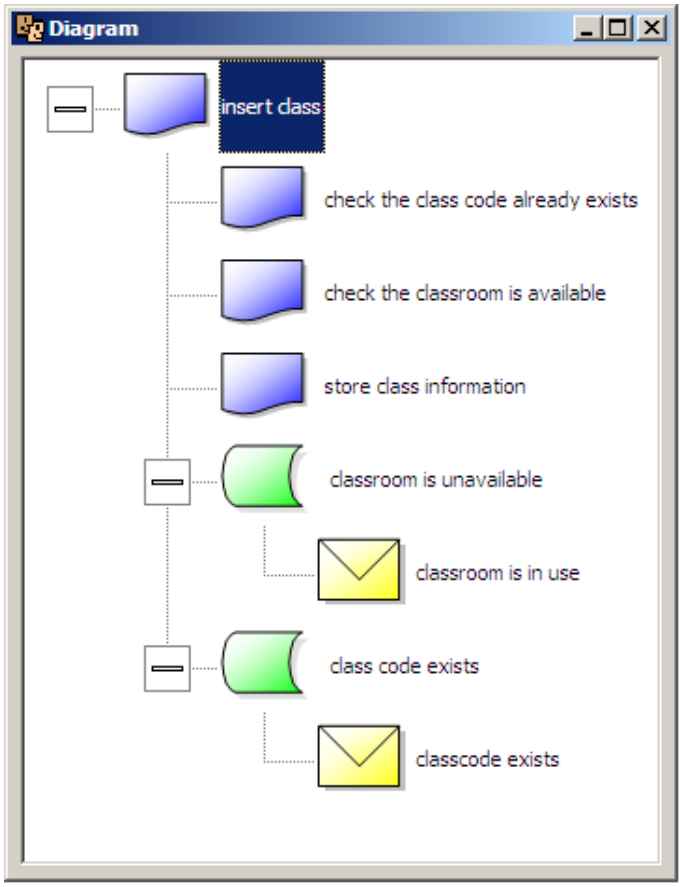


Figura 26. Fluxograma das ações de cadastro de turma

Na Tabela 19, descrevemos as ações da operação de cadastro de notas. Iteramos sobre a coleção de notas e, se a nota na disciplina foi superior a 5, o aluno é aprovado. Caso contrário, o aluno é reprovado.

Tabela 19. Cadastro de notas

1	*For each* item *of the* student grade list,
2	*If* the student grade *is greater or equal to* 5, approve the student. *Otherwise,*
3	reprove the student.

A Tabela 20 mostra a implementação gerada a partir da especificação das ações da Tabela 19. Os mesmos passos executados no processo de transformação explicado no primeiro caso de uso foram repetidos neste.

Tabela 20. Código gerado para cadastro de notas

```
public void addGrades(br.ufrj.academic.domain.Class classObj,
        java.util.Collection grades) {
    for (java.util.Iterator var1 = grades.iterator(); var1.hasNext();) {
        br.ufrj.academic.service.GradeItemTO var2 =
(br.ufrj.academic.service.GradeItemTO) var1.next();
        if (var2.getGrade() >= 5) {
            br.ufrj.academic.domain.Enrollment.approveStudent(classObj, var2);
        } else {
            br.ufrj.academic.domain.Enrollment.reproveStudent(classObj, var2);
        }
    }
}
```

4.1.3 Avaliação

Analisando o exemplo de aplicação é possível notar que a linguagem Natural MDA permite especificar ações de forma precisa e completa e, ao mesmo tempo, com um nível de abstração alto e voltado ao domínio da aplicação fazendo com que o projetista não precise se preocupar com detalhes de sintaxe dependente de plataforma. Contudo, sabemos que esse exemplo - e outros executados após o desenvolvimento da linguagem – não são suficientes para demonstrar de forma definitiva a precisão e completude da linguagem. Para tanto, pretende-se desenvolver um trabalho de pesquisa que visa provar formalmente essas características da Natural MDA.

Como esperado, constatou-se uma significativa redução da quantidade de caracteres que se deve escrever para representar o comportamento de sistemas, se comparada com a quantidade de caracteres de código gerado. A Tabela 21 mostra as proporções de caracteres na especificação e na implementação dos casos de uso apresentados. Mais importante que a redução de caracteres, temos como vantagem a facilidade de leitura, devido à redução das especificidades dos construtores das linguagens de programação.

Tabela 21. Proporção de caracteres na especificação x implementação

	Inscrição em turma	Cadastro de turma	Cadastro de nota
Caracteres especificação	300	188	116
Caracteres implementação	803	480	401
Proporção de caracteres (especificação/implementação)	37%	39%	28%

Uma vantagem que pôde ser observada no exemplo de aplicação do Sistema Acadêmico é que o fluxo principal do caso de uso se mantém separado do fluxo alternativo – o que facilita a legibilidade das regras de negócio. Essa vantagem pode ser vista na Tabela 17, onde o fluxo principal é especificado nas primeiras três linhas e os fluxos alternativos são especificados nas duas últimas linhas.

Como limitação, identificamos a dificuldade de *refactoring* (por exemplo, troca da descrição dos métodos) pois se alterarmos a descrição de uma determinada operação, seria necessário alterar a especificação de ações de todas as operações que a utilizam. Acreditamos que isso poderia ser tratado pelo editor da linguagem, já que as ferramentas de modelagem de sistemas e o processador da linguagem não têm esse conhecimento. Atualmente, o editor altera apenas a especificação das ações. A partir do momento em que o editor controlar as descrições das ações, ele poderá efetuar o *refactoring* corretamente.

Outra limitação identificada foi o fato de que as especificações de ações que invocam operações que lançam exceções deveriam tratá-las. Acreditamos que o editor da linguagem poderia identificar esses casos e alertar ao usuário, oferecendo opções de tratamento de eventos.

4.1.4 Expressividade dos diagramas comportamentais da UML

Para refinar o caso de uso, o projetista tem a opção de fazê-lo utilizando a linguagem Natural MDA ou os diagramas de seqüência e de colaboração da UML. A seguir, há algumas afirmações a respeito das limitações desses diagramas.

> "One of the great strengths of an interaction diagram is its simplicity. (…) They do, however, have weaknesses, the principal one is that although they are good at describing behavior: they do not define it. They typically do not show all the iteration and control that is needed to give a computationally complete description. (…). Jacobson uses pseudo-code in conjunction with sequence charts to provide a more executable model". (STOTTS, 2000)

> "While message sequence charts (MSCs) are widely used in industry to document the interworking of processes or objects, they are expressively weak, being based on the modest semantic notion of a partial ordering of events". (DAMM; HAREL, 2001)

Utilizando os recursos atuais da UML, não é possível gerar completamente o código a partir de modelos. O principal problema nas especificações de comportamentos de sistemas é que os diagramas de interação da UML são baseados em cenários (DAMM; HAREL, 2001). Um caso de uso pode ser modelado por um conjunto de diagramas de seqüência, onde cada um descreve um possível cenário. Isso é problema para a geração de código pois é necessário reunir os cenários definidos em cada diagrama em uma única implementação (SELONEN; SYSTA; KOSKIMIES, 2001).

Mesmo sabendo das limitações dos diagramas de interação, tentamos representar ações equivalentes na linguagem Natural MDA e nesses diagramas. A Figura 27 e a Figura 28 mostram o diagrama de colaboração e o diagrama de seqüência correspondentes ao caso de uso "Inscrição de aluno em turma".

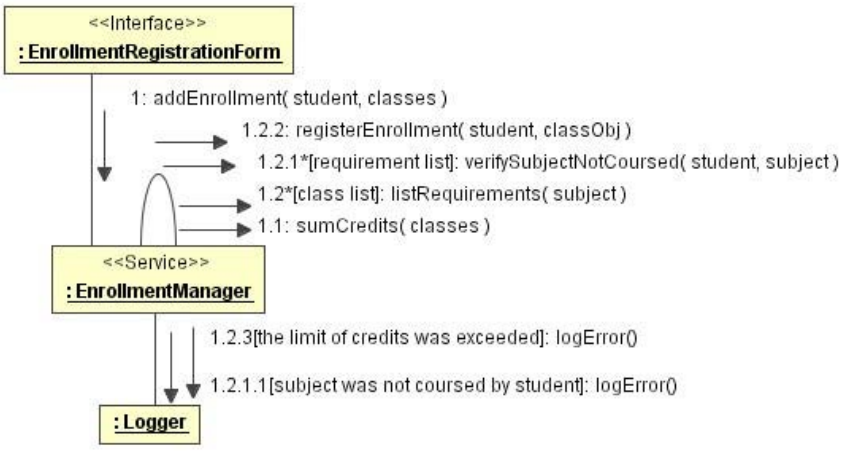


Figura 27. Diagrama de colaboração do caso de uso "Inscrição de aluno em turma"

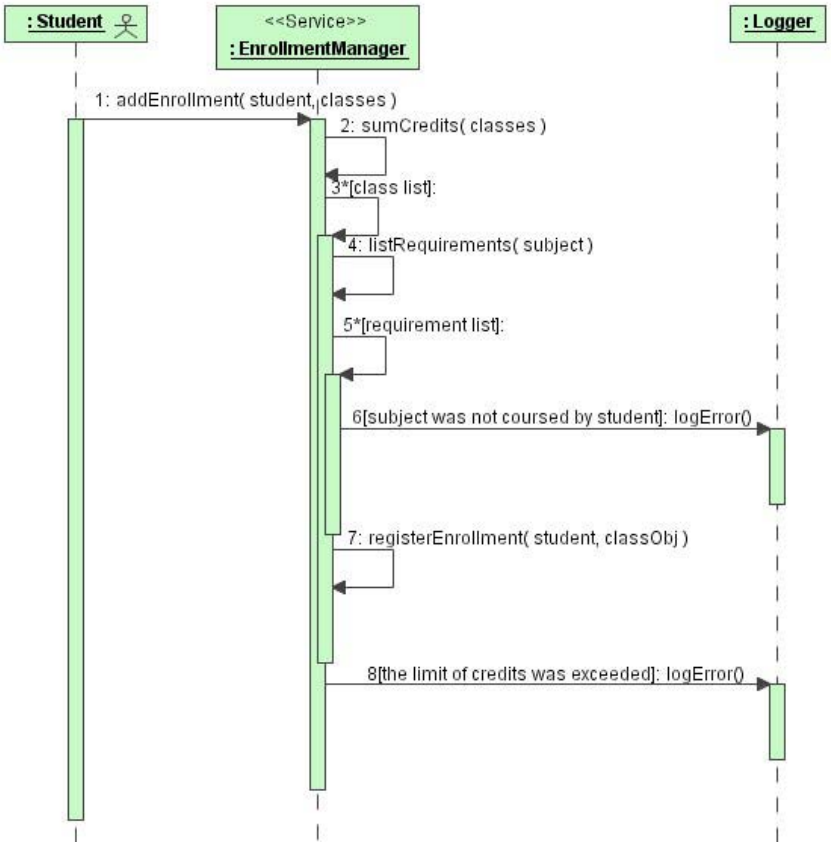


Figura 28. Diagrama de seqüência do caso de uso "Inscrição de aluno em turma"

Ao fazer os diagramas de seqüência e de colaboração, identificamos as seguintes dificuldades:

- Especificação de iterações: Na ferramenta de modelagem utilizada, *Magic Draw 9.5 Personal Edition*, foi possível declarar iterações, mas a coleção a ser iterada, que precisa ser informada, pode ser qualquer *string*. Isso pode ser observado na iteração sobre a lista de requisitos. Utilizamos a descrição do parâmetro de retorno do método que lista os requisitos de uma determinada disciplina, mas a UML não impõe restrições sobre esse valor.
- Especificação de condições: A dificuldade encontrada na especificação de iterações também foi encontrada na especificação de condições. Além disso, não conseguimos representar condições aninhadas e declarar uma seqüência de mensagens a serem enviadas no caso da condição não ser satisfeita.

- Tratamento de eventos: Não encontramos recursos para declarar que determinados eventos poderiam ocorrer e, em caso de ocorrência, como devem ser tratados.

4.2 ESTUDO DE OBSERVAÇÃO

O exemplo de aplicação descrito anteriormente consiste de um estudo de viabilidade, realizado com o intuito de prover uma avaliação inicial e informações para o aprimoramento dos requisitos da linguagem de ações e testes da implementação das ferramentas de apoio à linguagem Natural MDA. Após este estudo de viabilidade, um estudo de observação foi realizado para coletar avaliações de outras pessoas. O plano desse experimento seguiu o modelo definido por (BARROS; WERNER; TRAVASSOS, 2005).

> A realização de um estudo experimental geralmente pode ser dividida em cinco fases: a definição, o planejamento, a execução, a análise e o empacotamento do estudo. A definição do estudo consiste em resumir seus objetivos, seu foco de qualidade e os objetos que serão analisados. O planejamento envolve a descrição do perfil dos participantes, dos instrumentos, do processo de execução e uma avaliação crítica dos problemas que podem ser encontrados ao longo desta execução. A execução consiste na realização do estudo experimental pelos participantes, utilizando os instrumentos e o processo definidos no planejamento. A análise consiste na organização dos resultados gerados pelos participantes durante a execução e a realização de inferências sobre estes resultados. Finalmente, o empacotamento consiste na organização e armazenamento dos documentos construídos nas etapas anteriores, com o intuito de facilitar a repetição do estudo experimental no futuro. (BARROS; WERNER; TRAVASSOS, 2005)

Nesta seção, apresentamos um estudo de observação que avalia a viabilidade da utilização da linguagem Natural MDA. Inicialmente, definimos as etapas envolvidas no estudo e apresentamos como estas etapas foram realizadas. Em seguida, apresentamos as observações obtidas durante o estudo.

4.2.1 Definição

O objetivo do estudo foi a utilização da linguagem Natural MDA no desenvolvimento de sistemas, onde é adotada uma abordagem orientada a modelos, para geração automática de

artefatos a partir de modelos UML, visando identificar a viabilidade da utilização da linguagem no desenvolvimento de sistemas.

O foco de qualidade do estudo são as dificuldades encontradas pelos usuários no entendimento e utilização da linguagem, e a aderência aos objetivos da MDA (independente de plataforma, nível de abstração acima da implementação, executável e completa). O estudo foi desenvolvido tendo em vista a continuidade do desenvolvimento de pesquisas relacionadas com esta abordagem. Neste estudo, não estamos diretamente interessados em mensurar o ganho de produtividade derivado da utilização da linguagem. Mas sim, consideramos que melhorias podem ser identificadas através de análises e estudos futuros.

Seguindo a notação *Goal-Question-Metric* (GQM) (SOLINGEN; BERGHOUT, 1999), a definição do estudo é:

Analisar *a utilização da linguagem Natural MDA no desenvolvimento de sistemas*

Com o propósito de *avaliar a viabilidade de sua utilização*

Referente *aos ganhos obtidos por sua utilização e as dificuldades encontradas*

No contexto de *desenvolvimento de sistemas*

4.2.2 Planejamento

O contexto global considerado nesse trabalho é que as linguagens de especificação de ações estudadas até o momento oferecem mecanismos que permitem especificar sistemas com completude, de forma precisa e independente de plataforma. No entanto, estas ainda não alcançaram um nível de abstração compatível com o nível de abstração associado à modelagem de sistemas. Por outro lado, existem algumas linguagens naturais controladas já definidas e com o nível de abstração desejado pelas abordagens orientadas a modelo, mas que não são aderentes a essas abordagens.

O contexto local desse estudo tem como objetivo avaliar a viabilidade da utilização da linguagem Natural MDA no desenvolvimento de sistemas. Os participantes do estudo foram

requisitados a identificar se a linguagem engloba os requisitos de linguagens de especificação de ações (executável, completa, nível de abstração acima da implementação) e responder se as especificações de ações escritas nesta linguagem são legíveis para pessoas que não possuem conhecimento em Computação.

O treinamento dos participantes que utilizaram a linguagem Natural MDA foi realizado em duas sessões com duração de duas horas cada uma. O treinamento foi composto de duas fases, que puderam ser interrompidas a qualquer momento pelos participantes para perguntas. Na primeira fase, realizamos uma exposição sobre MDA e linguagens de especificação de ações, utilizando transparências que também foram distribuídas para os participantes. Estas transparências encontram-se no Apêndice 7. Na segunda fase, apresentamos três casos de uso e os participantes puderam acompanhar o processo de refinamento desses casos de uso e a transformação em código executável.

O estudo foi realizado em um ambiente de desenvolvimento de sistemas na área de telecomunicações. Os participantes do estudo foram desenvolvedores de sistemas, os mesmos que participaram do projeto original. Cada participante atuou como projetista. Como o custo de realização completa de um caso de uso para cada participante dificultaria o estudo experimental, decidimos criar o diagrama de classes e deixar para o participante apenas a especificação de ações de um caso de uso. Todos os participantes receberam o modelo do sistema e a definição da gramática e descreveram as ações na linguagem Natural MDA. Os participantes também receberam um questionário de avaliação da satisfação do usuário na utilização da linguagem. Neste treinamento, as ferramentas de apoio à linguagem não foram utilizadas pelo fato de exigirem a utilização de um laboratório, a instalação das ferramentas e uma capacitação nas mesmas.

O foco de qualidade do estudo exige critérios que avaliem os ganhos obtidos e as dificuldades encontradas na utilização da linguagem pelos usuários. Tanto os ganhos quanto

as dificuldades foram avaliados qualitativamente, através dos questionários. Esta análise tem o objetivo de avaliar a dificuldade de aplicação da linguagem e a qualidade do material utilizado no estudo. A análise qualitativa foi realizada através de um questionário. A qualidade do material utilizado no estudo foi avaliada por perguntas que tratam da utilidade do modelo entregue aos usuários, da qualidade do treinamento oferecido e da descrição inicial do problema.

Com este estudo de observação, pretende-se mostrar alguns indícios de que a linguagem Natural MDA é legível pelos clientes do sistema e não apresenta dificuldades de utilização.

4.2.3 Execução

Para a execução deste estudo, primeiramente, foram selecionados os desenvolvedores envolvidos no projeto. Estes participantes foram selecionados propositalmente por já conhecerem o domínio do sistema a ser desenvolvido. Desta forma, os participantes puderam focar o entendimento na utilização da linguagem. Os participantes do estudo foram alunos de mestrado com experiência de 2 anos (em média) em desenvolvimento de sistemas e 3 projetos (em média). Todos os participantes tinham experiência prática em desenvolvimento de sistemas orientado a objetos, na abordagem MDA e em levantamento de requisitos. Esses dados foram extraídos do questionário apresentado no Apêndice 4.

O sistema escolhido foi o Configurador de Ordens de Serviços. Esse sistema é uma implementação simplificada do padrão *Operation Support Systems through Java* (OSSJ) (OSSJ, 2002), que consiste de criação, envio e inicialização de ordens de serviços para aprovisionamento de assinantes de serviços de telefonia. As ordens possuem serviços e os serviços possuem parâmetros. Uma ordem tem um tipo de ordem, um serviço está associado a um tipo de serviço e um parâmetro está associado a um tipo de parâmetro. Ordens podem ter ordens dependentes. Ordens que dependem de outras só podem ser executadas quando a

ordem da qual ela depende for finalizada com sucesso. A Figura 29 mostra o diagrama de classes dos elementos da camada de domínio do sistema. A Figura 30 mostra o diagrama de classes dos elementos da camada de serviço. Algumas operações têm relacionamento de dependência com exceções, mas por questões de simplificação, optamos por não mostrá-los no diagrama.

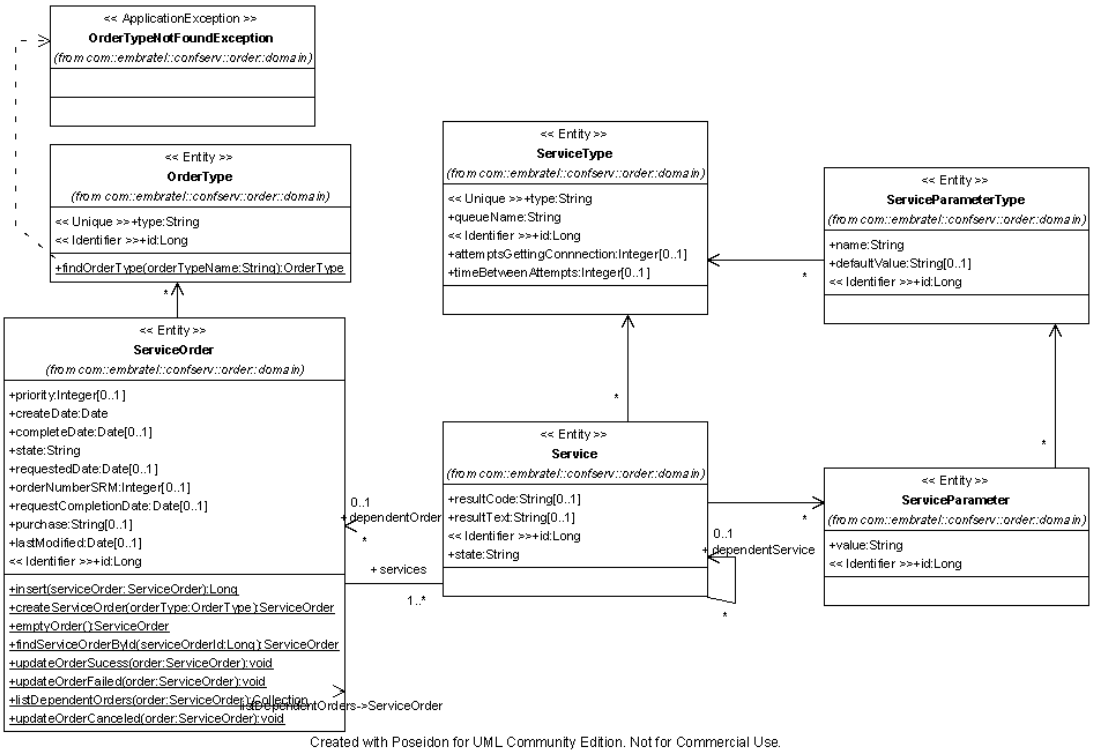


Figura 29. Diagrama de classes de domínio do configurador de serviços

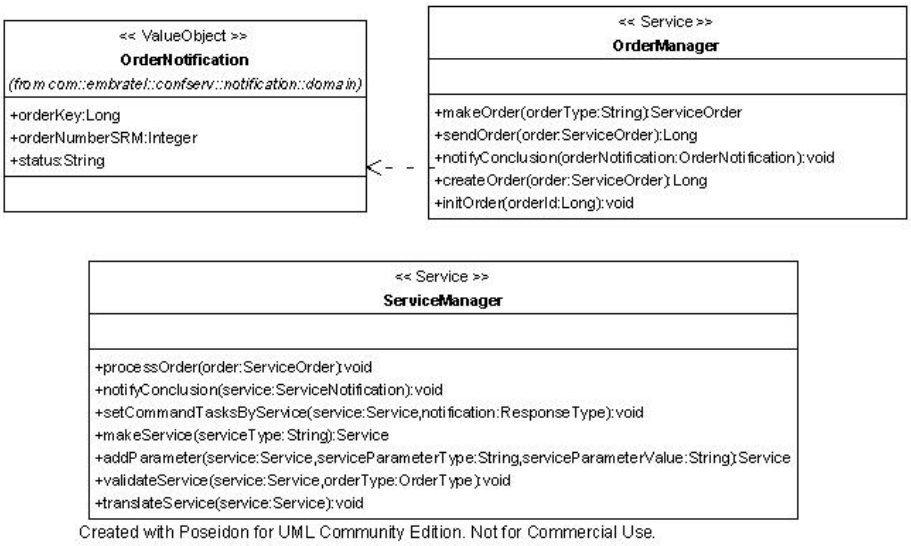


Figura 30. Diagrama de classes de serviço do configurador de serviços

A seguir, mostraremos os três casos de uso que foram apresentados durante o treinamento. A Tabela 22 mostra o caso de uso de envio de ordens descrito com a linguagem Natural MDA. Os passos desse caso de uso são receber uma ordem, persistí-la e iniciá-la.

Tabela 22. Descrição de ações da operação sendOrder

```
Create the order and initialize the order.
```

A Tabela 23 mostra a implementação gerada a partir da especificação das ações da Tabela 22. Os mesmos passos executados no processo de transformação explicado no exemplo de aplicação foram repetidos neste.

Tabela 23. Código gerado para a operação sendOrder

```
/**
  *  send the order
  *  create the order and initialize the order.
  * @param order  order
  */
public java.lang.Long sendOrder(
        com.embratel.confserv.order.domain.ServiceOrder order)
  throws com.embratel.confserv.order.service.OrderManagerException,
         com.embratel.confserv.order.service.ConfServException {
   java.lang.Long var1 = this.createOrder(order);
   this.initOrder(var1);
   return var1;
}
```

Na Tabela 24, apresentamos o caso de uso de inicialização de ordem descrito com a linguagem de ações. A inicialização da ordem consiste em buscá-la e iniciar o processamento.

Tabela 24. Descrição de ações da operação initOrder

```
Find service order and process the order.
When service errors occur, show error 'A service error ocurred'.
```

A Tabela 25 mostra a implementação gerada a partir da especificação das ações da Tabela 24. Os mesmos passos executados no processo de transformação explicado no primeiro caso de uso foram repetidos neste.

Tabela 25. Código gerado para a operação initOrder

```
/**
*  initialize the order
*  find service order and process the order. When service errors occur,
show error 'A service error occurred'.
* @param orderId  order identifier
*/
public void initOrder(java.lang.Long orderId)
        throws
com.embratel.confserv.order.service.OrderManagerException {
      try {
       com.embratel.confserv.order.domain.ServiceOrder var1 =
com.embratel.confserv.order.domain.ServiceOrder.findServiceOrderById
(orderId);
       com.embratel.confserv.order.service.ServiceManagerBI var2 =
locator.ServiceLocator.instance().getServiceManagerBI();
       var2.processOrder(var1);
    } catch
(com.embratel.confserv.order.service.ServiceManagerException var3) {
       logger.error("a service error occurred");
    }
 }
```

A Tabela 26 mostra a especificação de ações do caso de uso de notificação de conclusão de ordens. Caso a ordem finalize com sucesso, as ordens dependentes devem ser inicializadas. Caso contrário, as ordens dependentes devem ser canceladas.

Tabela 26. Descrição de ações da operação notifyConclusion

```
Find service order.
```

```
If order status is equal to success, update order to success, list the
dependent orders and for each dependent of the dependent list, process
the order.
Otherwise, update order to failed, list the dependent orders and for
each dependent of the dependent list, update order to canceled.
When service errors occur, show error 'A service error occurred'.
```

A Tabela 27 mostra a implementação gerada a partir da especificação das ações da Tabela 26. Os mesmos passos executados no processo de transformação explicado no primeiro caso de uso foram repetidos neste.

Tabela 27. Código gerado para a operação notifyConclusion

```
/**
  *  Notify the conclusion of the order
  *  Find service order. If order status is equal to success, update
order to success, list the dependent orders and for each dependent of
the dependent list, process the order. Otherwise, update order to
failed, list the dependent orders and for each dependent of the
dependent list, update order to canceled. When service errors occur,
show error 'A service error occurred'.
  * @param orderNotification  conclusion notification of the order
  */
public void notifyConclusion(
   com.embratel.confserv.notification.domain.OrderNotification
orderNotification) {
   try {
      com.embratel.confserv.order.domain.ServiceOrder var1 =
com.embratel.confserv.order.domain.ServiceOrder.findServiceOrderById
(orderNotification.getOrderKey());
      if (orderNotification.getStatus().equals(var1)) {
com.embratel.confserv.order.domain.ServiceOrder.updateOrderSucess(var1);
         java.util.Collection var2 =
com.embratel.confserv.order.domain.ServiceOrder.listDependentOrders
(var1);
         for (java.util.Iterator var3 = var2.iterator();
var3.hasNext();) {
            com.embratel.confserv.order.domain.ServiceOrder var4 =
(com.embratel.confserv.order.domain.ServiceOrder) var3.next();
            com.embratel.confserv.order.service.ServiceManagerBI var5 =
locator.ServiceLocator.instance().getServiceManagerBI();
            var5.processOrder(var4);
         }
      } else {
com.embratel.confserv.order.domain.ServiceOrder.updateOrderFailed(var1);
         java.util.Collection var2 =
com.embratel.confserv.order.domain.ServiceOrder.listDependentOrders(var1
);
         for (java.util.Iterator var3 = var2.iterator();
var3.hasNext();) {
            com.embratel.confserv.order.domain.ServiceOrder var4 =
(com.embratel.confserv.order.domain.ServiceOrder) var3.next();
com.embratel.confserv.order.domain.ServiceOrder.updateOrderCanceled(var4
);
            }
         }
      } catch
(com.embratel.confserv.order.service.ServiceManagerException var2) {
            logger.error("a service error occurred");
      }
```

```
}
```

Depois de mostrar a gramática, o sistema a ser desenvolvido e a especificação de alguns casos de uso utilizando a linguagem Natural MDA, os participantes especificaram um outro caso de uso de mesma complexidade. Para finalizar o estudo, os participantes responderam o questionário de avaliação da linguagem apresentado no Apêndice 5. Os comentários dos participantes são apresentados na seção seguinte.

4.2.4 Análise dos resultados

Alguns participantes sentiram falta de informações durante o treinamento porque nem todas as descrições dos elementos do modelo foram disponibilizadas. Alguns participantes disseram que o treinamento carece de uma explicação mais detalhada sobre a transformação em código.

A partir das respostas dos questionários, pudemos observar que a linguagem teve boa aceitação pelos participantes do estudo. No questionário, os participantes fizeram os seguintes comentários:

- A busca das descrições dos elementos do modelo não é prática → Nossa opinião é que isto pode ser melhorado com o editor da linguagem, que guia o especialista. A ferramenta lista as descrições das classes, atributos, associações, métodos, parâmetros e faz sugestões coerentes. Além de oferecer a funcionalidade de gerar fluxogramas a partir de descrição de ações.
- As declarações dos pacotes das classes no corpo do método dificultam a leitura do método → Se assumirmos que o código não precisa ser analisado, isto não é problema. Para dar um maior grau de confiança à geração de código, como trabalhos futuros, podemos gerar testes para o código gerado.
- O tempo de aprendizagem da linguagem não deve ser desconsiderado → Consideramos que, assim como qualquer técnica de especificação de sistemas, a

linguagem Natural MDA exige um tempo de capacitação. No entanto, como essa linguagem é próxima da linguagem natural, tende-se a obter menor tempo de capacitação nessa, se comparado com outras que definam sintaxe ou notação muito diferente da linguagem natural.

- Os nomes das variáveis poderiam ser mais descritivos, evitando var1, var2...
- A inclusão de construtores para terminar a execução de métodos (*return*) e para sair de iterações (*break*).
- Uma vantagem da linguagem é evitar mau entendimento entre clientes e projetistas de sistemas → Quando a especificação do sistema está representada por uma notação ilegível pelo cliente, dificulta-se a obtenção da correta interpretação. Nesse caso, pode-se prosseguir a especificação e construção do sistema com defeitos que poderiam ser evitados se a especificação pudesse ser validada pelo cliente. Além disso, segundo Pressman (2002), quanto mais cedo é descoberto um defeito, menor o custo envolvido para a correção do mesmo.
- Uma outra vantagem observada neste estudo é que o fluxo principal do caso de uso se mantém separado do fluxo alternativo. Isto facilita a legibilidade das regras de negócio.

Em relação à avaliação do treinamento, acreditamos que se os participantes pudessem executar as transformações, eles ficariam mais confiantes e entenderiam melhor o funcionamento. Para uma próxima aplicação do estudo, o treinamento pode ser aprimorado com a utilização de uma ferramenta de modelagem, o editor da linguagem e as ferramentas de processamento da linguagem. Nesse caso, torna-se necessário dedicar um tempo para preparação do ambiente para a realização dos experimentos.

Durante o estudo, foi observado que: (i) nenhum participante rejeitou o uso da linguagem, (ii) um deles comentou que a solução parece simples para um problema não tão

simples, (iii) todos participaram até o final do experimento, (iv) alguns se preocuparam muito em como a especificação seria traduzida em código, perguntando sobre os parâmetros de entrada e saída de cada operação.

A Tabela 28 mostra a proporção de caracteres da especificação e da implementação. Nessa contagem de caracteres, consideramos apenas o código gerado referente à implementação da operação - não consideramos a documentação da operação e os artefatos gerados pela ferramenta MDA. Podemos observar que a quantidade de caracteres necessários para a especificação é consideravelmente inferior à quantidade de caracteres necessários na implementação equivalente.

Tabela 28. Proporção de caracteres na especificação x implementação

	Envio de ordens	Inicialização de ordens	Notificação de conclusão
Caracteres especificação	36	90	312
Caracteres implementação	277	489	1330
Proporção de caracteres (especificação/implementação)	13%	19%	23%

5 CONCLUSÃO

As linguagens atuais de especificação de ações para a abordagem MDA carecem de um nível de abstração adequado à tarefa dos projetistas de sistemas. Consideramos que essas linguagens devem ser mais próximas do nível de abstração dos modelos e mais independentes da sintaxe de implementação. Para resolver esse problema, definimos uma linguagem de especificação de ações baseada em linguagem natural controlada, que chamamos de Natural MDA. Além da linguagem Natural MDA ter nível de abstração mais alto, ela é complementar à UML, independente de plataforma e pode ser transformada em código executável de forma automatizada.

A linguagem Natural MDA é alinhada aos objetivos da MDA. Através dos estudos realizados, mostramos indícios de que a adoção dessa linguagem contribui na melhoria do processo de desenvolvimento orientado a modelos, reduzindo o *gap* entre o domínio do problema e a linguagem de implementação, e facilitando a comunicação entre clientes e projetistas. Além disso, como essa linguagem é uma linguagem natural controlada, ela não tem as dificuldades de entendimento inerentes das linguagens totalmente visuais. Ao mesmo tempo, é integrada aos diagramas da UML, o que aumenta o poder de expressividade da linguagem.

Neste trabalho, buscamos utilizar as linguagens textuais e as linguagens visuais onde essas forem mais adequadas, dependendo da necessidade, do conhecimento e do tipo de perfil do usuário para a tarefa a ser realizada. Julgamos que as linguagens gráficas são mais adequadas às definições estruturais, enquanto as linguagens textuais são mais adequadas às definições comportamentais.

Dentre as mais importantes medidas de qualidade apontadas por Pressman (2002), duas são diretamente afetadas pela abordagem aqui proposta:

- "Correção é o grau em que o software desempenha sua necessária função" (PRESSMAN, 2002). A abordagem baseada em modelos, complementada por mecanismos que permitam a execução e validação das funções mais cedo, reduz a taxa de defeitos (problemas relatados pelo usuário).
- "Manutenibilidade é a facilidade com que um programa pode ser corrigido, se um erro é encontrado, adaptado, se seu ambiente se modifica, ou aperfeiçoado, se o cliente deseja uma modificação nos requisitos" (PRESSMAN, 2002). Certamente, em ambientes de modelos executáveis, o tempo necessário para fazer alteração no software é menor do que nos ambientes não automatizados.

Dentre as contribuições deste trabalho, podemos citar:

- O levantamento e a comparação de recursos, atualmente existentes, de especificação de sistemas de forma precisa e completa. Buscou-se identificar quais desses recursos são aderentes aos requisitos de linguagens de especificação de ações e, mais especificamente, aderentes à abordagem MDA.
- O projeto de uma linguagem de especificação de ações que atenda aos requisitos da MDA, acompanhado de ferramentas de apoio ao processamento e uso da linguagem.
- O aumento do nível de abstração na especificação de sistemas, permitindo melhor legibilidade pelos clientes.

Reconhecemos que a gramática da linguagem Natural MDA não é completa, pois algumas operações aritméticas e de manipulação de cadeia de caracteres não são possíveis no momento. Por esse motivo, utilizamos a idéia de funções auxiliares. Em caso de necessidade de novos requisitos, a gramática pode ser estendida para incluir outros tipos de expressões.

O exemplo de aplicação desenvolvido serviu para identificar os requisitos da linguagem de ações e fazer testes preliminares da implementação. Através do estudo de

observação, apresentamos uma aplicação prática da linguagem, a viabilidade de sua utilização nos processos de desenvolvimento de sistemas e alguns trabalhos futuros. O estudo mostrou indícios de questões como:

- Garantia de um mínimo de completude da gramática para especificação de ações, dado que foi possível especificar parte do sistema (que foi o escopo do estudo) com a linguagem;
- Facilidade de utilização da linguagem através de questionário de avaliação qualitativa.

Apesar de termos realizado um estudo de observação, sabemos que este não é suficiente para uma validação efetiva do trabalho, embora ofereça indícios de uma boa aceitação da proposta. Questões que consideramos importantes e que poderão ser tratadas em trabalhos futuros são:

- Integração com um ambiente de execução, de forma que os artefatos possam ser executados, depurados e validados;
- Extensão da gramática para contemplar o uso de elipses e anáforas para tornar os recursos dessa linguagem mais próximos dos recursos lingüísticos das linguagens naturais;
- Identificação de *bad smells*[3] (FOWLER *et al.*, 1999) na especificação de ações de forma equivalente à identificação desses em especificações OCL (CORREA; WERNER, 2004) e em declarações de aspectos (PIVETA *et al*., 2006).
- Uma validação formal a respeito da equivalência entre a entrada (especificação) e a saída (implementação) do processo de transformação;

[3] *Bad smells* podem ser vistos como alertas de que existem problemas no software e esses podem ser corrigidos através da aplicação de refatorações. Alguns exemplos de *bad smells* são: trechos de código que não estão sendo utilizados, duplicações de código, classes com poucas responsabilidades ou atribuições demais. Esses problemas dificultam o reuso e podem ser minimizados através da identificação de seus sintomas e da remoção das causas desses problemas (PIVETA *et al*., 2006).

- Uma análise detalhada do impacto de utilizar a linguagem Natural MDA nas diversas metodologias existentes de desenvolvimento de sistemas, verificando quais delas são compatíveis com o desenvolvimento orientado a modelos e, mais especificamente com a linguagem Natural MDA, e identificando quais adaptações seriam necessárias para melhorar o processo de desenvolvimento;
- Um estudo de caso mais detalhado, envolvendo uma aplicação de maior complexidade, com mais participantes e métricas mais objetivas e quantitativas, dos ganhos obtidos quando da utilização da linguagem;
- A criação de um modelo de estimativa de esforço para desenvolvimento orientado a modelos utilizando a linguagem Natural MDA, da mesma forma como existem atualmente os modelos de estimativa de esforço baseados em pontos de função e casos de uso.

REFERÊNCIAS

AHO, A. V., SETHI, R., ULLMAN, J. D. **Compilers: Principles, Techiniques, and Tools**. Addison-Wesley, ISBN 0-2011-0088-6, 1986.

ALLEN, J. **Natural Language Understanding**. The Benjamin/Cummings Publishing Company Inc, ISBN: 0-8053-0334-0, 1995.

ALTWARG, R. **Controlled Languages: An Introduction**, 2000. Disponível em http://www.shlrc.mq.edu.au/masters/students/raltwarg/clindex.htm. Acesso em dezembro 2005.

ANDROMDA **Getting started**, 2005. Disponível em http://www.andromda.org. Acesso em março 2006.

ANTLR **Another Tool for Language Recognition**, 2005. Disponível em http://www.antlr.org. Acesso em março 2006.

BARROS, M. O., WERNER, C. M. L., TRAVASSOS, G. H. **Um Estudo Experimental sobre a Utilização de Modelagem e Simulação no Apoio à Gerência de Projetos de Software**, XIX Simpósio Brasileiro de Engenharia de Software (SBES), Outubro, 2005, Uberlândia, MG, Brasil.

BRIDGEPOINT. **Object Action Language**, 2005. Disponível em http://www.acceleratedtechnology.com. Acesso em dezembro 2005.

BROWN, A. W.; CONALLEN, J. **An introduction to model-driven architecture – Part III: How MDA affects the iterative development process**, 2005. Disponível em http://www-106.ibm.com/developerworks/rational/library/may05/brown/index.html. Acesso em junho 2005.

BRYANT, B., LEE, B. S. **Two-Level Grammar as an Object-Oriented Requirements Specification Language**, XXXV Hawaii International Conference on System Sciences (HICSS), 2002, IEEE Computer Society, Volume 9, p. 280. ISBN:0-7695-1435-9.

CIANNI, N. M., CABEÇO, T. S. **Um Editor para Especificação de Ações em Linguagem Natural Controlada**, Projeto final de curso, Universidade Federal do Rio de Janeiro, RJ, Brasil, 2006. Disponível em http://dataware.nce.ufrj.br:8080/dataware_en.

CORBA. **Common Object Request Broker Architecture**, 2006. Disponível em http://www.omg.org/corba. Acesso em janeiro 2006.

CORREA, A. L., WERNER, C. M. L. **Applying Refactoring Techniques to UML/OCL Models**, VII International Conference on the Unified Modeling Language (UML'04), Volume 1, p. 173-187, Lisboa, Portugal. Lecture Notes in Computer Science, Springer, Heidelberg, Alemanha, ISSN 0302-9743

CUNHA, M. F. **Experiência prática na construção de aplicações utilizando Model Driven Architecture**. Projeto final de curso. Universidade Federal do Rio de Janeiro, RJ, Brasil, 2005.

DAMM, W., HAREL, D. **LSCs: Breathing Life into Message Sequence Charts**, Formal Methods in System Design, Volume 19, p. 45-80, Julho, 2001. ISSN:0925-9856

DEELSTRA, S., SINNEMA, M., van GURP, J., BOSCH, J. **Model Driven Architecture as Approach to Manage Variability in Software Product Families**, Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAFA), 2003, CTIT Technical Report TR-CTIT-03-27, University of Twente, p. 109-114.

DESIDERATI, F. **Geração de Código-Fonte Automatizado para Aplicações CRUD Utilizando Model Driven Architecture**, Projeto final de curso, Universidade Federal do Rio de Janeiro, RJ, Brasil, 2006.

DINH-TRONG, T. **JAL: Java like Action Language**, Specification 1.1, Department of Computer Science, Colorado State University, Outubro, 2005.

DSLTOOLS **Domain-Specific Language Tools**, 2006. Disponível em http://msdn.microsoft.com/vstudio/dsltools. Acesso em abril 2006.

EJB. **Enterprise Java Beans**, 2003. Disponível em http://java.sun.com/products/ejb. Acesso em abril 2006.

FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., ROBERTS, D. **Refactoring: Improving the Design of Existing Code**, Addison-Wesley Professional, 1999. ISBN 0-2014-8567-2.

HARS, A.; MARCHEWKA, J. T. **Eliciting and mapping business rules to IS design: introducing a natural language case tool**. Proceedings of Decision Sciences Institute, Vol.2, p. 533-535, 1996.

HIBERNATE. **Hibernate**, 2006. Disponível em http://www.hibernate.org. Acesso em abril 2006.

HUTN. **Human-Usable Textual Notation Specification**, versão 1.0, OMG, 2004. Acesso em agosto 2006.

JAVA. **Java Techonology**, Sun Microsystems, 2006. Disponível em http://java.sun.com. Acesso em março 2006.

KABIRA **Kabira Action Semantics**, 2005. Disponível em http://www.kabira.com. Acesso em novembro 2005.

KLEPPE, A., WARMER, J. **Extending OCL to include actions,** III International Conference on the Unified Modeling Language (UML), p. 440-450, 2000.

KLEPPE, A.; WARMER, J.; BAST, W. **MDA Explained: The Model Driven Architecture: Practice and Promisse**. Addison-Wesley, 2002. ISBN: 4-8443-1869-1

KRUCHTEN, P. **The Rational Unified Process – An Introduction**, Segunda edição, Addison-Wesley, 2000. ISBN: 0-2016-0459-0.

LARRUCEA, X., DIEZ, A. B. G., MANSELL, J. X. **Practical Model Driven Development Process**, II European Workshop on Model Driven Architecture (EWMDA), Canterbury, England, Setembro, 2004.

LEAL, L. N., PIRES, P. F., CAMPOS, M. L. M. **An Action Specification Language based on Controlled Natural Language**, X Simpósio Brasileiro de Linguagens de Programação (SBLP), Itatiaia, RJ, Brasil , p. 161-174. Maio, 2006. ISBN: 85-7669-071-3.

LEAL, L. N., PIRES, P. F., CAMPOS, M. L. M., DELICATO, F. C. **Natural MDA: Controlled Natural Language for Action Specifications on Model Driven Development**, XIV International Conference on Cooperative Information System (CoopIS), França, Novembro, 2006. http://www.cs.rmit.edu.au/fedconf/index.html?page=coopis2006cfp.

LINHALIS, F., MOREIRA, D. A. **Semantic Mapping between UNL Relations and Software Components to the Execution of Natural Language Requisitions**, Proceedings of International Information and Telecommunication Technologies Symposim, São Carlos, SP, Brasil, Dezembro, p. 109-116, 2004.

LINHALIS, F., MOREIRA, D. A. **Geração de Programas Utilizando Linguagem Natural Restrita , Ontologias e Componentes de Software**, WebMedia, MG, Brasil, 2005.

LIU, H.; LIEBERMAN, H. **Metafor: Visualizing stories as code**. X International Conference on Intelligent user interfaces (IUI´05). San Diego, California, USA. ACM 1-58113-894-6/05/0001, p. 305-307. Janeiro, 2005. ISBN:1-58113-894-6.

LIU, H.; LIEBERMAN, H. **Toward a programmatic semantics of natural language,** Proceedings of VL/HCC'04, XX IEEE Symposium on Visual Languages and Human-Centric Computing, p. 281-282. IEEE Computer Society Press. Setembro, 2004.

LIU, H., LIEBERMAN, H. **Feasibility studies for programming in natural language**. End-User Development. Kluwer Academic Publishers/Springer. Printed in the Netherlands, 2005.

McNEILE, A., SIMONS, N. **Methods of behaviour modeling: a commentary on behaviour modeling techniques for MDA**, 2004. Disponível em http://www.metamaxim.com/download/documents/Methods.pdf. Acesso em abril 2005.

MDA. **MDA Guide V1.0.1**. Disponível em http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2001. Acesso em maio 2005.

MELLOR, S. J.; BALCER, M. J. **Executable UML: A Foundation for Model-Driven Architecture**. Addison Wesley. Maio, 2002. ISBN: 0-201-74804-5.

MELLOR, S. J. et al. **Software-Platform-Independent, Precise Action Specifications for UML**. Proceedings of UML 1998, p. 281-286.

MELLOR, S. J., SCOTT, K., UHL, A., WEISE, D. **MDA Distilled: Principles of Model-Driven Architecture**, Addison Wesley, Março, 2004. ISBN: 0-201-78891-8.

MELOCHE, T. **The Rational Unified Process. A well documented, complete yet complex methodology**, The Menlo Institute LLC, 2002. Disponível em http://www.menloinstitute.com/freestuff/whitepapers. Acesso em maio 2005.

METACASE. **Domain-Specific Modeling with MetaEdit+: 10 times faster than UML**, 2005. Disponível em http://www.metacase.com. Acesso em maio 2005.

MOF **MetaObject Facility**, 2006. Disponível em http://www.omg.org/mof. Acesso em março 2006.

MOSSES, P. D. **Theory and Practice of Action Semantics**, BRICS Report Series. RS-96-53. ISSN 0909-0878. Dezembro, 1996.

NET. **Microsoft .NET**, 2006. Disponível em http://www.microsoft.com/net. Acesso em abril 2006.

OMG. **Action Semantics for UML – Request for Proposal**, 1999. Disponível em http://www.omg.org/docs/ad/98-11-01.pdf. Acesso em agosto 2005.

OMG. **Object Management Group**, 2005. Disponível em http://www.omg.org. Acesso em agosto 2005.

OSSJ. **OSS Service Activation API**, 2002. Disponível em http://java.sun.com/products/oss/downloads. Acesso em fevereiro 2006.

PENDER, T. **UML A Bíblia**. Editora Campus, 2002. ISBN: 8-5352-1408-9.

PIVETA, E., HECHT, M., PIMENTA, M., PRICE, R. T. **Bad Smells em Sistemas Orientados a Aspectos**, XIX Simpósio Brasileiro de Engenharia de Software (SBES), Uberlândia, MG, Brasil, 2005.

PRESSMAN, R. S. **Engenharia de Software**, McGrawHill, 5a edição, 2002. ISBN: 8-5868-0425-8.

PROLOG **Prolog**, 2006. Disponível em http://www.swi-prolog.org. Acesso em março 2006.

PYTHON. **Python Programming Language**, 2006. Disponível em http://www.python.org. Acesso em março 2006.

RATIONAL. **PRJ270: Essentials of Rational Unified Process – Student Manual**, Version 2003.06.00, 2003.

RATIONAL. **Rational Unified Process V7.0 Evaluation**, 2006. Disponível em http://www-128.ibm.com/developerworks/downloads/r/rup. Acesso em maio 2006.

RODRIGUES, C. L. C. **Proposta de adequação do RUP ao desenvolvimento de software baseado em MDA**. Monografia do Curso de Graduação em Ciências da Computação, UFPE, Recife, 2004.

SCHWITTER, R.; FUCHS, N. E. **Attempto: From Specifications in Controlled Natural Language towards Executable Specifications**, Proceedings of the GI Enterprise Modelling and Information Systems Architectures (EMISA) Workshop, Tutzing, Germany, 1996.

SELONEN, P., SYSTA, T., KOSKIMIES, K. **Generating Structured Implementation Schemes from UML Sequence Diagrams**, Proceedings of TOOLS USA, Santa Barbara, California, USA, Julho-Agosto, p. 317-328. IEEE Computer Society. 2001.

SILVA, S. R. P. **Um Modelo Semiótico para Programação por Usuários Finais**. Tese de doutorado. PUC, Rio de Janeiro. Março, 2001.

SILVA, S. R. P.; PINHEIRO, J. P. **Um Framework para Criação de Linguagens de Domínio Específico**. VIII Simpósio Brasileiro de Linguagens de Programação (SBLP), Rio de Janeiro, Maio, 2004.

SILVA, S. R. P.; PINHEIRO, J. P. **O uso de Linguagem Natural Controlada para Representação de Conhecimento por Usuários Finais**, 2004.

SOLINGEN, R. Van, BERGHOUT, E. **The Goal / Question / Metric Method: A Practical Guide for Quality Improvement of Software Development**, McGrawn Hill, 1999. ISBN 0-0770-9553-7.

STOTTS, D. **Interaction diagrams**, 2000. Disponível em http://www.cs.unc.edu/~stotts/COMP145/CRC/Interactions.html. Acesso em abril 2006.

STRINGTEMPLATE. **String Template**, 2004. Disponível em http://www.stringtemplate.org. Acesso em agosto 2005.

SWT. **Standard Widget Toolkit**, 2006. Disponível em http://www.eclipse.org/swt. Acesso em janeiro 2006.

TAM, R. Chung-Man; MAULSBY, D.; PUERTA, A. R. **U-TEL: A Tool for Eliciting User Task Models from Domain Experts.** Proceedings of International Conference on Intelligent User Interfaces (IUI'98), p. 77-80, 1998.

THOMAS, D.; BARRY, B. M. **Model Driven Development – The Case for Domain Oriented Programming**. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA´03), California, Outubro, 2003. ACM 1-58113-751-6/03/0010.

TOLVANEN, Juha-Pekka., ROSSI, M. **MetaEdit+: Defining and Using Domain-Specific Modeling Languages and Code Generators**. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA´03), California, Outubro, 2003. ACM 1-58113-751-6/03/0010.

TRAVASSOS, G. H., GUROV, D., AMARAL, E. A. G. **Introdução à Engenharia de Software Experimental**, Relatório técnico, RT-ES-590-02, Programa de Engenharia de Sistemas e Computação, Rio de Janeiro, Brasil, 2002.

UML **Unified Modeling Language**, 2004. Disponível em http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf. Acesso em janeiro 2006.

USHIDA, H., ZHU, M. **The Universal Networking Language Beyond Machine Translation**, International Symposium on Language and Cyberspace, Setembro, 2001.

VELOCITY. **Velocity**, 2005. Disponível em http://jakarta.apache.org/velocity. Acesso em julho 2005.

VISUAL RULES, **Visual Rules**, 2006. Disponível em http://www.visual-rules.de. Acesso em janeiro 2006.

WARMER, J.; KLEPPE, A. **The Object Constraint Language - Getting Your Models Ready for MDA**. Segunda edição. Addison Wesley. August, 2002. ISBN: 0-321-17936-6.

WIERINGA, R. J., SAAKE, G. **A Formal Analysis of the Shlaer-Mellor Method: Towards a Toolkit of Formal and Informal Requirements Specification Techniques**, Requirements Engineering Journal, Volume 1, Número 2, p. 106-131. 1997.

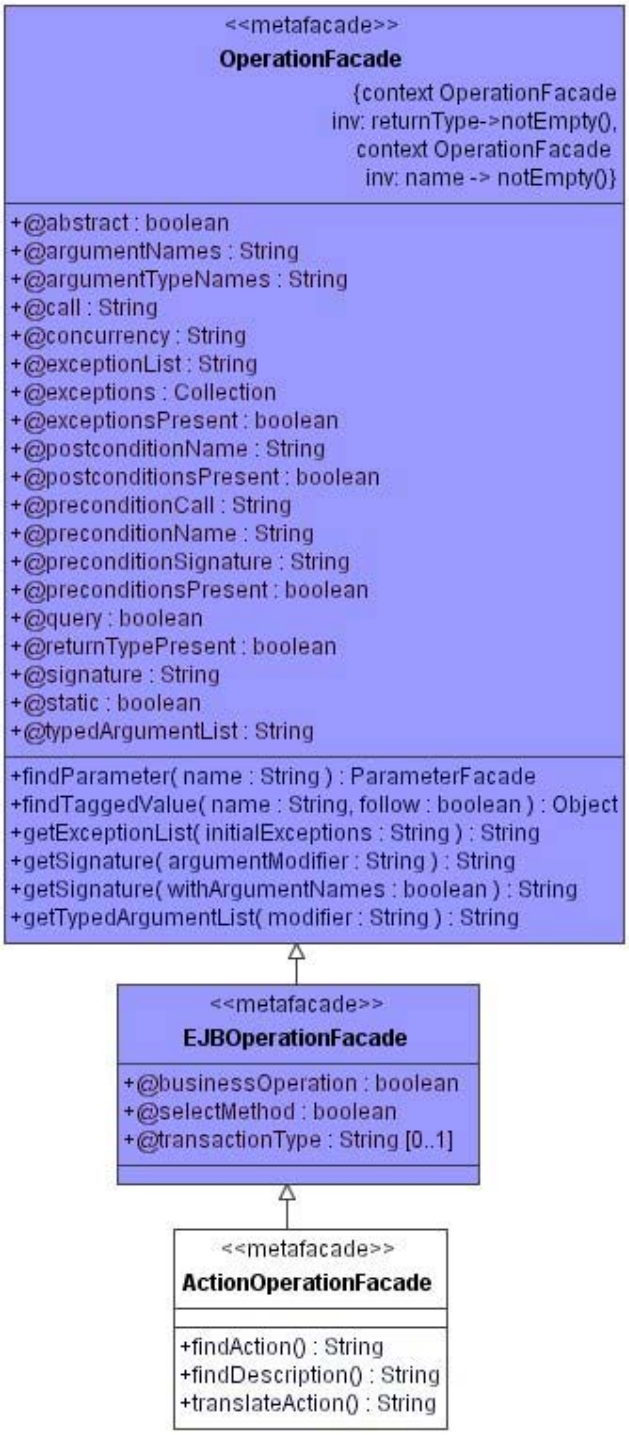
WILKIE, I. et al. **ASL Language Manual**, 2001. Kennedy Carter Ltd. Disponível em http://www.kc.com. Acesso em maio 2005.

XMI. **XML Metadata Interchange Specification version 2.1**, 2005. Disponível em http://www.omg.org/technology/documents/formal/xmi.htm. Aceso em fevereiro 2006.

APÊNDICES

Apêndice 1

MODELAGEM DA EXTENSÃO DAS METAFACADES DO ANDROMDA



Apêndice 2

TEMPLATE PARA GERAÇÃO DE CÓDIGO DAS OPERAÇÕES DE SERVIÇO

```
#parse("templates/ejb/SessionEJBGlobals.vm")
// license-header java merge-point
#if ($stringUtils.isNotBlank($service.packageName))
package $service.packageName;
#end

/**
 * @see $service.fullyQualifiedName
 */
public class ${service.name}BeanImpl
    extends ${service.name}Bean
    implements javax.ejb.SessionBean
{


#foreach ($operation in $service.operations)
#set ($description = $operation.findTaggedValue("element.description"))
##set ($description = $stringUtils.toPhrase($description))
##set ($description = $stringUtils.toSingleLine($description))
#set ($description = $stringUtils.replace($description, '\n', ' * '))
#set ($behaviour = $operation.findTaggedValue("operation.behaviour"))
##set ($behaviour = $stringUtils.toPhrase($behaviour))
##set ($behaviour = $stringUtils.toSingleLine($behaviour))
#set ($behaviour = $stringUtils.replace($behaviour, '\n', ' * '))
    /**
       * #if ($stringUtils.isNotBlank($description)) $description #end

       * #if ($stringUtils.isNotBlank($behaviour)) $behaviour #end

     #foreach ($argument in $operation.arguments)
     * @param $argument.name #if
($stringUtils.isNotBlank($argument.findTaggedValue("element.description")))
$argument.findTaggedValue("element.description") #end

     #end
     */
      $operation.visibility  $operation.returnType.fullyQualifiedName
$operation.signature
     #if ($operation.exceptionsPresent)
         throws $operation.exceptionList{
     #else
        {
     #end
     #set ($action = $operation.translateAction())
     $action
      }
#end
}
```

Apêndice 3

IMPLEMENTAÇÃO DA EXTENSÃO DA METAFACADE DO ANDROMDA

```
package org.andromda.cartridges.ejb.metafacades;

import java.util.Collection;

import org.apache.log4j.Logger;

import br.antlr.ProcessTokens;
import br.ufrj.nce.al.control.ExpressionTranslator;
import br.ufrj.nce.al.control.TranslationResult;
import br.ufrj.nce.al.control.Variable;
import br.ufrj.nce.al.facade.XMIClassFacade;
import br.ufrj.nce.al.facade.XMIFacade;
import br.ufrj.nce.al.facade.XMIFacadeFactory;
import br.ufrj.nce.al.facade.XMIOperationFacade;

/**
 * MetafacadeLogic implementation for
 * org.andromda.cartridges.ejb.metafacades.ActionOperationFacade.
 *
 * @see org.andromda.cartridges.ejb.metafacades.ActionOperationFacade
 */
public class ActionOperationFacadeLogicImpl extends ActionOperationFacadeLogic {

    protected Logger logger = Logger
                .getLogger(ActionOperationFacadeLogicImpl.class);

    public ActionOperationFacadeLogicImpl(Object metaObject, String context) {
          super(metaObject, context);
    }

    /**
     * @see
org.andromda.cartridges.ejb.metafacades.ActionOperationFacade#translateAction()
     */
    protected java.lang.String handleTranslateAction() {
          XMIFacadeFactory.setParameter(getModel());
          XMIFacade xmiFacade = XMIFacadeFactory.getInstance();

          String behaviour = (String) findTaggedValue("operation.behaviour");
          String description = (String) findTaggedValue("element.description");
          logger.info("behaviour = " + behaviour);
          logger.info("description = " + description);

          XMIOperationFacade operation = xmiFacade.getOperation(description);
          XMIClassFacade returnType = operation.getReturnType();
          logger.info("Return Type: " + returnType.getName());
          TranslationResult translationResult = null;
          if (behaviour != null) {
                logger.info("operation to be translated = " +
operation.getName());
                Collection expressions = ProcessTokens.process(behaviour
                            .toLowerCase());
                logger.info("expressions = " + expressions);
                ExpressionTranslator translator = new ExpressionTranslator(
                            expressions, xmiFacade, operation);
                translationResult = translator.translateExpression();
                logger.info("implementation = "
                            + translationResult.getTranslation());
                if (!translator.hasReturnExpression(expressions)) {
                      logger.info("There is no return expression");
                      //add the return
                      if (!returnType.getName().equals("void")) {
                            logger
                                        .info("Finding variable by type: "
                                                    + returnType.getName());
                            Variable returnObj =
translationResult.getTranslationContext()
```

```
    .findVariableByType(returnType.getName());
                                logger.info("Variable found: " +
returnObj.getName());
                                translationResult.setTranslation(translationResult
                                            .getTranslation()
                                            + "\n return " + returnObj.getName() +
";");
                          }
                    }
             } else {
                    translationResult = new TranslationResult();
                    translationResult
                                .setTranslation("//TODO: put your implementation
here");
                    if (!returnType.getName().equals("void")) {
                          translationResult.setTranslation(translationResult
                                      .getTranslation()
                                      + "\n return " +
returnType.getJavaNullString() + ";");
                    }
             }

             return translationResult.getTranslation();
      }

      /*
       * (non-Javadoc)
       *
       * @see
org.andromda.cartridges.ejb.metafacades.ActionOperationFacadeLogic#handleFindDesc
ription()
       */
      protected String handleFindDescription() {
             String description = (String) findTaggedValue("element.description");
             if (description == null) {
                    description = "";
             }
             return description;
      }

      /*
       * (non-Javadoc)
       *
       * @see
org.andromda.cartridges.ejb.metafacades.ActionOperationFacadeLogic#handleFindActi
on()
       */
      protected String handleFindAction() {
             String behaviour = (String) findTaggedValue("operation.behaviour");
             if (behaviour == null) {
                    behaviour = " ";
             }
             return behaviour;
      }

}
```

Apêndice 4

QUESTIONÁRIO DE AVALIAÇÃO DE PERFIL DO PARTICIPANTE

Nas perguntas abaixo, quando duas ou mais alternativas forem válidas, marque a alternativa que mais se aplica a seu caso.

Determine seu nível de formação.
[ ] Graduando [ ] Graduado
[ ] Mestrando [ ] Mestre
[ ] Doutorando [ ] Doutor

Determine a sua experiência no desenvolvimento de software?

[ ] Eu nunca desenvolvi software
[ ] Eu desenvolvi software para uso pessoal
[ ] Eu desenvolvi software como parte de trabalhos de curso (____ anos)
[ ] Eu desenvolvi software em ambiente industrial (____ anos)

Determine a sua experiência em projeto de software ?

[ ] Eu nunca fui projetista de software
[ ] Eu fui projetista em projetos de curso
[ ] Eu fui projetista em poucos (<3) projetos na indústria (___ projetos)
[ ] Eu fui projetista em muitos (>=3) projetos na indústria (___ projetos)

Por favor, indique sua experiência nas questões abaixo, assinalando uma das alternativas da escala de 1 a 5, segundo a tabela abaixo:

1 – nenhuma
2 – estudo em cursos ou livros
3 – praticado em projetos de curso
4 – praticado na indústria
5 – praticado em diversos (>=3) projetos da indústria

Experiência em desenvolvimento de software orientado a objetos
Experiência como desenvolvedor em desenvolvimento de software seguindo a MDA
Experiência como projetista em desenvolvimento de software seguindo a MDA
Experiência em levantamento de requisitos junto ao cliente

Apêndice 5

QUESTIONÁRIO DE AVALIAÇÃO DA LINGUAGEM

Você considera que a linguagem de especificação de ações teve utilidade no desenvolvimento do projeto proposto? Você a utilizaria novamente? Justifique.

_____

_____

_____

Você considera que as especificações das ações tornaram-se mais legíveis para pessoas que não têm muito conhecimento específico de Computação? Justifique.

_____

_____

_____

É possível gerar código executável a partir da descrição de ações? Se sim, o código estava bem organizado e documentado?

_____

_____

_____

Você considera que a linguagem permite descrição de ações independente de detalhes de plataforma específica?

_____

_____

_____

Você tem alguma sugestão de evolução, melhoria ou ferramentas de apoio à utilização da linguagem?

_____

_____

_____

As informações disponibilizadas sobre o sistema a ser desenvolvido, a linguagem proposta e sua aplicação foram de fácil compreensão? Você sentiu falta de alguma informação neste treinamento?

_____

_____

_____

Apêndice 6

DETALHAMENTO DA GRAMÁTICA NATURAL MDA

O principal termo da linguagem é “rule”. Como mostrado na Figura a, este termo consiste em um conjunto de operações, seguido de um ponto final, ou sentenças ou comandos a serem executados ao final de uma iteração.

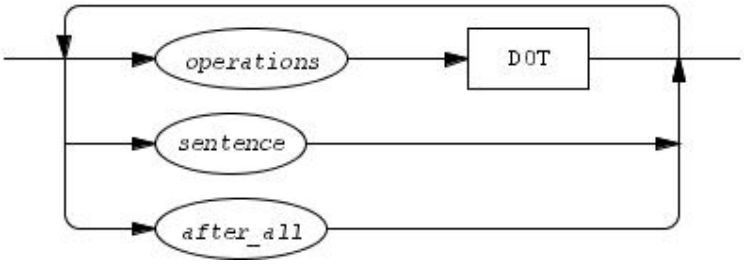


Figura a. Termo “rule” da gramática

O conjunto de operações, que na gramática é chamado “operations”, consiste de uma seqüência formada por uma operação, seguida de um conector “and” ou vírgula, e outra operação; ou mensagens de erro ou alerta (“show_error_warning”) ou sentenças de sinalização de retorno. A Figura b ilustra esse termo da gramática.

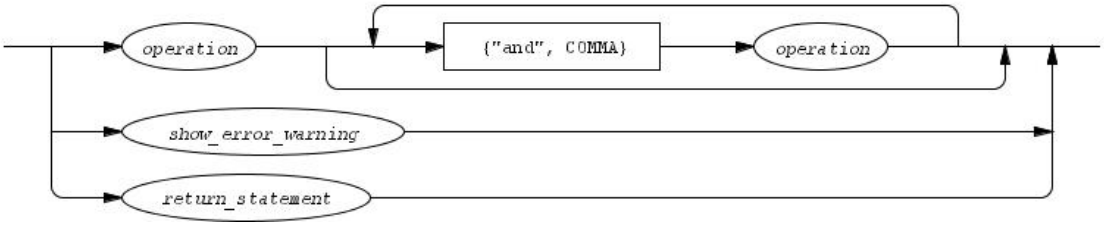


Figura b. Termo “operations” da gramática

Como mostrado na Figura c, uma mensagem de erro ou alerta é uma seqüência dos termos: “show”, seguido de “error” ou “warning”, em seguida a mensagem entre aspas duplas.

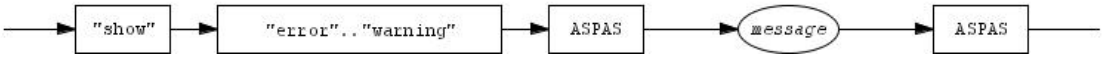


Figura c. Termo “show_error_warning” da gramática

Agora que já descrevemos o termo “operations”, veremos na Figura d o termo “sentence”. Uma sentença pode corresponder a uma expressão condicional, uma iteração ou eventos. Expressões condicionais podem ser seguidas de um conjunto de operações caso a condição não seja satisfeita.

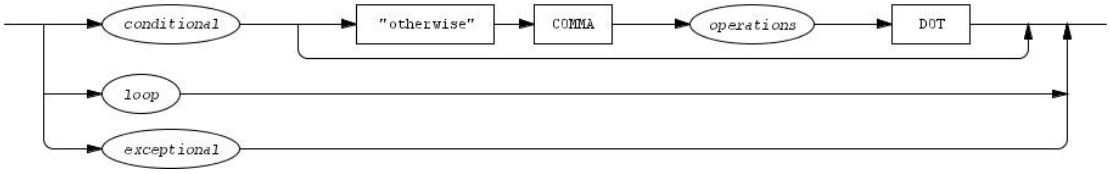


Figura d. Termo “sentence” da gramática

Na Figura e, veremos que uma expressão condicional é composta por um conjunto de condições e um conjunto de operações a serem executadas caso as condições sejam verdadeiras.

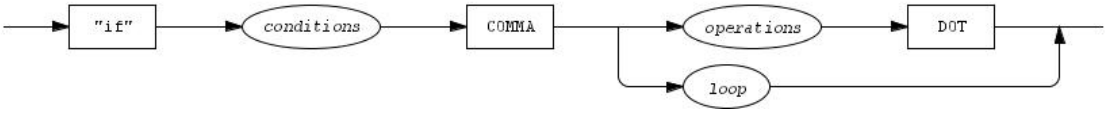


Figura e. Termo “conditional” da gramática

Como podemos ver na Figura f, o conjunto de condições é expresso na forma de uma condição, seguida de um conector “and” ou “or”, e outra condição, e assim sucessivamente.

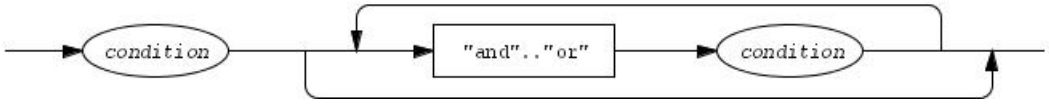


Figura f. Termo “conditions” da gramática

Cada condição é composta por dois operandos e um operador. Como podemos ver na Figura g, os possíveis operadores da gramática são: “equal”, “not_equal”, “greater_or_equal_to”, “greater_than”, “lower_or_equal_to” e “lower_than”.

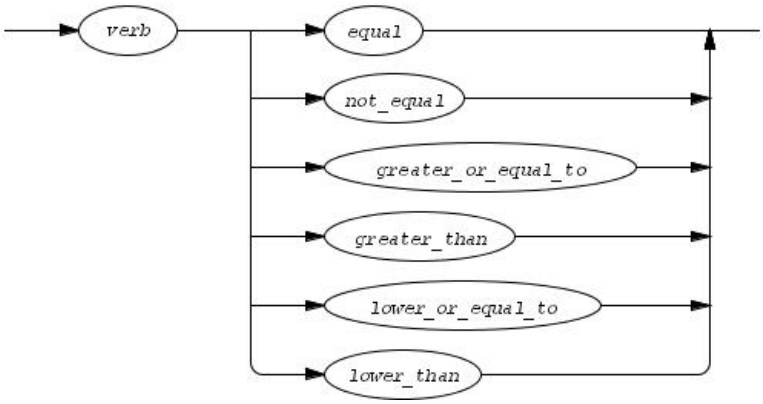


Figura g. Termo “condition” da gramática

As iterações são representadas por uma coleção a ser iterada e um conjunto de operações e expressões condicionais a serem executadas, como mostrado na Figura h.

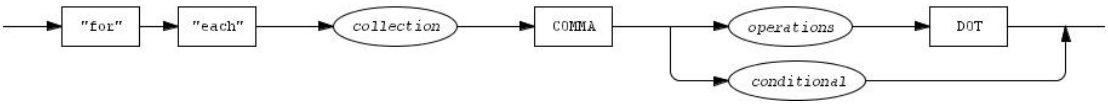


Figura h. Termo “loop” da gramática

Para representar os eventos, criamos a expressão “exceptional”, que está representada na Figura i. Existe uma pequena diferença entre a expressão condicional e a expressão de eventos. As expressões de eventos devem ser utilizadas para representar fluxos alternativos de casos de uso.

Figura i. Termo “exceptional” da gramática

A expressão “after all”, mostrada na Figura j, foi criada para atender a necessidade de declarar comandos a serem executados ao final do escopo de uma iteração.

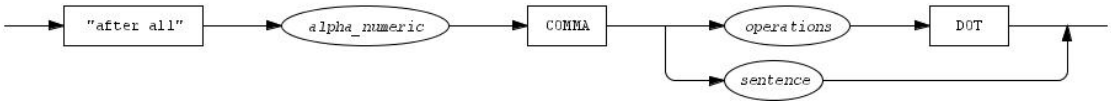
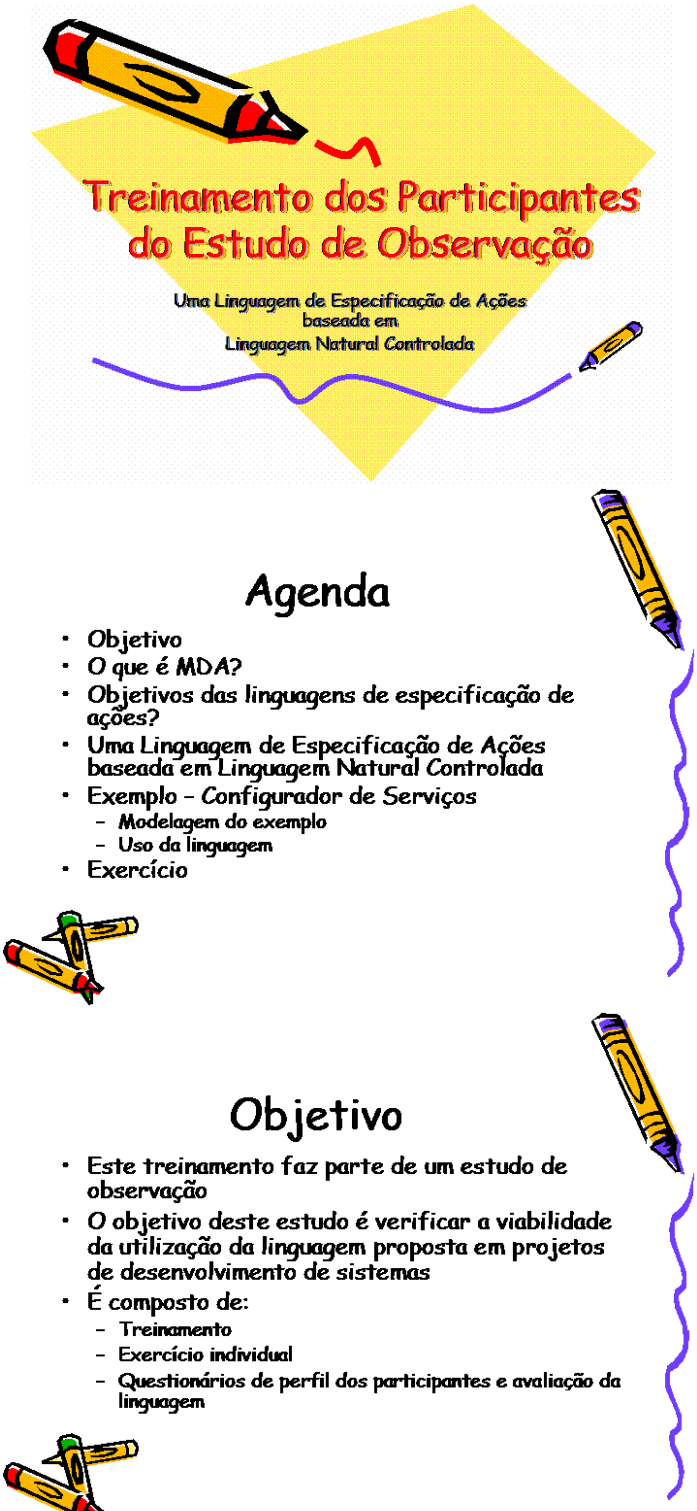


Figura j. Termo “after all” da gramática

Apêndice 7

TREINAMENTO DA LINGUAGEM NATURAL MDA COM OS PARTICIPANTES DO ESTUDO DE OBSERVAÇÃO



O que é MDA?

- *Model Driven Architecture*
- É uma abordagem de desenvolvimento de sistemas baseada em modelos UML, que foi definida pela *OMG*
- Objetivo:
  - Prover uma separação entre a especificação de sistemas e os detalhes específicos de tecnologia

Como funciona?

- Níveis dos modelos
  - CIM (Computation Independent Model)
  - PIM (Platform Independent Model)
  - PSM (Platform Specific Model)
- Transformações sucessivas são aplicadas nos modelos com o objetivo de gerar o código-fonte

Por quê linguagens de especificação de ações?

Linguagem de ações

- Uma linguagem de ações é uma linguagem de especificação para especificar operações em um diagrama de classes
- Requisitos de linguagens de ações:
  - Padrão na indústria
  - Verificação através de simulação
  - Completude na geração de código através de modelos
- Características
  - Compatível com a UML
  - Executável e completa
  - Independente de implementação
  - Nível de abstração acima da implementação

Uma Linguagem de Ações baseada em Linguagem Natural Controlada

- Tipos de sentenças
  - Invocações de operações
  - Iterativas
  - Condicionais
  - Eventuais

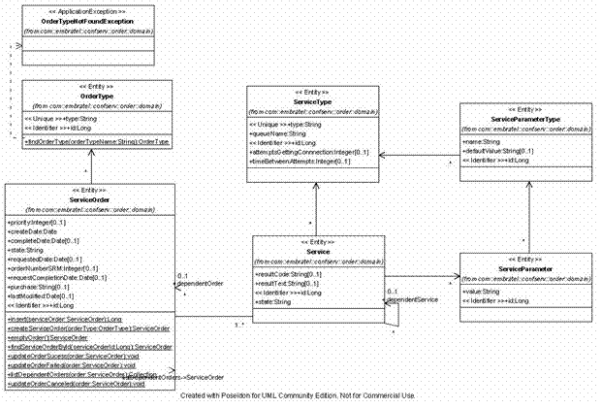
Uma Linguagem de Ações baseada em Linguagem Natural Controlada

```
rule         : ((operations DOT) | (sentence) | (after_all) )+;
operations: (operation ((COMMA | AND) operation)*) |
              (show_error_warning) |
              (return_statement);
show_error_warning : SHOW (ERROR | WARNING) ASPAS message ASPAS;
return_statement   : RETURN alpha_numeric;
sentence: (conditional (OTHERWISE COMMA operations DOT)?) |
          (loop) | (exceptional) ;
conditional : IF conditions COMMA ((operations DOT) | loop);
Loop         : FOR EACH collection COMMA
               ((operations DOT) | (conditional )) ;
exceptional : WHEN event COMMA operations DOT;
after_all    : AFTER_ALL alpha_numeric COMMA
               ((operations DOT) | (sentence));
Conditions : condition ((AND | OR ) condition)*;
condition : operand verb ( (equal | not_equal | greater_or_equal_to | greater_than | lower_or_equal_to
                           | lower_than) ) operand;
```

Exemplo - Descrição

- Configurador de ordens de serviços
  - Criação de ordens
  - Envio de ordens
  - Inicialização de ordens
  - Notificação de conclusão

Modelagem de domínio

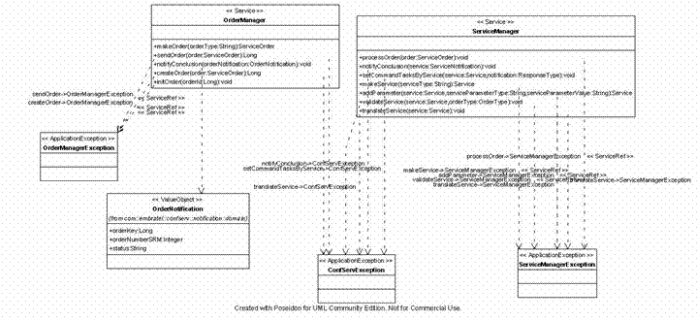


Descrição dos elementos do modelo

- Service Order
  - Insert
    - element.description= insert the order
  - CreateOrder
    - element.description= build the service order
    - return.description= the order
  - FindServiceOrderById
    - element.description= find service order
  - UpdateOrderSucess
    - element.description= update order to success
  - UpdateOrderFailed
    - element.description= update order to failed
  - UpdateOrderCanceled
    - element.description= update order to canceled
  - ListDependentOrders
    - element.description= list the dependent orders
    - return.description= dependent list
  - EmptyOrder
    - Element.description= empty order

- OrderTypeNotFoundException
  - element.description= order type was not found
- OrderType
  - FindOrderType
    - element.description= find order type
- ServiceManagerException
  - element.description= service errors occur
- OrderNotification
  - orderKey
    - element.description= order identifier
  - status
    - element.description= order status

Modelagem de serviço



Descrição das operações de serviço

- *ServiceManager*
  - ProcessOrder
    - element.description: Process the order
  - ValidateService
    - element.description: validate the service
- *OrderManager*
  - makeOrder
    - descrição dos parâmetros: order type
    - element.description: Make the order
    - operation.behaviour: Find order type, build the service order *and return* the order. *When* order type was not found, *show error* 'order type not found' *and return* empty order.
  - initOrder
    - parâmetros: order identifier
    - element.description: Initialize the order
    - operation.behaviour: Find service order *and* process the order. *When* service errors occur, *show error* 'A service error occurred'.

Descrição das operações de serviço (cont.)

- *OrderManager*
  - notifyConclusion
    - parâmetros: conclusion notification of the order
    - element.description: Notify the conclusion of the order
    - operation.behaviour: Find service order. *If* order status *is equal to* success, update order to success, list the dependent orders *and for each* dependent *of the* dependent list, process the order. *Otherwise*, update order to failed, list the dependent orders *and for each* dependent *of the* dependent list, update order to canceled. *When* service errors occur, *show error* 'A service error occurred'.
  - sendOrder
    - descrição dos parâmetros: order
    - retorno: order identifier
    - element.description: Send the order
    - operation.behaviour: Create the order *and* initialize the order.

Exercício

- Descrever a operação "createOrder" da classe "OrderManager" utilizando a linguagem proposta
  - Pré-condição: todos os serviços devem ser válidos
  - Pós-condição: a ordem deve ser persistida

Questionário de Avaliação

- Responder os questionários de avaliação de perfil dos participantes e avaliação da linguagem proposta

Método makeOrder

```
/**
 * make the order
 * find order type, build the service order and return the order. When order type was not found, show
   error "order type not found" and return empty order.
 * @param orderType  order type
 */
public com.embratel.confserv.order.domain.ServiceOrder makeOrder(
    java.lang.String orderType) {
    try {
        com.embratel.confserv.order.domain.OrderType var1 =
com.embratel.confserv.order.domain.OrderType.findOrderType(orderType);
        com.embratel.confserv.order.domain.ServiceOrder var2 =
com.embratel.confserv.order.domain.ServiceOrder.createServiceOrder(var1);

        return var2;
    } catch (com.embratel.confserv.order.domain.OrderTypeNotFoundException var3) {
        logger.error("order type not found");

        return com.embratel.confserv.order.domain.ServiceOrder.emptyOrder();
    }
}
```

Método sendOrder

```
/**
 * send the order
 * create the order and initialize the order.
 * @param order  order
 */
public java.lang.Long sendOrder(
    com.embratel.confserv.order.domain.ServiceOrder order)
    throws com.embratel.confserv.order.service.OrderManagerException,
      com.embratel.confserv.order.service.ConfServException {
    java.lang.Long var1 = this.createOrder(order);
    this.initOrder(var1);

    return var1;
}
```

Método notifyConclusion

```
/**
 * notify the conclusion of the order
 * Find service order, if order status is equal to success, update order to success, list the dependent orders and for each dependent of the dependent list,
      process the order. Otherwise, update order to failed, list the dependent orders and for each dependent of the dependent list, update order to
      canceled. When service errors occur, show error "A service error occurred".
 * @param orderNotification  conclusion notification of the order
 */
public void notifyConclusion(
    com.embratel.confserv.notification.domain.OrderNotification orderNotification) {
    try {
        com.embratel.confserv.order.domain.ServiceOrder var1 =
        com.embratel.confserv.order.domain.ServiceOrder.findServiceOrderById(orderNotification.getOrderReq());

        if (orderNotification.getStatus().equals(var1)) {
            com.embratel.confserv.order.domain.ServiceOrder.updateOrderSuccess(var1);

            java.util.Collection var2 = com.embratel.confserv.order.domain.ServiceOrder.listDependentOrders(var1);

            for (java.util.Iterator var3 = var2.iterator(); var3.hasNext();) {
                com.embratel.confserv.order.domain.ServiceOrder var4 = (com.embratel.confserv.order.domain.ServiceOrder) var3.next();
                com.embratel.confserv.order.service.ServiceManagerBI var5 = locator.ServiceLocator.instance()
                                                        .getServiceManagerBI();
                var5.processOrder(var4);
            }
        }
        else {
            com.embratel.confserv.order.domain.ServiceOrder.updateOrderFailed(var1);

            java.util.Collection var2 = com.embratel.confserv.order.domain.ServiceOrder.listDependentOrders(var1);

            for (java.util.Iterator var3 = var2.iterator(); var3.hasNext();) {
                com.embratel.confserv.order.domain.ServiceOrder var4 = (com.embratel.confserv.order.domain.ServiceOrder) var3.next();
                com.embratel.confserv.order.domain.ServiceOrder.updateOrderCancelled(var4);
            }
        }
    } catch (com.embratel.confserv.order.service.ServiceManagerException var2) {
        logger.error("A service error occurred");
    }
}
```

Método initOrder

```
/**
 * initialize the order
 * find service order and process the order. When service errors occur, show error 'A
   service error occurred'.
 * @param orderId  order identifier
 */
public void initOrder(java.lang.Long orderId)
    throws com.embratel.confserv.order.service.OrderManagerException {
    try {
        com.embratel.confserv.order.domain.ServiceOrder var1 =
  com.embratel.confserv.order.domain.ServiceOrder.findServiceOrderById(orderId);
        com.embratel.confserv.order.service.ServiceManagerBI var2 =
  locator.ServiceLocator.instance()
                                                        .getServiceManagerBI();
        var2.processOrder(var1);
    } catch (com.embratel.confserv.order.service.ServiceManagerException var3) {
        logger.error("a service error occurred");
    }
```